ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING
project report for the examination
of
Computational Intelligence and Deep Learning

Student:
Alessandro Diana

AA. 2022/2023

Contents

List of Figures

1 Introduction

Forest fires are the most common natural disaster and represent a serious risk to the health of people and animals, the integrity of infrastructure, and the economy. According to the NFPA(National Fire Protection Association [1]) [2], about 1,353,500 fires were reported in the US during 2021, which resulted in 3,800 civilian death, 14,700 civilian injuries, and estimated $15.9 billion in direct property damage.
For these reasons, it is important to boost fire detection research in the safest way possible. Timely forest fire recognition is crucial and can greatly reduce the number of fatalities and damage caused by these events.
Fast fire recognition can also play a key role in rapidly extinguishing fires, preventing them from turning much dangerous and difficult to manage large-scale wildfires. Examples of this phenomenon are the Australian bushfires, one of which began in 2019 and lasted until March 2020, burning at least half a billion animals alive. Another case started in early March 2023 that burned 18 000 hectares of land, destroyed six houses, and killed at least 200 livestock[3].

Today, there are several methods for fire detection. Conventional methods are based on sensors that detect smoke, atmospheric temperature, ionization, and so on.
These sensors are widely popular due to their simple use and low cost. However, they have weaknesses: they have late alarm activation, problems in covering large areas, and signal transmission.
In fact, almost all of these sensors require proximity to smoke or fire. As a result, the alarm is triggered after the smoke/fire reaches the sensor, causing a delay that can be critical.

Therefore, these methods are unsuitable for early warning, and detection in large spaces, open spaces, or large infrastructures, making them unsuitable for the case of forest fires in large open areas.
Given these scenarios, an image-based fire detection system can be optimal and effective. For this problem of image-based recognition, artificial intelligence, in particular, CNN ('Convolutional Neural Networks')[4] can give an enormous contribution and excellent results, given the great development and improvement of CNNs in recent years.

The objective of my work is to create a CNN that can recognize the presence of forest fires. I will start from already existing networks (AlexNet and GoogLeNet) in order to build a customized network for the problem analyzed. Real images of forest fires are used for training and testing.

2 State of Art

2.1 Why use CNN?

As mentioned in the paragraph above forest fire detection is a non-simple problem in which "traditional" sensors cannot be deployed with sufficient effectiveness. It is a problem that requires monitoring of large forest areas and in which early warning can save lives and avoid extensive damage.
For these reasons, a system that can reliably detect fire via images or video would be an excellent solution to the problem.
In recent years, in the field of image recognition, there has been strong interest and development through the use of artificial intelligence. In particular, excellent results are obtained through CNNs (Convolutional Neural Networks), which are a type of network. This type of neural network specializes in processing data in grid form, such as an image. For these reasons, CNNs were chosen to be used to be able to do fire detection through images.

2.2 Common CNN Architectures used

There are numerous articles that have approached the problem of fire detection using CNN and working on both images and video. I will briefly mention a few to give a point of reference and insight into the problem.

In this article[5] they start from the architectures of the AlexNet and SqeezeNet to create their CNN, which had to be light in order to run efficiently on Jatson Nano using PyTorch. Their problem consists of outdoor fire detection, more specifically a 3-class classification (fire, smoke, neutral). They used 2 datasets: the first [6] consisting of 900 images and the second [7] consisting of about 550 images (also used data augmentation). These datasets are both available on GitHub, I did not use them for the project because they consisted of too varied images not focused on the forest focus . The final model they made is 683.1kB in size and achieved 79.66% accuracy on the first dataset and 84% on the second dataset (with data augmentation)

In this paper[8] they took several models of existing networks (AlexNet, Resnet50, VGG16, GoogLeNetV3, and EfficientNetB0) on which transfer learning was used and compared the various results obtained on each. Their problem was a generic outdoor fire detection. The proposed dataset is available on Kaggle [9]. I did not consider it for the project for several reasons.

- The number of images stated in the paper does not correspond with the actual size of the dataset on Kaggle (much smaller, a few hundred images);
- The images in the dataset are very varied, not representing forest fire well.

The best results were obtained from EfficientNetB0, implemented using TensorFlow and Keras framework while training and testing were done on Google Colab. EfficientNetB0 has 4,096,378 parameters and an accuracy of 95.4%.
In this paper[10] a network for indoor flame and smoke localization is made. They took several models of existing networks: Faster R–CNN Inception V2, SSD MobileNet V2 models, GoogleNet and SqueezeNet.

All these custom networks start from known architectures mentioned above by simplifying them and trying to optimize and adapt them as best as possible for fire detection. The results

obtained are excellent in all cases.

This project aims to go for a custom network that can give good performance for the outdoor fire detection problem through a network that has a number of parameters to train limited.

3 System configuration

3.1 Hardware and Software set up

For training and testing the CNN models for this project, I used different configurations. More specifically, I used two of my own computers and Google Colab.

Two different computers with different hardware and software were used:

- computer1(laptop): has Windows 10 as its operating system on which Python 3.10, cuda 11.2, cuDNN 8.1, and TensorFlow 2.10.0 are installed. On the hardware side, it has an RTX 3060 mobile (3840 CUDA cores, 6GB GDDR6, 120 Tensor Cores, 30 RT Cores), an i7-10870H (8 cores and 16 threads), and 16 GB DDR4;
- computer2(desktop): has Windows 10 as its operating system on which Python 3.10, cuda 11.2, cuDNN 8.1, and TensorFlow 2.10.0 are installed. On the hardware side, it features a GTX 1080 OC (2560 CUDA cores, 8 GB GDDR5X), i7 7700k (4 cores and 8 threads), and 16 GB RAM DDR4.

All network training was performed on both computers. The performance of the trained models in the two computers is similar to each other, what changes is the training time and GPU utilization. The configuration with 1080 is slower, has mean utilization of 18-20%, and has a peak utilization of 35% while the configuration with the 3060 is faster, has mean utilization of 10-12%, and has a peak utilization of 18%.

The configuration with the RTX 3060 has advantages due to the higher power of the graphic card and especially for the presence in it of tensor cores, which are missing in the GTX 1080. Briefly, tensor cores are specialized cores for computing mixed-precision matrix multiplications (you can learn more about them on Nvidia's dedicated page[11]). An increase in the speed of this type of operation brings advantages in training in a deep neural network (which makes extensive use of this operation).

Google Colab is a free environment that runs via the cloud and can save files to google drive. It allows to program in Python using notebooks and allows you to run them using the GPU computing power provided for free. A fast environment that needs no configuration (it already has several libraries built in such as TensorFlow and Keras that are useful for NN). In the project I ran the programs on Google Colab using Python 3 and as a hardware accelerator a T4 GPU (runtime settings). In addition, Colab provides 12.7 GB of RAM and 78.2 GB of disk space as execution resources. The Nvidia Tesla T4 GPU is a professional card specialized for AI work in which there are 320 Tensor Cores, 2560 CUDA cores, and 16 GB GDDR6.
This environment was the slowest of all those I tested. The training and testing time of the CNNs is on average 1.5 to 3 times slower with peaks up to 7-8 times longer than the other configurations (e.g., training steps with batch size 128 done in 1s compared to 144ms in the laptop configuration).

From the experience for this project, Google Colab is a good tool because it gives everyone the possibility to be able to do calculations taking advantage of performance given by GPUs made especially for AI purposes however it is very limited both as time of use and as work you can do on the free account.
Graphics cards for home use (most video game utilities) are good options for those who want to work and also a lot in AI without having to spend on specific tools and for AI use only. These

cards have become viable, especially in the last few years when in the Nvidia video game field the increase in AI has grown considerably and thus consequently also the presence of specific hardware and software.
As found in the project, solutions (GPU) for laptops are also valid, especially if belonging to the latest generations even if they have the defect of having much less memory which can be a high obstacle when dealing with complex network architectures and very large and expensive datasets in terms of the space occupied by their individual elements.
This is why I think the best solution for the home are GPUs for desktop computers, which can count on greater computing power and memory than their counterparts for laptops, at a cost of greater consumption and space used.
In fact, from what I was able to test in this project, the GTX 1080 (even if from a generation not designed for AI) has had good performance anyway. In some cases of programs that had to run for a long time and with greater use of memory (k-cross validation program for the parameters), it managed to perform better than the RTX 3060, especially for the greater memory available which avoided any crashes due when the memory is full.

3.2 Dataset

3.2.1 Dataset explanation

The dataset used to train the CNN models is a combination of other smaller datasets. All images, both with and without fire, are related to natural outdoor landscapes.

The dataset used can be found on Kaggle at the link:
https://www.kaggle.com/datasets/mohnishsaiprasad/forest-fire-images
Its volume is approximately 404 MB.

this dataset contains 5050 images belonging to two classes: Fire and Non_Fire. The images are divided into 2 folders named Train_Data and Test_Data.
Train_Data folder contains 5000 images split equally into Fire and Non_Fire folders and classes.
Test_Data folder contains 50 images split equally into Fire and Non_fire classes taken from Google.

An analysis of the dataset revealed that 391 images were corrupted and therefore unusable. These corrupted images mostly belong to the fire class, which means that the dataset is no longer perfectly balanced, but remains balanced. The dataset now consists of 4609 images in total of which 2499 belong to the non_fire class and 2110 belong to the fire class.

All the images in the dataset are of the same format, which is jpg. The images are all in color and follow the RGB model, so they have 3 channels to represent the colors.
The shape of the images varies a lot, probably due to the nature of the dataset which is composed of several different small datasets. In this dataset, there are 719 different shapes for the images. The most common shape is 250x250x3 which is represented by 1908 images. The second most common resolution is 1080x1920x3 which is represented by 206 images. 65.16% of the images (3258) belong to the 10 most common shapes, this indicates that most of the images belong to a few formats and that the remaining hundreds of different formats are represented by 1 or a few images.

All images will be resized to the 224x224x3 shape before being used.
These images resized with this shape and saved as Numpy arrays occupy 0.694 Gb of memory.

Image resizing is done using the OpenCV-python library via the resize method. Image resizing can be done through various interpolation methods:

- INTER_NEAREST: This method uses the nearest neighbor concept for interpolation. This is one of the simplest and fastest methods, using only one neighboring pixel from the image for interpolation, but loses substantial information when resizing;
- INTER_LINEAR: This method is somewhat similar to the INTER_CUBIC interpolation. But unlike INTER_CUBIC, this uses 2×2 neighboring pixels to get the weighted average for the interpolated pixel. Is less fast than inter_nearest, but will not result in information loss unless you're shrinking the image, but causes some undesirable softening of details and can still be somewhat jagged;
- INTER_CUBIC: this method uses bicubic interpolation for resizing the image. While resizing and interpolating new pixels, this method acts on the 4×4 neighboring pixels of the image. It then takes the weights average of the 16 pixels to create the new interpolated pixel. Cubic interpolation is computationally more complex and hence slower than linear interpolation. However, the quality of the resulting image will be higher;
- INTER_AREA: this method uses pixel area relation for resampling. This is best suited for reducing the size of an image (shrinking). When used for zooming into the image, it is similar to the INTER_NEAREST method. INTER_AREA works better in image decimation and avoiding false inference patterns in images but at the cost of the longest processing time;

In general, if you have to:

- shrink the image, it is best to use Area interpolation;
- zoom in on the image, it is best to use the Linear, Cubic, or LANCZOS4 interpolation methods (in the decreasing order of speed).

For our case, we will mostly have a reduction of the images to fit the network model. We have no restrictions on the loading time of the dataset, which is not very large. For these reasons, I chose area interpolation, at the cost of a longer resize time (an extra 30% from empirical tests on this dataset) we have better quality.

Our fire detection problem has bias due to its nature. One expects images containing a fire to have a large prevalence of red color, but one must be careful that the network does not just learn that images with a lot of red are fire because this is not always the case.

For this reason and more analysis, I did a check on the distribution of colors in the class images.

For each image in each class, the average value of the B, G, and R values is taken and then these contributions are added together. At the end I have for each class the contribution of the colors red, green, and blue given by the images, in this way, you can see the color balance for the classes in the dataset.

As shown in bar graphs 1, the fire class is characterized by a predominance of red, as expected, and the Non_Fire class is balanced on all 3 colors with a slight predominance of green.
This balance in Non_Fire class, the good presence of red, should help to avoid the net always learning that red indicates fire. Something that is not expected is the strong presence of blue,

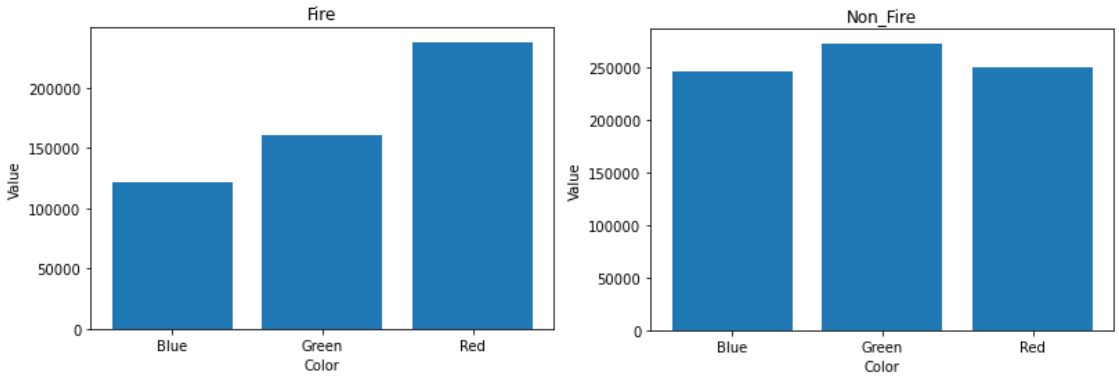


Figure 1: Illustrative image of colour balance in Fire class images (left) and in Non_Fire class images (right).

a quick visual analysis of the dataset led to finding many images in the Non_Fire class given by bare snow-capped mountains that probably give a large contribution of blue.

More in-depth analysis of the dataset and its improvement will be explained in the following paragraph.

3.2.2 Dataset improvement

This paragraph describes a more depth analysis done on the dataset.
Before going into the analysis it is good to remember the problem to which our network must respond, this is because different analyzes can be made and different judgment methods can be applied to the images and data obtained to discriminate which are useful and which are useless. Some of these judgment methods are not influenced by the specific problem (e.g. damaged images or images with high noise) while others are (e.g. understanding the relevance of the image or the congruence of the image with the given problem). In my analysis, I have discarded images both for quality reasons not related to the problem and for reasons of congruence with the problem.

The problem faced in this project is forest fire detection, we want to train a network that is able to understand the presence of forest fires through images in order to raise the alarm, so the best images are those that show different forest landscapes in different seasons and of different types and then different types of forest fires.

Now I can start to describe the analysis.
I started the analysis of the dataset from the images belonging to the non_fire class. I viewed the images one by one and found several images that in my opinion do not fit the problem of this project and therefore need to be removed to make the dataset more congruous for the purpose. I have divided the removed images into categories that express their subject. Below I list these categories with my motivation for why they do not fit the forest fire detection problem and show an example for each category.

- snow-capped mountains: these are images depicting snow-capped mountains, mainly made up of snow and rock. this type of images do not depict any type of forest, if they had mainly framed wooded areas they would have been useful for the network because they

would have represented winter forests. But since there are no trees they are of no use to our problem. The images removed from the dataset belonging to this category are 209;

- barren ground: these are images that depict landscapes without vegetation, or with sporadic vegetation but no trees. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 71;
- towns: these are images in which urban environments are present which for the most part are without vegetation. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 69;
- rocks: these are images in which there are rocky environments without trees. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 61;
- fireworks: these images show fireworks at night near a beach and buildings. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 41;
- beach: these are images of a beach where you don't see any type of vegetation and there are sports facilities. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 34;
- satellite: these are satellite images of different types of areas (countryside, cities, etc...). The idea for this application is to make fire detention through images of cameras or video cameras placed on the ground that have a more or less wide view of portions of the surrounding forest. for this reason, satellite photos even if of wooded areas are not useful. The images removed from the dataset belonging to this category are 20;
- not real: they are images taken from a game, so they are not real images but rendered by a game engine. furthermore, these images represent fantasy and disturbing elements such as characters and monsters and fights between them. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 4;
- damaged: these are images that contain errors making it impossible to view them correctly and therefore unusable. The images removed from the dataset belonging to this category are 4;
- fire: images showing flames that are in this section of the dataset by mistake. The images removed from the dataset belonging to this category are 2;
- sea: images in which the sea was present, I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 2;
- plant zoom: images of plants taken very close up are not useful for a forest monitoring application, they are a view of the plants in too much detail. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 2;
- others: The images in this category belong to several categories that are represented by only one image. There is an image removed because it is an exact copy of another image, an image of a motorcycle, an image with a copyright layer that distorts it and an image showing a person.

Figure 2: Illustrative image of some examples of images discarded by the Non_fire class.

Image 2 shows sample images from the categories described above. The images shown were all present in the original dataset and were removed with this analysis, they are shown to better understand the nature of the discarded images.
In total, I removed 523 images belonging to the non_fire class.

Then I continued the analysis of the dataset from the images belonging to the fire class. I viewed the images one by one and found several images that in my opinion do not fit the problem of this project and therefore need to be removed to make the dataset more congruous for the purpose. I have divided the removed images into categories that express their subject. Below I list these categories with my motivation for why they do not fit the forest fire detection problem

and show an example for each category.

- city fire: these are images showing fires in the city setting, without vegetation. I removed these images because they are not suitable for our problem. The images removed from the dataset belonging to this category are 223;
- non_fire: images in which no flames are present. In some cases, there are forests without fire and thus were included by mistake. In other cases, they are fire images that focus on elements outside the flames (e.g., firefighter planes and helicopters). The images removed from the dataset belonging to this category are 24;
- others: The images in this category belong to several categories that are represented by only a few images. There are zoom images of a fireplace fire, there is an image where there is mostly road, there are images that are zoomed on people, and an image with a copyright layer that distorts it.

Figure 3: Illustrative image of some examples of images discarded by the fire class.

Image 3 shows sample images from the categories described above. The images shown were all present in the original dataset and were removed with this analysis, they are shown to better understand the nature of the discarded images.

In total, I removed 256 images belonging to the fire class.

Overall after these analyses, I removed 779 images from the original dataset.
Now the dataset consists of 3832 images, of which 1854 belong to the fire class and 1978 belong to the non_fire class. After these changes, the dataset remains balanced.
These images resized with this shape (224,224,3) and saved as Numpy arrays occupy 0.577 Gb of memory (17% less than the old dataset).

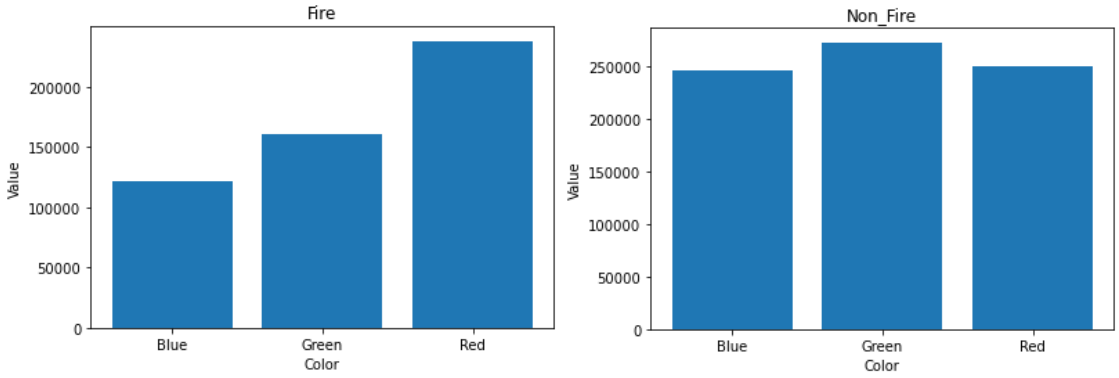


Figure 4: Illustrative image of colour balance in Fire class images (left) and in Non_Fire class images (right) in the new dataset.

I now analyze the color distribution in the images, shown in the bar graphs 4, to see if there are any differences with that in the dataset before the changes.
Comparing the distributions of both classes before and after the changes to the dataset one can see that the distribution of colors has remained almost unchanged. This is a good thing because the distribution was good and if it remained the same in the new dataset which is composed of images more congruent to our problem this should lead to an improvement in the quality of our CNNs.

A good quality dataset from which to take the input data is essential to train CNNs as correctly and as well as possible. As mentioned above, in my opinion, the original dataset had errors and many images that were not consonant with the problem. After my analysis and editing, its quality has improved but it can still be improved (for example):

- improve the images in the fire category by including images with different angles and where the fire occupies different percentages of the image and zones;
- improve the images in the non_fire category by putting more of them relevant to forested areas of different types and from different angles.

Considering that images for these types of problems are difficult to find or to recreate artificially and that this work was done as a project for a university exam, thus with limited resources and time, I judge the dataset to be improvable but of an acceptable quality to carry on with the project and the making of a CNN.

3.3 Epochs and early stopping

An epoch means training the network with all training data at once. So with each epoch, we will only use all training data once. The choice of the number of epochs to be made when training a network is very important. A number of epochs too high would lead to network overfitting, while a number of epochs too low would lead to network underfitting.

- Overfitting occurs when the model cannot be generalized and overfits the training data set. In this case, the accuracy of the model will be very high for the data in the training set and lower for the data in the validation and test set;
- Underfitting occurs when the model is unable to determine a meaningful relationship between the input and output data. In this case, the accuracy of the model will be low for both the data in the training set and the test set.

To handle overfitting, one can set the number of epochs high and use the early stopping techniques. This technique checks at each epoch made the performance of the network, on the validation set, and stops training when: the desired performance is reached, when there is no increase from a specified number of epochs or when the maximum number of epochs has been reached.
This method makes it possible to manage overfitting and save time in training. However, depending on the set parameters early stopping may stop the training when overfitting has already started, returning not the best model. To avoid this, checkpoints are used in combination with the callback function. This allows a history of the best models during training to be saved and then retrieved.

In the training phase for the models in this network, I used high epoch numbers (e.g. 100,200) and I used early stopping which controlled the loss function by going to stop training if no improvement occurred for a number of epochs in a row (e.g. 10,15,20).
I chose the loss function as the metric for early stopping training because it represents the average value of the error of the generated outputs of the network with respect to the given labels. An improvement in the loss function will represent an overall improvement in network performance.

3.4 Optimizer

In order to train a network, one must also specify an optimizer, which is an algorithm used to minimize loss by adjusting various parameters and weights, so as to provide better model accuracy in a shorter time.

Some of them are:

- SGD (Stochastic Gradient Descent): method executes a parameter update for every training example. In the case of huge datasets, SGD performs redundant calculations resulting in frequent updates having high variance causing the objective function heavily fluctuates during training. This fluctuation can be an advantage because it allows the function to jump to better local minima, but at the same time, it can represent a disadvantage with respect to the convergence in specific local minima. A solution to this problem is to slowly decrease the learning rate value in order to make the updates smaller and smaller, avoiding high oscillations. This method required less memory but is computationally expensive

- AdaGrad (Adaptive Gradient Algorithm): adapts the learning rate to the parameters performing different updates depending on the occurrence of features. With frequent features the updates are small and with rare features the update is large. With this method, the network is able to take information from rare features and give it the right weight to highlight them. The learning rate becomes small with an increase in the depth of the neural network and could result in not being able to learn anymore;

- RMSprop (Root Mean Square Propagation): this method only accumulates gradients in a particular history fixed window which sets a number of past gradients to take into consideration during the training. RMSProp can be considered as an updated version of AdaGrad with few improvements and doesn't have the problem of the vanishing learning rate.

- Adam (Adaptive Moment Estimation): combines advantages of both RMSprop and Adadelta and stores an exponentially decaying average of past gradients similar to momentum. This method does not need to focus on the learning rate value, its implementation is easy, requires less memory, and is computationally efficient but can have a weight decay problem and may not converge to an optimal solution

For the model in this project, I chose the Adam optimizer, with default parameter value, because is the best optimizer in general.
I used parameters different from the default ones only at the end to see if it was possible to have better performance from the chosen model through a special choice of learning rate. The learning rate is a parameter that establishes the magnitude of the jump in the direction toward the minimum of the loss function. A learning rate that is too low could take too long to converge and could get stuck in local minima while a learning rate that is too high could give oscillation problems and miss the global minimum. By default, the value for the learning rate in the Keras Adam optimizer is 0.001. I will use Keras hyper tuning to go and try different learning rate values and see which gives the best result.

3.5 Activation function used

In the networks used by this project, these activation functions are used: ReLU (shown in fig 5) and Softmax.

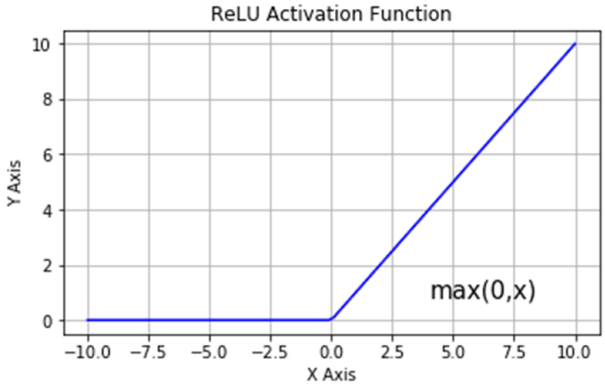


Figure 5: Illustrative image of ReLU activation function.

ReLU is the most common and widely used activation function due to its simplicity in computation and backpropagation.

$$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

Figure 6: Illustrative image of the Softmax activation function formula.

Softmax is a function that transforms numbers into probabilities, its output is a vector of size equal to the number of classes and each of its individual values indicates the probability of belonging to that particular class. Mathematically, it is defined as shown in fig.6, where:

- y is an input vector of n dimension;
- y_i is the ith element of the input vector y and may have any value between -inf and +inf;
- $\exp(y_i)$ is an exponential function calculated on y_i, its value can range from almost zero if y_i is less than 0, to + infinity if y_i) is large.
- $\sum_{j=1}^{n} exp(y_i)$ is the normalization term that ensures that the output vector has the following properties: the sum of its elements is one and the value of each of them is between 0 and 1 (so that it is a valid probability distribution);
- n is the number of classes.

I used the Softmax function in the output layer.

3.6 Batch size

The batch size defines the number of samples that will be propagated through the network. The smaller the batch size is, the less memory is needed and the less accurate the estimation of the gradient is.
In this project, the batch size used are 32, 64, and 128.

3.7 Dropout utilization

Another technique to avoid overtraining and help generalization is Dropout. This technique drops out the nodes (input and hidden layer) in a neural network. All the forward and backward connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p.
In the models of this project, dropout is usually used in the last fully connected layers. The usual p-values used are 0.3 and 0.5.

3.8 General approach

This section describes the general approach I used for the creation of the CNN models, their training, and testing.

1. I applied the AlexNet and GoogLeNet networks to the fire detection problem. The results obtained from these networks will be used as a comparison with the results obtained from the Ifrit networks;

2. I created several models of Ifrit networks taking the above-mentioned networks as a basis;

3. I analyzed the performance of the networks on validation and test sets and also the training times. I compared the performance of the models to see which were the best;

4. By analyzing the results of the models, I improved and extended their structure to improve their results and then retested them.

I followed this method by redoing the steps several times until I was able to make an Ifrit CNN that was able to achieve very good results.

4 CNN Architectures

4.1 Known CNNs used

In this section I will briefly discuss two of the most widely used and well-known CNNs of recent years, these networks served as a starting point and comparison benchmark for the CNN models created and tested in this work.
The two networks examined are AlexNet and GoogLeNet both winners of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)[12].

4.1.1 AlexNet

AlexNet is a CNN model designed by Alex Krizhevsky and Ilya Sutskever, under the supervision of Geoffrey Hinton. AlexNet represented a fundamental breakthrough in image classification problems and won the ImageNet Large Scale Visual Recognition Challenge in 2012.
The special feature added by this network is to have several convolutional layers in succession instead of alternating them in pooled layers. AlexNet takes 224x224x3 images as input, and has three channels, allowing us to handle color images (RGB scheme).
AlexNet has eight layers: the first five layers are convolutional, of which the first two use max-pooling, while the last three levels do not, being fully connected.
Respectively, the filters applied to the convolutional layers are:

- the first with 96 kernels of size 11×11×3 and stride 4;
- the second with 256 kernels of size 5×5×48 and stride 1 (48 because divided between 2 GPU);
- the third with 384 kernels of size 3×3×256 and stride 1 (256 because it is a communicating layer between the 2 GPUs);
- the fourth with 384 kernels of size 3×3×192 and stride 1 (192 because it is divided between 2 GPUs);
- the fifth with 256 kernels of size 3×3×192 and stride 1(192 because divided between 2 GPUs).

The last three layers are fully connected layers and consist of 4096,4096,1000 neurons respectively.

The network uses local Response Normalization (LRN) and the ReLU activation function, except for the last layer which uses the Softmax activation function. With the exception of the last layer, the rest of the network was split into two copies, running separately on two GPUs as shown in the figure7.
In this project was redesigned a version running on a single GPU.

The trained model of this network occupies 189.06 MB and has 24,754,242 trainable parameters.
For more details on AlexNet see the authors' paper [13].

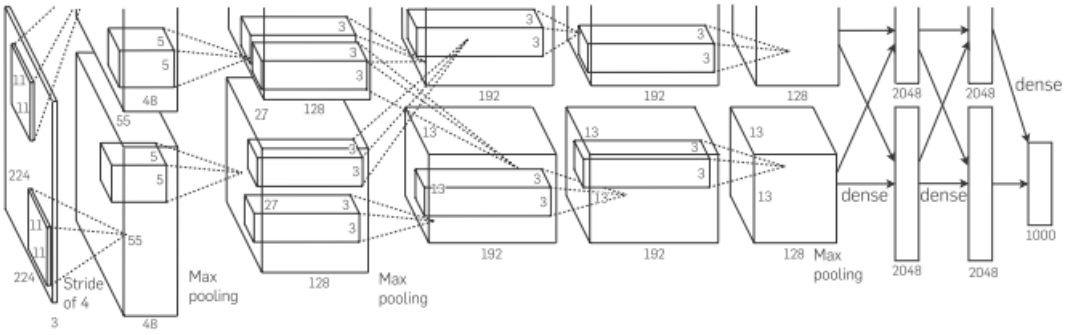


Figure 7: Illustrative image of the AlexNet architecture divided into the two training GPUs.

4.1.2 GoogLeNet

GoogLeNet was created by Google Inc. and the model was published in the paper "Going Deeper with Convolutions".GoogLeNet won ILSVRC-2014 and is one of the most successful models of the earlier years of CNN.
This architecture takes input images of size 224x224x3 with RGB color channels. The overall architecture has a depth of 22 layers (as shown in the figure 8) and also contains two auxiliary classifier layers connected to the output via inception layers. These layers help combat the problem of gradient vanishing. The architecture was designed with computational efficiency in mind. The idea behind it is that the architecture can be executed on individual devices even with reduced computational resources.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 8: Illustrative image of the GoogLeNet architecture.

All convolutions in this architecture use rectified linear units (ReLUs) as activation functions. This network uses different types of interesting methods that allow it to create a deeper architecture.

- 1×1 convolution: These convolutions are used to decrease the number of parameters (weights and biases) of the architecture and increase the depth of the architecture;
- Global Average Pooling: this pooling is used at the end of the network and takes a feature map of 7×7 and an average of 1×1. In this way, the number of trainable parameters is reduced to 0 and accuracy is improved.
- Inception module: In this module the 1×1, 3×3, 5×5 convolution and 3×3 max-pooling are executed in parallel to the input and the output of these is stacked together to generate the final output. The idea behind this is that convolution filters of different sizes handle multiple-scale objects better.

The trained model of this network occupies MB and has 5,244,726 trainable parameters. For more details on GoogLeNet see the authors' paper [14].

4.2 Ifrit models

In this section, I discuss the CNN models I made to solve the forest fire detection problem treated for this project. To be precise, I developed, tested, and compared 4 main models. The name of each model consists of the prefix "IfritNet," which is the name given to my CNNs for this project, and the suffix "vx" with x as the version indicator. Each model with its own characteristics is explained below.

4.2.1 IfritNet v1

This first version has the simplest architecture of all those proposed. This architecture takes input images of size 224x224x3 with RGB color channels.

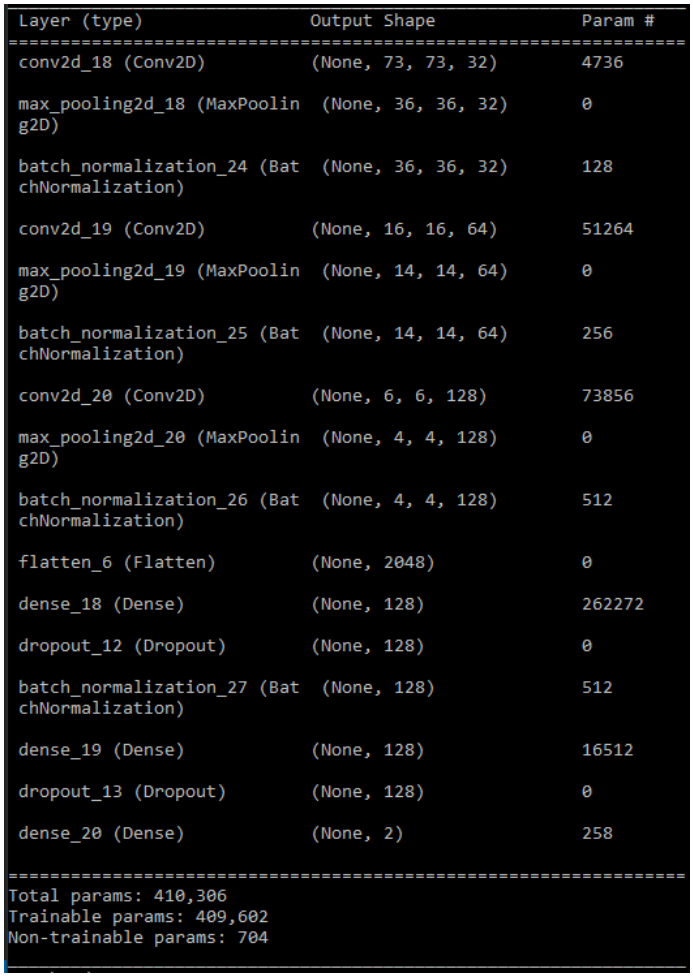
IfritNetv1 has six layers(as shown in the figure 9): the first three layers are convolutional, all followed by max pooling and batch normalization, while the last three levels are fully connected. Respectively, the filters applied to the convolutional layers are:

- the first with 32 kernels of size 7×7×3 and stride 3;
- the first with 64 kernels of size 5×5×3 and stride 2;
- the first with 128 kernels of size 3×3×3 and stride 2;

The first two fully connected layers have drop out of 0.3 and each layer consists of 128 neurons and the last one consists of 2 neurons (the number of classes). In all layers, the ReLU function is used as the activation function except for the last neuron where the softmax activation function is used.

The basic idea is to initially apply filters that take into account a large area(kernel size) and then as the depth of the network increases, increase the number of filters but decrease the kernel size. In this way, the network at first takes large portions (larger kernel size) and summarizes them (max pooling), and then goes deep by extracting more information (more filters) in smaller areas (focus on the most important image details and information left and emerged from pooling).

The trained model of this network occupies 4.79 MB and has 409,602 trainable parameters.



Layer (type)	Output Shape	Param #
conv2d_18 (Conv2D)	(None, 73, 73, 32)	4736
max_pooling2d_18 (MaxPooling2D)	(None, 36, 36, 32)	0
batch_normalization_24 (BatchNormalization)	(None, 36, 36, 32)	128
conv2d_19 (Conv2D)	(None, 16, 16, 64)	51264
max_pooling2d_19 (MaxPooling2D)	(None, 14, 14, 64)	0
batch_normalization_25 (BatchNormalization)	(None, 14, 14, 64)	256
conv2d_20 (Conv2D)	(None, 6, 6, 128)	73856
max_pooling2d_20 (MaxPooling2D)	(None, 4, 4, 128)	0
batch_normalization_26 (BatchNormalization)	(None, 4, 4, 128)	512
flatten_6 (Flatten)	(None, 2048)	0
dense_18 (Dense)	(None, 128)	262272
dropout_12 (Dropout)	(None, 128)	0
batch_normalization_27 (BatchNormalization)	(None, 128)	512
dense_19 (Dense)	(None, 128)	16512
dropout_13 (Dropout)	(None, 128)	0
dense_20 (Dense)	(None, 2)	258

Total params: 410,306
Trainable params: 409,602
Non-trainable params: 704

Figure 9: Illustrative image of the IfritNet v1 architecture.

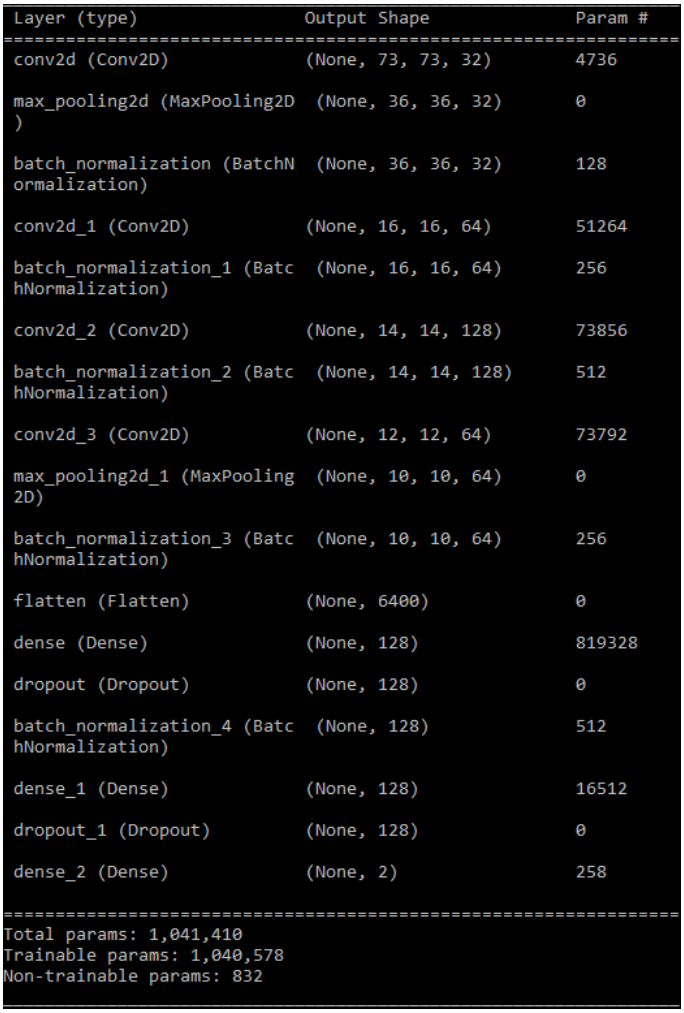
4.2.2 IfritNet v2

This version comes from wanting to merge the previous version (IfritNetv1) with the concepts exploited in AlexNet, more specifically having multiple convolutional layers in sequence without pooling in between.

This architecture takes input images of size 224x224x3 with RGB color channels.

IfritNetv2 has seven layers(as shown in the figure 10): the first four layers are convolutional, only after the first and fourth there is a max pooling, while the last three levels are fully connected. Respectively, the filters applied to the convolutional layers are:

- the first with 32 kernels of size 7×7×3 and stride 3;
- the first with 64 kernels of size 5×5×3 and stride 2;
- the first with 128 kernels of size 3×3×3 and stride 1;
- the first with 64 kernels of size 3×3×3 and stride 1;



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 73, 73, 32)	4736
max_pooling2d (MaxPooling2D)	(None, 36, 36, 32)	0
batch_normalization (BatchNormalization)	(None, 36, 36, 32)	128
conv2d_1 (Conv2D)	(None, 16, 16, 64)	51264
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_2 (Conv2D)	(None, 14, 14, 128)	73856
batch_normalization_2 (BatchNormalization)	(None, 14, 14, 128)	512
conv2d_3 (Conv2D)	(None, 12, 12, 64)	73792
max_pooling2d_1 (MaxPooling2D)	(None, 10, 10, 64)	0
batch_normalization_3 (BatchNormalization)	(None, 10, 10, 64)	256
flatten (Flatten)	(None, 6400)	0
dense (Dense)	(None, 128)	819328
dropout (Dropout)	(None, 128)	0
batch_normalization_4 (BatchNormalization)	(None, 128)	512
dense_1 (Dense)	(None, 128)	16512
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 2)	258

Total params: 1,041,410
Trainable params: 1,040,578
Non-trainable params: 832

Figure 10: Illustrative image of the IfritNet v2 architecture.

The first two fully connected layers have drop out of 0.3 and each layer consists of 128 neurons and the last one consists of 2 neurons (the number of classes). In all layers, the ReLU function is used as the activation function except for the last neuron where the softmax activation function is used.

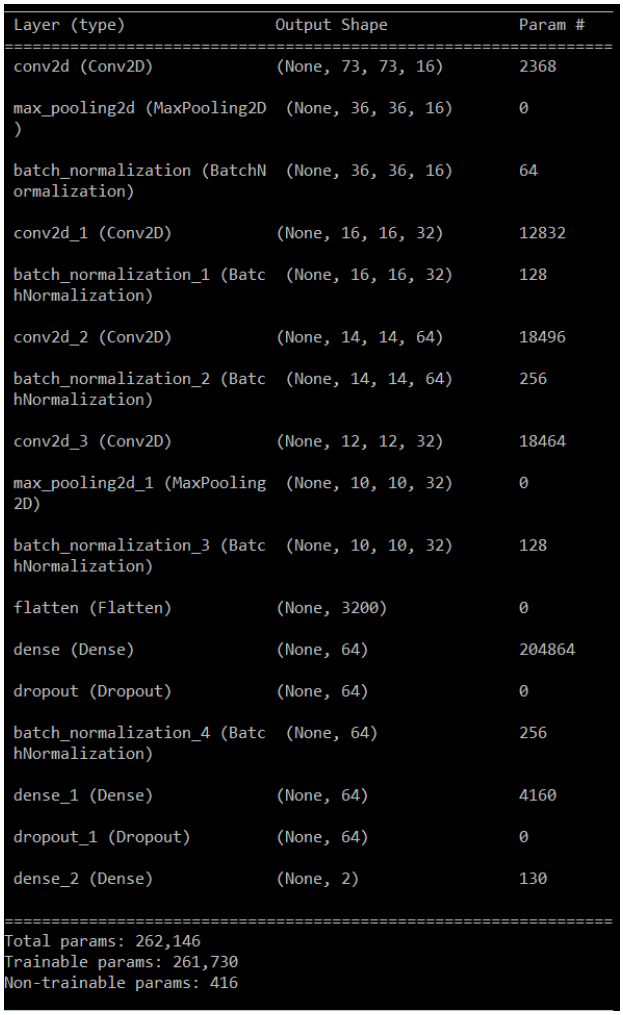
The basic idea is the same as the previous model except for a few aspects.

- Pooling was not employed after each convolutional layer but only after the first and the last;
- Compared with the previous model, one more convolutional layer was added and a wave pattern was given to the number of their filters. Indeed, in the last convolutional layer, the number of filters goes from 128 to 64 going down, to avoid having too many filters compared to the number of image features. This should help against the dispersion of features for network learning.

The trained model of this network occupies 12.02 MB and has 1,040,578 trainable parameters.

4.2.3 IfritNet v3

A net with fewer weights is easier and faster to train and takes up less space. For these reasons, I wanted to try to have a smaller, optimized net for this problem that could perform similarly to the previous ones but have a much smaller number of parameters.
This third model is inspired by a "lite" version of the second model. In this network we tried to go to reduce the number of training weights in order to have a network similar to the architecture proposed before but much lighter and faster.
I decreased the number of network weights by cutting the number of filters in the convolutional layers and decreasing the number of neurons in the fully connected layers.
This architecture takes input images of size 224x224x3 with RGB color channels.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 73, 73, 16)	2368
max_pooling2d (MaxPooling2D)	(None, 36, 36, 16)	0
batch_normalization (BatchNormalization)	(None, 36, 36, 16)	64
conv2d_1 (Conv2D)	(None, 16, 16, 32)	12832
batch_normalization_1 (BatchNormalization)	(None, 16, 16, 32)	128
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
batch_normalization_2 (BatchNormalization)	(None, 14, 14, 64)	256
conv2d_3 (Conv2D)	(None, 12, 12, 32)	18464
max_pooling2d_1 (MaxPooling2D)	(None, 10, 10, 32)	0
batch_normalization_3 (BatchNormalization)	(None, 10, 10, 32)	128
flatten (Flatten)	(None, 3200)	0
dense (Dense)	(None, 64)	204864
dropout (Dropout)	(None, 64)	0
batch_normalization_4 (BatchNormalization)	(None, 64)	256
dense_1 (Dense)	(None, 64)	4160
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 2)	130

Total params: 262,146
Trainable params: 261,730
Non-trainable params: 416

Figure 11: Illustrative image of the IfritNet v3 architecture.

IfritNetv3 has seven layers(as shown in the figure 11): the first four layers are convolutional,

only after the first and fourth there is a max pooling, while the last three levels are fully connected. Respectively, the filters applied to the convolutional layers are:

- the first with 16 kernels of size 7×7×3 and stride 3;
- the first with 32 kernels of size 5×5×3 and stride 2;
- the first with 64 kernels of size 3×3×3 and stride 1;
- the first with 32 kernels of size 3×3×3 and stride 1;

The first two fully connected layers have drop out of 0.3 and each layer consists of 64 neurons and the last one consists of 2 neurons (the number of classes). In all layers, the ReLU function is used as the activation function except for the last neuron where the softmax activation function is used.

The trained model of this network occupies 3.11 MB and has 261,730 trainable parameters. The trained model of IfritNet v3 occupies 75% less than IfritNet v2 and even more than 98% less than AlexNet. Regarding the number of trainable parameters, IfritNet v3 has about 99% fewer than AlexNet and about 75% fewer than IfritNet v2.

4.2.4 IfritNet v4

Also for this fourth model, I wanted to try to reduce the number of network parameters while trying to maintain performance. For the reduction, I took inspiration from GoogLeNet, specifically the 1x1 convolution and the inception module (look at section 4.1.2).

This architecture takes input images of size 224x224x3 with RGB color channels.

This network consists of a convolutional layer, with max pool, followed by four inception modules, a fully connected layer, with dropout, and the output layer (as shown in the figure 12). In particular, in the four inception modules, there is a max pool after the first and third, and at the end, there is global average pooling.

Remind that in the inception module 1×1, 3×3, 5×5 convolution and 3×3 max-pooling are executed in parallel to the input and the output of these is concatenated together to generate the final output. there is a 1x1 convolutional layer before the 3x3 and 5x5 convolutional layers and before the max pool.
Respectively, the filters applied to the inception module layers are:

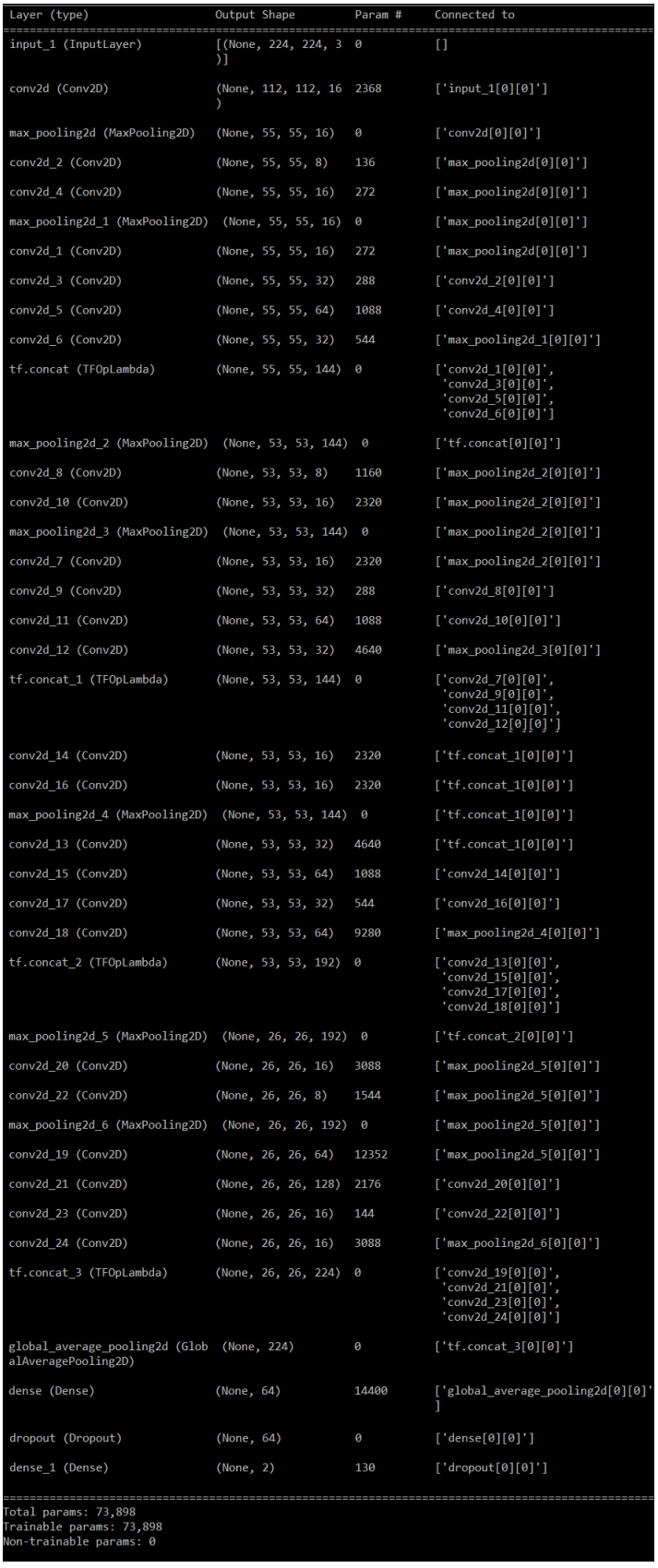
- 16 filters for 1x1, 8 filters for 1x1 convolutional layer before 3x3, 32 filters for 3x3, 16 filters for the 1x1 convolutional layer before 5x5, 64 filters for 5x5, and 32 filter for 1x1 convolutional layer after max pool;
- 16 filters for 1x1, 8 filters for 1x1 convolutional layer before 3x3, 32 filters for 3x3, 16 filters for the 1x1 convolutional layer before 5x5, 64 filters for 5x5, and 32 filter for 1x1 convolutional layer after max pool;
- 32 filters for 1x1, 16 filters for 1x1 convolutional layer before 3x3, 64 filters for 3x3, 16 filters for the 1x1 convolutional layer before 5x5, 32 filters for 5x5, and 64 filter for 1x1 convolutional layer after max pool;
- 64 filters for 1x1, 16 filters for 1x1 convolutional layer before 3x3, 128 filters for 3x3, 8 filters for the 1x1 convolutional layer before 5x5, 16 filters for 5x5, and 16 filter for 1x1 convolutional layer after max pool.

The idea behind inception modules is to be able to process images at different scales and then aggregate them so that the next layer can extract features from the different scales simultaneously.
A key contribution against uncontrolled parameter increase is the 1x1 convolutional layers before the other convolutional layers. This module also handles multiple-scale objects better.
For these reasons I wanted to adopt these modules in IfriNetv1, in this way I tried to go and improve feature extraction while at the same time, I went and cut the number of network weights.

I decided on the number of filters of the inception layers maintaining the same basic idea in which at first prioritize filters with larger kernel sizes and then as the depth of the network increases, prioritize filters with small kernel sizes (as can be seen in the description above).
For this reason, in the first inception modules, the 5x5 convolutional filter number will be predominant over the others, and the 1x1 convolutional filter will be the least present While in the later inception modules, it will be the opposite, more space and importance will be given to the 1x1 filter and especially the 3x3 while they will have less space for the 5x5 filter.

The first fully connected layer has a drop out of 0.3 and consists of 64 neurons and the second layer consists of 2 neurons (the number of classes). In all layers, the ReLU function is used as the activation function except for the last neuron where the softmax activation function is used.

The trained model of this network occupies 1.22 MB and has 73,898 trainable parameters.
The trained model of IfritNet v4 occupies 60% less than IfritNet v2 and even more than 99% less than AlexNet. Regarding the number of trainable parameters, IfritNet v4 has about 99.7% fewer than AlexNet and about 72% fewer than IfritNet v2.



Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 224, 224, 3)]	0	[]
conv2d (Conv2D)	(None, 112, 112, 16)	2368	['input_1[0][0]']
max_pooling2d (MaxPooling2D)	(None, 55, 55, 16)	0	['conv2d[0][0]']
conv2d_2 (Conv2D)	(None, 55, 55, 8)	136	['max_pooling2d[0][0]']
conv2d_4 (Conv2D)	(None, 55, 55, 16)	272	['max_pooling2d[0][0]']
max_pooling2d_1 (MaxPooling2D)	(None, 55, 55, 16)	0	['max_pooling2d[0][0]']
conv2d_1 (Conv2D)	(None, 55, 55, 16)	272	['max_pooling2d[0][0]']
conv2d_3 (Conv2D)	(None, 55, 55, 32)	288	['conv2d_2[0][0]']
conv2d_5 (Conv2D)	(None, 55, 55, 64)	1088	['conv2d_4[0][0]']
conv2d_6 (Conv2D)	(None, 55, 55, 32)	544	['max_pooling2d_1[0][0]']
tf.concat (TFOpLambda)	(None, 55, 55, 144)	0	['conv2d_1[0][0]', 'conv2d_3[0][0]', 'conv2d_5[0][0]', 'conv2d_6[0][0]']
max_pooling2d_2 (MaxPooling2D)	(None, 53, 53, 144)	0	['tf.concat[0][0]']
conv2d_8 (Conv2D)	(None, 53, 53, 8)	1160	['max_pooling2d_2[0][0]']
conv2d_10 (Conv2D)	(None, 53, 53, 16)	2320	['max_pooling2d_2[0][0]']
max_pooling2d_3 (MaxPooling2D)	(None, 53, 53, 144)	0	['max_pooling2d_2[0][0]']
conv2d_7 (Conv2D)	(None, 53, 53, 16)	2320	['max_pooling2d_2[0][0]']
conv2d_9 (Conv2D)	(None, 53, 53, 32)	288	['conv2d_8[0][0]']
conv2d_11 (Conv2D)	(None, 53, 53, 64)	1088	['conv2d_10[0][0]']
conv2d_12 (Conv2D)	(None, 53, 53, 32)	4640	['max_pooling2d_3[0][0]']
tf.concat_1 (TFOpLambda)	(None, 53, 53, 144)	0	['conv2d_7[0][0]', 'conv2d_9[0][0]', 'conv2d_11[0][0]', 'conv2d_12[0][0]']
conv2d_14 (Conv2D)	(None, 53, 53, 16)	2320	['tf.concat_1[0][0]']
conv2d_16 (Conv2D)	(None, 53, 53, 16)	2320	['tf.concat_1[0][0]']
max_pooling2d_4 (MaxPooling2D)	(None, 53, 53, 144)	0	['tf.concat_1[0][0]']
conv2d_13 (Conv2D)	(None, 53, 53, 32)	4640	['tf.concat_1[0][0]']
conv2d_15 (Conv2D)	(None, 53, 53, 64)	1088	['conv2d_14[0][0]']
conv2d_17 (Conv2D)	(None, 53, 53, 32)	544	['conv2d_16[0][0]']
conv2d_18 (Conv2D)	(None, 53, 53, 64)	9280	['max_pooling2d_4[0][0]']
tf.concat_2 (TFOpLambda)	(None, 53, 53, 192)	0	['conv2d_13[0][0]', 'conv2d_15[0][0]', 'conv2d_17[0][0]', 'conv2d_18[0][0]']
max_pooling2d_5 (MaxPooling2D)	(None, 26, 26, 192)	0	['tf.concat_2[0][0]']
conv2d_20 (Conv2D)	(None, 26, 26, 16)	3088	['max_pooling2d_5[0][0]']
conv2d_22 (Conv2D)	(None, 26, 26, 8)	1544	['max_pooling2d_5[0][0]']
max_pooling2d_6 (MaxPooling2D)	(None, 26, 26, 192)	0	['max_pooling2d_5[0][0]']
conv2d_19 (Conv2D)	(None, 26, 26, 64)	12352	['max_pooling2d_5[0][0]']
conv2d_21 (Conv2D)	(None, 26, 26, 128)	2176	['conv2d_20[0][0]']
conv2d_23 (Conv2D)	(None, 26, 26, 16)	144	['conv2d_22[0][0]']
conv2d_24 (Conv2D)	(None, 26, 26, 16)	3088	['max_pooling2d_6[0][0]']
tf.concat_3 (TFOpLambda)	(None, 26, 26, 224)	0	['conv2d_19[0][0]', 'conv2d_21[0][0]', 'conv2d_23[0][0]', 'conv2d_24[0][0]']
global_average_pooling2d (GlobalAveragePooling2D)	(None, 224)	0	['tf.concat_3[0][0]']
dense (Dense)	(None, 64)	14400	['global_average_pooling2d[0][0]']
dropout (Dropout)	(None, 64)	0	['dense[0][0]']
dense_1 (Dense)	(None, 2)	130	['dropout[0][0]']

Total params: 73,898
Trainable params: 73,898
Non-trainable params: 0

Figure 12: Illustrative image of the IfritNet v4 architecture.

5 Results and discussions

In this section, I present the training results of the networks examined for this project.
The basic training methodology was the same for all networks. As mentioned earlier in the dedicated paragraphs, I trained each network with different batch sizes and early stopping patience parameters (the observed metric was always the loss function). The batch size values tested are 32, 64, and 128 while the patience values tested are 10, 15, and 20. I performed 10 cross-validation for each pair of parameters tested and for each cross-validation done I saved the average accuracy and loss value and the average confusion matrix.
I performed the cross-validation to get an average of the performance of the networks using different data, and samples, in the training, validation, and test set. This is because training our model on different random splits can lead to different values of the model performance, using the average should avoid reporting values that statistically can change a lot from those reported and have a reference value that is also statistically valid.
The dataset is almost perfectly balanced so accuracy is sufficient as a metric of network performance.
In addition to accuracy, I also analyzed the confusion matrix related to trained models. This is because of the nature of our problem, a false alarm will have a strongly lower weight than an alarm not sent because the fire was not detected. This condition for a good network is that there are as few instances as possible of images belonging to the fire class but classified as belonging to the non-fire class.

In the appropriate sections for each model treated in this project, I will show the graphs of these results obtained to identify the best network and the best parameters for each of them.
In the graphs that show these results, the names in the horizontal axis identify the values of the parameters tested in the images. These names have the following format "bX_eY" where X indicates the batch size value while Y indicates the value for early stopping patience, which indicates after how many epochs without improvement in the loss function of the validation set the training is stopped.
The confusion matrices may report slight inconsistencies between the numbers and the percentages, this is because each confusion matrix is the result of averaging the k confusion matrices obtained during cross-validation. The numbers are slightly skewed because the average is rounded to give an integer, while the percentages are approximated to the second decimal place so they give a more truthful measure (they will be given more consideration).

Another general consideration, valid for all the analyses reported in the next sections, regards the difference in the trends of the curves of the metrics analyzed in the training and validation set. The trends of the validation set will be more or less of the same trend but more irregular, this is due to the fact that the metrics on those elements are calculated after each epoch of the training as for the elements of the training set but the updating of the weights is done only on the elements of the training set.
So the network updates are done to adapt and improve the performance as much as possible on the training set and the improvement of the validation set happens indirectly because even though the weights are not updated on their elements but on elements belonging to the same and similar classes. Thus the improvements will be greater and more constant on the training set (set used to adapt the network weights) and more irregular, less constant, on the validation set (set on which it is verified that the adaptations made on the network actually lead to a general improvement in performance and not only specific to the elements of the training set, overtraining).

5.1 AlexNet

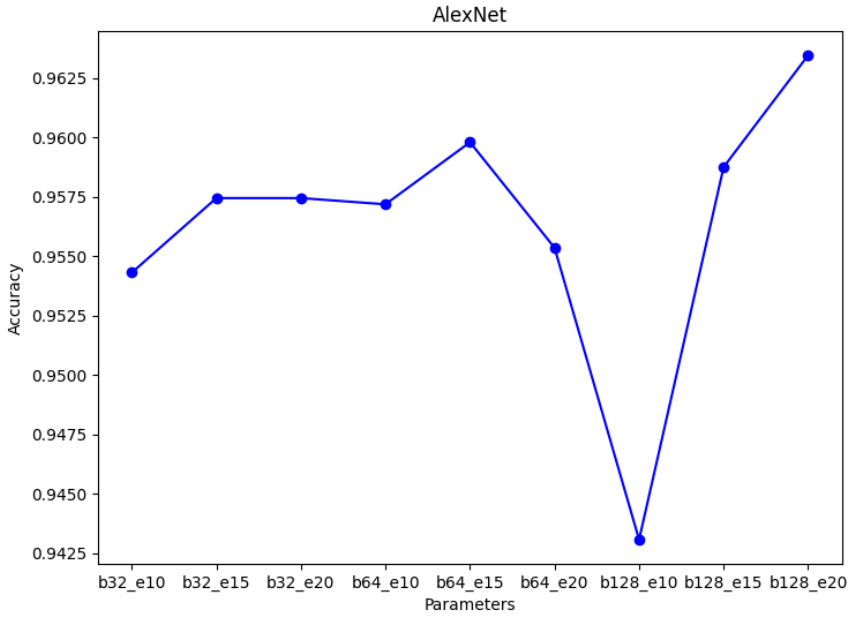


Figure 13: Illustrative image of accuracy results obtained with AlexNet.

In the figure 13 average accuracy for each parameter setting tested are shown.

The performance of the network with the different settings is very good, ranging from 94% to 96.5% in accuracy. In particular, we can observe that the setting that performed best is a batch size of 128 and patience of 20 which achieved 96.5% in accuracy. While the worst result was with a batch size of 128 and patience of 10 which achieved just over 94% in accuracy. The gap between the best setting and the second best, batch size 64 and patience of 20, is 0.5%, and the performances of 7 out of 9 settings are between 95.4% and 96%, about 0.6%. It can be seen that all the tested settings produce results very close to each other in terms of accuracy.

In figure 14 average values of the loss function for each parameter setting tested are shown. The values shown are in agreement with the accuracy values obtained, the graph for loss function values is almost the specular of the graph of accuracy values.
They are not perfectly specular of each other because accuracy is not the inverse of loss, however, there is an inverse relationship between them. The loss function values shown represent the average of the error for each individual sample, that is the average between the difference of the value given in output from the network and the value of the label associated with the given sample (ground truth).
The accuracy will depend on the classification made by the network, then on a threshold of the class probability predicted by the model. Thus accuracy will be strictly dependent on the distribution of loss values, not their mean. Therefore network models with the same value for the loss function may have different accuracy values, and vice versa network models with the

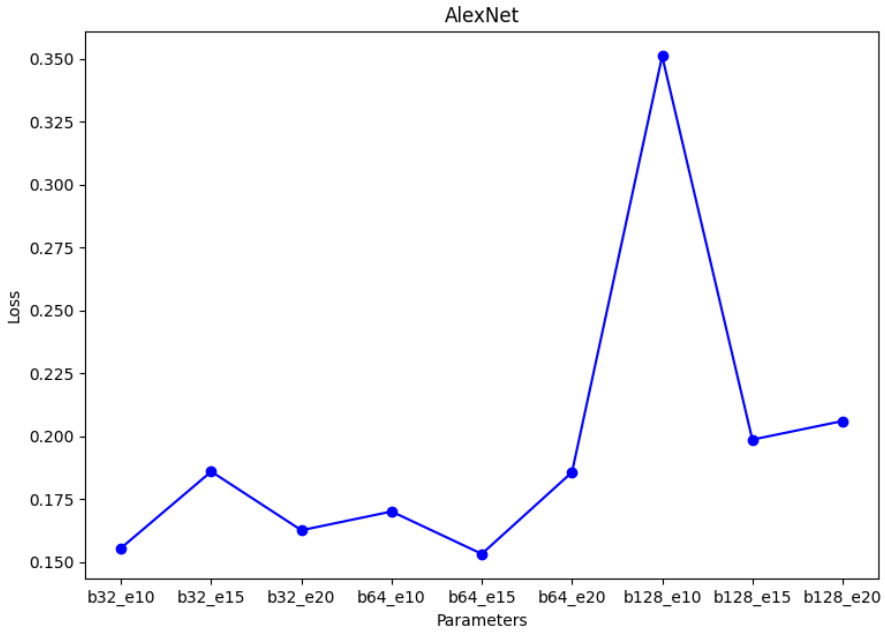


Figure 14: Illustrative image of loss function results obtained with AlexNet.

same accuracy may have different loss values.
But in general, there is an inverse relationship between loss and accuracy because the lower the mean of the loss values the more it will mean that in general the network is better trained (has a smaller error between predicted value and label) and this will also lead to better accuracy.

Now I analyze the confusion matrix related to the two settings with better accuracy, fig 15. For brevity and clarity from now on when we talk about the confusion matrix we will consider the fire class as positive and the non-fire class as negative. As mentioned earlier in our problem a false positive (a fire alarm when no fire is present) is much less important than a false negative (a case where there is fire but no alarm is given).

The confusion matrix confirms that the selection of classes in the test set mirrors the distribution in the full dataset, which is almost perfectly balanced. Confusion matrixes of both the first and second best models are shown because given the small difference in accuracy between them (0.5%) if the second model results with lower false negatives than the other it might be convenient to evaluate it as better, given the importance of having low false negatives. Now for clarity, I will denote as the first model the one with batch size settings equal to 128 and patience 20 and denote as the second model the one with batch size settings of 64 and patience of 15. The confusion matrix of the first model is slightly better in all 4 fields (TP, FP, TN, FN) than that of the second model, this confirms that the first model is the best, both in accuracy and for false negatives. The first model has an average false negative percentage of 1.67%, a low value therefore good. The false positive is just under 2%, this shows a slight model bias but it is really minimal and can be ignored.

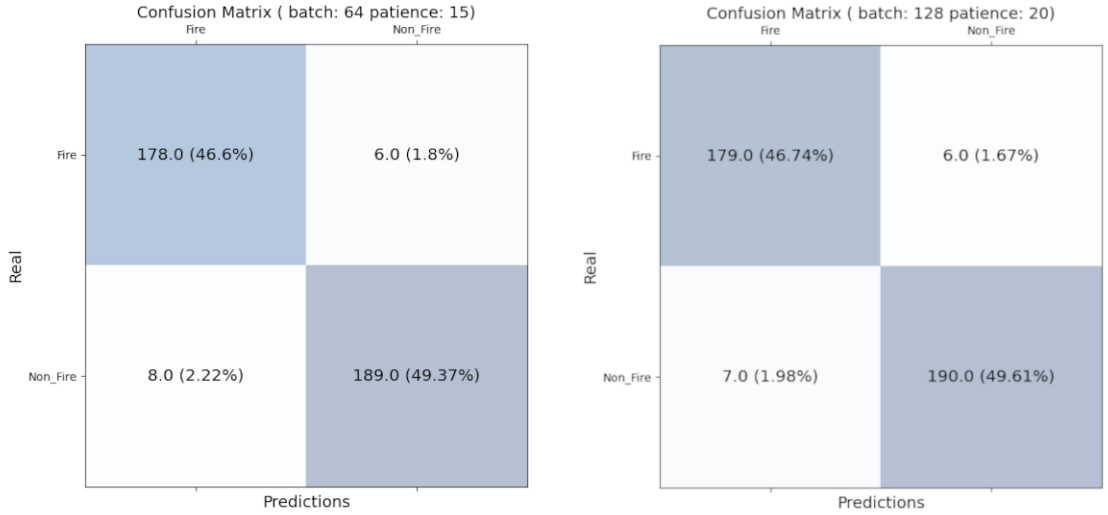


Figure 15: Illustrative image of the confusions matrix obtained with AlexNet with settings batch size 128 and patience 20 and with batch size 64 and patience 15.

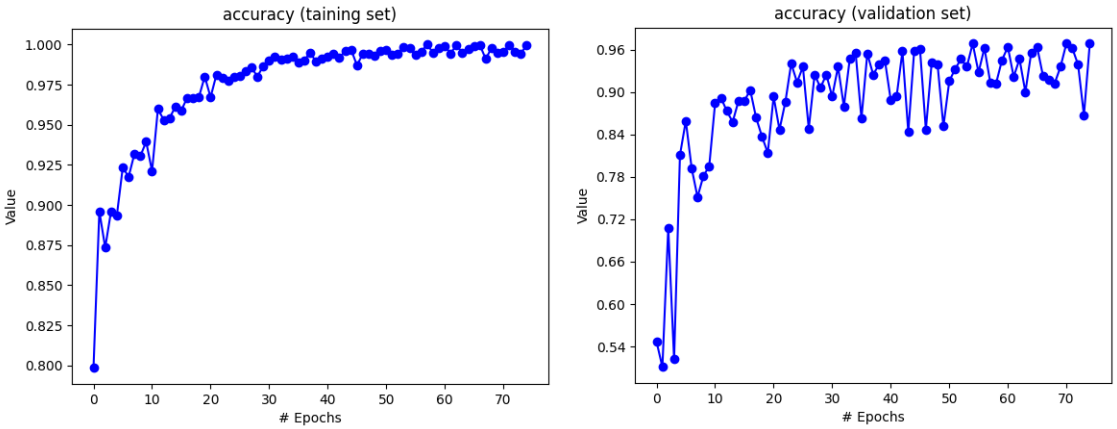


Figure 16: Illustrative image of accuracy results obtained with the best AlexNet settings.

Now I train the model, without cross-validation, using the best parameters and showing the results obtained.
In the dataset, the provided test set consists of only 50 images, 25 per class. Given the small number of the provided test set, I used a random selection from the training set (the 20%) as the test set.
The accuracy of the training and validation set during network training are shown in the figure 16.

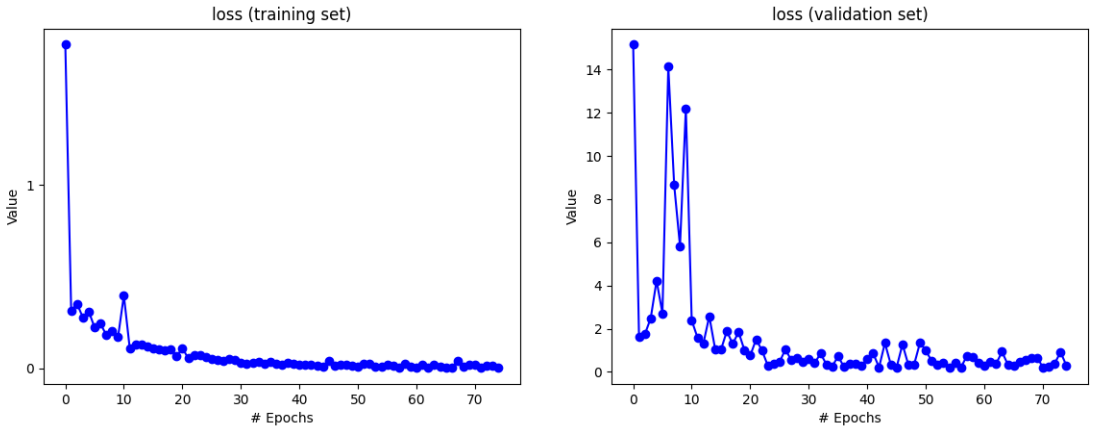


Figure 17: Illustrative image of loss function results obtained with the best AlexNet settings.

The values of the loss function of the training and validation set during network training are shown in figure 17.

Network training ran for 75 epochs before being interrupted by early stopping, so at the 55th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 30th epoch, then the descent slows down until it remains almost constant, about from the 55th epoch. Looking at the values for the validation set I notice that the curve is much more irregular but has much the same trend as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around epoch 30, and then tends to get slightly worse from the 55th epoch onward.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 30th epoch, then the increase slows down until it remains almost constant, about from the 55th epoch. Looking at the values for the validation set I notice that the curve is much more irregular but has much the same pattern as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around the 25th epoch, and then tends to get slightly worse from the 55th epoch onward.

The values shown are consistent with the training that ran for 75 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 55) we were able to have the model with better performance, that is, before the beginning of overtraining.

The results on the test set of this final model achieved 94.99% accuracy and 0.2992 for the loss function and 2.22% for false negatives (as shown in the figure 18). These are slightly lower performances than those obtained with cross-validation but not discordant.

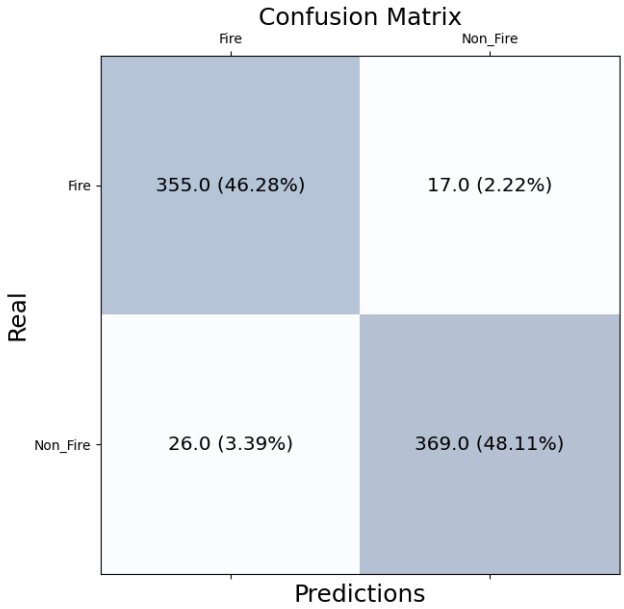


Figure 18: Illustrative image of the confusions matrix obtained with final AlexNet model with settings batch size 128 and patience 20.

This trained model is saved in the "Model/best_model" folder of the project, with the name "AlexNet" (In the GitHub folder it is not present because it is too big, it exceeds 100 MB).

5.2 GoogLeNet

In figure 19 average accuracy for each parameter setting tested is shown.
Values for both the main output and auxiliary outputs are shown in this plot. I have shown all outputs for completeness since the GoogLeNet architecture provides 3 outputs of which one main output, called out, and two auxiliary outputs, called aux1 and aux2 (the only architecture of this type among those analyzed in the project). The auxiliary outputs will not be considered for the final performance of the network because their use is only during training, in fact, they help against the gradient vanishing phenomenon, but they are not needed in the classification phase after the network is trained.
The vanishing of the gradient is a problem that occurs with certain types of activation functions (such as the Sigmoid function that has a very small derivative and for large portions very close to 0) that does not allow for proper training of the network. In a few words, you may find that the derivatives for each layer are very small, and with backpropagation and chain rule to find the gradient you would go and multiply the derivatives of each layer (which as mentioned before are close to 0) and this leads to an exponential decrease in the gradient as it propagates through the network. A very small gradient leads to an update of the biases and weights of the initial layers that are not sufficient for the network to function properly.
The performance of the network with the different settings is very good, ranging from 96.74% to 97.55% in accuracy. In particular, we can observe that the setting that performed best is a batch

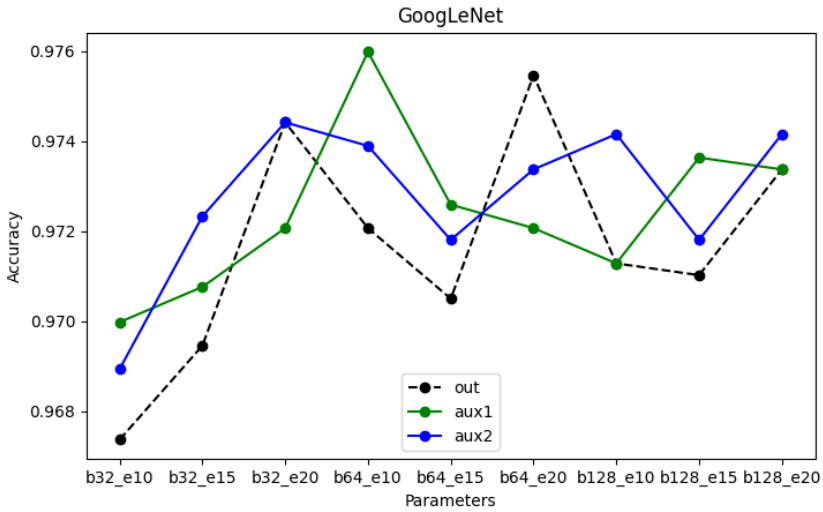


Figure 19: Illustrative image of accuracy results obtained with GoogLeNet.

size of 64 and patience of 20 which achieved 97.55% in accuracy. While the worst result was with a batch size of 32 and patience of 10 which achieved just over 96.74% in accuracy. The gap between the best setting and the second best, batch size 32 and patience of 20, is around 0.11%, and the performances of 7 out of 9 settings are between 96.95% and 97.44%, about 0.5%. It can be seen that most of the tested settings produce results very close to each other in terms of accuracy.

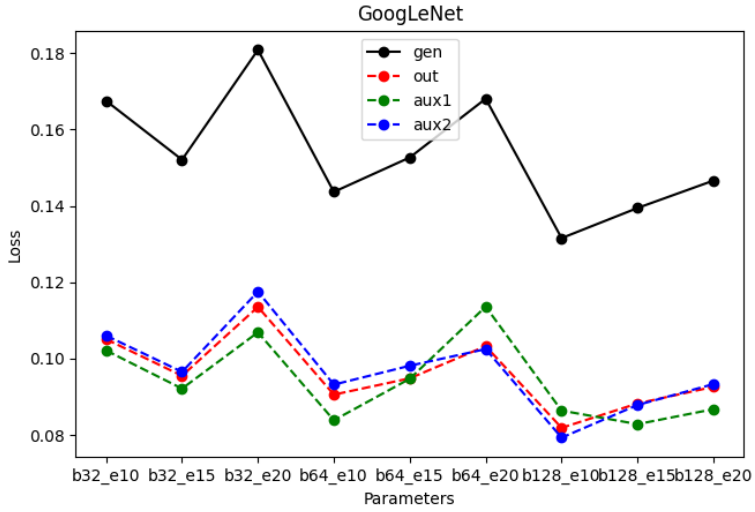


Figure 20: Illustrative image of loss function results obtained with GoogLeNet.

In figure 20 average values of the loss function for each parameter setting tested are shown.

The values shown are in agreement with the accuracy values obtained, the graph for loss function values is almost the specular of the graph of accuracy values.
Four curves of loss function values are represented in the plot. One for each output, a main one called out and two auxiliary ones called aux1 and aux2, and a general curve resulting from the contribution of each output. As mentioned earlier these three outputs have different weights, 1 for the main one and 0.3 for the auxiliary ones, which serve only in the training phase. The general loss value (the one in black in the figure) is the weighted sum, made with the weights said above, of the loss values for each output. This is also evident from the fact that the general loss graph has the same trends as the loss graphs of the various outputs. From the graph, I noticed that the loss values of the various outputs are really very similar to each other and almost always have exactly the same trends.

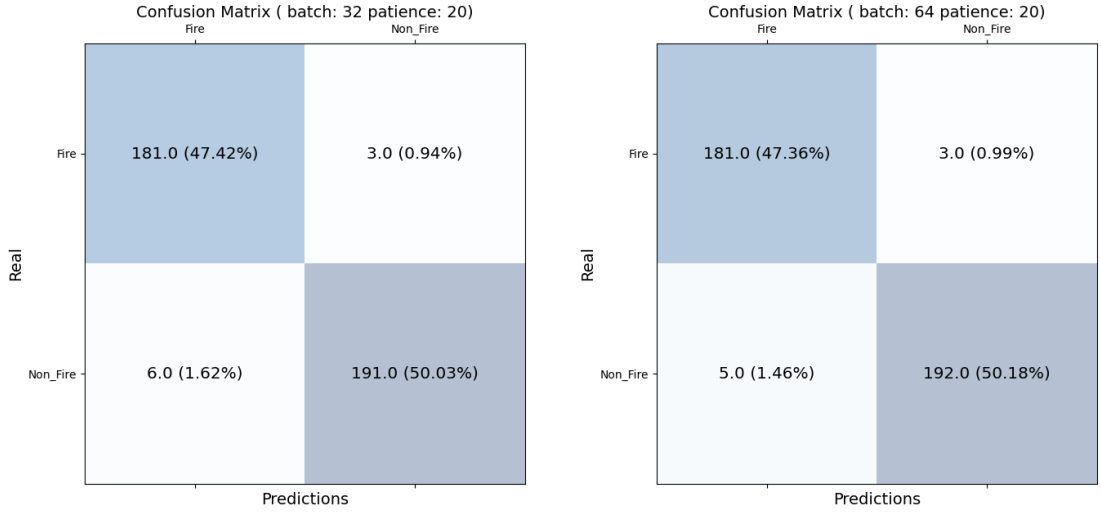


Figure 21: Illustrative image of the confusions matrix obtained with GoogLeNet with settings batch size 32 and patience 20 and with batch size 64 and patience 20.

Now I analyze the confusion matrix related to the two settings with better accuracy, fig 21. Now for clarity, I will denote as the first model the one with a batch size equal to 64 and patience of 20 and denote as the second model the one with a batch size equal to 32 and patience of 20. Confusion matrices of both the first and second best models are shown because given the small difference in accuracy between them (0.11%) if the second model results with lower false negatives than the other it might be convenient to evaluate it as better, given the importance of having low false negatives.

The confusion matrix of the first model is slightly better on 2 fields over 4 (FP, TN) than the second model. The second model is better for TP and FN that are which are the most important parameters.
The differences between the models are really minimal, 0.05% for false negative, 0.06% for true positive, 0.2% for false positive, and 0.15% for true negative.
Given the really small difference between the two models, one cannot accurately identify which one is better than the other. The first model could always be considered better than the second

given the higher accuracy and the difference in false negatives of only 0.02%, which can be considered insignificant. If one wants to prioritize the false negative independently of the real amount of improvement then the second model will be the best at the cost of a very small loss in accuracy.

Now I train the model, without cross-validation, using the two best parameters and showing the results obtained to analyze them and help to determine which of the two settings is better.

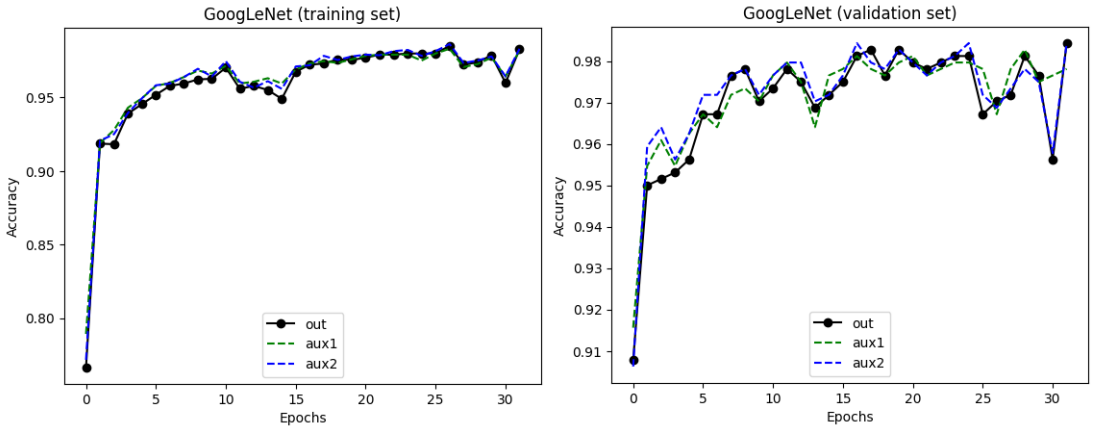


Figure 22: Illustrative image of accuracy results obtained with the GoogLeNet settings batch size 64 and patience 20.

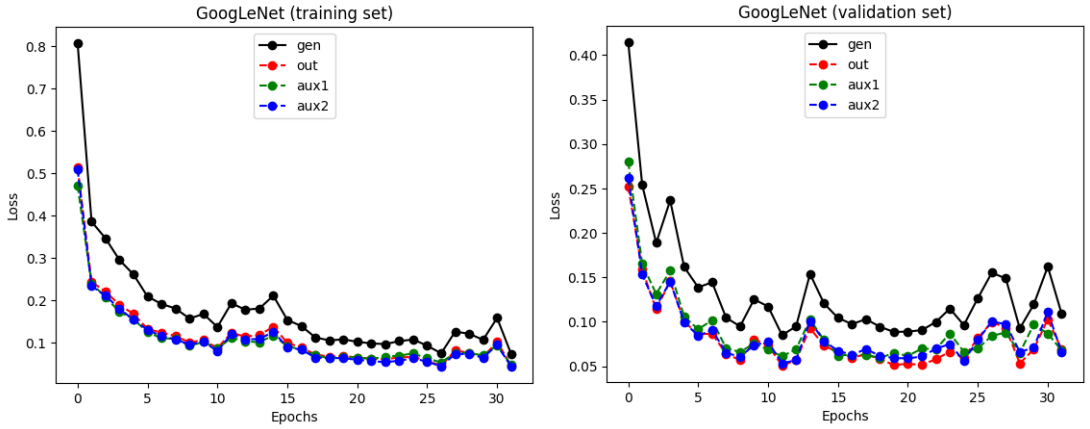


Figure 23: Illustrative image of loss function results obtained with the GoogLeNet settings batch size 64 and patience 20.

In the dataset, the provided test set consists of only 50 images, 25 per class. Given the small number of the provided test set, I used a random selection from the training set (the 20%) as the test set.
The accuracies of the training and validation set during network training for the first model are shown in figure 22.

The values of the loss function of the training and validation set during network training for the first model are shown in figure 23.

Network training ran for 32 epochs before being interrupted by early stopping, so at the 12 epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
All four loss function curves are present in the plot: the general one, the one of the main output, and the two of the auxiliary outputs. It is clear from the plot that all the curves follow the exact same trends. Early stopping monitors the values of the general loss function so from now on when I talk about the behavior of the loss function I will refer to the general curve.

Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 5th epoch, then the descent slows down until the 10th epoch and then has a small upward peak and then has a very low, almost stationary, downward trend until the end.
Looking at the values for the validation set I notice that the curve is more irregular with a downward trend until the 10th epoch and then an upward peak and an upward trend after the 20th epoch.
I noticed that the difference from the training set is that in the validation set the curve is much more irregular and tends to rise in the final epochs (a mark of overtraining). There are also more peaks, three in the middle part and two in the end part.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 5th epoch, then the increase slows down until it remains almost constant, about from the 15th epoch, until the 26th epoch where performance goes down and then goes up again, a small valley. There is also a small valley between the 10th and 15th epochs where performance goes down.
Looking at the values for the validation set I notice that the curve is much more irregular but has much the same pattern as the values for the training set. I also notice that the values of the three outputs are always similar and close but differ much more than at the training set, especially in the early stages of training where the auxiliary outputs have higher accuracy than the main output.

The values shown are consistent with the training that ran for 32 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 12) we were able to have the model with better performance, that is, before the beginning of overtraining.

The accuracies of the training and validation set during network training for the second model are shown in figure 24.
The values of the loss function of the training and validation set during network training for the second model are shown in figure 25.

Network training ran for 38 epochs before being interrupted by early stopping, so at the 18th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.

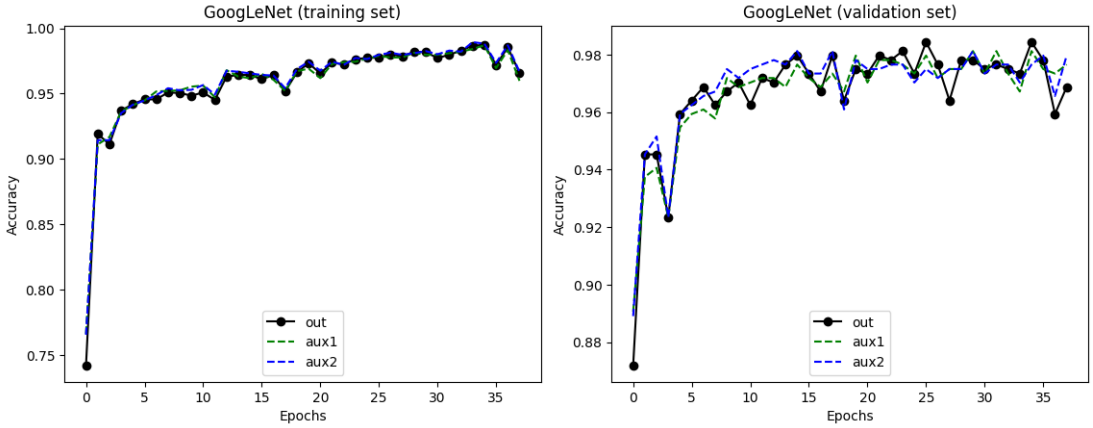


Figure 24: Illustrative image of accuracy results obtained with the GoogLeNet settings batch size 32 and patience 20.

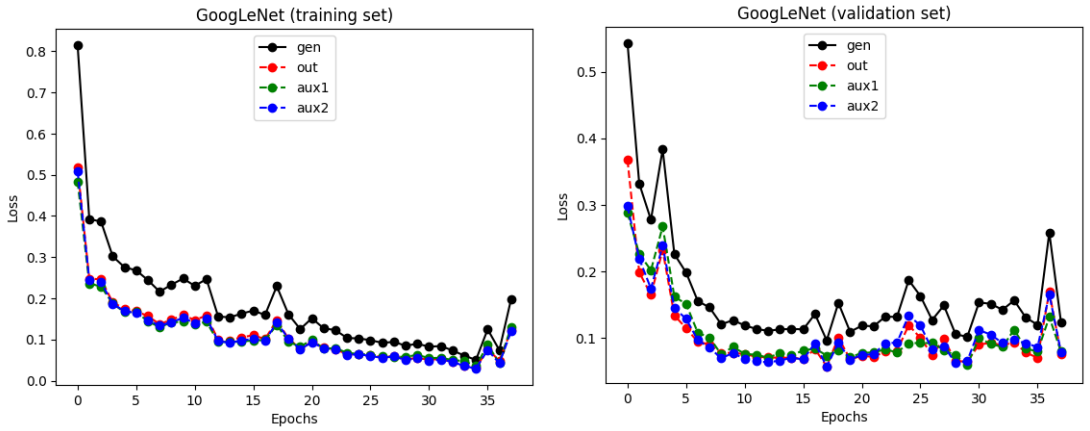


Figure 25: Illustrative image of loss function results obtained with the GoogLeNet settings batch size 32 and patience 20.

Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 5th epoch, then the descent slows down until the 7th epoch and then has two small upward peaks and then has a very low, almost stationary, downward trend until the end when there is an upward trend.
Looking at the values for the validation set I notice that the curve is more irregular with a downward trend until the 10th epoch and then an upward trend with 3 major peaks.
However, the curves appear to be quite smooth, especially in the first phase, in their trend except for the peaks. The tendency to rise from the middle of the training set, not present in the training set, is a marker of overtraining.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 5th epoch, then the increase slows down until it remains almost constant, about from

the 30th epoch, until the 32nd epoch where performance goes down. Looking at the values for the validation set I notice that the curve is much more irregular and has a slightly different trend. It reaches its peak first, towards the middle of the training, and then declines from the 25th epoch to the end. I also notice that the values of the three outputs are always similar and close but differ much more than at the training set, especially in the middle stages of training.

The values shown are consistent with the training that ran for 38 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly and except for the last few epochs) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 18) we were able to have the model with better performance, that is, before the beginning of overtraining.

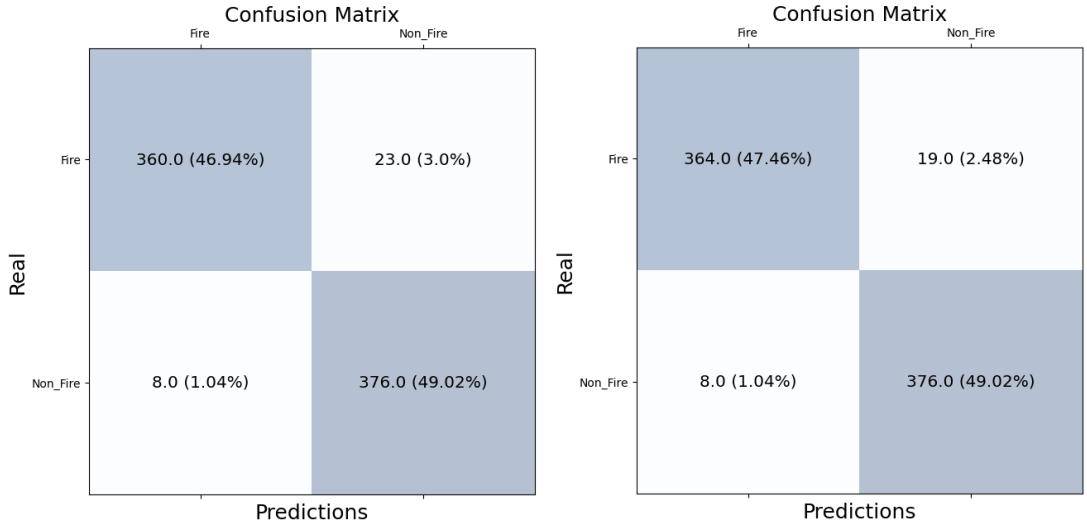


Figure 26: Illustrative image of the confusions matrix obtained with bests GoogLeNet model with settings batch size 32 and patience 20(left) and batch size 64 and patience 20(right).

With the help of the confusion matrices of the two models examined, fig 26, I make a final report to determine the best one:

- batch size 64 and patience 20 (first model): achieved an accuracy of 96.47%, a loss value of 0.1746, and false negatives of 2.48%;
- batch size 32 and patience 20 (second model): achieved an accuracy of 95.95%, a loss value of 0.1860, and false negatives of 3.0%.

The first model has a better accuracy, 0.52% difference, false negative, 0.52% difference, and true positive. The false negative and true negative are the same in the two models. I consider the first model better due to the better false negative percentage and accuracy of the second one.

The first trained model is saved in the "Model/best_model" folder of the project, with the name "GoogLeNet_b64_e20.hdf5"

The second trained model is saved in the "Model/best_model" folder of the project, with the name "GoogLeNet_b32_e20.hdf5"

5.3 IfritNetv1

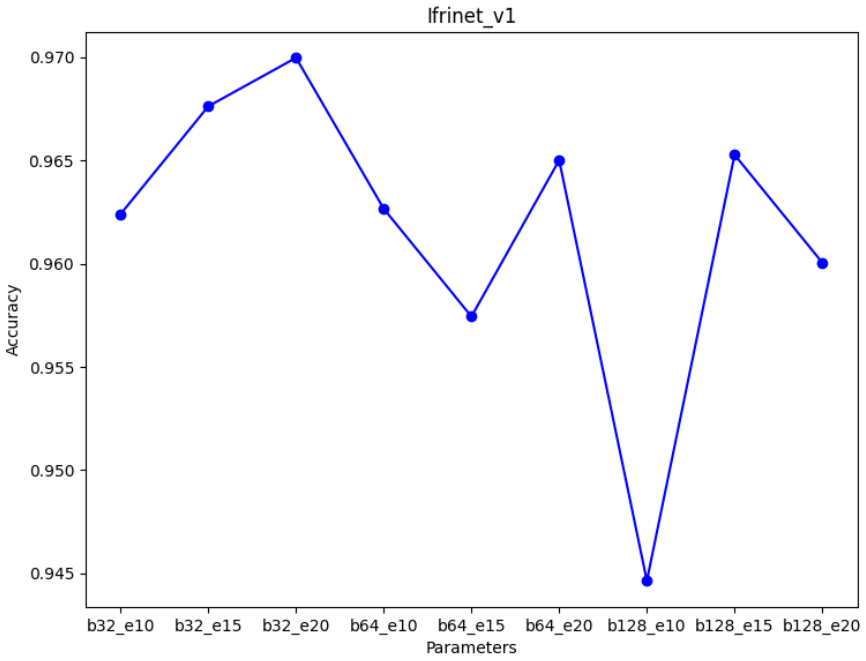


Figure 27: Illustrative image of accuracy results obtained with IfritNetv1.

In figure 27 average accuracy for each parameter setting tested are shown.
The performance of the network with the different settings is very good, ranging from 94.5% to 97% in accuracy. In particular, we can observe that the setting that performed best is a batch size of 32 and patience of 20 which achieved 97% in accuracy. While the worst result was with a batch size of 128 and patience of 10 which achieved just over 94.5% in accuracy. The gap between the best setting and the second best, batch size 32 and patience of 15, is around 0.25%, and the performances of 8 out of 9 settings are between 96% and 97%, about a 1%. It can be seen that all the tested settings, except one, produce results very close to each other in terms of accuracy.

In figure 28 average values of the loss function for each parameter setting tested are shown. The values shown are in agreement with the accuracy values obtained, the graph for loss function values is almost the specular of the graph of accuracy values.

Now I analyze the confusion matrix related to the two settings with better accuracy, fig 29.

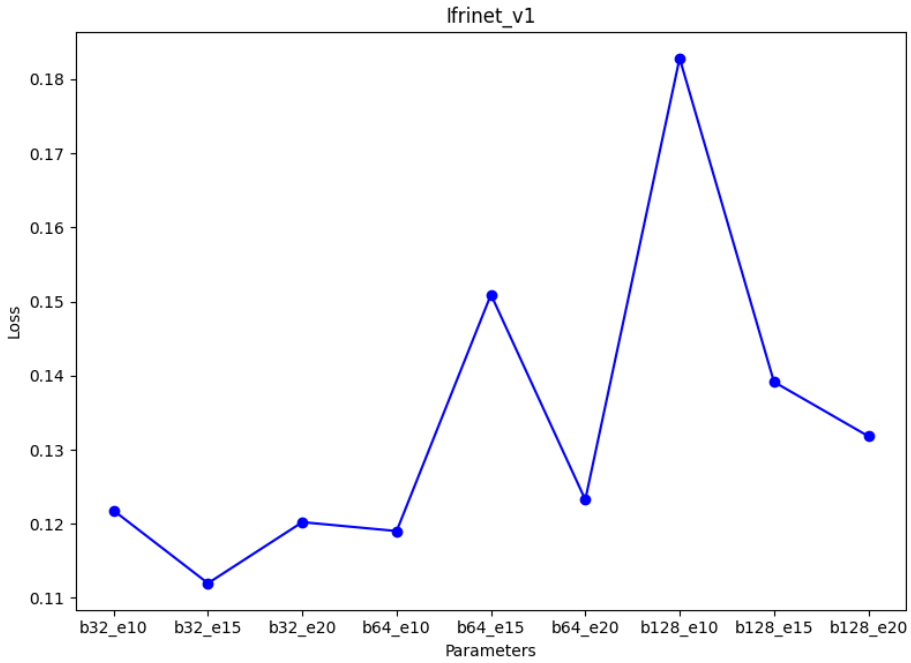


Figure 28: Illustrative image of loss function results obtained with IfritNetv1.

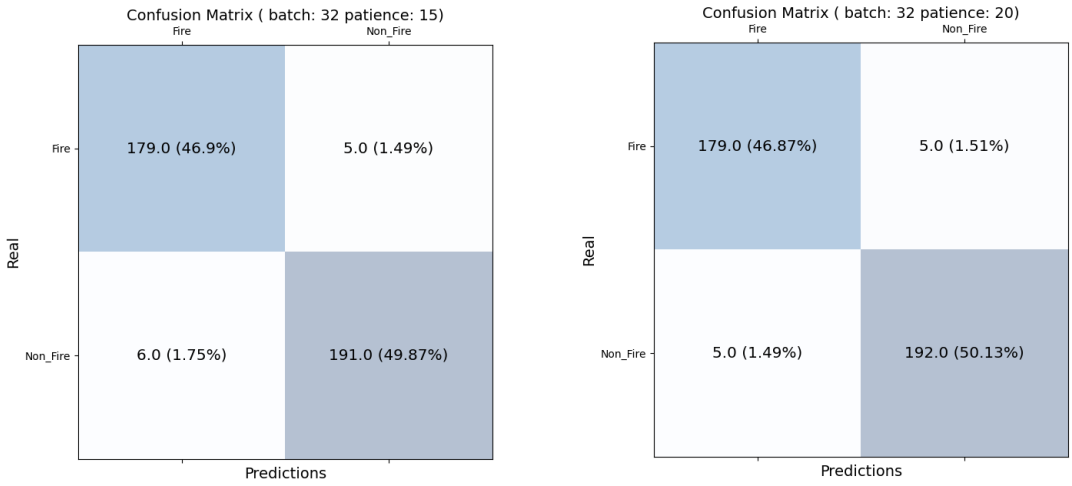


Figure 29: Illustrative image of the confusions matrix obtained with IfritNetv1 with settings batch size 32 and patience 15 and with batch size 32 and patience 20.

The confusion matrix confirms that the selection of classes in the test set mirrors the distri-

bution in the full dataset, which is balanced but not perfectly balanced.
Now for clarity i will denote as the first model the one with batch size equal to 32 and patience 20 and denote as the second model the one with batch size equal 32 and patience of 15. Confusion matrices of both the first and second best models are shown because given the small difference in accuracy between them (0.25%) if the second model results with lower false negatives than the other it might be convenient to evaluate it as better, given the importance of having low false negatives.
The confusion matrix of the first model is slightly better on 2 fields over 4 (FP, TN) than the second model. The second model is better for TP and FN that are the which are the most important parameters.
The differences between the models are really minimal, 0.02% for false negative and 0.03% for true positive. Given the really small difference between the two models, one cannot accurately identify which one is better than the other. The first model could always be considered better than the second given the higher accuracy and the difference on false negatives of only 0.02%, which can be considered insignificant. If one wants to prioritize the false positive independently of the real amount of improvement then the second model will be the best at the cost of a small loss in accuracy.

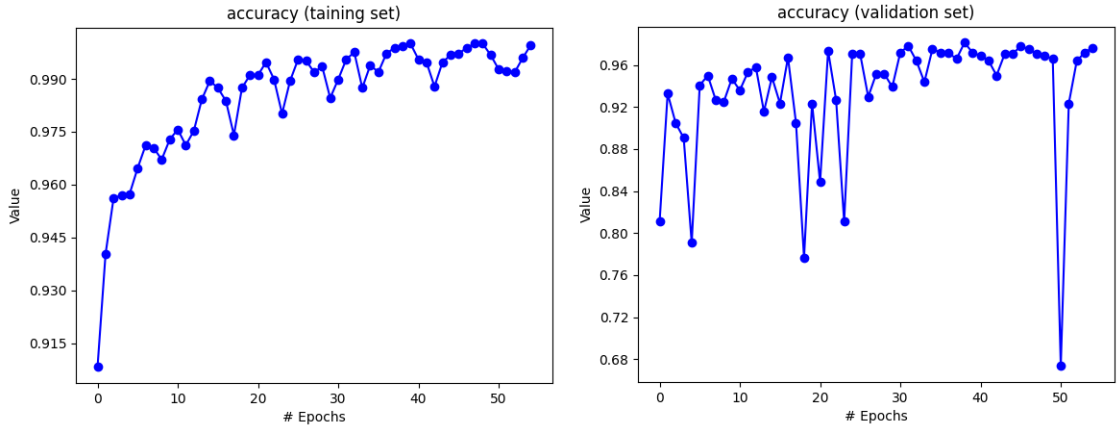


Figure 30: Illustrative image of accuracy results obtained with the IfritNetv1 settings batch size 32 and patience 20.

Now I train the model, without cross validation, using the two best parameters and showing the results obtained to analyze them and help to determine which of the two settings is better.
In the dataset, the provided test set consists of only 50 images, 25 per class. Given the small number of the provided test set, I used a random selection from the training set (the 20%) as the test set. The accuracy of the training and validation set during network training for the first model are shown in the figure 30.
The values of the loss function of the training and validation set during network training for the first model are shown in the figure 31.
Network training ran for 55 epochs before being interrupted by early stopping, so at the 35th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced

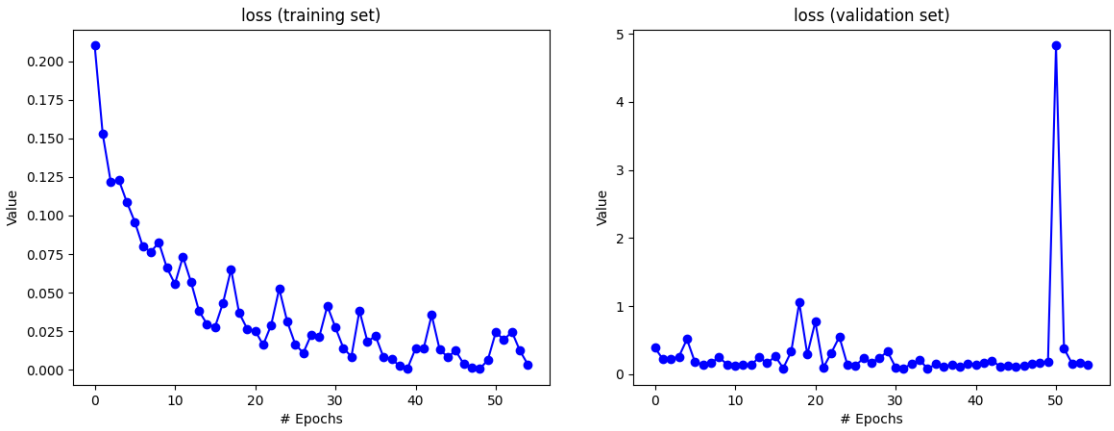


Figure 31: Illustrative image of loss function results obtained with the IfritNetv1 settings batch size 32 and patience 20.

descent until the 20th epoch, then the descent slows down until it remains almost constant, about from the 40th epoch.
Looking at the values for the validation set I notice that the curve is very linear with a slight downward trand until the 35th epoch and then having a slight upward trand. The trands in the curve are very minimal and difficult to notice. I notice as differences from the training set that it tends to become more stationary from earlier, at the beginning, and then tends to get slightly worse from the 35th epoch onward.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 20th epoch, then the increase slows down until it remains almost constant, about from the 35th epoch. Looking at the values for the validation set I notice that the curve is much more irregular and linear but has much the same pattern as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around the 15th epoch, and then tends to get slightly worse from the 35th epoch onward.

The values shown are consistent with training that ran for 55 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 35) we were able to have the model with better performance, that is, before the beginning of overtraining.

The accuracy of the training and validation set during network training for the second model are shown in the figure 32.
The values of the loss function of the training and validation set during network training for the second model are shown in the figure 33.
Network training ran for 35 epochs before being interrupted by early stopping, so at the 20th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.

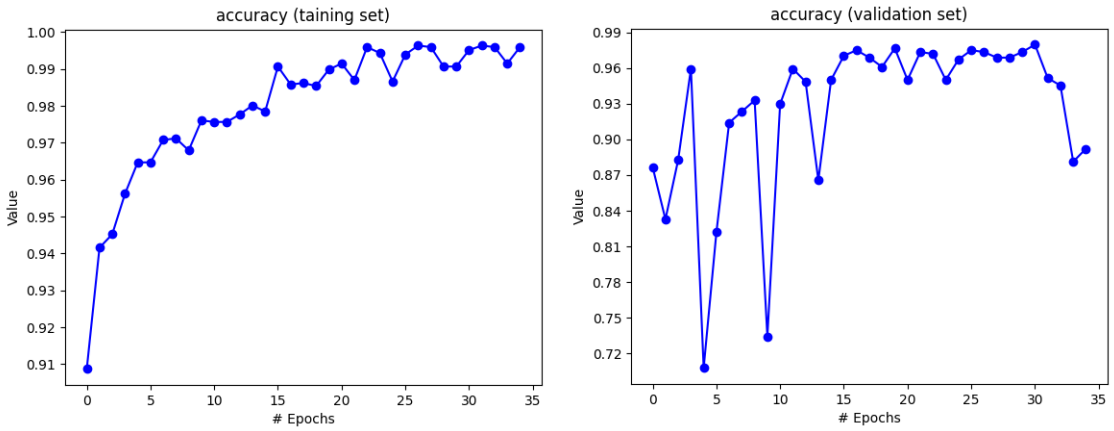


Figure 32: Illustrative image of accuracy results obtained with the IfritNetv1 settings batch size 32 and patience 15.

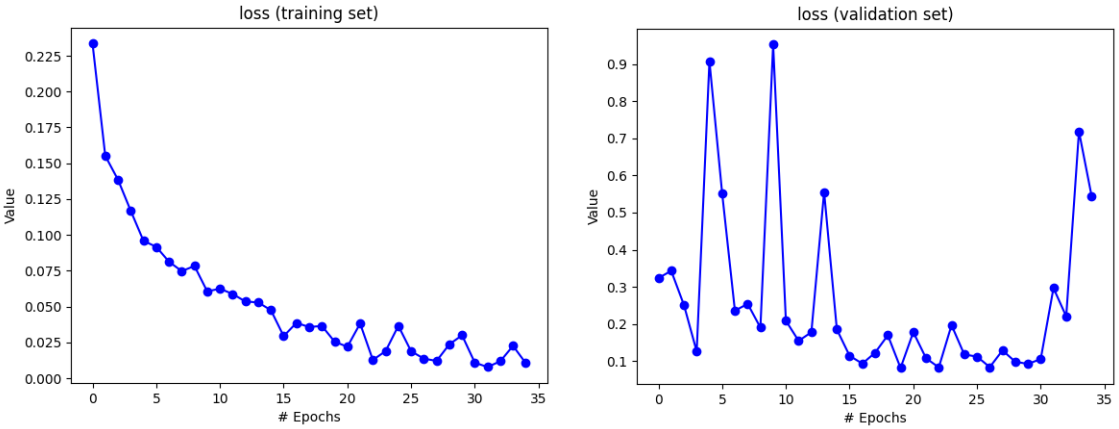


Figure 33: Illustrative image of loss function results obtained with the IfritNetv1 settings batch size 32 and patience 15.

Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 10th epoch, then the descent slows down until it remains almost constant, about from the 25th epoch.
Looking at the values for the validation set I notice that the curve is morelinear with strong irregularities (peaks) with a slight downward trand until the 15th epoch and then having a slight upward trand, except for the end where the trand is pronounced. I notice as differences from the training set that it tends to become more linear (except for the irregularities) from earlier, at the beginning, and then tends to get worse from the 30th epoch onward.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 10th epoch, then the increase slows down until it remains almost constant, about from the 15th epoch. Looking at the values for the validation set I notice that the curve is much more

irregular and linear but has much the same pattern as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around the 10th epoch, and then tends to get slightly worse from the 25th epoch onward.

The values shown are consistent with training that ran for 35 epochs using early stopping with patience 15 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 20) we were able to have the model with better performance, that is, before the beginning of overtraining.

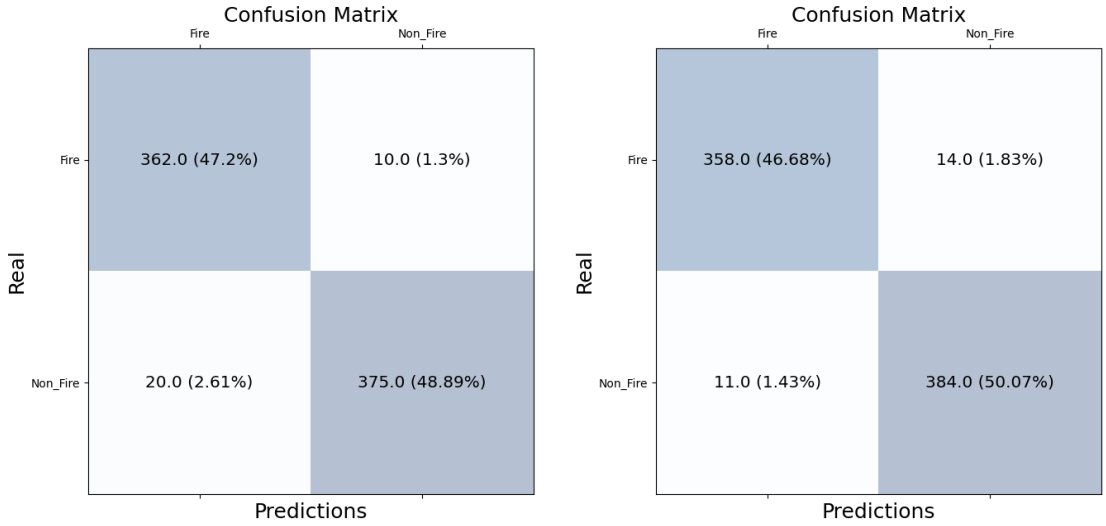


Figure 34: Illustrative image of the confusions matrix obtained with bests IfritNetv1 model with settings batch size 32 and patience 20(left) and batch size 32 and patience 15(right).

With the help of the confusion matrices of the two models examined, fig 34, I make a final report to determine the best one:

- batch size 32 and patience 20 (first model) : achieved an accuracy of 96.09, a loss value of 0.1684, and false negatives of 1.3%;
- batch size 32 and patience 15 (second model) : achieved an accuracy of 96.74, a loss value of 0.1655, and false negatives of 1.83%.

The first model has a worse accuracy, 0.65% difference and a slightly worse loss however it has a better false negative, 0.53% difference. I consider the first model better due to the better false negative percentage and a similar accuracy to the second one.

The first trained model is saved in the "Model/best_model" folder of the project, with the name "IfritNet_1_b32e20.hdf5"
The second trained model is saved in the "Model/best_model" folder of the project, with the

name ”IfritNet_1_b32e15.hdf5”

5.4 IfritNetv2

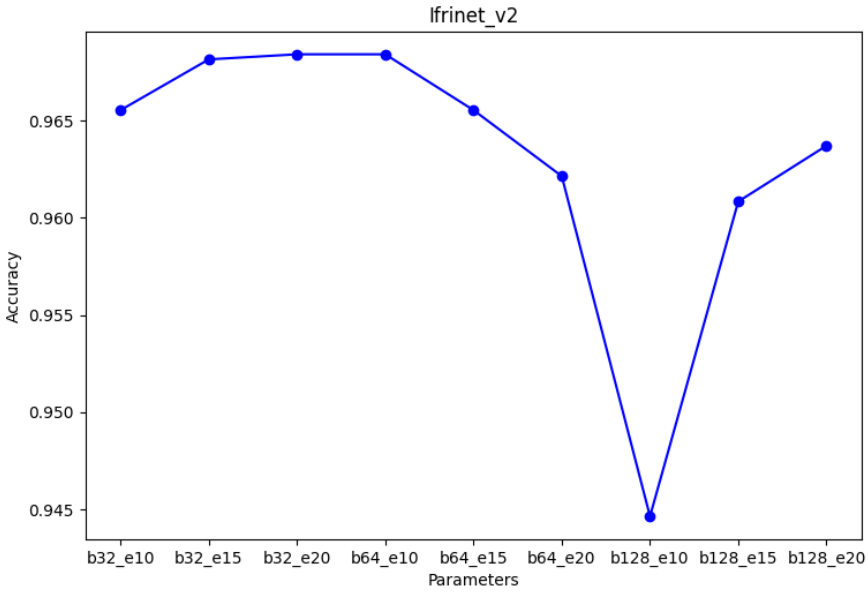


Figure 35: Illustrative image of accuracy results obtained with IfritNetv2.

In the figure 35 average accuracy for each parameter setting tested are shown.
The performance of the network with the different settings is very good, ranging from around 94.5% to around 96.8% in accuracy. For this model we have 3 settings that have higher performance than the others and are really close to each other. These settings are:

- batch size equals 32 and patience equals 15, accuracy of 96.81%
- batch size equals 32 and patience equals 20, accuracy of 96.84%
- batch size equals 64 and patience equals 10, accuracy of 96.84%

The difference in accuracy is minimal 0.03% and in some cases even zero. Accuracy alone cannot really tell us what the best model is. The worst result is given by the setting with batch size equal to 128 and patience equal to 10, which realizes an accuracy of about 94.5%. The gap between the best setting and the worst is around 2.5%, and the performance of 8 out of 9 settings are between 96.1% and 96.84%, less than 0.8%. It can be seen that all the tested settings, except one, produce results very close to each other in terms of accuracy.

In the figure 36 average value of loss function for each parameter setting tested are shown.
The three best settings also have similar values for loss function values, in fact:

- batch size equals 32 and patience equals 15, loss function of 0.099

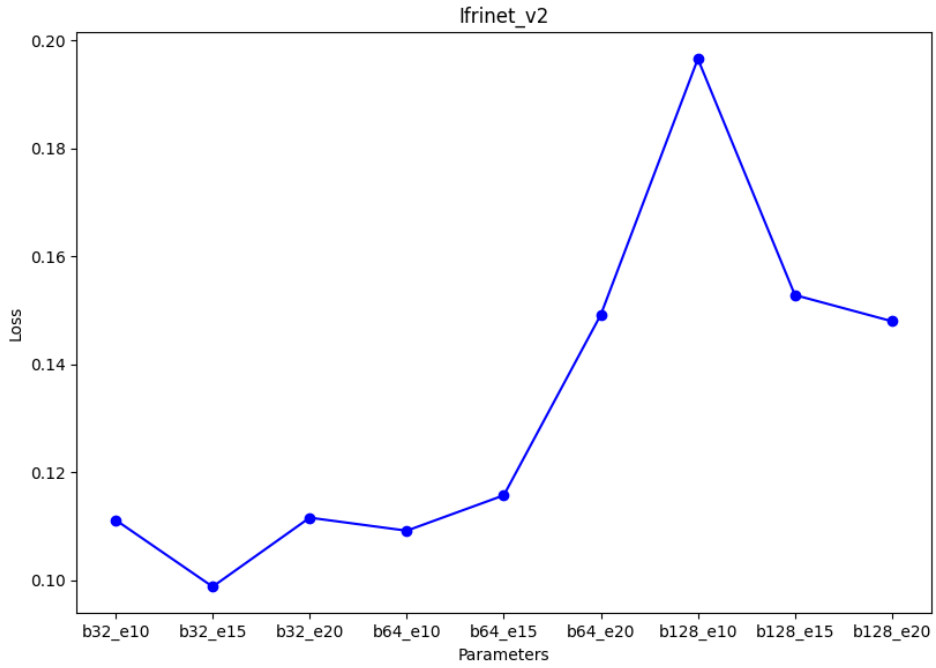


Figure 36: Illustrative image of loss function results obtained with IfritNetv2.

- batch size equals 32 and patience equals 20, loss function of 0.112
- batch size equals 64 and patience equals 10, loss function of 0.109

Looking at the loss we see that the model with worst accuracy of the three has the lowest loss function value, but again it is a very small gap.

In our problem the number of false negatives is discriminating factor for the performance of the models, so now I analyze the confusion matrices related to these 3 settings, fig 37.

Now for clarity i will denote as the first model the one with batch size equal to 32 and patience 15, denote as the second model the one with batch size equal to 32 and patience of 20, and denote as the third model the one with batch size equal to 64 and patience of 10.

The first model, which has lower accuracy, is better than the other 2 with regard to true negatives and false positives but is the worst with regard to false negatives. Since of both accuracy and false negative it is the worst among the three models it can be discarded.
Analyzing the second and third models shows that the second is better than the third for true positives and false negatives. This combined with the fact that the accuracy of the two models is identical leads to choosing the second model as the best.

Now I train the model, without cross validation, using the best parameters and showing the results obtained. In the dataset, the provided test set consists of only 50 images, 25 per class. Given the small number of the provided test set, I used a random selection from the training set (the 20%) as the test set.

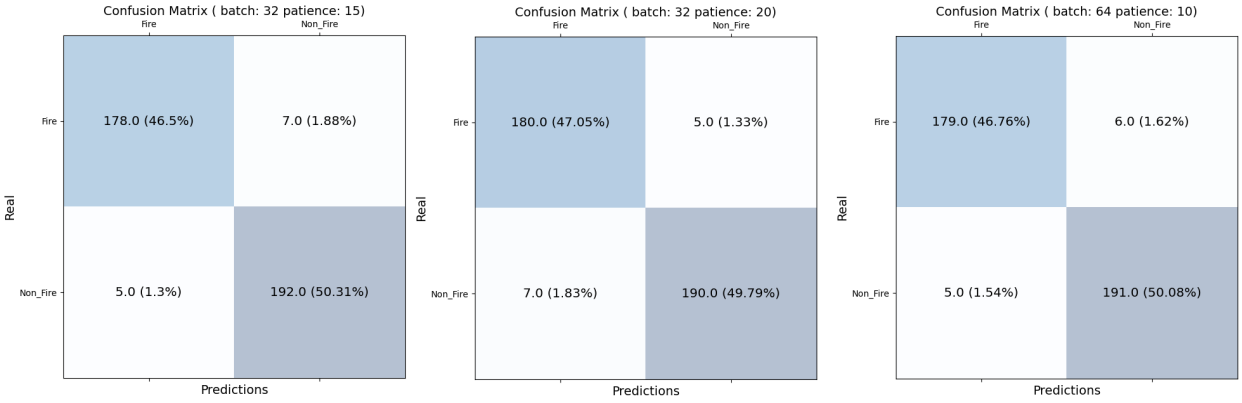


Figure 37: Illustrative image of the confusions matrix obtained with IfritNetv2 with settings batch size 32 and patience 15, with batch size 32 and patience 20 and with batch size 64 and patience 10.

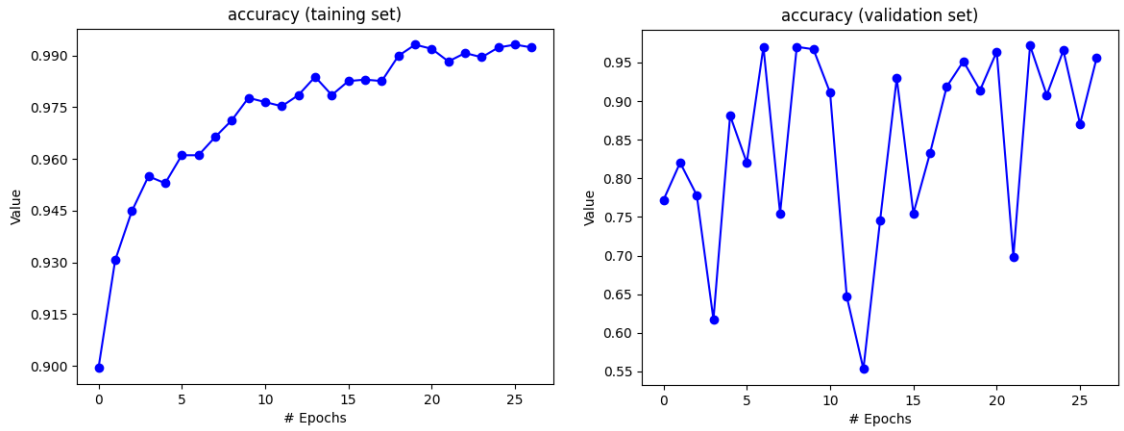


Figure 38: Illustrative image of accuracy results obtained with the best IfritNetv2 settings.

The accuracy of the training and validation set during network training are shown in the figure 38.
The values of the loss function of the training and validation set during network training are shown in the figure 39.

Network training ran for 27 epochs before being interrupted by early stopping, so at the 7th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 9th epoch, then the descent slows down until it remains almost constant, about from the 17th epoch. Looking at the values for the validation set I notice that the curve is much more irregular and linear but has much the same trend as the values for the training set. I

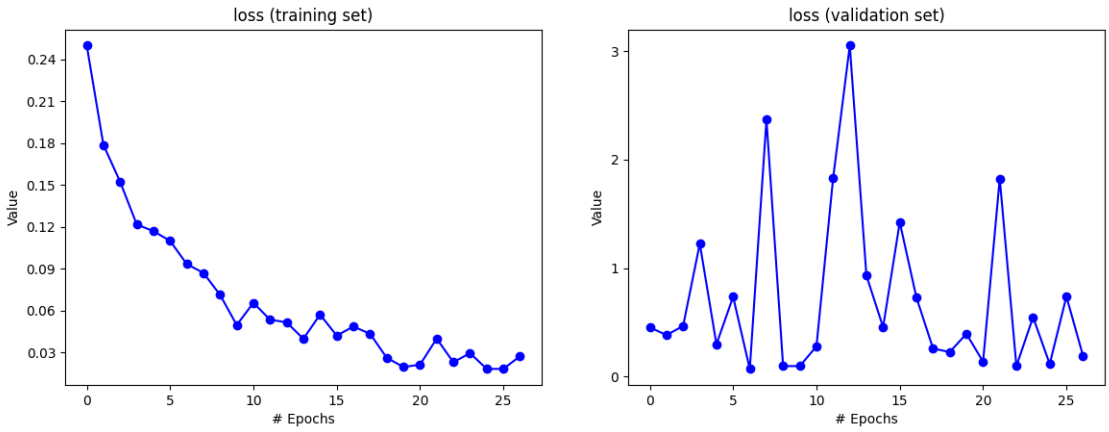


Figure 39: Illustrative image of loss function results obtained with the best IfritNetv2 settings.

noticed that the differences from the training set is that in the validation set the curve is much more irregular and tends to be flat. In the beginning you have a downward trend, up to the 10th epoch, then you have a big spike in the middle , from the 10th to the 17th epoch, and then towards the end a slight upward trend, from the 22nd epoch.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 10th epoch, then the increase slows down until it remains almost constant, about from the 20th epoch. Looking at the values for the validation set I notice that the curve is much more irregular but has much the same pattern as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around the 6th epoch, and then tends to get slightly worse from the 22th epoch onward.

The values shown are consistent with training that ran for 27 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 7) we were able to have the model with better performance, that is, before the beginning of overtraining.

The results on the test set of this final model achieved 95.05% accuracy and 0.1535 for the loss function and 1.96% for false negatives (as shown in the figure 40). These are slightly lower performances than those obtained with cross validation but not discordant.

This trained model is saved in the "Model/best_model" folder of the project, with the name "IfritNet_2.hdf5"

5.5 IfritNetv3

In the figure 41 average accuracy for each parameter setting tested are shown.
The performance of the network with the different settings is very good, ranging from 96.2% to

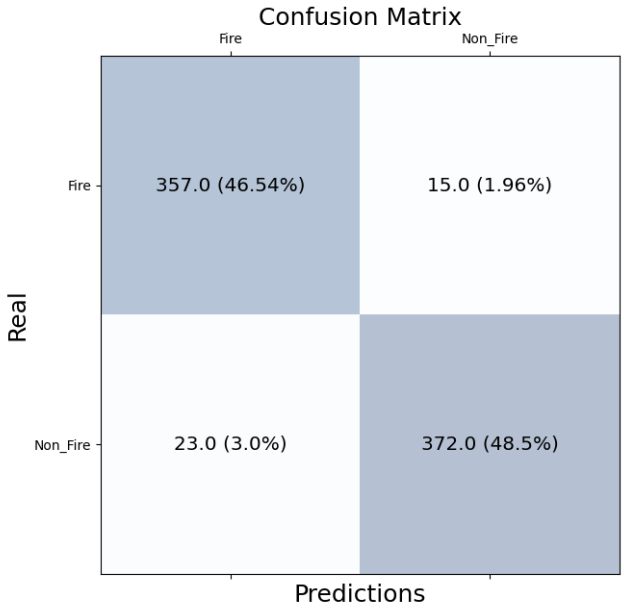


Figure 40: Illustrative image of the confusions matrix obtained with final IfritNet v2 model with settings batch size 32 and patience 20.

97.2% in accuracy. In particular, we can observe that the setting that performed best is a batch size of 64 and patience of 20 that achieved 97.18% in accuracy. While the worst result was with batch size of 64 and patience of 15 which achieved just over 96.3% in accuracy. The gap between the best setting and the second best, batch size 32 and patience of 15, is around 0.3%. The area of highest concentration is between 96.5% and 96.9% where there are the performance of 5 out of 9 settings, a range of 0.4%. It can be seen that all the tested settings, produce results close to each other in terms of accuracy.

In figure 42 average values of loss function for each parameter setting tested are shown.

Now I analyze the confusion matrix related to the two settings with better accuracy, fig 43. Now for clarity, I will denote as the first model the one with batch size equal to 64 and patience 20 and denote as the second model the one with a batch size equal to 32 and patience of 15. Confusion matrices of both the first and second best models are shown because given the small difference in accuracy between them (0.3%) if the second model results with lower false negatives than the other it might be convenient to evaluate it as better, given the importance of having low false negatives.
The confusion matrix of the first model is slightly better on all fields (TP, FP, TN, FN) than the second model.

Now I train the model, without cross-validation, using the two best parameters and showing the results obtained to analyze them and help to determine which of the two settings is better.

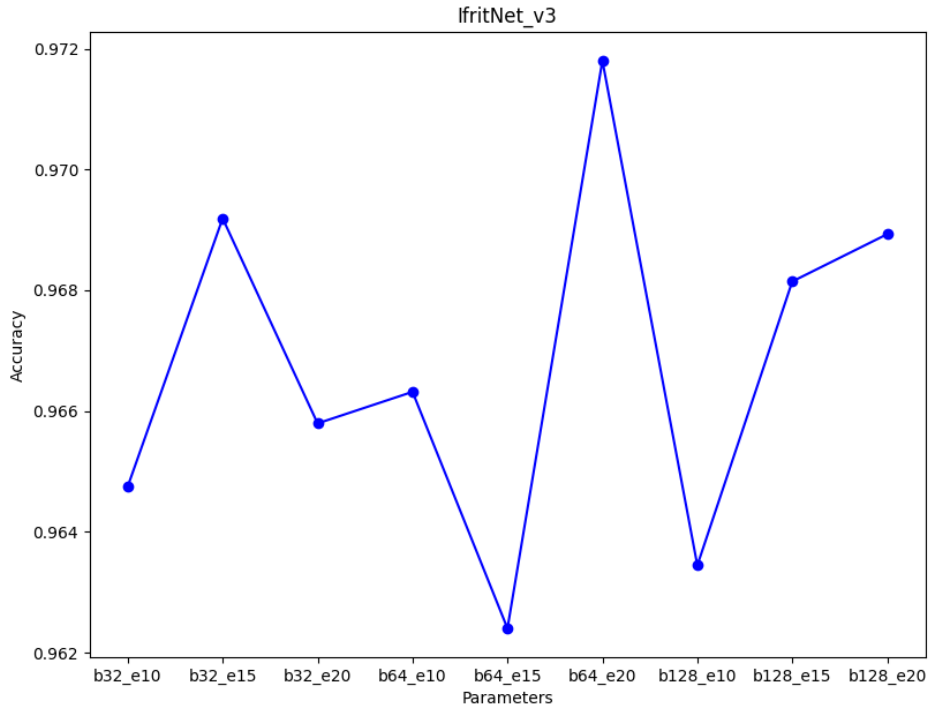


Figure 41: Illustrative image of accuracy results obtained with IfritNetv3.

In the dataset, the provided test set consists of only 50 images, 25 per class. Given the small number of the provided test set, I used a random selection from the training set (the 20%) as the test set.
The accuracy of the training and validation set during network training for the first model are shown in the figure 44.
The values of the loss function of the training and validation set during network training for the first model are shown in figure 45.

Network training ran for 44 epochs before being interrupted by early stopping, so at the 24th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 10th epoch, then the descent slows down until it remains almost constant, about from the 20th epoch.
Looking at the values for the validation set I notice that the curve is irregular with a downward trend until the 8th epoch and then a slight upward trend, after the 35th epoch. I noticed that the difference from the training set is that in the validation set the curve is much more irregular and tends to be more flat in the middle. In the beginning, you have a downward trend, up to the 8th epoch, then you have a some spike in the middle, from the 10th to the 35th epoch, and then towards the end a slight upward trend, from the 35th epoch.

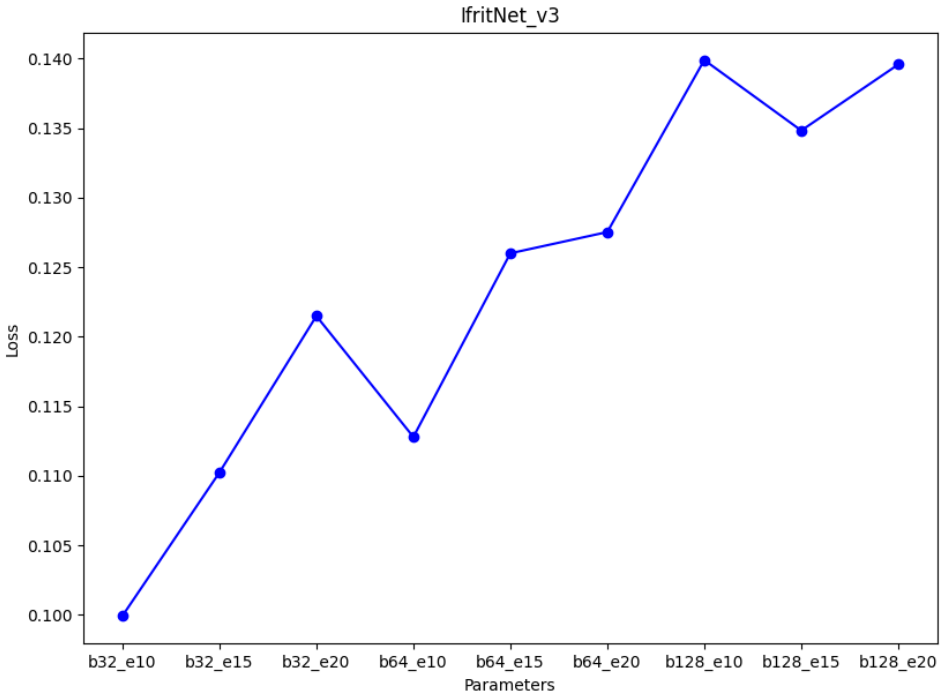


Figure 42: Illustrative image of loss function results obtained with IfritNetv3.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 10th epoch, then the increase slows down until it remains almost constant, about from the 15th epoch. Looking at the values for the validation set I notice that the curve is much more irregular and linear but has much the same pattern as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around the 13th epoch, and then tends to get slightly worse from the 34th epoch onward.

The values shown are consistent with the training that ran for 44 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 24) we were able to have the model with better performance, that is, before the beginning of overtraining.

The accuracy of the training and validation set during network training for the second model are shown in the figure 46.
The values of the loss function of the training and validation set during network training for the second model are shown in figure 47.

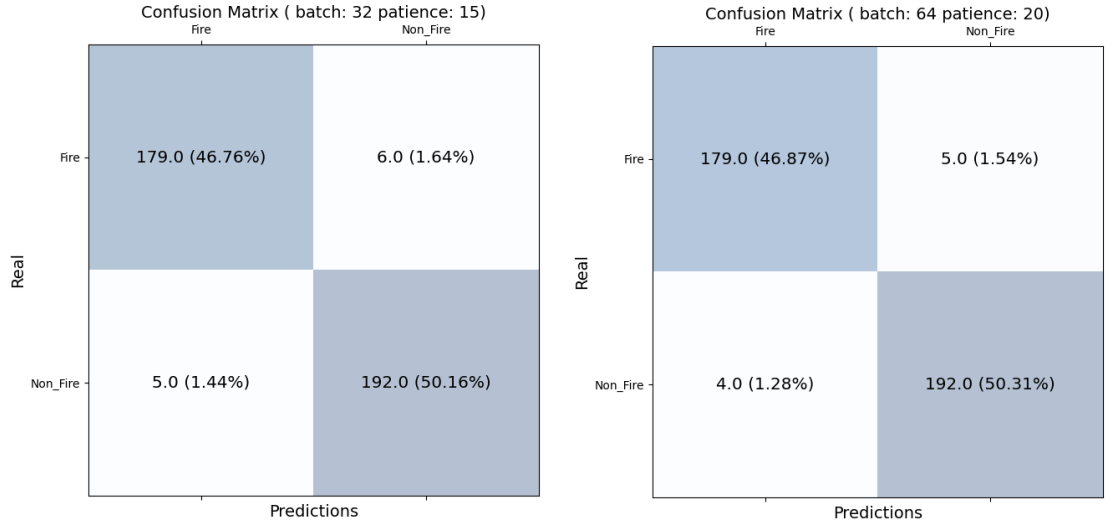


Figure 43: Illustrative image of the confusions matrix obtained by IfritNetv3 with settings batch size 32 and patience 15 and with batch size 64 and patience 20.

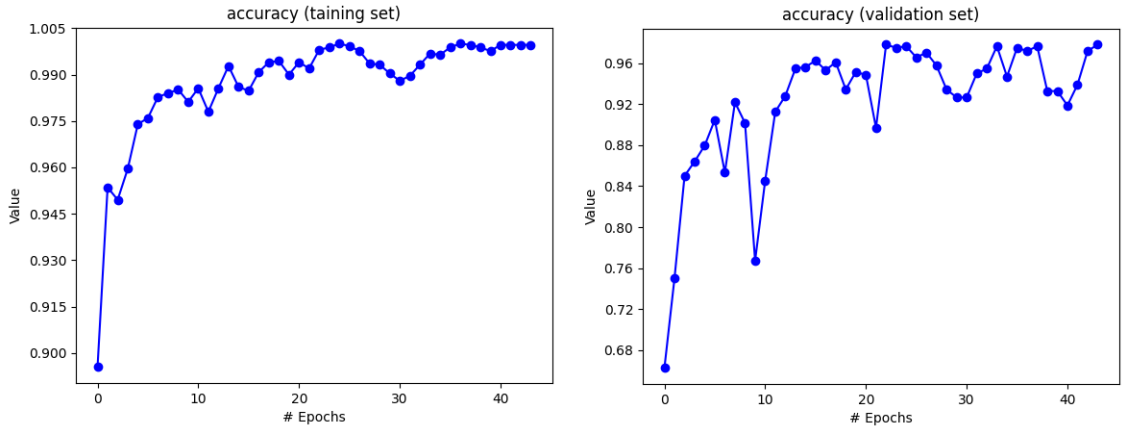


Figure 44: Illustrative image of accuracy results obtained with the IfritNetv3 settings batch size 64 and patience 20.

Network training ran for 38 epochs before being interrupted by early stopping, so at the 23th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 10th epoch, then the descent slows down until it remains almost constant, about from the 25th epoch.
Looking at the values for the validation set I notice that the curve is very linear and irregular with a slight downward trend until the 23rd epoch and then a slight upward trend. I notice

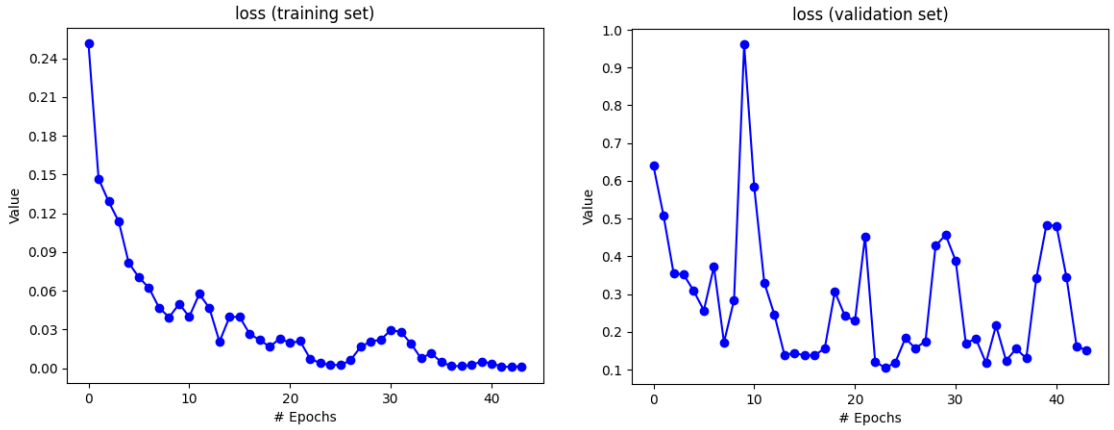


Figure 45: Illustrative image of loss function results obtained with the IfritNetv3 settings batch size 64 and patience 20.

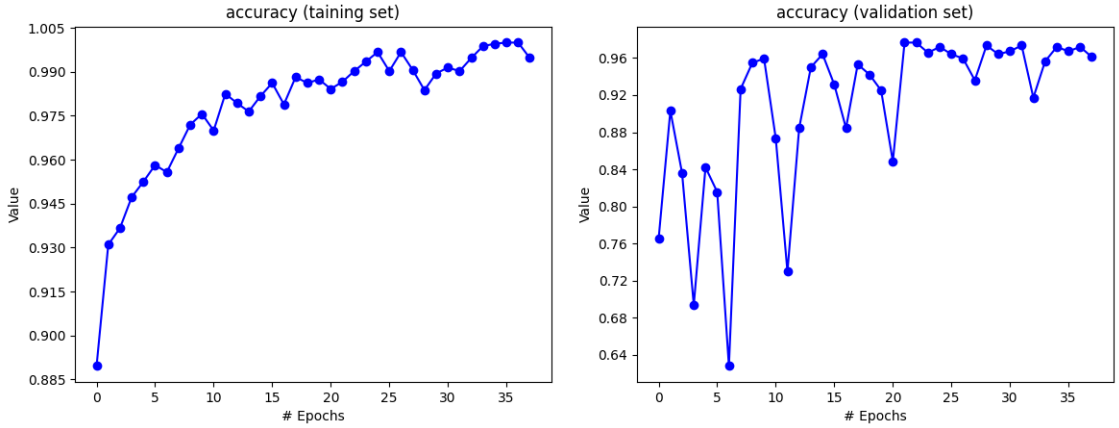


Figure 46: Illustrative image of accuracy results obtained with the IfritNetv3 settings batch size 32 and patience 15.

differences from the training set in that it tends to become more stationary from earlier, at the beginning, and then tends to get slightly worse from the 34th epoch onward.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 10th epoch, then the increase slows down until it remains almost constant, about from the 25th epoch. Looking at the values for the validation set I notice that the curve is much more irregular and linear but has much the same pattern as the values for the training set. I notice as differences from the training set that it tends to become more stationary from earlier, around the 10th epoch, and then tends to get slightly worse from the 33rd epoch onward.

The values shown are consistent with the training that ran for 38 epochs using early stopping with patience 15 on the loss function metric. It can be seen that there is a slight overtraining,

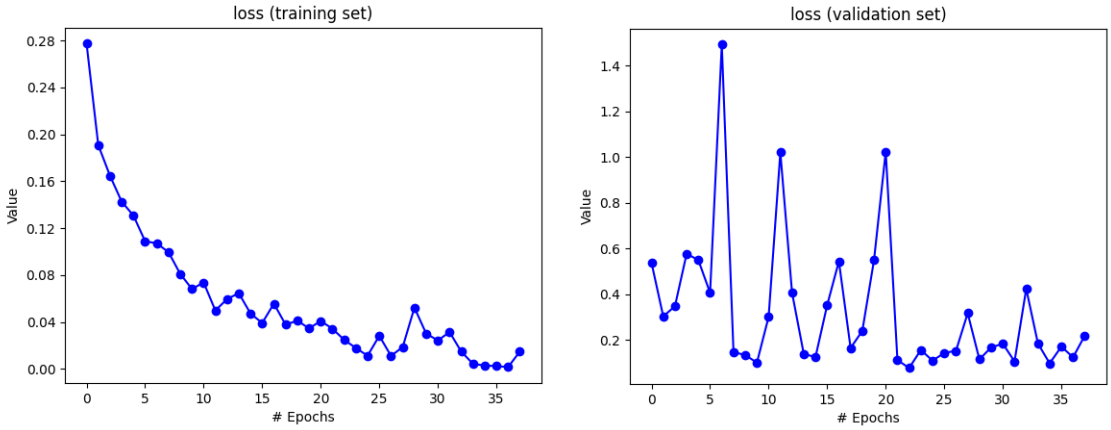


Figure 47: Illustrative image of loss function results obtained with the IfritNetv3 settings batch size 32 and patience 15.

the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 23) we were able to have the model with better performance, that is, before the beginning of overtraining.

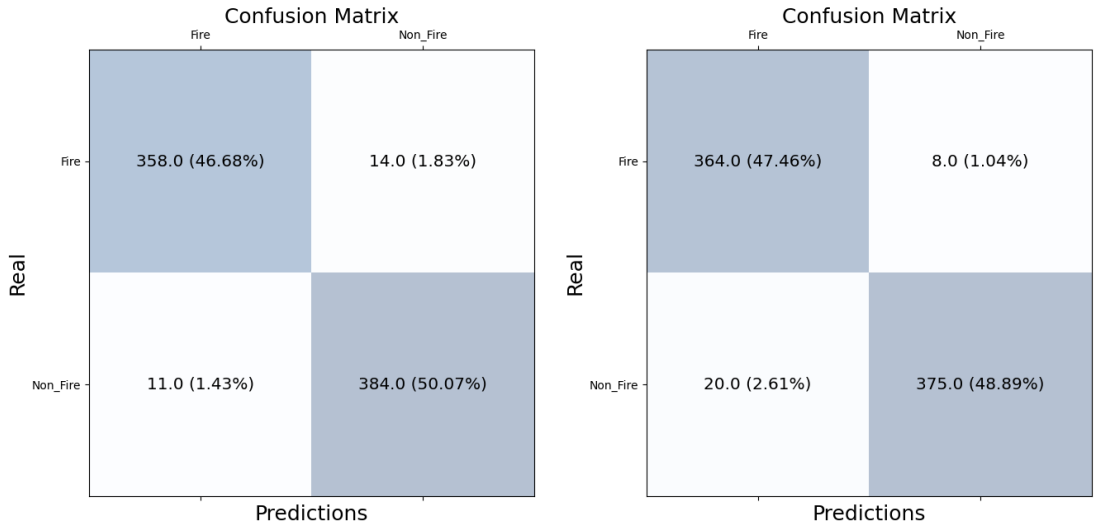


Figure 48: Illustrative image of the confusions matrix obtained with bests IfritNetv3 model with settings batch size 64 and patience 20(left) and batch size 32 and patience 15(right).

With the help of the confusion matrices of the two models examined, fig 48, I make a final

report to determine the best one:

- batch size 64 and patience 20 (first model) : achieved an accuracy of 96.74%, a loss value of 0.2046, and false negatives of 1.83%;
- batch size 32 and patience 15 (second model) : achieved an accuracy of 96.35%, a loss value of 0.1821, and false negatives of 1.04%.

The second model has a worse accuracy, 0.39% difference, a slightly better loss however and a better false negative, 0.79% difference. I consider the second model better due to the better false negative percentage and a similar accuracy to the first one.

The first trained model is saved in the "Model/best_model" folder of the project, with the name "IfritNet_3_b64e20.hdf5"
The second trained model is saved in the "Model/best_model" folder of the project, with the name "IfritNet_3_b32e15.hdf5"

5.6 IfritNetv4

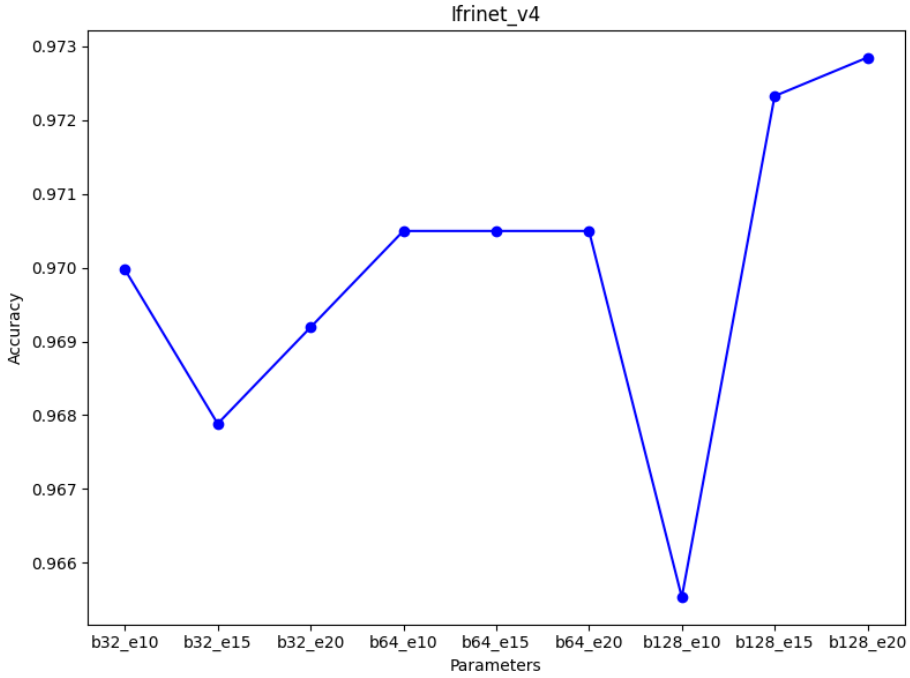


Figure 49: Illustrative image of accuracy results obtained with IfritNetv4.

In the figure 49 average accuracy for each parameter setting tested are shown.

The performance of the network with the different settings is very good, ranging from 96.5% to 97.2% in accuracy. In particular, we can observe that the setting that performed best is a batch size of 128 and a patience of 20 that achieved 97.28% in accuracy. While the worst result was with batch size of 128 and patience of 10 which achieved just over 96.55% in accuracy. The gap between the best setting and the second best, batch size 128 and patience of 15, is very small, around 0.05%. In particular, we can see that 3 settings with the same batch size of 64 and different in patience for early stopping led to the same result of 97.05%. Another thing to note is that 6 out of 9 settings achieved accuracy levels of 97% or higher, a threshold difficulty achieved by previous models. It can be seen that all the tested settings, produce results very close to each other in terms of accuracy, the max gap between two settings is of 0.7%.

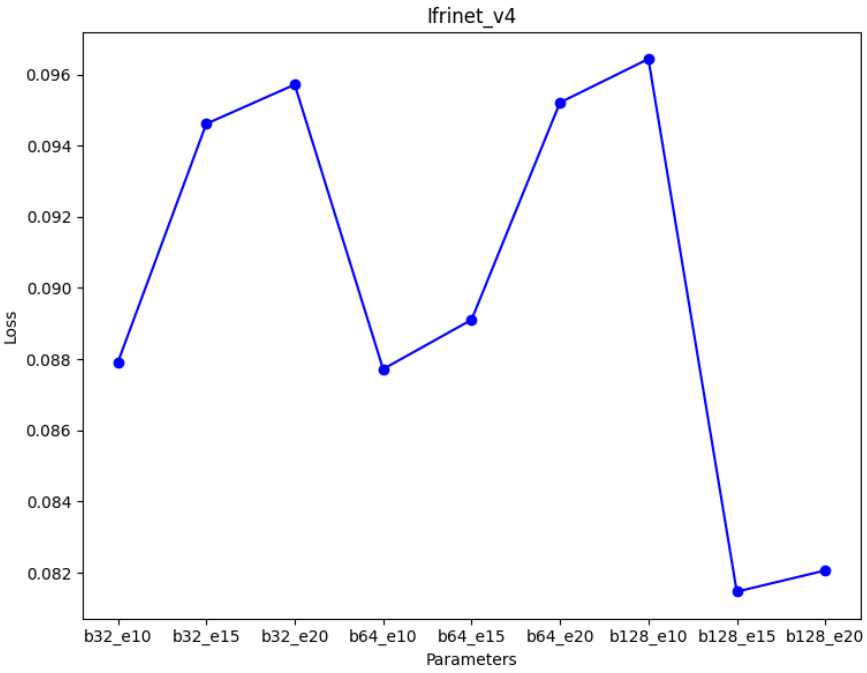


Figure 50: Illustrative image of loss function results obtained with IfritNetv4.

In figure 50 average values of the loss function for each parameter setting tested are shown.

Now I analyze the confusion matrix related to the two settings with better accuracy, fig 51. Now for clarity, I will denote as the first model the one with batch size equal to 128 and patience 20 and denote as the second model the one with batch size equal to 128 and patience of 15.
Confusion matrices of both the first and second best models are shown because given the small difference in accuracy between them (0.05%) if the second model results with lower false negatives than the other it might be convenient to evaluate it as better, given the importance of having low false negatives.
The confusion matrix of the first model is slightly better on 2 fields over 4 (FP, TN) than the second model. The second model is better for TP and FN that are which are the most important

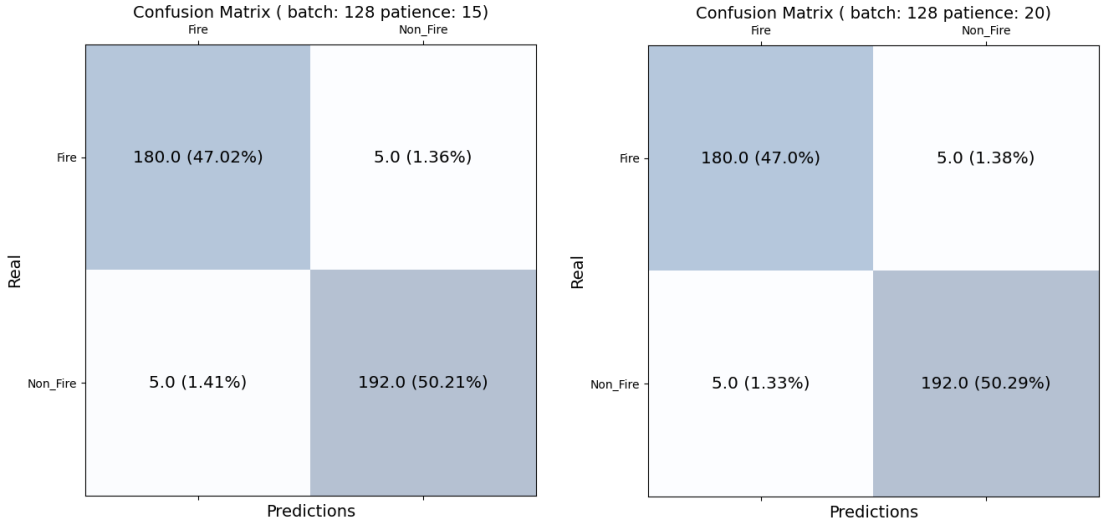


Figure 51: Illustrative image of the confusions matrix obtained by IfritNetv4 with settings batch size 128 and patience 15 and with batch size 128 and patience 20.

parameters.
The differences between the models are really minimal, 0.02% for false negative, 0.02% for true positive, 0.08% for false positive, and 0.07% for true negative.
Given the really small difference between the two models, one cannot accurately identify which one is better than the other. The first model could always be considered better than the second given the higher accuracy and the difference in false negatives of only 0.02%, which can be considered insignificant. If one wants to prioritize the false negative independently of the real amount of improvement then the second model will be the best at the cost of a very small loss in accuracy.
In this case, I choose the second as the better model for these reasons:

- a false negative better than 0.02% has more weight than an accuracy less than 0.05% considering that these two solutions have better accuracy than all settings of all other models;
- better false positive of 0.08%.

Now I train the model, without cross-validation, using the two best parameters and showing the results obtained. Although in my opinion, the second model is better I still wanted to do the training of both and compare them given the very limited difference in performance.
In the dataset, the provided test set consists of only 50 images, 25 per class. Given the small number of the provided test set, I used a random selection from the training set (the 20%) as the test set.

By training the model with the parameters of batch size 128 and patience of 20 I reached the maximum limit of 100 epochs before early stopping was triggered, so training was stopped not complete, with still room for improvement. So I remade the training by setting the maximum epoch parameter equal to 200.

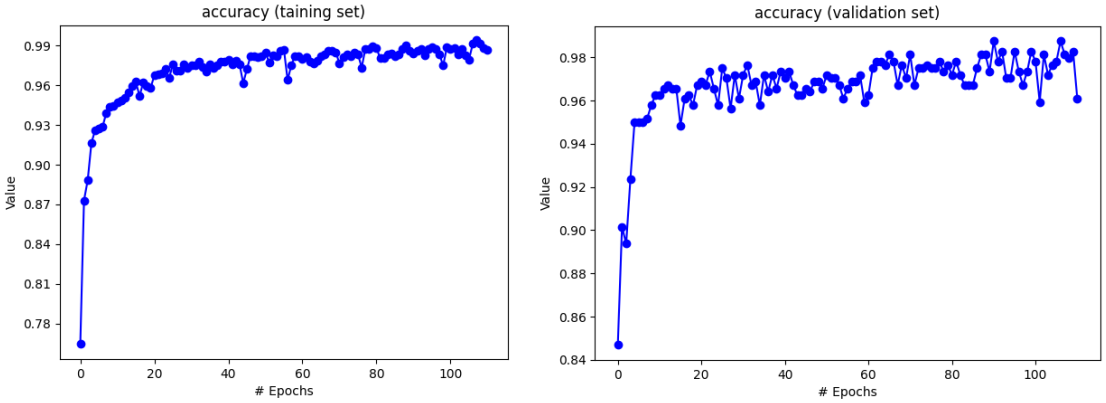


Figure 52: Illustrative image of accuracy results obtained with the IfritNetv4 settings batch size 128 and patience 20.

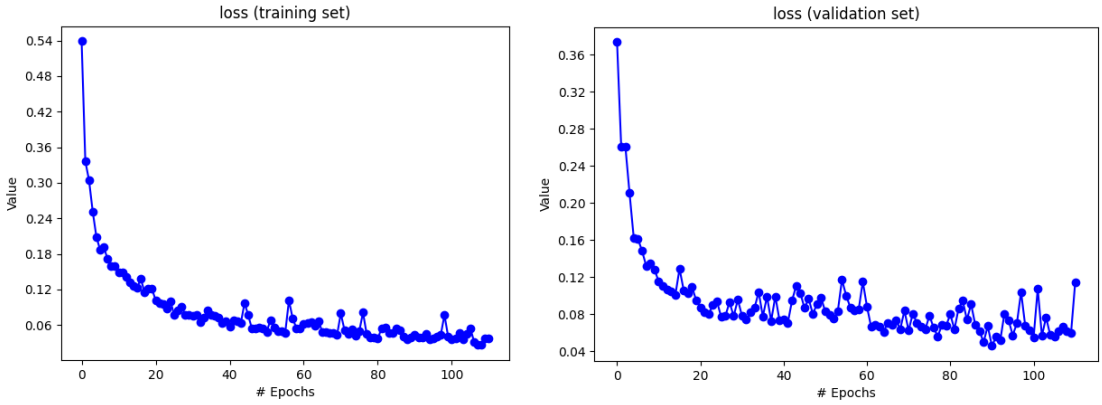


Figure 53: Illustrative image of loss function results obtained with the IfritNetv4 settings batch size 128 and patience 20.

The accuracy of the training and validation set during network training for the first model are shown in the figure 52.
The values of the loss function of the training and validation set during network training for the first model are shown in figure 53.

Network training ran for 111 epochs before being interrupted by early stopping, so at the 91th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 20th epoch, then the descent slows down until it remains almost constant, about from the 60th epoch.
Looking at the values for the validation set I notice that the curve is more irregular with a downward trand until the 20th epoch and then having a slight upward trand, after 90th epoch.

I noticed that the differences from the training set is that in the validation set the curve is much more irregular and tends to be more flat in the middle, except for the irregular peaks. In the beginning you have a downward trend, up to the 20th epoch, then you have a some spike, and then towards the end a slight upward trend, from the 91th epoch.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 20th epoch, then the increase slows down until it remains almost constant, about from the 40th epoch. Looking at the values for the validation set I notice that the curve is much more irregular and linear but has much the same pattern as the values for the training set. I notice as differences from the training set that the average trend tends to become more linear from earlier, around the 13th epoch, and then tends to get slightly worse from the 91th epoch onward.

The values shown are consistent with training that ran for 111 epochs using early stopping with patience 20 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 91) we were able to have the model with better performance, that is, before the beginning of overtraining.

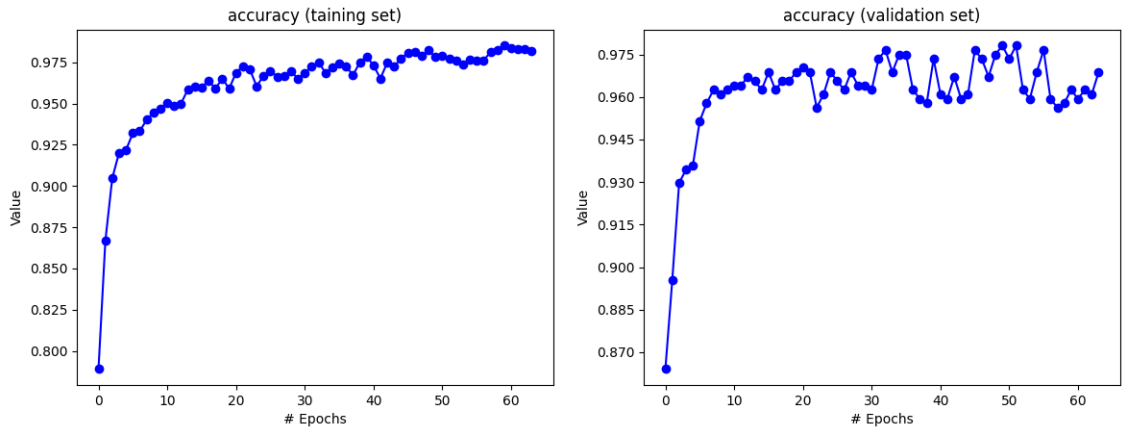


Figure 54: Illustrative image of accuracy results obtained with the IfritNetv4 settings batch size 128 and patience 15.

The accuracy of the training and validation set during network training for the second model are shown in the figure 54.
The values of the loss function of the training and validation set during network training for the second model are shown in figure 55.

Network training ran for 64 epochs before being interrupted by early stopping, so at the 49th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 15th epoch, then the descent slows down until it remains almost constant, about

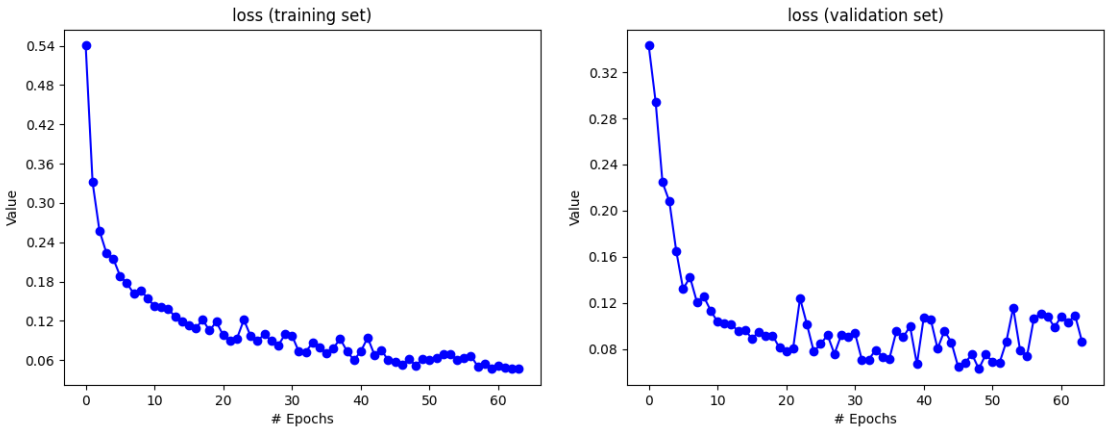


Figure 55: Illustrative image of loss function results obtained with the IfritNetv4 settings batch size 128 and patience 15.

from the 40th epoch.
Looking at the values for the validation set I notice that the curve is more irregular with a downward trend until the 20th epoch and then a slight upward trend, after the 49th epoch. I noticed that the difference from the training set is that in the validation set the curve is much more irregular. In the beginning, you have a downward trend, up to the 20th epoch, then you have a some spike, and then towards the end a slight upward trend, from the 49th epoch.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 10th epoch, then the increase slows down until it remains almost constant, about from the 20th epoch. Looking at the values for the validation set I notice that the curve is much more irregular but has more or less the same pattern as the values for the training set. I notice as differences from the training set that the average trend tends to become more linear from earlier, around the 7th epoch, and then tends to get slightly worse from the 49th epoch onward.

The values shown are consistent with the training that ran for 64 epochs using early stopping with patience 15 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 49) we were able to have the model with better performance, that is, before the beginning of overtraining.

With the help of the confusion matrices of the two models examined, fig 56, I make a final report to determine the best one:

- batch size 128 and patience 20 (first model): achieved an accuracy of 97.26%, a loss value of 0.0847, and false negatives of 1.43%;
- batch size 128 and patience 15 (second model): achieved an accuracy of 97.91%, a loss value of 0.0880, and false negatives of 1.17%.

The second model has better accuracy, a gap of 0.65%, and is better in all fields of the confusion

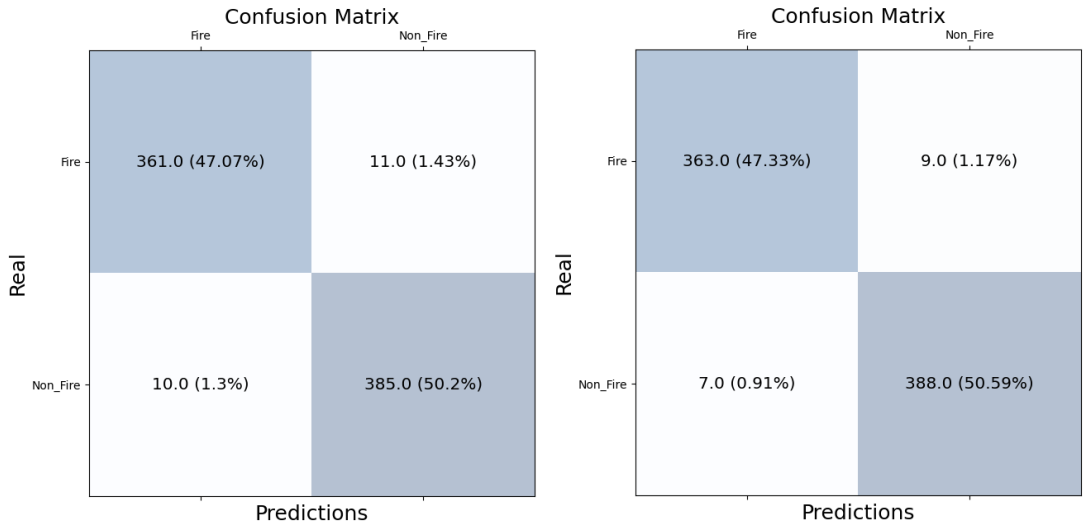


Figure 56: Illustrative image of the confusions matrix obtained with bests IfritNetv4 model with settings batch size 128 and patience 20(left) and batch size 128 and patience 15(right).

matrix, especially in false negatives, a gap of 0.26%. I consider the second model better.

The first trained model is saved in the "Model/best_model" folder of the project, with the name "IfritNet_4_b128e20.hdf5"
The second trained model is saved in the "Model/best_model" folder of the project, with the name "IfritNet_4_b128e15.hdf5"

5.7 Model comparison

In this section, I will compare all the networks tested in the project to select the best one for our problem. I now list the results of the best setting for each network examined:

- AlexNet achieved 94.99% of accuracy, 2.22% of false negative and its model weighs 189.06 MB. Prediction time of an image of 18 ms;
- GoogLeNet
- IfritNetv1 achieved 96.09% of accuracy, 1.3% of false negative and its model weighs 4.79 MB. Prediction time of an image of 19 ms;
- IfritNetv2 achieved 95.05% of accuracy, 1.96% of false negative and its model weighs 12.02 MB. Prediction time of an image of 19 ms;
- IfritNetv3 achieved 96.35% of accuracy, 1.04% of false negative and its model weighs 3.11 MB. Prediction time of an image of 18 ms;
- IfritNetv4 achieved 97.91% of accuracy, 1.17% of false negative and its model weighs 1.22 MB. Prediction time of an image of 19ms.

IfritNet v3 is the model that has the best false negative, the second-best accuracy, and the second-lowest weight and number of trainable parameters in the network. IfritNet v4 is the model that has the best accuracy, the second best false negative, and the lowest weight and number of trainable parameters in the network. IfritNet v4 has a higher accuracy of 1.56% and lower false negatives of 0.13% than IfritNet v3.
I chose IfritNet v4 as the best network in the project for the following reasons:

- highest accuracy of all and with an excellent value, almost 98%;
- second best false negative value, very close to the best value, and overall very low, 1.17%;
- smallest and lightest network among all those analyzed in this project.

The results so far with this network are very good, but I want to see if it is possible to improve further. In the next section we will use techniques, such as hypertuning for the learning rate or applying exponential learning rate decay, to try to improve the optimizer and consequently the training of the network.

5.8 Improvements

In this section I will expose additional techniques used on the model chosen as the best in the previous section, IfritNet v4 with batch size equal to 128 and patience equal to 15, to go on to further improve its performance. Below I will detail all the techniques adopted and the results obtained with them.

5.8.1 Learning rate and Dropout optimization

For analyzing and choosing the best learning rate for the optimizer and best value for dropout of the dense layer I used hypertuning (or hyperparameter tuning), which is a process that leads to choosing the best set of hyperparameters for the model. For this process I used the Tuner library from Keras(for more information see [15]).

After setting up the model, the hyperparameters and their values to be tested you have to choose the tuner model to be used.
There are 4 different types to choose from:

- RandomSearch: is based on the concept that trying every possible combination of the parameters to find the best solution (GridSearch) is not optimal since the number of combinations grows exponentially as the number of parameters increases. This approach would take too long to explore the entire hyperparameter search space. This approach randomly tries the various parameters thus going for a random exploration of the hyperparameter space. This takes less time than gridSearch but does not guarantee to find the optimal result, only to come close.
  Has several parameters, some of them of interest are:
  - The object function: representing the optimization goal during tuning;
  - Max_trail: is the maximum number of combinations that are tried to search for the optimum. is set in case of very large hyperparameter space that would make the random search go on for a long time (execution can stop before it is reached, by default equal to 10);

- Directory and project_name: these are two parameters for specifying where to save the details of the tests performed.

- Hyperband: random search has the problem that it might take a set of values for parameters that are clearly bad and go and do a complete training and evaluation, going to waste resources and time. Hyperband tries to solve this problem by going and randomly taking samples of all combinations of hyperparameters doing not full training and evaluation on them but partial training and evaluation. More precisely, it will go and train the models for a few epochs (less than max_epochs) and select only the best models (candidates) from those tested for a few epochs. It will go and repeat this process (keep only the best ones for each iteration) until it then does a full training and evaluation with only the final candidates. Hyperband determines the number of models to train in a bracket by calculating 1 + log factor (max_epochs) and rounding it to the nearest integer. Has several parameters, some of them of interest are:
  - The object function: representing the optimization goal during tuning;
  - factor: the reduction factor for the number of epochs and number of models for each bracket. Defaults to 3;
  - max_epochs: rapresent the maximum number of epochs to train one model. It is recommended to set this to a value slightly higher than the expected epochs to convergence for your largest Model, and to use early stopping during training, defaults is 100; - Directory and project_name: these are two parameters for specifying where to save the details of the tests performed.

- BayesianOptimization: technique to solve the problem given by the random selection of combinations of hyperparameters. Random selection can be a problem because it helps the exploration of the hyperparameter space but it doesn't guarantee to find the best solution. The idea is to randomly choose only the first combinations and then, based on their performance, choose the next best possible parameter, so as to take into consideration the history of the combinations of the values of the already tested hyperparameters. This process of choosing the next best value for a hyperparameter goes on until the optimum is reached or the maximum possible attempts are reached. Has several parameters, some of them of interest are:
  - The object function: representing the optimization goal during tuning;
  - max_trials: total number of trials (model configurations) to test at most, default value is 10;
  - num_initial_points: number of randomly generated samples as initial training data for Bayesian optimization. Default value is of 3 times the dimensionality of the hyperparameter space;
  alpha: value that represents the expected amount of noise in the observed performances in Bayesian optimization, default value is 1e-4;
  beta: rapresents balancing factor of exploration and exploitation. The larger it is, the more explorative it is. Default value is 2.6;
  - Directory and project_name: these are two parameters for specifying where to save the details of the tests performed.

- Sklearn: is a tuner for Scikit-learn Models, performs cross-validated hyperparameter search for Scikit-learn models.

I chose to use Hyperband because the values of the hyperparameter space are not large so there are no problems of size and random choice and with this method it is more efficient than

GridSearch.

The values tested for dropout are from 0 to 0.5 with steps of 0.1, and the values tested for learning rate are: 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005 and 0.00001.

Hypertuning for dropout and learning rate lasted 13m 40s and the results showed 0.0 as the best dropout value for the last fully connected layer and 0.005 as the best value for learning rate.

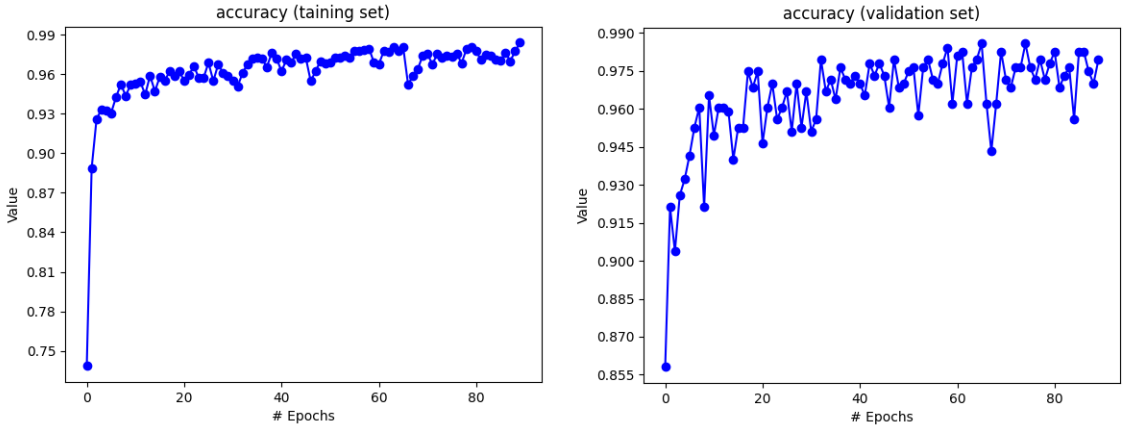


Figure 57: Illustrative image of accuracy results obtained with the IfritNetv4 settings batch size 128, patience 15, dropout 0.0 and learning rate 0.005.

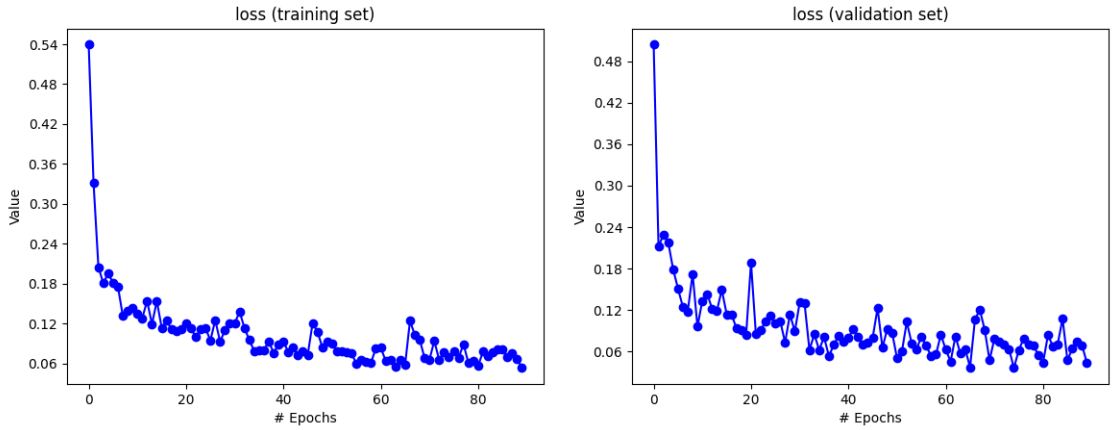


Figure 58: Illustrative image of loss function results obtained with the IfritNetv4 settings batch size 128, patience 15, dropout 0.0 and learning rate 0.005.

The accuracy of the training and validation set during network training are shown in the figure 57.
The values of the loss function of the training and validation set during network training are

shown in the figure 57.

Network training ran for 90 epochs before being interrupted by early stopping, so at the 75th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
Looking at the values for the loss function of the training set I notice that they have a pronounced descent until the 10th epoch, then the descent slows down until it remains almost constant, about from the 40th epoch. Looking at the values for the validation set I notice that the curve is more irregular but has much the same trend as the values for the training set.

Looking at the values for training accuracy I notice that they have a pronounced increase until the 5th epoch, then the increase slows down until it remains almost constant, about from the 10th epoch. Looking at the values for the validation set I notice that the curve is much more irregular but has much the same pattern as the values for the training set. I noticed that the network achieves in a few epochs (3) good levels of accuracy , above 90%, and then the rest of the training, almost 90 epochs, used it to improve performance by 9 percentage points. So the network learns very quickly how to solve our classification problem with good performance and then has a very long refinement phase that leads to really good performance.

The values shown are consistent with training that ran for 90 epochs using early stopping with patience 15 on the loss function metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. With early stopping we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 75) we were able to have the model with better performance, that is, before the beginning of overtraining.

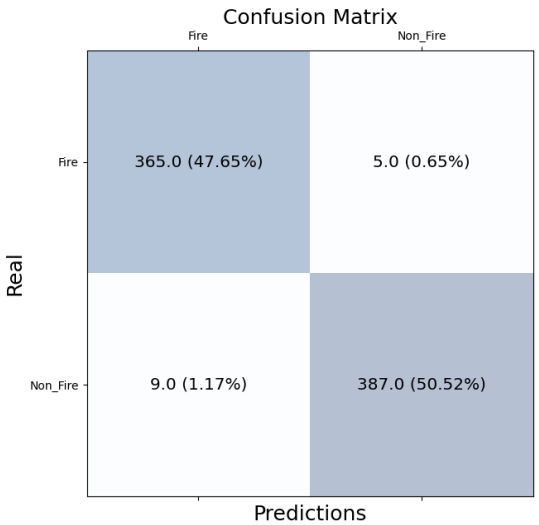


Figure 59: Illustrative image of the confusions matrix obtained with bests IfritNetv4 settings batch size 128, patience 15, dropout 0.0 and learning rate 0.005.

The results on the test set of this final model achieved 98.17% accuracy and 0.0632 for the loss function and 0.65% for false negatives (as shown in the figure 59). These results are really great, this model has the best accuracy and the best false negatives obtained so far.

The model trained is saved in the "Model/best_model" folder of the project, with the name "IfritNet_4_B_LR.hdf5"

5.8.2 Class weights

Another technique that can be used is to act on the weights given to the classes during training. Usually this is done when you have a highly unbalanced dataset, going to give much greater weight to the minority class samples to give them more importance during training. Acting on the class weights and increasing the weight of the "Fire" class could go to improve false negatives and true positives but this could lead to a worsening of the overall accuracy and false positives. In our case where a failure to detect fire is considered more important than a false positive is an operation that may make sense. I decided not to do it because the last obtained network model has very good performance in both accuracy (greater than 98%) and false negatives (about 0.6%). The false negatives obtained are already very low so I think it is not useful to go and act on the class weights to try to improve false negatives (very limited range of improvement) by going to compromise accuracy (greater than 98%) and false positive (which is very good 1.17%) and true negative. Although I have not performed this technique I have reported it for completeness.

5.8.3 Validation Accuracy as early stop metrics

Another method that can be tried to improve model performance is to go to use early stopping by monitoring the accuracy of the validation set instead of its loss.
As mentioned above, the loss function represents the average error of the output generated by the network with respect to the labels. So a lower loss in general indicates a more accurate network and leads to improved accuracy, but the accuracy does not depend so much on the average error of the outputs but more on the distribution of the individual errors. This explains why same values of loss can give different values of accuracy and different values of loss can give same values of accuracy. Until now I have been using the loss function because the goal was not only very good accuracy but also low false negatives so I opted to improving the loss value which i supposed to give models with good performance (as actually happened). I can now try to take the best settings found so far and perform early stopping by monitoring the accuracy of the validation set to see if we get a model with better performance.

The accuracy of the training and validation set during network training are shown in the figure 60.
The values of the loss function of the training and validation set during network training are shown in the figure 61.

Network training ran for 64 epochs before being interrupted by early stopping, so at the 49th epoch there was the last checkpoint, and from there on the training produced only worse values for the loss function.
The first thing I noticed is that the trend of both the loss function value and accuracy is much

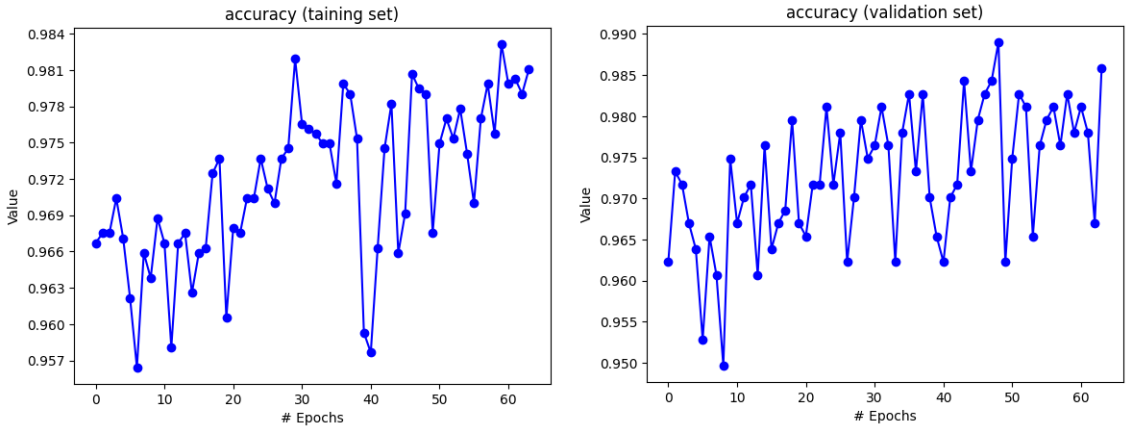


Figure 60: Illustrative image of accuracy results obtained with the IfritNetv4 settings batch size 128, patience 15, dropout 0.0, learning rate 0.005 and early stopping monitoring val accuracy.

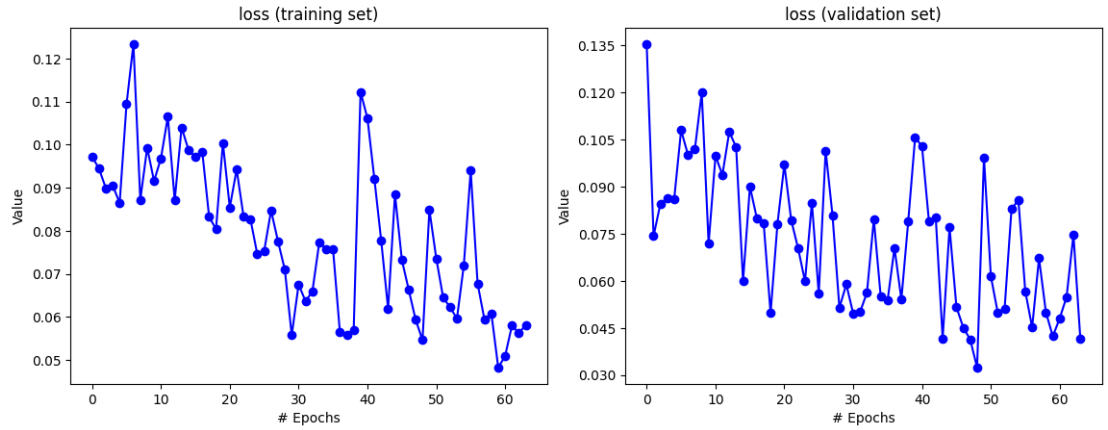


Figure 61: Illustrative image of loss function results obtained with the IfritNetv4 settings batch size 128, patience 15, dropout 0.0, learning rate 0.005 and early stopping monitoring val accuracy.

more irregular than the model trained with early stopping that monitors the loss function. However, one can see clear training consistent with the training parameters as in the previous cases.

Looking at the accuracy values of the validation set I notice an (irregular but constant) upward trend for the first almost 50 epochs after which I notice a downward trend with progressive worsening.
Looking at the accuracy values of the training set I notice more discordant and irregular trends compared to the validation set. In the first 20 epochs there is a stable, slightly downward trend. After until about the 30th epoch there is a strong upward trend followed by another stable trend until about the 55th epoch. From the 55th epochs until the end of the training instead there is a slight upward trend.

Looking at the loss function values of the training set I notice for the first 15 epochs an

irregular but stable trend followed by a strong downward trend until the 30th epoch. After that until the end there is a stable trend although with very irregular values.
Looking at the loss function values of the validation set I notice ceh from the beginning to the end they have a slight irregular downward trend with some small stable sequences.

The values shown are consistent with training that ran for 64 epochs using early stopping with patience 15 on the validation accuracy metric. It can be seen that there is a slight overtraining, the accuracy of the training set is higher and tended to rise (although slowly) while the accuracy of the validation set has stabilized and was starting to fall. Also with early stopping monitoring val_accuracy we were able to see the beginning of overtraining and stop while using the checkpoints at the best result (epoch 49) we were able to have the model with better performance, that is, before the beginning of overtraining.

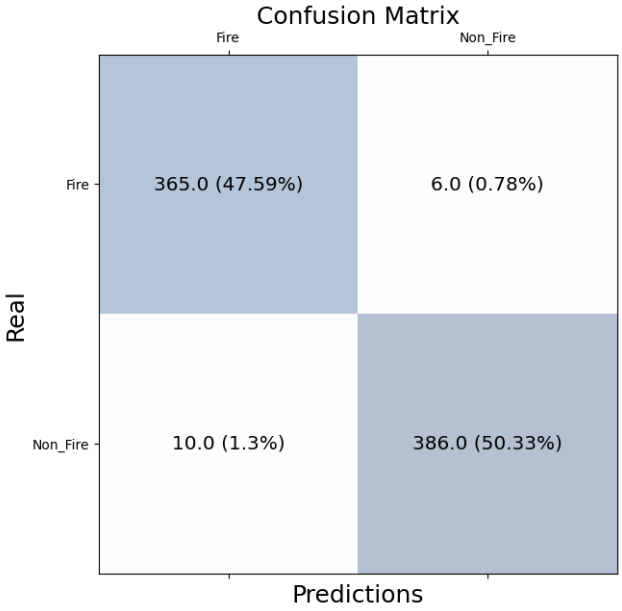


Figure 62: Illustrative image of the confusions matrix obtained with IfritNetv4 settings batch size 128, patience 15, dropout 0.0, learning rate 0.005 and early stopping monitoring val accuracy..

The results on the test set of this final model achieved 97.91% accuracy and 0.0759 for the loss function and 0.78% for false negatives (as shown in the figure 62).
This model has very good performance but does not exceed the performance of the model with the best settings and with early stopping that monitors the loss function.
Comparing the models, it can be seen that the latter model is slightly worse in accuracy (0.26% less) and as a false negative (0.13% more), and as a false positive (0.13% more). The difference between the models is not very pronounced, but the latter one, although slightly, is worse in all metrics examined in this project.
For these reasons I consider this model to be worse than the previous model which I consider to be the best model in the whole project.

This trained model is saved in the "Model/best_model" folder of the project, with the name

"IfritNet_4_vall_acc.hdf5"

6 Conclusion

In this section, I summarize the work done on the project and future improvements or evolutions of the project idea.

In this project, I addressed the problem of forest fire detection and tried to give a solution that uses a CNN to detect the presence of fires through images.
I watched and learned from two known CNN models (AlexNet and GoogLeNet) and carried out several tests starting with a version of CNN as simple as possible to then evolve it with the techniques seen (inception module, 1x1 filters, etc.) and create my CNN. In the evolution, I tried to make each version better for our problem by making it lighter than the previous one while maintaining or improving performance.
Each network has hyperparameters that define its architecture and affect performance. Finding the ideal set of parameters is not easy and requires time and testing. To find the best parameters in the design, I adopted different techniques.

- For the batch size and early stopping parameters, I used 10-cross validation on every possible pair of values (range of values chosen by me) to get statistically reliable performance data;
- For the choice of the best dropout in the last layer (fully connected) and the learning rate I used hypertuning from the Keras library;

I also tried changing the early stopping metric on the best model to see if I could see any improvement. Thanks to this project I was able to get a lot of practice on the:

- using online tools (google Colab);
- the use of physical machines with GPUs (with the consequent setting up of special software and libraries for the use of AI) Consequent analysis of benefits and disadvantages between laptop and fixed PC versions of Nvidia GPUs;
- analysis of datasets for the purposes of coherence and image quality for training a CNN.

The possible improvements for this project are numerous and can cover several aspects:

- dataset improvement: an optimal dataset can greatly improve network performance on the problem addressed. A possible improvement could be to improve the quantity and quality of both Fire and Non_Fire class images or to add new images belonging to classes for future improvements such as "Smoke";
- network size reduction and optimization for execution on embedded and specialized AI execution devices (e.g. jetson nano);
- early fire detection: a future improvement of the network is to add functionality for early fire detection. Being able to identify a fire from its early stages (when it may be small or there is only smoke) could reduce damage considerably and provide an even earlier warning of danger. A technique that can be added to the network and which could be useful in early detection are residual connections (connections in which the output of a previous layer is added to the input of a following layer) which can help safeguard information from small fires or smoke.

A possible future application and use of the designed network is the inclusion of the network in a special device connected to a camera. This monitoring device would control a portion of the forest and in case of detection of a fire, it could send an alarm together with a photo of the situation, in this way false alarms could also be identified immediately.

7 Implementation

All of the project code, the results obtained, the modified dataset and the trained networks are on both GitHub and google drive (shared with professors for review). In this section, I illustrate all the programs and their breakdown and location on both GitHub and GoogleDrive.
I also provide a brief guide for using the program with GUI through also screenshots of the application.

7.1 GitHub

The application code is divided into four files (present in the project's root folder):

- Fire_cls_GUI: this file contains the GUI realized by Python Tkinter. It is the main file and manages the interface with the user and all the operations required for the correct functioning of the application;
- AlexNet_class: in this file, there is the AlexNet class, realized in Python. This class implements one method for making the model, one for compiling it, and one for returning the made model. The structure of the network is generic and customizable in terms of the size of input images and the number of output classes;
- GoogLeNet_class: this file contains the GoogLeNet class realized in Python. This class implements one method for making the model, one for compiling it, and one for returning the made model. The structure of the network is generic and customizable in terms of the size of input images and the number of output classes;
- IfritNet_class: in this file, there is the IfritNet class, realized in Python. This class implements one method for making the model of each version of IfritNet CNN realized for this project, one for compiling it, and one for returning the made model. The structure of the network is generic and customizable in terms of the size of input images and the number of output classes.

In the project's root folder, there are these folders:

- Dataset: folder containing the dataset images used for training and testing networks in the project. This folder is divided into:
  - old_ds: contains the original dataset except for the corrupted images that have been removed;
  - new_ds: contains the dataset after the image analysis and selection phase, it does not contain the images that I found not congruent with the problem addressed.
- Model: folder containing the saved model of the CNN trained during the project. This folder is divided into:
  - train_hdf5: Contains checkpoint saves of models made during network training phases and other test saves;
  - best_model: contains the saves of the best models.
- Others: folder containing other utility or test python files or for future implementations;

- result_test_CNN: folder containing a record of all network training done for the project. It is divided into subfolders, one for each type of CNN seen in the project. There is data pertaining to workouts done with GTX 1080, RTX 3060, and Google Colab. The training reports are both in notepad form (text lines) and saved images (accuracy and loss trends and confusion matrix).

The complete application code is available on GitHub[16].

7.2 Shared Google Drive (for project examination)

The shared folder in google drive where all the material related to the delivery of the exam is located is called "Project_CIDL", inside it is a folder called "Diana" in which all the delivery material is located.

In the "Diana" folder, there are these folders and notebook files:

- Dataset: folder containing the dataset images used for training and testing networks in the project. This folder is divided into:
  - old_ds: contains the original dataset except for the corrupted images that have been removed;
  - new_ds: contains the dataset after the image analysis and selection phase, it does not contain the images that I found not congruent with the problem addressed.
- Model: folder containing the saved model of the CNN trained during the project. This folder is divided into:
  - train_hdf5: Contains checkpoint saves of models made during network training phases and other test saves;
  - best_model: contains the saves of the best models.
- result_test_CNN: folder containing a record of all network training done for the project. It is divided into subfolders, one for each type of CNN seen in the project. There is data pertaining to workouts done with GTX 1080, RTX 3060, and Google Colab. The training reports are both in notepad form (text lines) and saved images (accuracy and loss trends and confusion matrix).
- dataset_Analyzer: notebook in Python that contains the code for analyzing the dataset. A program that given a dataset counts the number of images present, their format, their shape, the presence or absence of corrupted images, the number of classes, and the RGB distribution of images for each class;
- model_k_cross_evaluation: a program that given a set of values for batch size, patience for early stopping, model, and specified k performs a k-cross validation for each pair of batch size and patience values. It shows the confusion matrix, accuracy, and loss averages for each pair of values;
- networks_fit_evaluate: notebook in Python that contains classes for each CNN used in the project and allows you to train (by specifying various parameters), load, and save a model for each type of CNN in the project. In addition to training, evaluation, and visualization of model performance are also performed;

- IfritNet_4_acc: notebook in Python that contains code for the IfrtitNet version 4 network training test with the best parameters and with early stopping that monitors validation accuracy instead of validation loss;
- Ifrit4_LR_tuning: notebook in python that contains code to do hypertuning on the dropout value of the last layer and the learning rate value to find the best setting for the IfritNet version 4 model.

there is also a pdf of this report regarding the work done for this project.

7.3 Fire_cls_GUI view

In this section, I will briefly explain the functioning and features of the application by showing a view of it.

A graphical interface was created to make management, training, and testing of the CNNs used in this work easier even for those who are not familiar with programming or TensorFlow. Via the graphical interface (shown in the figure) the user can load the dataset, choose the CNN model, set the parameters, and train and test the CNN. The user can also load a network already trained and ready to use or save the prepared network. The user can also display images from the test set (either generated by the program or given by the user himself) for the CNN to predict, the graphical interface will display both the ground truth and the class predicted by the network. One of the main reasons for implementing to make an application with a graphical interface is to be able to visualize the images that the network will predict and to be able to verify with one's own eyes the images and the results given by CNN.
In my opinion, this has different advantages:

- An inexpert user or only interested in testing the functioning of a CNN will be more involved in this visual aspect, making everything more concrete;
- An expert user can test the network not only in the form of numerical accuracy values (or other metrics) but by seeing for himself the images that are classified well or not, and can also detect a possible trend of correct or incorrect recognitions if present. This could help the analysis and improvement of the network results.

The GUI of the newly opened application can be seen in figure 63. We can see that the GUI is divided into 5 blocks. Now, I briefly describe them starting from top to bottom:

- first: contains the welcome phrase in the application;
- second: contains the interface concerning the management of dataset loading, choice of CNN, including saving and loading an already trained model, and all the parameters for new training of the network;
- third: contains the visualization of the image under consideration at the moment;
- fourth: contains the interface to change the displayed image, display its associated label and have the class predicted from the previously trained CNN model and also the part of loading other test data external to the original dataset;
- fifth: contains the text explaining any errors made by the user while using the application.

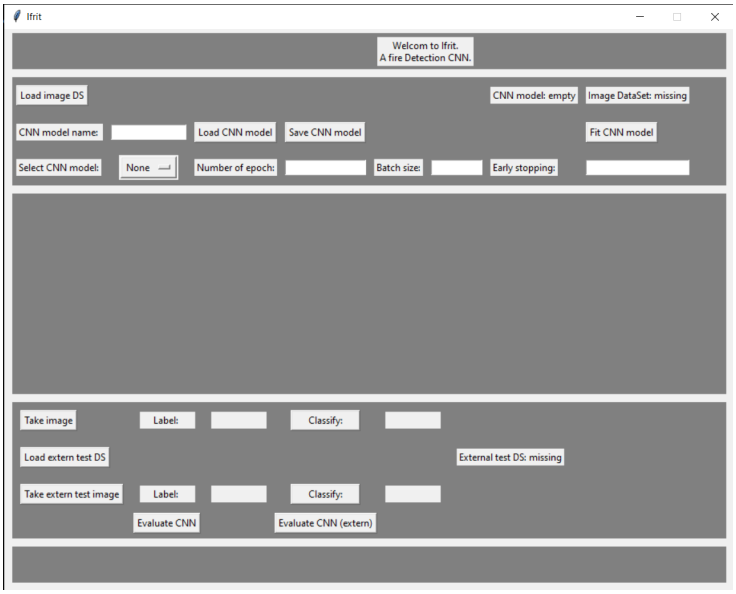


Figure 63: The GUI of the newly opened application.

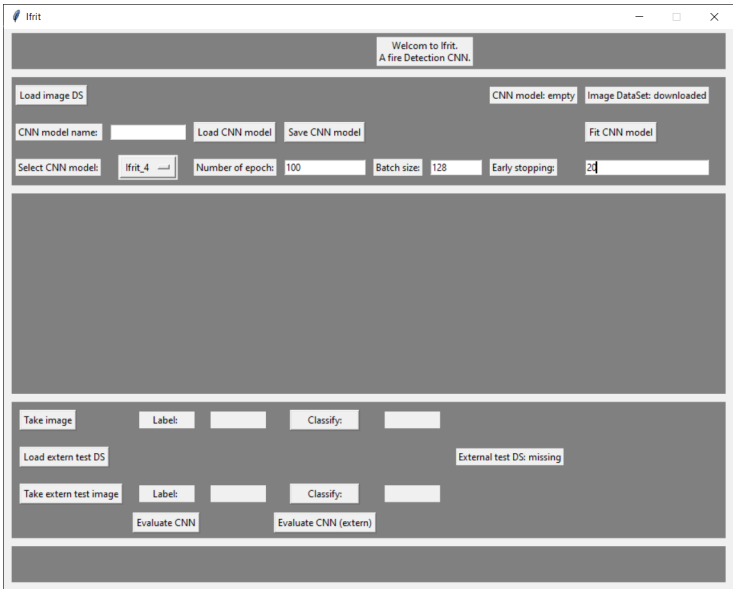


Figure 64: The GUI after the user has loaded the dataset and entered the parameters for training.

The first thing to do is to load the dataset via the button in the upper left corner. When the dataset is loaded it will be shown in the status text at the top right. Once the dataset is loaded, the user can enter the parameters they want for training, as shown in Figure 64, and train the network. For parameters not entered by the user, the application will use default values.

After training the model or uploading it the users will be notified of a ready CNN model via the status line in the upper right corner (next to the one for the dataset), as shown in Figure 65.

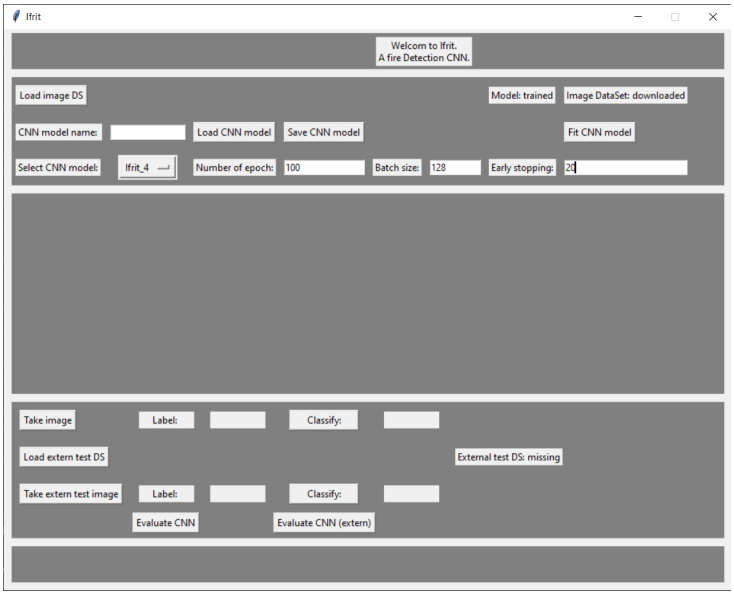


Figure 65: The GUI after the CNN model training.

After training the model the results of training (accuracy and loss function curves for training and validation set), evaluation on the test set, and the associated confusion matrix will be displayed.

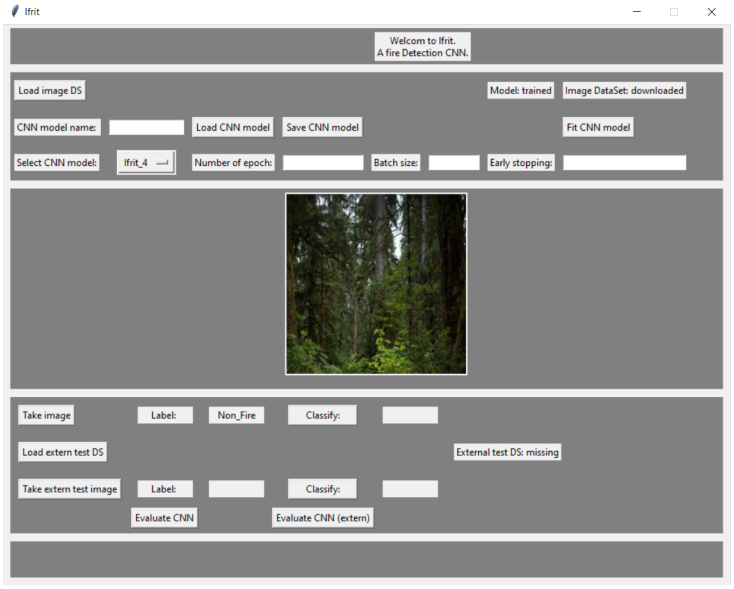


Figure 66: The GUI shows a random image take by the test set.

Afterward, the user can upload a random image from the test set, through the "take image" button, which will be displayed in the center of the GUI, as shown in Figure 66. Also displayed is the label, the class, associated with the image.

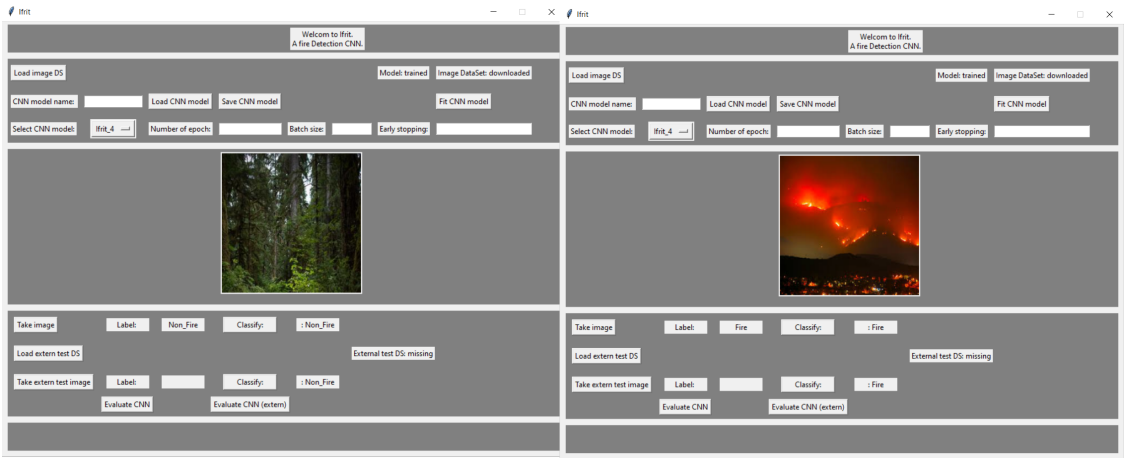


Figure 67: Pair of examples of image classification and label display predicted by the network. On the left is an image belonging to the "Non_fire" class, and on the right is an image belonging to the "Fire" class.

Now the user by pressing on the "classify" button can have the displayed image classified to the trained or loaded CNN model, the label predicted by the model will be displayed in the field to the right of the "classify" button, as shown in figure 67.

Bibliography

[1] *Link to nfpa home page*, https://www.nfpa.org/, Accessed: 2023-05-31.

[2] *Link to nfpa report - fire loss in the united states*, https://www.nfpa.org/News-and-Research/Data-research-and-tools/US-Fire-Problem/Fire-loss-in-the-United-States, Accessed: 2023-05-31.

[3] *Link to 2022–23 australian bushfire season wikipedia page.* https://en.wikipedia.org/wiki/2022%E2%80%9323_Australian_bushfire_season, Accessed: 2023-05-31.

[4] *Link to cnn wikipedia page.* https://en.wikipedia.org/wiki/Convolutional_neural_network, Accessed: 2023-05-31.

[5] R. M. Jakub Gotthans Tomas Gotthans, *Deep convolutional neural network for fire detection*, https://ieeexplore.ieee.org/abstract/document/9092344, Accessed: 2023-06-15.

[6] AbimbolaOO, *Github deepquestai/fire-smoke-dataset*, https://github.com/DeepQuestAI/Fire-Smoke-Dataset, Accessed: 2023-07-05.

[7] jivitesh-sharma, *Github cair/fire-detection-image-dataset*, https://github.com/cair/Fire-Detection-Image-Dataset, Accessed: 2023-07-05.

[8] S. Majid, F. Alenezi, S. Masood, M. Ahmad, E. S. Gündüz, and K. Polat, *Attention based cnn model for fire detection and localization in real-world images*, https://www.sciencedirect.com/science/article/abs/pii/S0957417421014445, Accessed: 2023-06-15.

[9] A. Kumar, *Kaggle fire detection dataset*, https://www.kaggle.com/datasets/atulyakumar98/test-dataset, Accessed: 2023-07-05.

[10] J. Pincott, P. W. Tien, S. Wei, and J. K. Calautit, *Indoor fire detection utilizing computer vision-based strategies*, https://www.sciencedirect.com/science/article/abs/pii/S2352710222011615, Accessed: 2023-06-15.

[11] *Link to tensor cores nvidia's dedicated page*, https://www.nvidia.com/en-us/data-center/tensor-cores/, Accessed: 2023-05-31.

[12] *Link to imagenet challenge.* https://www.image-net.org/challenges/LSVRC/index.php, Accessed: 2023-06-01.

[13] G. E. H. Alex Krizhevsky Ilya Sutskever, *Imagenet classification with deep convolutional neural networks*, https://dl.acm.org/doi/abs/10.1145/3065386, Accessed: 2023-06-01.

[14] C. Szegedy, W. Liu, Y. Jia, *et al.*, *Going deeper with convolutions*, https://arxiv.org/abs/1409.4842, Accessed: 2023-06-01.

[15] *Keras tutorial page for keras tuner*, https://keras.io/keras_tuner/, Accessed: 2023-06-27.

[16] *Project link on github*, `https://github.com/Arzazrel/Project_CIDL`, Accessed: 2023-06-9.