UNIVERSITA' DEGLI STUDI DI MESSINA

Department of Engineering

Engineering and Computer Science

Master's Degree

Fire Detection System

A Machine Learning Project

Students:

Antonino Maio

Ilenia Ficili

Introduction.

The goal of this project was to use machine learning models and techniques in order to build a fire detection system, able to recognize the presence of fire in a picture.

In daily life it is a great implementation that can help prevent dangerous situations or minimize the damage.

To achieve this, a convolutional neural network was used to construct a proper model suitable with the chosen dataset.

Dataset

The dataset that has been chosen is the 'FIRE Dataset', available on Kaggle. It was created during the NASA Space Apps Challenge in 2018.

Data is divided into 2 folders, **fire_images,** is a folder that contains 755 outdoor-fire images some of them with the presence of heavy smoke, while the remaining 244 are nature images labeled as **non-fire_images**.

The images available of variable dimensions, and they all are RGB (channels Red Green and Blue).

The dataset is skewed and for this reason it is important that the test set has an equally-sized number of images per class.

To generate the datased used to train the model is firstly created a main dataframe, called image_df, in which are stored filepaths and relative labels of each image in the dataset. This dataframe is created by calling the **make_dataframe(path)** function which is a custom function that takes the path of the directory containing the dataset as input and returns the head of the dataframe. The function then iterates for each file in a directory and appends the filename to its path. Then, for each **image_file** the filepath is splitted from its label and two Pandas series are created, these two series will be concatenated into the image_df dataframe.

From this main dataframe a subdataframe, test_df, is created sampling 40 instances per category, in order to have a balanced test set, then these instaces are dropped from the main dataframe.

At this point image_df has 919 images and test_df contains the remaining 80. Using the train_test_split utility provided from sklearn the main dataframe is splitted into two dataframes: one used to generate the training set and the other for the validation set. The validation dataframe will contain the 30% of the total images. Data is also shuffled and splitted in a stratified fashion: this means that relative class frequencies is approximately preveserved in the splits. Once the three dataframes have been created it's possible to generate the DataFrameIterators which consist of batches of couples (image,label).

To create the DataFrameIterator are instantiated two ImageDataGenerator: one will be used for the validation and training data, while the other for the test data, the reason why we use two generators is because we're applying a certain preprocessing to the training and validation data, while we want the test data to remain "pure". The preprocessing used for training and validation is a random rotation in a range of 10° and a horizontal flip. However,

every dataset is preprocessed with the MobileNetV2 preprocessing function in order to remap the value range of the pixel in the range [-1,1].

Convnet

In order to achieve the desired result, deep learning techniques for computer vision were used to build up a Convolutional Neural Network.
Such an approach is preferable to models with densely connected layers, especially in computer vision problems, where the second kind of models have some performance issues with large images, since the dense layers involve a huge number of parameters compared with CNNs.
Convolutional neural networks emerged from the study of the brain's visual cortex; research showed that many neurons in the visual cortex have a small local receptive field, meaning that they react only to visual stimuli located in a limited region of the vision and to specific patterns called receptive fields. Moreover, as the visual signal makes its way through brain modules, neurons start assembling the low-level patterns into more complex ones.
The same principle has been used to develop the convolutional architecture, starting from the neocognitron in 1980.
The main building block of a CNN is the **convolutional layer**. In this kind of layer neurons are connected in order to emulate biological neurons in the visual cortex: neurons of a specified layer are connected only to neurons in their receptive fields in the underlying layer.

There are two ways to connect consecutive layers: **padding** and **striding**. Padding is used in order for a layer to have the same height and width of the previous layer, to obtain this result are usually added zeros around the inputs; whereas striding is a technique to reduce the dimension of the output and this is done by skipping columns and rows when overlapping receptive fields.

This architecture allows the network to concentrate itself on low-level features in the first layers, and then evolve them into higher-level features gradually along with the layers. This particular hierarchical structure is common in real-world images, and this is a reason why CNNs work so well in image classification tasks.
Neuron’s weights can be represented as a small image with the size of the receptive field, called **filters** or **convolution kernels**. A convolutional layer is full of neurons that use the same filter, thus the output of this layer will be a **feature map** which highlights the areas in the picture that activate the filter the most. During training the model will automatically learn

which are the most useful kernels for the specified problem and the above layers will learn how to combine the low-level features into more abstract concepts that could be useful for the task.

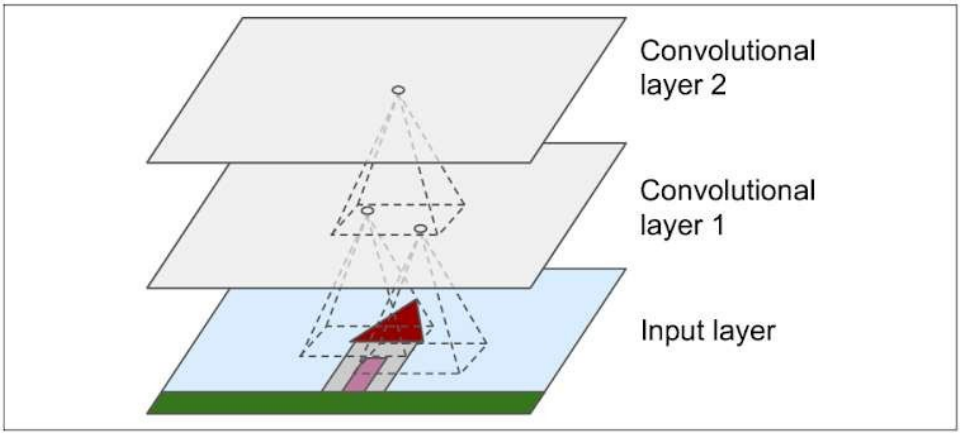
In a convolutional layer there are multiple filters and multiple outputs, each representing a feature map. So each convolutional layer simultaneously applies to its inputs multiple filters in order to make it capable of recognizing different features. Input images, as said before, are RGB images, meaning that they have 3 color channels, so feature maps of a convolutional layer are applied to each channel.

The algorithm used to train weight in CNNs is the **Backpropagation**. It involves taking the error rate of a forward propagation and feeding this loss backward through the layers to fine-tune the weights. Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization. In the Backpropagation algorithm the chain rule is used to compute the gradient of the network in order to tune each neuron weight. The algorithm is composed of two phases: **Forward pass** and **Backward pass**.

In the forward pass all the values associated with the “inputs” of the neural network are propagated to all nodes from top to bottom, until the loss value is calculated.

In the backward pass the algorithm will go through the computational graph from bottom to top and compute the contribution that each parameter had in the final loss value, in order to minimize the loss with respect to the input values by computing the gradient using the chain rule.

Another important layer when building CNNs is the pooling layer whose goal is to subsample the feature map in order to reduce the computational load, memory usage and number of parameters (thereby reducing the risk of overfitting). As in a convolutional layer, each neuron of this layer is connected to a limited number of neurons of the previous one. Neurons of the pooling layer include every parameter that concerns the convolutional layer except weights, in fact it aggregates its inputs using a function such as the max or the mean value.
Another good reason to use the Pooling layer is that it introduces some level of invariance to small translations. Downside of this kind of layer is that it is very destructive, since a 2x2 window this stride 2 will drop about 75% of the input values.



Last kind of layer that is going to be used in the model is the Dropout layer. This layer randomly sets input units to 0, with a specified frequency. This is done only during training in order to prevent overfitting.

A common and highly effective method when approaching a deep learning task is to use a **pretrained model**, that is a model previously trained on a large scale image classification task. Given the vast compute and time resources required to develop a neural network, starting from one already trained on a larger problem could be timesaving. If the original dataset is large and general enough, the spatial hierarchy of features learned by the base model can effectively act as a generic model for the visual world, and hence its features can be useful for many computer vision problems. The main usage of a pretrained model is as a **feature extractor**, in order to extract interesting features from the samples used in the specific task. To use a pretrained model as feature extractor firstly it's important to remove the pretrained classifier since this is the densely connected part of the model which is very specific. Then the **convolutional base** and all the representations learned by it are reused as feature extractor.

As every convolutional neural network, even in the convolutional base the level of generality of representations depends on the depth of the layer in the model. To make sure that the convolutional base performs well as a feature extractor is important to freeze all weights of this model, otherwise the already learned representations would be modified and the performances of the total model would be much worse.

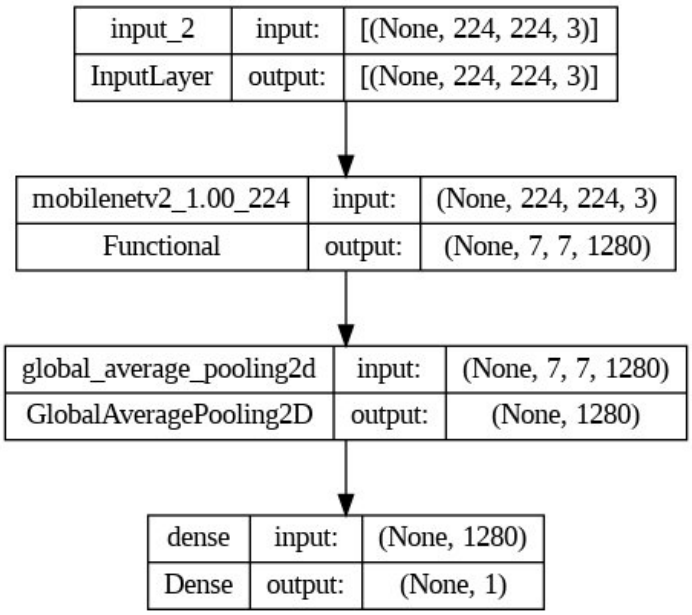
Lastly, to improve performances it's common to **fine-tune** the model, this is a technique which consists of unfreezing a few of the top layers base model and then jointly training both the densely connected classifier with these top layers.

This process slightly adjusts the more abstract representations in the feature extractor, making them more relevant for the specific problem at hand.

The model that has been built to perform the fire recognition task designed at the start of the project is a Binary classifier, which uses the MobileNetV2 model as feature extractor. This model has been chosen for its accuracy on Top-5 Classes of the ImageNet dataset (about 90%) and the time spent for inference step (3.8 ms) which makes this model very fast to compute predictions.

The base_model layers are also frozen to avoid that the feature extractor modifies its already learned representation, as has already been described in the Transfer Learning paragraph.
Then the **build_model()** function is defined where the model used to solve the task is created via functional API: the inputs of the model are processed first by the feature extractor, and then its outputs are flattened by the GlobalAveragePooling and at the end of our model there is a densely connected layer containing a single neuron, this is done because we are dealing with a binary classification so the probabilities of the elementbelonging to the two opposite classes are complementary, this is possible thanks to the useof the sigmoid as activation function.

It will be predicted, not the class, but the probability (p) of the instance to belong to a class, in this case the positive one. Of course, the remaining (1-p) is the probability to belong to the negative class, and their sum p + (1-p) is going to be 1, accordingly with the probability theory.
This behavior is due to the **sigmoid** activation function that bounds the output to the range [0,1]. Once the model has estimated the probability it is important to set a threshold over which the instance can be considered as belonging to class 1, this is done by rounding the probability to the closest integer, then the threshold is 0.5. In the following plot is represented the model used for this project:



The model defined above is built by calling the build_model() function and initially trained for 20 epochs. Before training it's important to compile the model, using the compile method, and specifying the optimizer used to update weights, the loss function used to compute the loss value and the metrics used to evaluate the performances.
The optimizer that has been chosen is the Adam optimizer, which is an algorithm that implements the stochastic gradient descent based on the adaptive estimation of first and second order moments.
The **loss function** used is the binary_crossentropy. which must be convex in order to compute the gradient descent and obtain the global minimum.
This uses the log function to penalize the prediction if it is far from the actual label.

$$J(\partial) = -\frac{1}{m}\sum_{i=1}^{m}[\log(h_6(x^i)) * y^i + \log(1 - h_6(x^i)) * (1 - y^i)]$$

The metrics used to evaluate performances of the model on both training and validation sets are **Accuracy, Precision and Recall**.

Accuracy is a metric used to compute the ratio between the correct and total predictions. This metric is mostly used in case of balanced classes, but it can be useful even in a skewed dataset like this one, since the test set is balanced. The problem about heavily skewed data is that accuracy is not reliable at all to verify the performance of the model.

$$Accuracy = \frac{\#number\ of\ correct\ predictions\ \#all\ predictions}$$

But in this case, we can also use precision and recall since these metrics are more meaningful.
Precision is defined as the ratio between the true positive and the total positive predictions. This metric is used to evaluate how good the model is when predicting positive class.

$$Precision = \frac{\#number\ of\ true\ positive}{\#\ true\ positive + false\ positive}$$

Precision is usually used along with recall, also called true positive rate (TPR), which is the ratio of true positive correctly detected by the classifier.

$$Recall = \frac{\#number\ of\ true\ positive\ \#true\ positive + false\ negative}$$

This metric highlights the ability of the algorithm to get some positive predictions, even if there are few examples.

Precision and Recall can be combined into F1-Score which is the harmonic mean between these two metrics, giving more weight to low values.

$$F1 - SCORE = 2 * \frac{Precison * Recall}{Precision + Recall}$$

Once the model is compiled the fit method is called to train the model and among its parameters in addition to training and validation dataset two more parameters are specified: epochs which clearly corresponds to the number of epochs for which we want to train the model and callbacks, which are object that can perform various actions during the training loop. In this case have been used two callbacks' classes: EarlyStopping and ModelCheckpoint.

ModelCheckpoint is a class which permits to save the model with some frequency: in this case it has been used to save only the best model, respect to the "val_loss" value in the training loop. The model is then saved in a file called fire_%d_%m_%Y_%H_%M_%S.keras. The name has in it the date and hour of the training in order to recognize different models used during the development phase.

EarlyStopping callback is used to stop the training and restore the best weights if the model stops improving its val_loss for 10 epochs.

After the initial training the value of the loss on the validation dataset is 0.07188, which is a good value, but it's possible to make the model even better.

In order to get the most out of the model the weights of the top layers of the feature extractor are unfrozen and the dense classifier is jointly trained with these layers for another 10 epochs. This is done to adapt the already learned representations of the feature extractor to this particular task and improve the performance of the model.

From the base_model are unfrozen only 11 layers, which is clear by looking at the model summary that it corresponds to the last block of the 16th convolutional blocks.

The reason why only few layers are unfrozen is because we want to tune the already learned representations and not change them completely.
The model is then recompiled with the same parameters as before except the optimizer's learning rate which is reduced from the default value (0.001) to the value of 1e-8.
After the fine-tuning the model keeps a loss_value similar to the best value achieved in the initial training, but its representation should be more specific about the problem at hand.

Since the dataset is not only skewed but also small (1k total images) there may be the risk of overfit or underfit, or even a high variance on the validation cost. This would prevent us from reliably evaluating the model.

To successfully evaluate the model then is used the **K-fold cross validation** which consists of splitting all the available training data into K partitions, instantiating K identical models and then training each model on k-1 folds, using the remaining one to compute the validation cost. At the end of each training phase the metrics are maintained while the model is discarded. This process is repeated until all the splits have been used for both training and validation. At each iteration the best model respect to the loss_value on the training split is saved and its metrics are computed and stored into the **results** list.

To assess the model even better it has been used the **Stratified KFold cross validation** which is an extension of the already described KFold cross validation where rather than splitting completely random the data, the ratio between the target classes in each fold are the same as in the full dataset.

The Stratified KFold is done on 5 models, so the dataset is splitted into 5 folds, 4 of them are used for the training and 1 for the validation. Training and validation DataFrameIterator in each iteration are generated starting from the image_df dataframe and using the indexes provided by the StratifiedKFold class instantiated right before the for loop. The

stratified fashion is then verified by printing the **value_counts()** method of each dataframe in order to verify if the class frequency is maintained among the two splits.

It is possible to look at the mean loss of all the models validated on different folds, and also the related metrics. The mean loss is lower than the one obtained on the used model, this is because the results are averaged among five models.

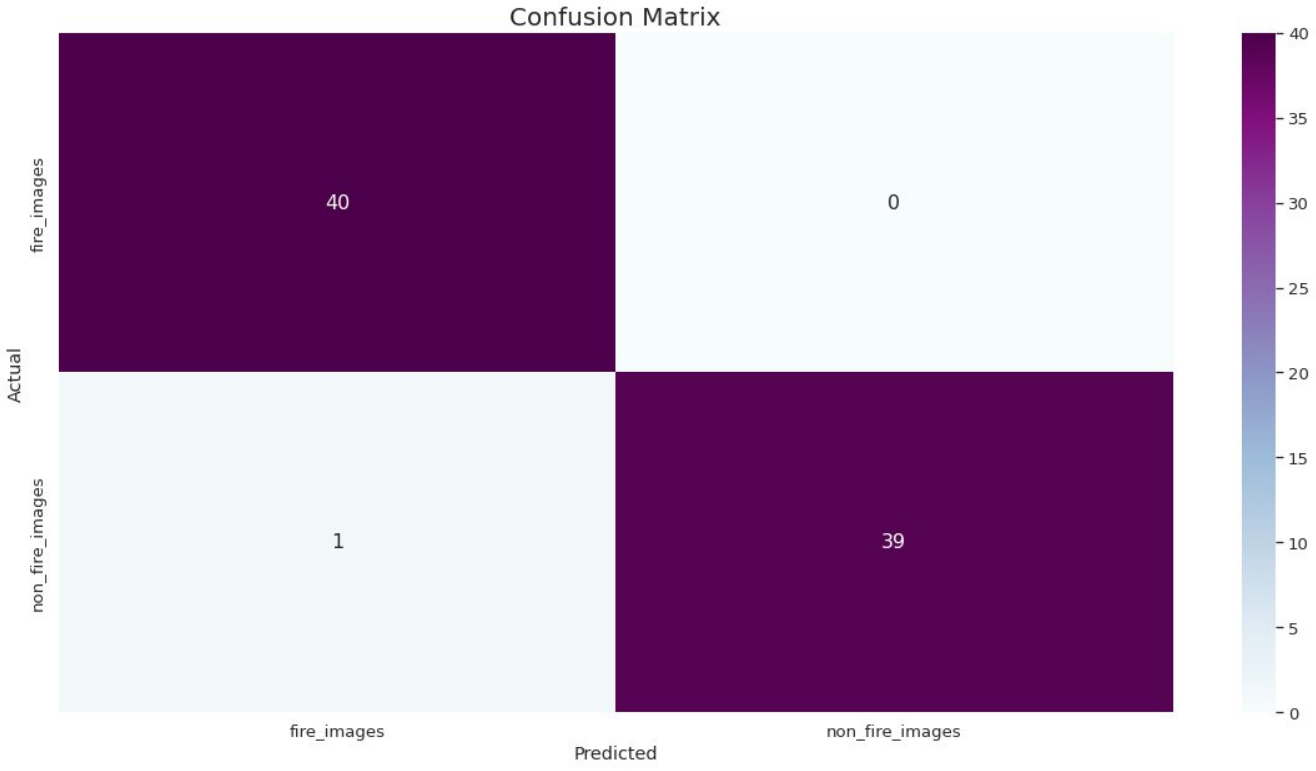
```
Mean Loss among 5 folds =  0.06430950239300728
Mean Precision among 5 folds =  0.9775757551193237
Mean Recall among 5 folds =  0.9383574962615967
Mean Accuracy among 5 folds =  0.979318130016327
```

METRICS EVALUATION

Once the model is trained it is evaluated the performance on the test dataset, that is created from the test dataframe previously described.

To evaluate the model all the metrics are computed and to have visual feedback is also plotted the confusion matrix that is a table layout in which the number of correct and incorrect predictions are summarized with count values and broken down by each class. It gives insight not only into the errors being made by the classifier but more importantly the types of errors that are being made.

Analyzing the Confusion matrix of the fine-tuned model, it can be noticed that only one of the 80 instances present on the test set is mislabeled.



By printing the classification report of the model, reported in the following graph, is possible to notice the high performance of the model itself.

	precision	recall	f1-score	support
0	0.98	1.00	0.99	40
1	1.00	0.97	0.99	40
accuracy			0.99	80
macro avg	0.99	0.99	0.99	80
weighted avg	0.99	0.99	0.99	80