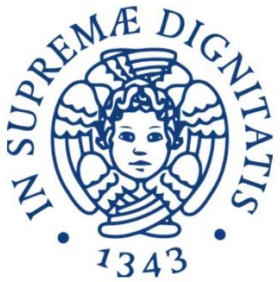
University of Pisa



Computational Intelligence and Deep Learning

Image classification

developing of a CNN from scratch, a pre-trained CNN and comparing the results

Student

Davide Vigna

Summary

Introduction

The purpose of this project is to perform a classification of different scenarios, in a home surveillance context using deep learning knowledge and techniques learned in this course.

This task arises from the need to find a solution to another past project that consisted in the realization of a low-cost video surveillance for a specific house using Raspberry Pi and a webcam. The free software used to remote monitoring and to record videos is called *motion*. It allows to send emails each time a movement is detected. It is used in a specific context, a house where 3 subjects live: a woman, a dog and a man. The main problem of this software is that it is very sensitive to changes in images. It sends emails even if a non-real movement happens e.g. (changing of lights). In addition, it would be necessary to detect all the possible combinations of the 3 subjects and an additional situation where a group of people (at least 3) is present in the detection.

For these reasons, the idea it to use a Convolutional Neural Network (CNN) in order to classify the detected movement each time it happens. This classical multi-classification task helps to define different levels of alarm depending on situation. The model obtained from the training of CNN will be used by an external software to handle the alarm level. This software is not a part of this project.

In this project some CNN are trained from scratch and others using a pre-trained network. The aim of this report is to analyse the various results of the experiments obtained during the implementation phase and to motivate the choices adopted for the selection of models.

Finally, a little insight is made on the branch of object detection in order to analyse what solutions are currently available. A simple implementation is proposed with the aim of understanding how object detection works and giving the possibility in the future to extend this project for a further customized implementation.

Motivation

The choice to use a CNN for the development of the task is due to the benefits that this kind of network offers respect to other deep learning solutions. First of all, these networks offer a level of invariance respect to little distortions, rotations and other kinds of noise in images. This allows to recognize a particular feature in different positions inside the image, preserving the spatial structure. In addition, it requires less parameters to train then fully connected networks, thanks to the fact of sharing weights. So, it's possible to train more difficult and powerful models in less time.

In this specific case, the CNN helps to overcome some limitations of the software *motion* like recognizing a situation where none is effectively in the room, so it does not send any email alarm (false-alarm).

This kind of networks are wildly used and suited for object detection. Object detection is one of the fundamental problems of computer vision. It forms the basis of many other downstream computer vision tasks, for example, instance segmentation, image captioning, object tracking, and more. For this project the *person detection* is useful. It is a variant of object detection used to detect a primary class "person" in images or video frames. Detecting people in video streams is an important task in modern video surveillance systems. [1]

Dataset composition

The dataset used for this project consists of a set of images collected using a *Raspberry Pi 3 Model B+* linked to a *Logitech Webcam C210*. The configuration of the software *motion* on Raspberry is set to take a picture each time a motion is detected and saved it on local memory. To get enough data, the software was run for several days in different situations, then a manual labelling has been done to all the pictures retrieved.

The final composition of the initial dataset is 4500 images (640x480), about 1GB, containing 9 possible scenarios (each one composed of 500 pictures) :

1. None is present
2. A man is present
3. A woman is present
4. A dog is present
5. A man and a woman are present
6. A man and a dog are present
7. A woman and a dog are present
8. A man, a woman and a dog are present
9. People is present (at least 3)

This base dataset has been chosen as a starting point to make experimentation on CNNs and considering the fact that it is possible to collect more data as last solution after applying all the deep learning techniques and strategies studied, if a no satisfying accuracy is achieved or if an overfitting situation is present. Of course, this has no cost in this case, just spending more time on it but in other contexts it could be more complicated and costly. Here, there are some examples of the 9 classes.

Figure 1 None

Figure 2 Man

Figure 3 Woman

Figure 4 Dog

Figure 5 Man-Woman

Figure 6 Man-Dog

Figure 7 Woman-Dog

Figure 8 Man-Woman-Dog

Figure 9 People

Related works

In the field of video surveillance, several applications stand out that can benefit from deep learning. Person detection and object detection is another area where deep learning has shown tremendous progress. For example, over the past five years, the IMAGENET database has organised the "large scale visual recognition challenge" in which image software algorithms are challenged to detect, classify and localise a database of over 150,000 photographs collected from Flickr and other search engines. The dataset is labelled into 1,000 object categories. Many deep learning systems are trained with over 1.2 million images from the IMAGENET dataset running on GPU based hardware accelerators. The improvements in accuracy range from 72% to over 90% from 2010 thru 2014. In 2015, all IMAGENET contestants used deep learning techniques. [2] [3] [4]

A key advantage of deep learning-based algorithms over legacy computer vision algorithms is that deep learning system can be continuously trained and improved with better and more datasets. Many applications have shown that deep learning systems can "learn" to achieve 99.9% accuracy for certain tasks, in contrast to rigid computer algorithms where it is very difficult to improve a system past 95% accuracy. The second advantage with deep learning system is the "abnormal" event detection. Deep learning systems have shown remarkable ability to detect undefined or unexpected events. This feature has the true potential of significantly reducing false-positive detection events that plague many security video analytics systems. In fact, the inability to reduce false-positive detection rate is the key problem in video surveillance industry; and has to-date prevented the wide scale acceptance of many vendors' intelligent video analytics solutions.

Another interesting work has been done for suspicious activity detection from surveillance video in the project [5]. The aim of the project is monitoring the suspicious activities in a campus using CCTV footages and alerts the security when any suspicious event occurs. This was done by extracting features from the frames using CNN. After the extraction was done, LSTM architecture is used to classify the frames as suspicious or normal class. The system classifies the videos as suspicious (students using mobile phone, fighting, fainting) or normal (walking, running). In the case of suspicious behaviours, an SMS (Short Message Service) will be sent to the respective authority [6]. The input videos are taken from CAVIAR dataset, KTH dataset, YouTube videos and videos taken from campus. Around 300 videos of different suspicious and normal behaviour videos are collected. The accuracy of the training phase is 76% for the initial 10 epochs. The accuracy of that model can be improved by increasing the number of iterations. The accuracy achieved is 87.15%.

A more sophisticated, but similar task has been done recently in another project that consist in a real-time violence detection in crime scene intelligent video surveillance systems. The aim is to collect the real time crime scene video of surveillance system and extract the features using spatio temporal (ST) technique with Deep Reinforcement neural network (DRNN) based classification technique. The input video has been processed and converted as video frames and from the video frames the features has been extracted and classified. Its purpose is to detect signals of hostility and violence in real time, allowing abnormalities to be distinguished from typical patterns. They used for testing a large-scale UCF Crime anomaly dataset. The experimental results reveal that the suggested technique performs well in real-time datasets, with accuracy of 98%, precision of 96%, recall of 80%t, and F-1 score of 78%. [7]

Data pre-processing

The data pre-processing phase is the most challenging and time-consuming part of data science, but it's also one of the most important parts. If the dataset is not prepared correctly, it could compromise the model and all the further analysis and considerations on it.

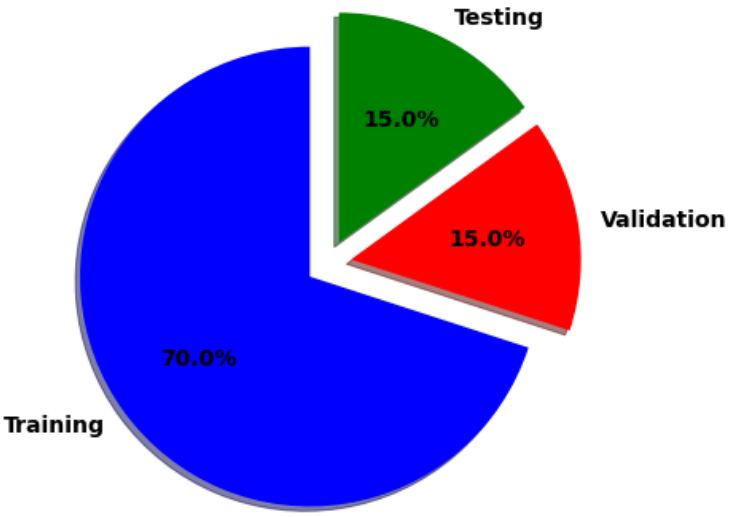
A first manual pre-processing *cleaning* phase has been done on all the collected images. All the cases in which it was not possible to distinguish a scenario from another due to distortion have been discarded in order to rise the quality of the dataset to be processed. The result of this elaboration is stored in a single folder on the Google Cloud. There wasn't the necessity to apply any transformation on the images dimensions because they all come from the same source *webcam*, so they can be correctly converted in tensors of the same size. This choice avoids to change the resolution of the image and allows also to recognize better, the presence of subjects that are not so near to the webcam going to deeper detail.

A greyscale transformation is adopted because the RGB colours does not provide a meaningful contribution to the models. The subjects that could pass in front of the webcam could have different dresses and colours and the network should be able to classify the different situation independently of what they dress in a specific moment. In this way the computation effort is reduced and a lighter model (fewer parameters and faster to train) could be obtained.

A rescaling technique is also used in order to have input values in the range [0,1]. This pre-processing operation has been used as first layer of all the models built. In this way there is no need to repeat the same rescaling operation on test data, when the model will be effectively used.

As last step, the images have been labelled and grouped together in the same subfolders according to their label. Therefore, a rebalancing operation has been done in order to have an equal number of samples per class.

For all the experiments the *hold-out validation* method is used because the dataset has been considered large enough for this task and the validation set, quite representative of the actual distribution. Images have been randomly selected from the initial dataset and the 70% of them have been used for training, the 15% for validation and the 15% to do tests.



Following this procedure, there is the guarantee to work on the same sets of data in all the experiments. These new splits have been saved in a different directory maintaining the same level of subdirectories, preserving the previous labelling action.

Before starting to work with the prepared dataset, the Python API command is used:

```
image_dataset_from_directory
```

This command returns a dataset of batch images specifying the path source and the size of the images. It allows to do different operations with a very low coding effort fixing particular parameters, like the greyscale operations cited above. This allows to retrieve in a very fast way the pre-processed dataset before feeding it to the network. It is used in all the experiments with different sets of parameters.

Methods and experiments

In this section are described all the experiments and the results obtained during the implementation of a CNN (from scratch and using a pre-trained network), analysing the critical aspects and situations encountered.

The approach adopted is the following: at first, a simple model is built and trained on the training set; then the validation set is used to see generalization capabilities and analysing underfitting /overfitting situations; finally, the test set is used to see the accuracy score on the unseen data. If an underfitting situation occurs, the model is gradually increased in complexity in order to have a higher accuracy. If an overfitting situation occurs, the model is regularized with some techniques in order to reduce this phenomenon. Before building each model, a proper set of hyperparameters is chosen and motivated according to the desired result to be obtained.

At the end of each experiment, a final comment about the goodness of the obtained result is given in order to highlight its characteristic. In addition, a bar char plot of miss-classification error is attached in order to understand in what classes error occurs and if they are common in all the networks trained.

CNN from scratch – common choices

One of the first choices made on the hyperparameters is the *batch size*. In all the experiments a mini-batch learning approach is used considering a batch size equal to 32. This number comes from several attempts where a too low value required too much training time, instead a high value (near to the number of trainset samples) required too much memory, so an intermediate solution is chosen. Respect to the batch learning, this approach has the benefit of reducing the risk of getting stuck at a local minimum, since different batches will be considered at each iteration, granting a robust convergence. Another advantage is that there is no need to store the errors for the whole dataset in the memory, but there is still the need to accumulate the sample errors to update the gradient after all mini-batches are evaluated.

Another important choice regards the number of *epochs* to use. By definition, an epoch is a one cycle through the full training dataset in presented to the network. When the number of epochs used to train a neural network model is more than necessary the overfitting may occur, so typically the model gives high accuracy on the training set but fails to achieve good accuracy on the validation set. This happens also in a specular way considering the loss (low loss on trainset and high loss on validation set). The number of max epochs used is 50 but in order to monitoring the trade of loss on validation set an *early stopping callback* is used choosing a value of patience equal to 5. The loss is chosen as monitoring metric to be minimized during the training procedure. By definition, accuracy score is the number of correct predictions obtained, instead loss values are the values indicating the difference from the desired target state. In this way, if the neural network model doesn't improve its loss on validation set until 5 epochs the training stops. In addition, a specific parameter is also set inside the callback function in order to restore model weights from the epoch with the best value of the monitored quantity.

Instead, for the evaluation of model performance, the *accuracy metric* is used. Accuracy is a valid choice of evaluation for classification problems which are well balanced and not skewed.

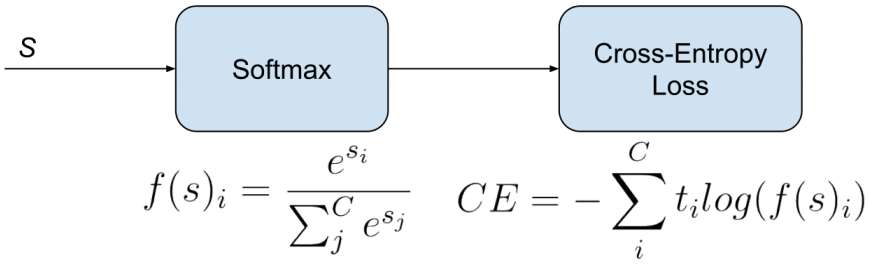
In this project, the dataset is balanced, each class has the same number of samples (see pre-processing section).

The *loss function* to be minimized in all the experiments is the *cross entropy* because it is a very good cost function and it is the most used in multi-classification problems. It is defined as:

$$CE = -\sum_{i}^{C} t_i log(s_i)$$

where t_i and s_i are the ground truth and the CNN score for each class i in C. As usually an activation function (Sigmoid / Softmax) is applied to the scores before the CE Loss computation. In this project the last layer has always a Softmax activation function with 9 neurons (the number of classes to be recognized). This is also called *Categorical Cross-Entropy loss* and the CNN will output a probability over the C classes for each image.

It can be summarised in the following schema [8] :



Different *optimizers* have been taken in consideration. An optimizer is an algorithm used to minimize an error function(loss function). In this project three optimizers have been considered, all belonging to the family of adaptive optimizers. This category has been introduced to solve the issues of the gradient descent's algorithms. Their most important feature is that they don't require a manually tuning of the learning rate value [9].

The three selected are:

- AdaGrad(Adaptive Gradient Descent) adapts the learning rate to the parameters performing small updates for frequently occurring features and large updates for the rarest ones. In this way, the network is able to capture information belonging to features that are not frequent, putting them in evidence and giving them the right weight. The problem of AdaGrad is that it adjusts the learning rate for each parameter according to all the past gradients. So, the possibility of having a very small learning rate after a high number of steps is relevant. If the learning rate is too much small, the weights cannot be updated and the consequence is that the network doesn't learn anymore.

- RMS-Prop (Root Mean Square Propagation) is a special version of AdaGrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients. RMS-Prop basically combines momentum with AdaGrad. The main advantage is the learning rate gets adjusted automatically and it chooses a different learning rate for each parameter. The disadvantage is that it is characterized by slow learning.

- Adam(Adaptive Moment Estimation) is one of the most popular and famous gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients , similar to momentum and also the decaying average of the past squared gradients , similar to RMS-Prop and AdaGrad. Thus, it combines the advantages of both the methods. [10]

Although they adjust automatically the *learning rate*, they give the possibility to choose an initial value. This parameter tells the optimizer how far to move the weights in the direction opposite of the gradient for a mini-batch. If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. If the learning rate is high, then training may not converge or even diverge. The training should start from a relatively large learning rate because, in the beginning, random weights are far from optimal, and then the learning rate can decrease during training to allow more fine-grained weight updates.

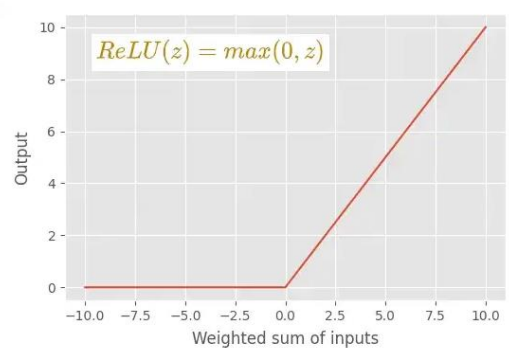
Multiple *activation functions* have been considered for training the network. The choice of activation function has a large impact on the capability and performance of the neural network, and different activation functions may be used in different parts of the model. For this reason, three of them have been analysed in details:

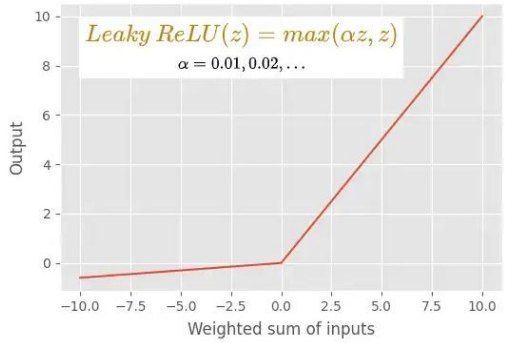
- ReLU (rectified linear unit), this function has several benefits. It does not have the vanishing gradient problem; it is computationally inexpensive and it has a faster convergence respect sigmoid and tanh functions. Its main characteristic is that, if the input value is 0 or greater than 0, the ReLU function outputs the input as it is. If the input is less than 0, the ReLU function outputs the value 0. It is typically used for hidden layer in MLP and CNN models. However, it has two drawbacks: the dying ReLU problem (a neuron can die because it is only activated by negative values and this function has a non-negative codomain $[0,+\infty]$), and the exploding gradient problem. The last one happens when large error gradients accumulate and result in very large updates to neural network model weights during training. When the magnitudes of the gradients accumulate, an unstable network is likely to occur, which can cause poor predication results or even a model that reports nothing useful.

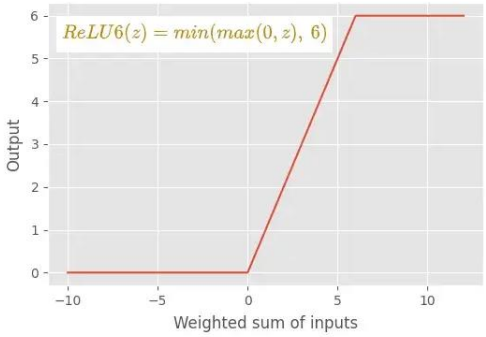
- Leaky ReLU, it is a modified version of the default ReLU function in which the dying ReLU problem is solved. In leaky ReLU, negative values are multiplied by small alpha and are not actually 0. This one is preferred when there are lots of negative activations and ReLU stops learning process.

- ReLU6, this is another variant of classical ReLU that solves the exploding gradient problem. It restricts to the value 6 on the positive side. Any input value which is 6 or greater than 6 will be restricted to the value 6 (hence the name). In comparison with classical ReLU, ReLU6 activation functions have shown to empirically perform better under low-precision conditions (e.g., fixed point inference) by encouraging the model to learn sparse features earlier. [11] [12] [13]

The next page shows a graphical representation of these three activation functions.







The following table contains a quick recap of the common set of hyperparameter and their values used in the experimentations.

BATCH SIZE	32
EPOCHS	50
EARLY STOPPING	Monitoring loss Patience = 5, Restore_best_weights=True save_best_only = True
EVALUATION METRIC	Accuracy
LOSS FUNCTION	Categorical Cross Entropy
OPTIMIZER	{AdaGrad, RMS-Prop, Adam}
ACTIVATION FUNCTIONS	{ReLu,Leaky ReLu, ReLu6}

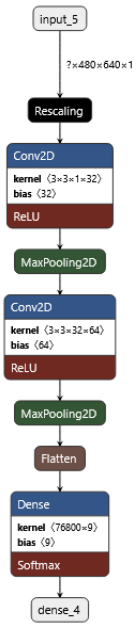
CNN from scratch – simple model

In this first experiment a simple model is used in order to do some observations and to have an idea of the behaviour of different optimizers during the training process.

The model used has the following structure:

```
Conv2D(filters=32, kernel_size=3, activation="relu",padding="same")
MaxPooling2D(pool_size=4)
Conv2D(filters=64, kernel_size=3, activation="relu",padding="same")
MaxPooling2D(pool_size=4)
Flatten()
Dense(9, activation='softmax')
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 480, 640, 1)]     0

 rescaling (Rescaling)       (None, 480, 640, 1)       0

 conv2d (Conv2D)             (None, 480, 640, 32)      320

 max_pooling2d (MaxPooling2D  (None, 120, 160, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 120, 160, 64)      18496

 max_pooling2d_1 (MaxPooling  (None, 30, 40, 64)       0
 2D)

 flatten (Flatten)           (None, 76800)             0

 dense (Dense)               (None, 9)                 691209

=================================================================
Total params: 710,025
Trainable params: 710,025
Non-trainable params: 0
```

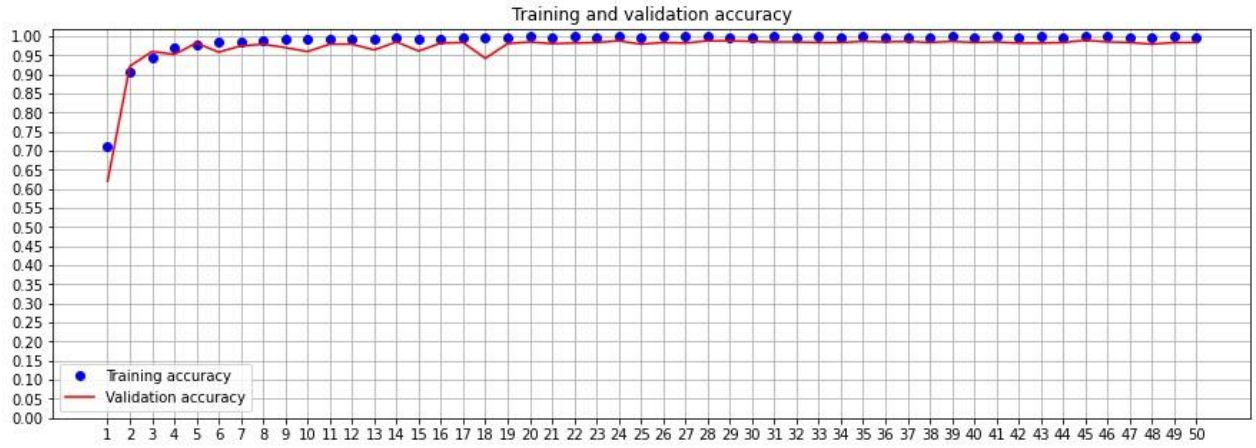


The model consists of two convolutional layers each one follwed by a max pooling layer with a pool size of 4. Next, it is present a flatten layer in which the multi-dimensional arrays are converted into a 1-D array of fully connected layers. The structure of this simple model is hinspired from the initial part of VGG16 architecture. The number of filters is chosen empirically. Both the convolutinal layers use the ReLU as activation function. In order to observe the different beahviour of the larning strategies, this model is trained using the three different optimizers cited above and the early stopping technique is not used initially.

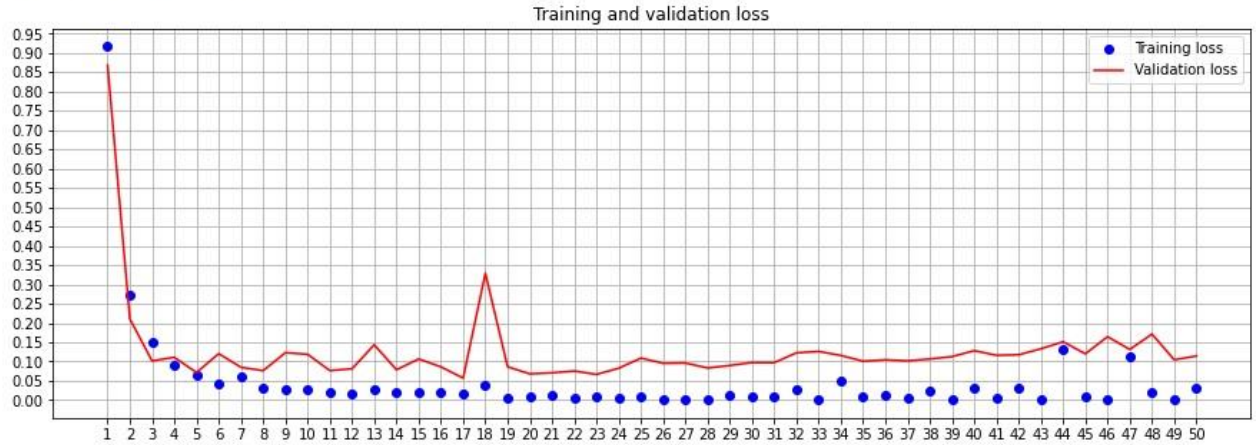
The first optimizer analized is the RMS-Prop, using the default initial learning rate (0.001) . The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
17	0.9971	0.9837	0.0171	0.0572

Test Accuracy	Test loss
0.9837	0.0797



<Figure size 432x288 with 0 Axes>



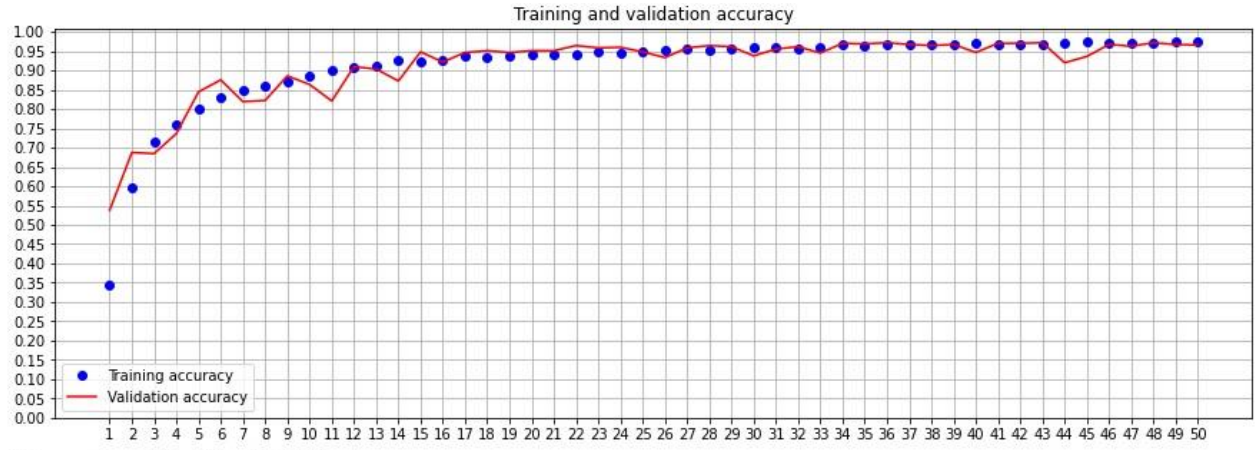
It has a very fast convergence and the best result is obtained at iteration 17 in which the lowest value of loss is achieved. From that point on, the loss gradually tends to rise. The reason why early stopping it is no used in these firsts experiments is because it wouldn't be possible to observe the full behaviour of the training in all the epochs.

The phenomenon of overfitting seems to be quite limited: training and validation accuracy are very similar to each other, same for the loss. The accuracy on test set is about 98%, that is a very good result. The next section is dedicated to try to improve these results.

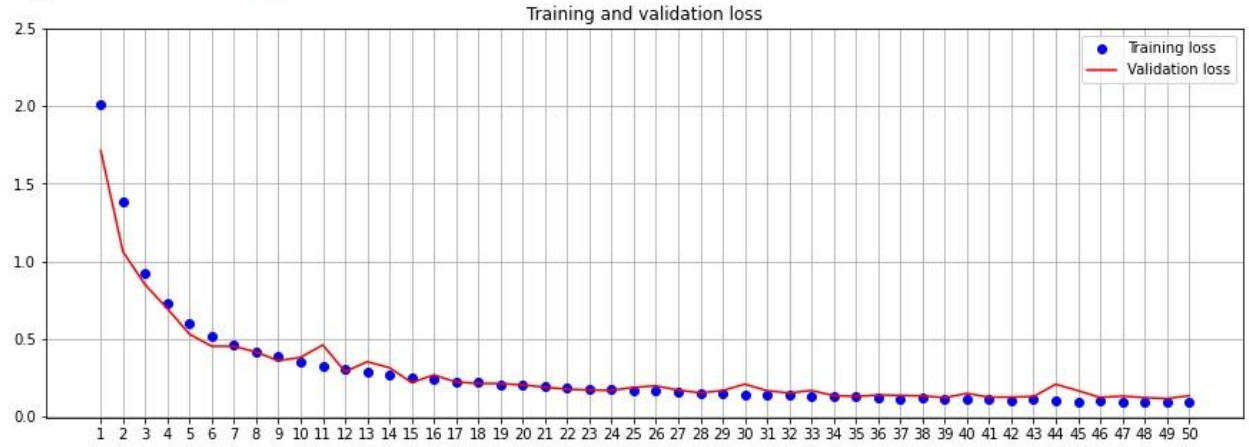
The second optimizer analized is the AdaGrad, using the default initial learning rate 0.001. The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
49	0.9740	0.9674	0.0942	0.1156

Test Accuracy	Test loss
0.9541	0.1469



<Figure size 432x288 with 0 Axes>



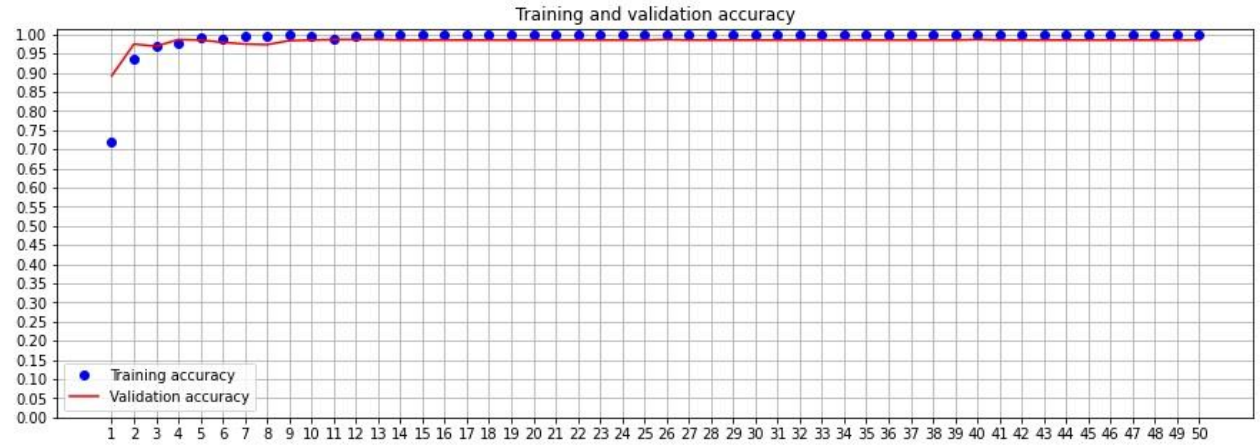
Differently from the previous one, it has a very slow convergence because AdaGrad keeps track of the sum of gradient squared and uses that to adapt the gradient in different directions. This is useful when features are very sparse. The average gradient for sparse features is usually small so such features get trained at a much slower rate. The loss graphic has lower oscillations respect to RMS-Prop.

In this experiment, the choice of using 50 epochs for training is maybe too low. The best result is obtained at iteration 49. It seems that the performance could be improved more if it would have more epochs available for training (probably a new local minimum is not reached). In fact, the performances are slightly worse than in the previous experiment but still acceptable considering the circumstances. The accuracy on test set is about 95% and the loss is about 0.15.

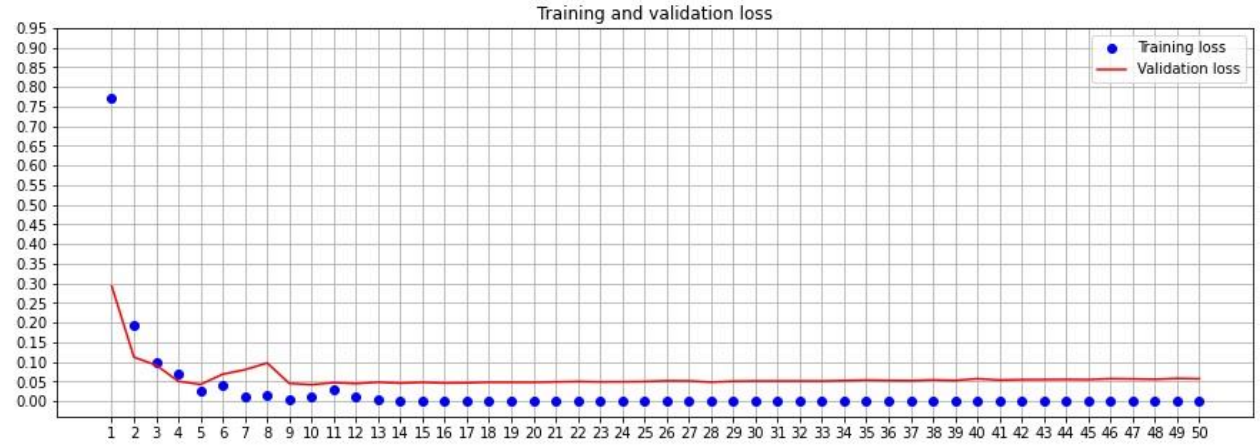
The third optimizer analized is the Adam, choosing the default values of parameters alpha, beta1, beta2 and epsilon. The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
10	0.9965	0.9852	0.0115	0.0423

Test Accuracy	Test loss
0.9778	0.0725



<Figure size 432x288 with 0 Axes>



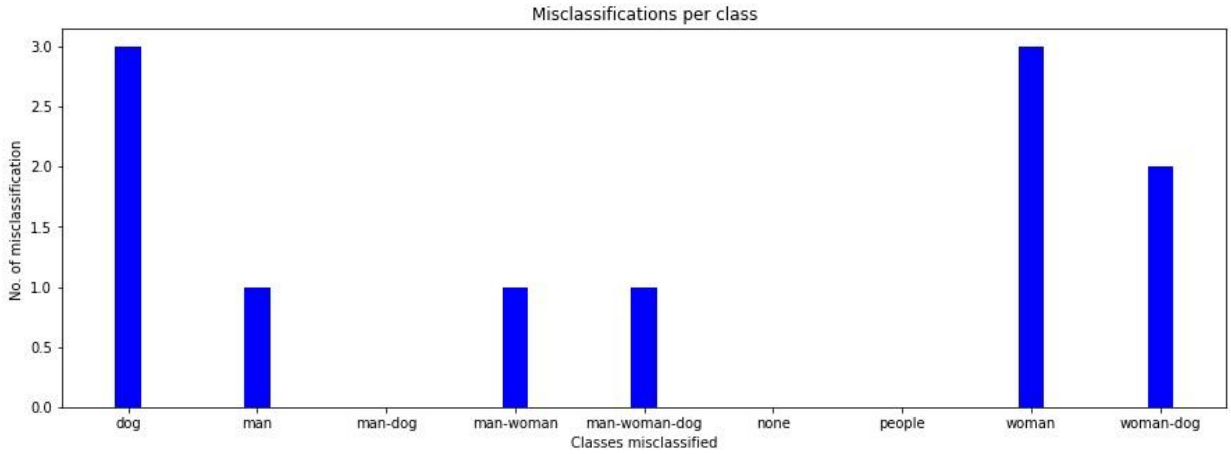
The result obtained is very similar to first one. The convergence is very fast and the best result is obtained at 10th iteration achieving the 97,78% of accuracy and the 0,725 of loss on test set. The Adam optimizer has an evident impact in this experiment and how the learning rate is adjusted very well during epochs. Once achieved the best result, the loss curve has a very small rise.

In general, the obtained results are very good. They have achieved a considerable accuracy and a quite low loss, so no underfitting situation is present. The overfitting is quite limited and is achieved in a situation close to the optimum. The main evident reason why this happens is because the dataset used for this task contains enough images for each class and the time spent in the pre-processing phase is paying off. Feeding to the network more data is one of the best solutions to fight overfitting, when they are available.

A brief error analysis is introduced here in order to understand what are the classes that are more misclassified. The following page contains the plots of the miss-classifications obtained on test set in each case. It seems that the class *woman* is the most miss-classified in all the cases.

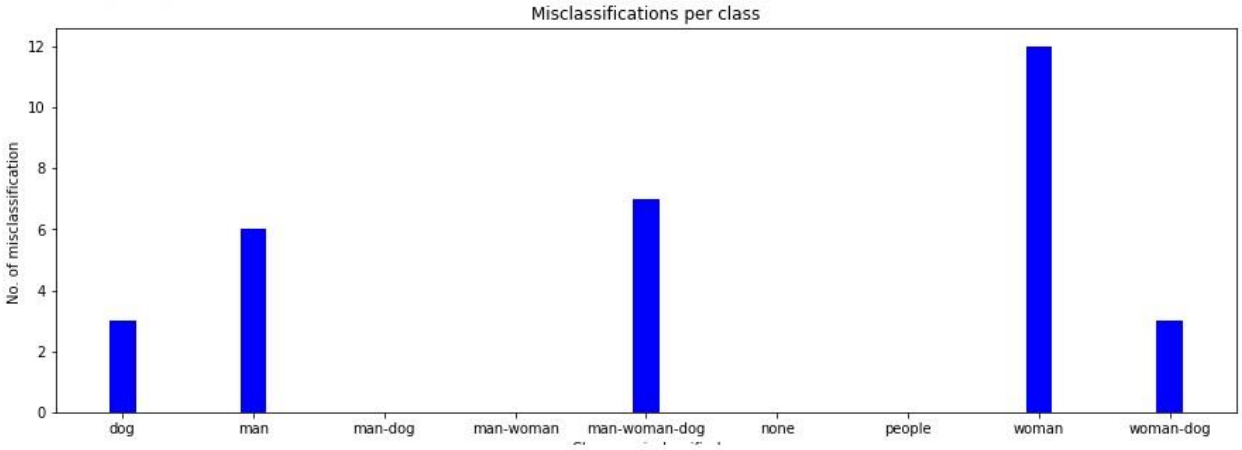
Using RMS-Prop.

```
Total errors :11/675
```



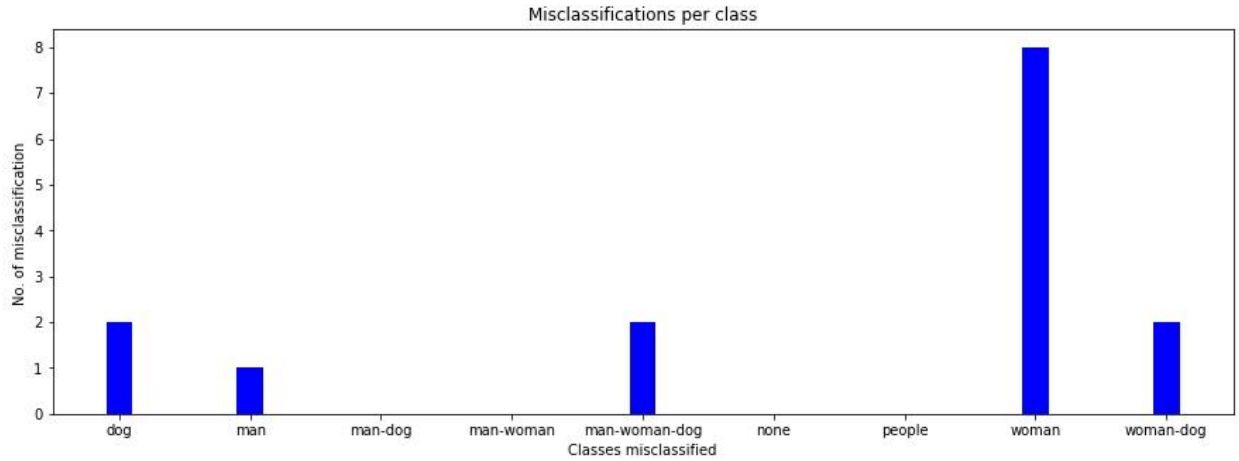
Using AdaGrad.

```
Total errors :31/675
```



Using Adam.

```
Total errors :15/675
```



CNN from scratch – deeper models

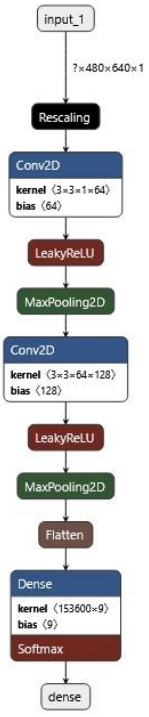
In this section different deeper models have been considered in order to try to improve as possible the very good result obtained so far. In all the further experiments, it is used is Adam optimizer. It is also used the early stopping regularization in order to stop training in overfitting situations.

The model used in the first attempt has the same structure of the previous one with the different of a greater number of convolutional filters. In fact, for each convolutional layer the number of filters has been doubled. This variation increases the number of trainable parameters so the time to train the network and the model complexity. This choice is made according to the resources available on Google Colab. The activation function used here is LeakyRelu with a value of alpha equal to 0.5.

The model has the following structure:

```
Conv2D(filters=64, kernel_size=3, padding="same")
LeakyReLU(alpha=0.5)
MaxPooling2D(pool_size=4)

Conv2D(filters=128, kernel_size=3, padding="same")
LeakyReLU(alpha=0.5)
MaxPooling2D(pool_size=4)

Flatten()
Dense(9, activation='softmax')
```

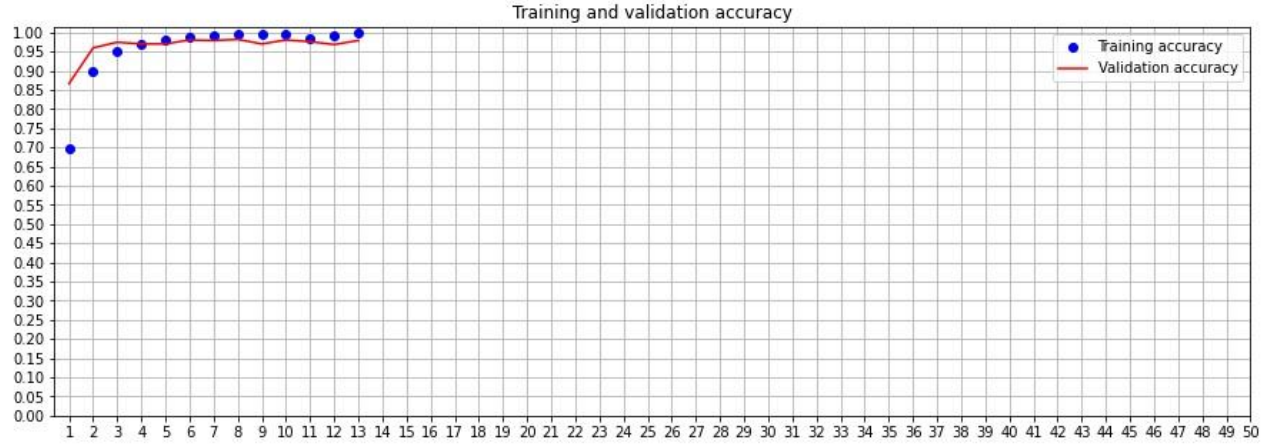
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 480, 640, 1)]     0

 rescaling (Rescaling)       (None, 480, 640, 1)       0

 conv2d (Conv2D)             (None, 480, 640, 64)      640

 leaky_re_lu (LeakyReLU)     (None, 480, 640, 64)      0

 max_pooling2d (MaxPooling2D  (None, 120, 160, 64)     0
 )

 conv2d_1 (Conv2D)           (None, 120, 160, 128)     73856

 leaky_re_lu_1 (LeakyReLU)   (None, 120, 160, 128)     0

 max_pooling2d_1 (MaxPooling  (None, 30, 40, 128)      0
 2D)

 flatten (Flatten)           (None, 153600)            0

 dense (Dense)               (None, 9)                 1382409

=================================================================
Total params: 1,456,905
Trainable params: 1,456,905
Non-trainable params: 0
_________________________________________________________________
```



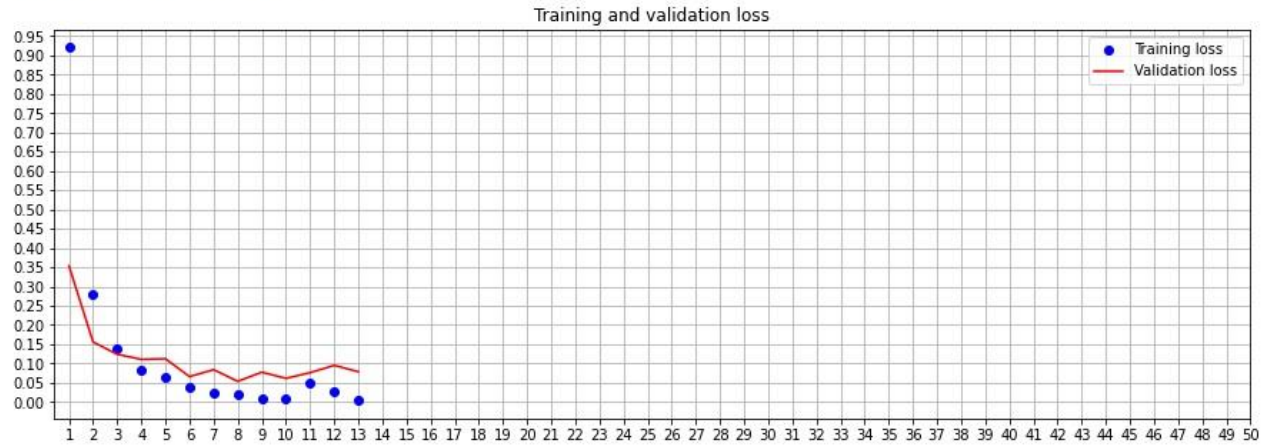
The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
8	0.9946	0.9822	0.0218	0.0538

Test Accuracy	Test loss
0.9778	0.0822

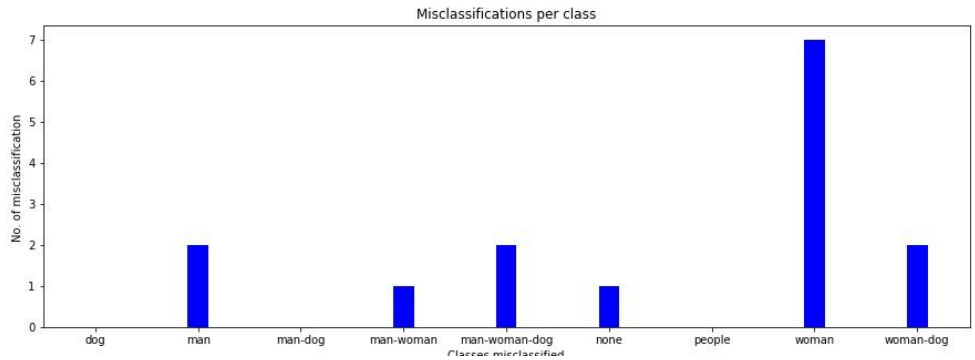



```
<Figure size 432x288 with 0 Axes>
```



The result obtained is still good but not significantly different from the results obtained in the previous section. The accuracy on test set is less than 98% and the loss is about 0.08. In a situation like this, for a similar result, a simpler model is preferable.

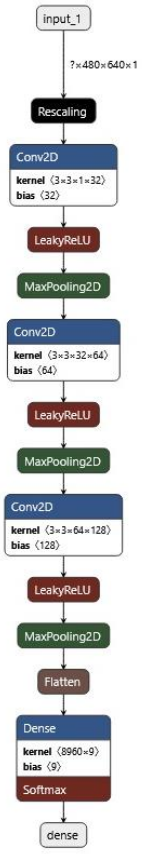
```
Total errors :15/675
```



The model used in this second attempt has an additional convolutional layer. The first two convolutional layers has the same identical structure and number of filters as the network trained in the section *simple model* with the difference of using a LeakyReLU as activation function. Althought a layer is introduced, this model has low parameters to train because of the presence of the additional max pooling that contributes to downsample the input data.

```
Conv2D(filters=32, kernel_size=3, padding="same")
LeakyReLU(alpha=0.5)
MaxPooling2D(pool_size=4)

Conv2D(filters=64, kernel_size=3, padding="same")
LeakyReLU(alpha=0.5)
MaxPooling2D(pool_size=4)

Conv2D(filters=128, kernel_size=3,padding="same")
LeakyReLU(alpha=0.5)
MaxPooling2D(pool_size=4)

Flatten()
Dense(9, activation='softmax')
```

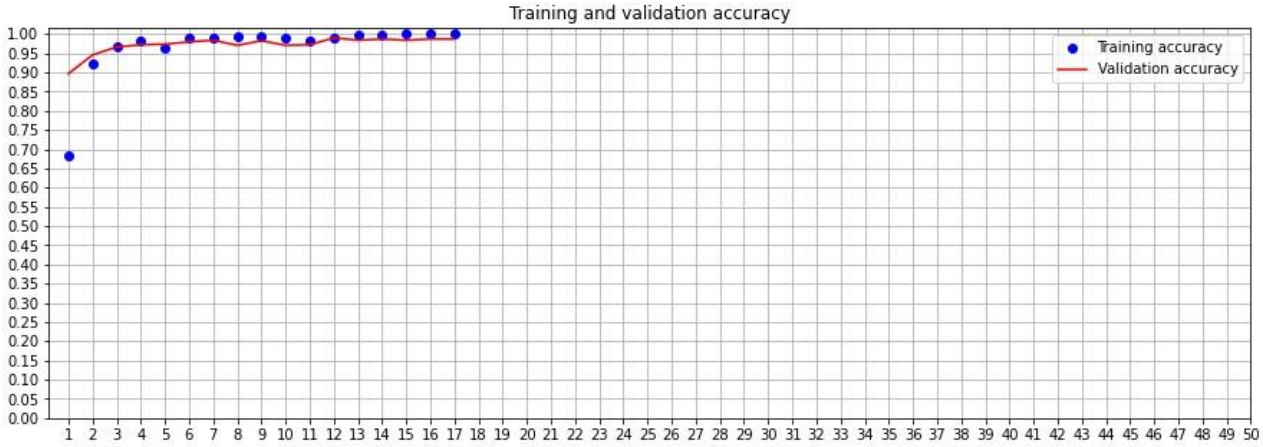
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 480, 640, 1)]     0

 rescaling (Rescaling)       (None, 480, 640, 1)       0

 conv2d (Conv2D)             (None, 480, 640, 32)      320

 leaky_re_lu (LeakyReLU)     (None, 480, 640, 32)      0

 max_pooling2d (MaxPooling2D  (None, 120, 160, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 120, 160, 64)      18496

 leaky_re_lu_1 (LeakyReLU)   (None, 120, 160, 64)      0

 max_pooling2d_1 (MaxPooling  (None, 30, 40, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 30, 40, 128)       73856

 leaky_re_lu_2 (LeakyReLU)   (None, 30, 40, 128)       0

 max_pooling2d_2 (MaxPooling  (None, 7, 10, 128)       0
 2D)

 flatten (Flatten)           (None, 8960)              0

 dense (Dense)               (None, 9)                 80649

=================================================================
Total params: 173,321
Trainable params: 173,321
Non-trainable params: 0
_________________________________________________________________
```



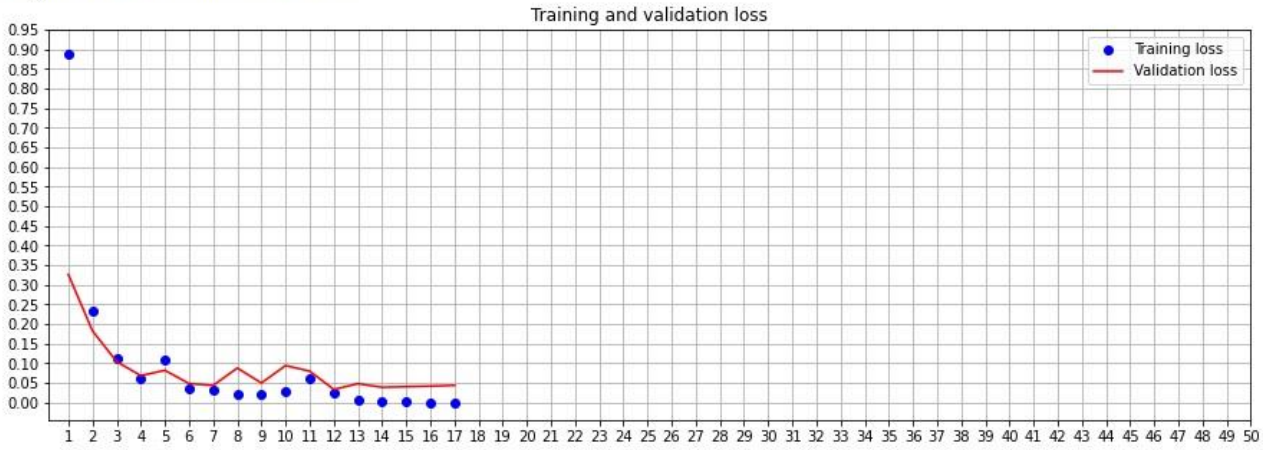
The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
12	0.9914	0.9896	0.0253	0.0334

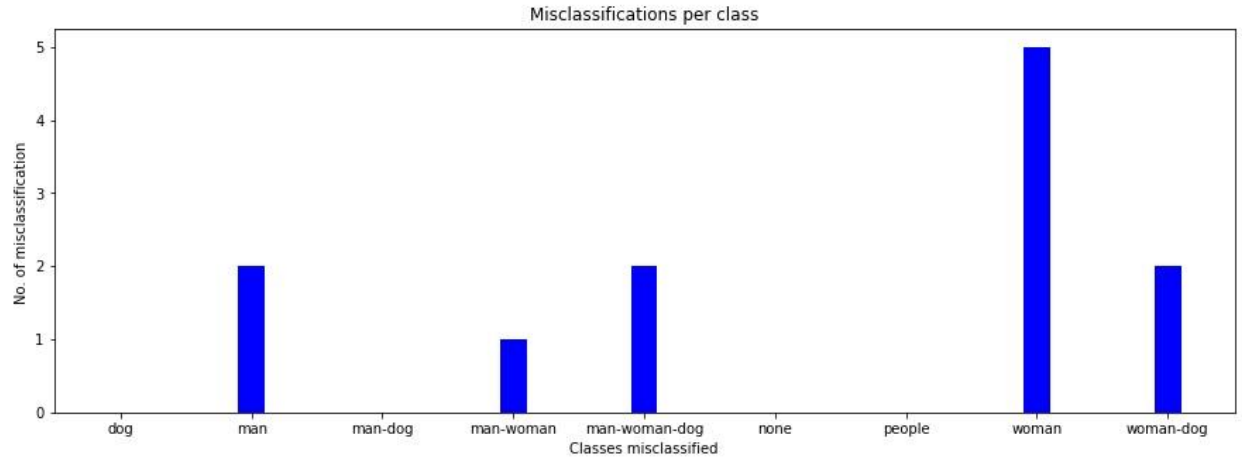
Test Accuracy	Test loss
0.9822	0.0689



<Figure size 432x288 with 0 Axes>



Total errors :12/675

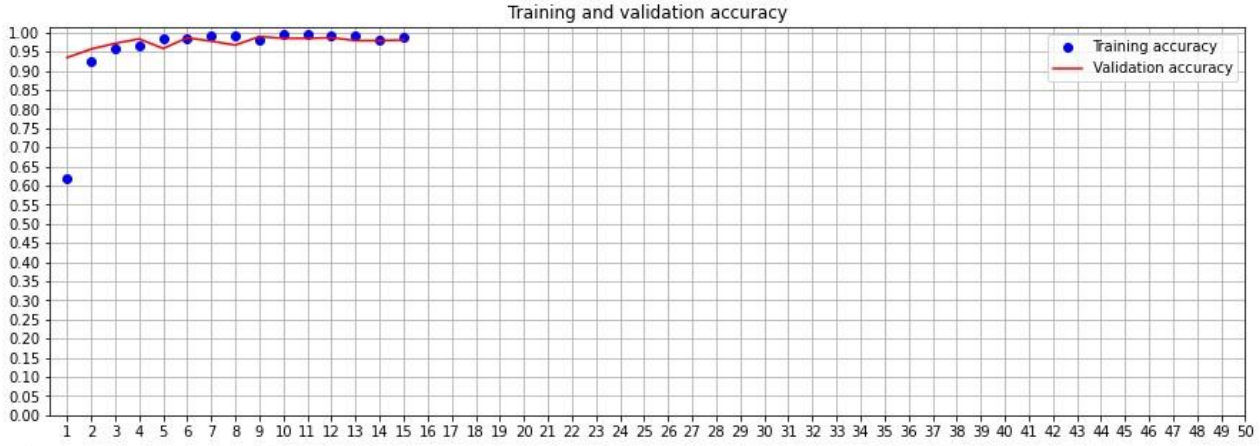


Looking at validation and test scores, it seems they have been improved a little respect to the other trials. This additional convolutional layer seems to give a benefit in this trial. Even if the accuracy on test is not the best encountered so far, the loss value results to be the lowest one.

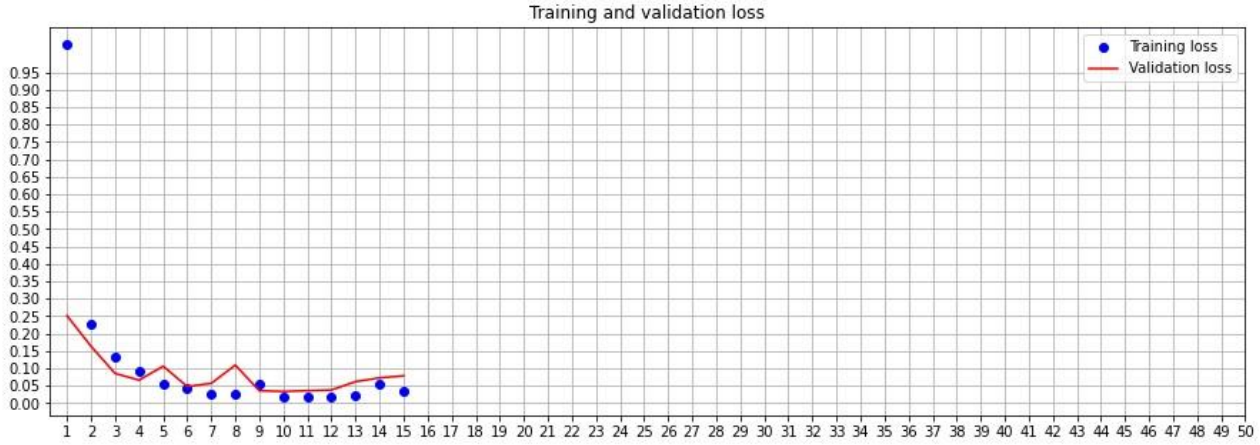
As last attempt, the same model is used with the ReLU6 activation function instead of LeakyRelu. The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
10	0.9956	0.9852	0.0181	0.0335

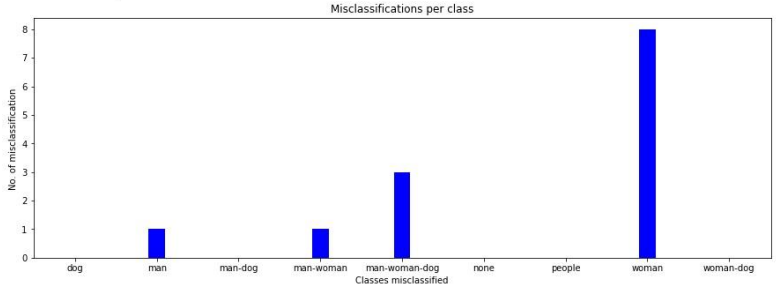
Test Accuracy	Test loss
0.9807	0.0551



<Figure size 432x288 with 0 Axes>



Total errors :13/675



Even in this case the loss metric is improved again.

CNN from scratch – using less data

In this section the previous models' structures are reused in order to analyse the training behaviour considering a reduced portion of the initial dataset. This is a non-common practice because not always who wants to train a deep neural network has a lot of data to work with. The task to be resolved is generally more difficult than the task proposed in this project and situations of overfitting or underfitting have to be fight. For this reason, this part is proposed just to have a confirmation of the benefits obtained from the all-available data.

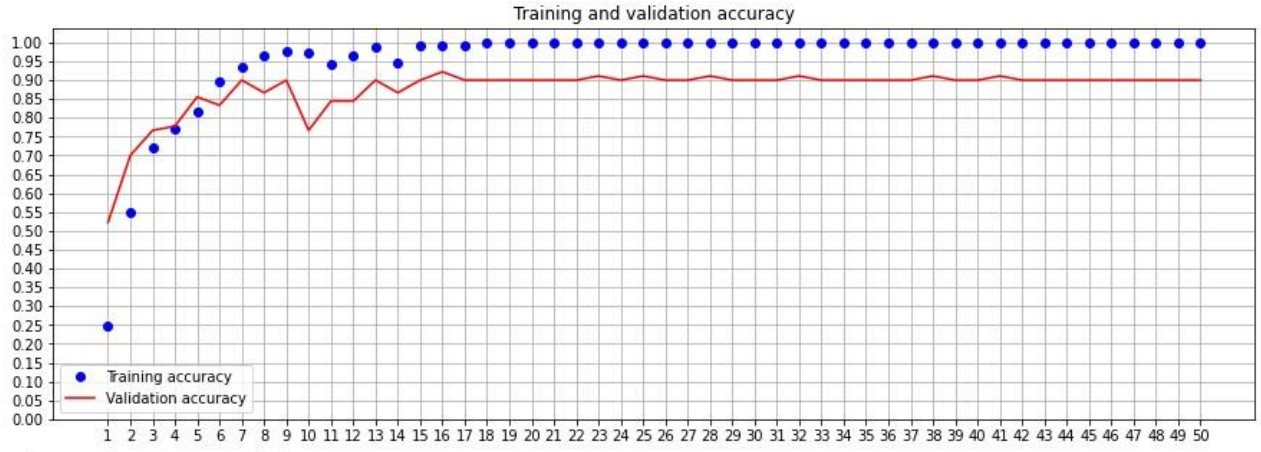
Instead of using the 70% - 15% - 15% to (train, validate and testing), it is used respectively the 10% (to train), 2% (to validate) and 88% (to test) of the initial dataset. The early stopping regularization is not used in order to have a complete visualization of the training process.

Here are reported the results obtained using this configuration on the *simple model* (using Adam optimizer) and the *deeper model* (with 3 convolutional layers, using LeakyReLU as activation function).

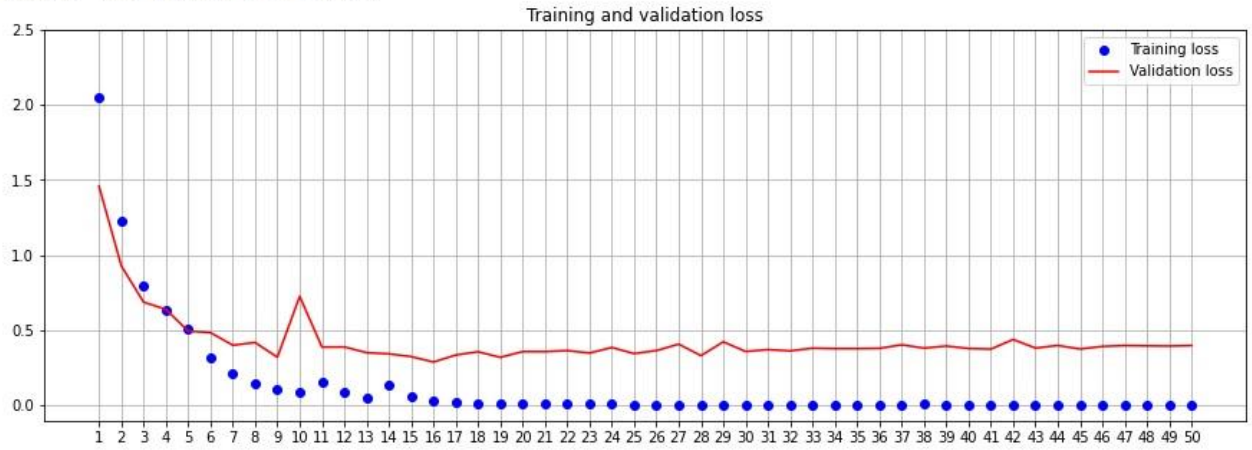
Using simple model.

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
16	0.9933	0.9222	0.0298	0.2883

Test Accuracy	Test loss
0.9207	0.2670



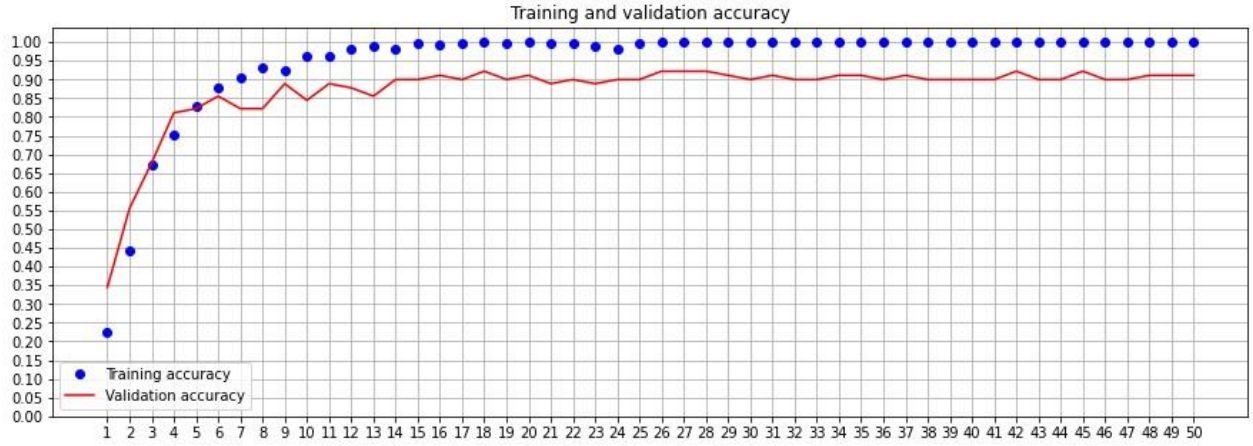
<Figure size 432x288 with 0 Axes>



Using deeper model.

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
10	0.9622	0.8444	0.1131	0.3701

Test Accuracy	Test loss
0.9033	0.3057



<Figure size 432x288 with 0 Axes>

In both the cases the phenomenon of overfitting is present and more evident. These two models have lower generalization capabilities respect to the model trained in previous sections. In a situation where no more data can be obtained, operations like dataset augmentation, reducing the capacity of the network, adding weight regularization or adding dropout layer can be done to reduce it.

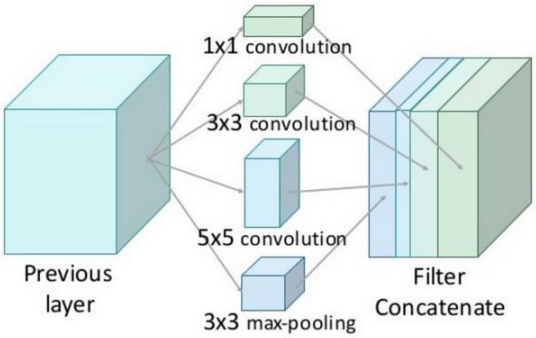
The great advantage of having a large number of data is to build simpler models that require less memory and less time to be trained but also to solve the overfitting phenomenon in a natural way.

Pre-trained CNN:

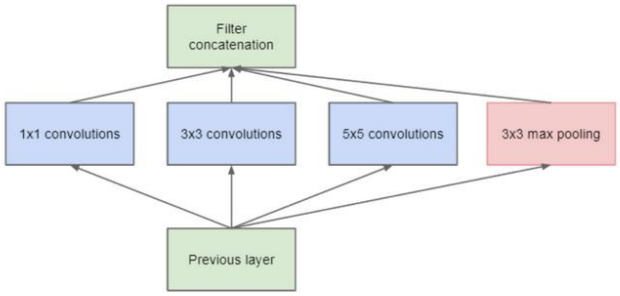
This part is dedicated to the usage of a pre-trained CNN in order to solve the task proposed in this project. For this purpose, Xception network is used because it is one of most performing models trained with ImageNet.

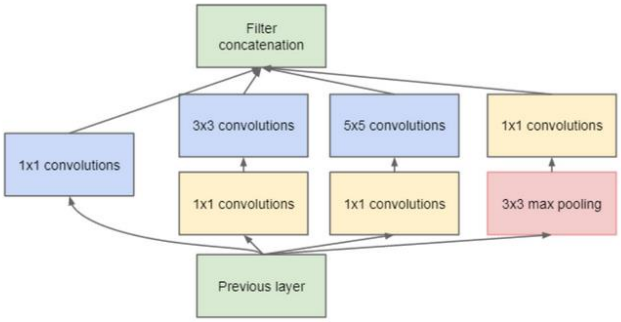
Xception is a deep convolutional neural network architecture that involves Depth wise Separable Convolutions. This network was introduced by Francois Chollet who works at Google, Inc. [14] Xception is also known as “extreme” version of an Inception module.

An Inception module computes multiple different transformations over the same input and then finally combining all the output which lets the model decide what features to take and by how much.



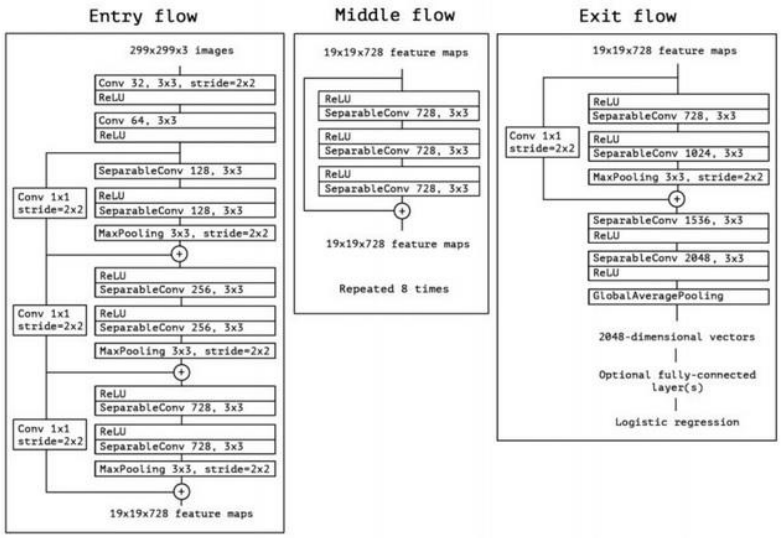
However, this is computationally inefficient because of convolutions. These convolutions not only happen spatially, but also across the depth. So, in each additional filter, it is necessary to perform convolution over the input depth to calculate just a single output map, and because of this, the depth becomes a huge bottleneck in the DNN. This depth can be reduced by doing 1x1 convolution across the depth. This convolution looks across multiple channel's spatial information and compress it down to a lower dimension. Due to this reduction, the researchers of Inception module were able to concatenate different layer transformations in parallel, resulting in DNN that was wide and deep.





Xception stands for *extreme inception*, it takes the principles of Inception to an extreme. In Inception, 1x1 convolutions were used to compress the original input, and from each of those input spaces different type of filters is used on each of the depth space. Xception just reverses this step. Instead, it first applies the filters on each of the depth map and then finally compresses the input space using 1x1 convolution by applying it across the depth.

Xception has other important characteristic: the presence or absence of a non-linearity after the first operation. In Inception model, both operations are followed by a ReLU non-linearity, however Xception doesn't introduce any non-linearity.

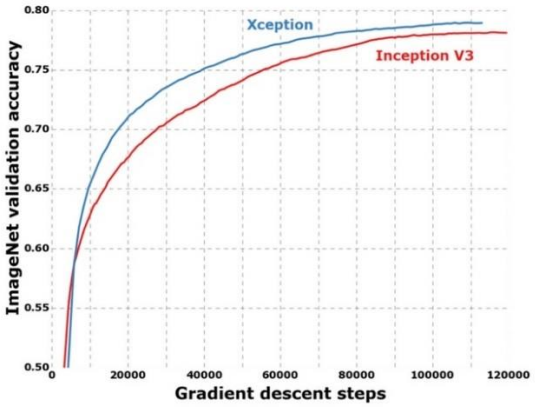


The data first goes through the entry flow, then it goes through the middle flow (repeating itself 8 times in this middle flow), and finally through the exit flow.

The next table shows how Xception outperforms every model in ImageNet dataset.

	Top-1 accuracy	Top-5 accuracy
VGG-16	0.715	0.901
ResNet-152	0.770	0.933
Inception V3	0.782	0.941
Xception	**0.790**	**0.945**

The next graph instead shows the different performance between Inception and Xception [15] :



Xception – feature extraction

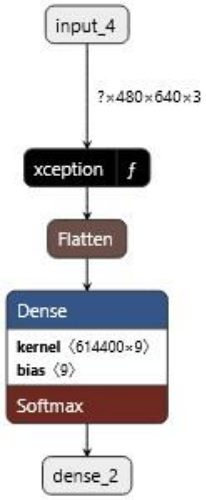
In this section the Xception model is used to apply one of the transfer learning techniques: feature extraction. The idea is to use the overall trained structure without the final part (fully connected layer) replacing it with a new fully connected layer with a number of neurons equal to number of classes to predict (9). In this way the features learned generically by the first part, also called *convolutional base*, are reused and the new fully connected layer is trained (ad hoc) for the custom task of this project. The big advantage is to leverage the training job done by others (which probably required a lot of time and a lot of data) and adapt it to the customized problem.

The Xception network requires as input, images on 3 RGB channels, so the previous pre-processing grayscale operation is not applied here. The rescaling operation is also removed due to limitation in memory of Google Colab account.

The model's structure is the following:

```
inputs = keras.Input(shape=(480, 640, 3))
x = Xception(inputs) #pre-trained network
x = layers.Flatten()(x)
outputs = layers.Dense(9, activation='softmax')(x)
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 480, 640, 3)]     0

xception (Functional)        (None, 15, 20, 2048)      20861480

flatten (Flatten)            (None, 614400)            0

dense (Dense)                (None, 9)                 5529609

=================================================================
Total params: 26,391,089
Trainable params: 5,529,609
Non-trainable params: 20,861,480
_________________________________________________________________
```

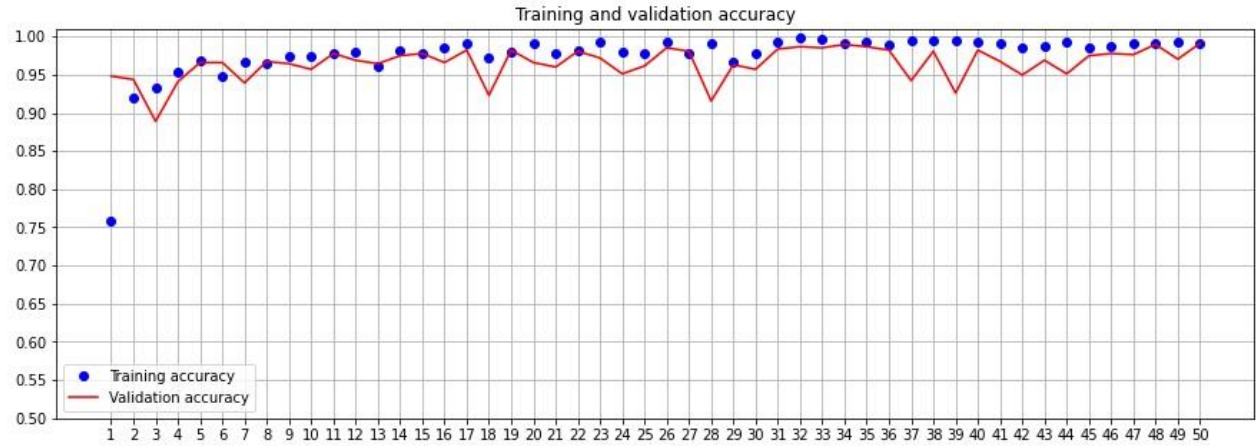


In this experiment, the optimizer used is Adam with an initial learning rate of 0. 0005, lower than the default previous used in order to achieve a slower convergence with lower oscillations. The early stopping technique is also removed to observe a full behaviour during training. The batch size dimension is still 32 and the number of epochs used is still 50. The evaluation metric is the accuracy and the loss function used is still the categorical cross entropy. Freezing the previous Xception layers, the number of trainable parameters result to be 5 529 609.

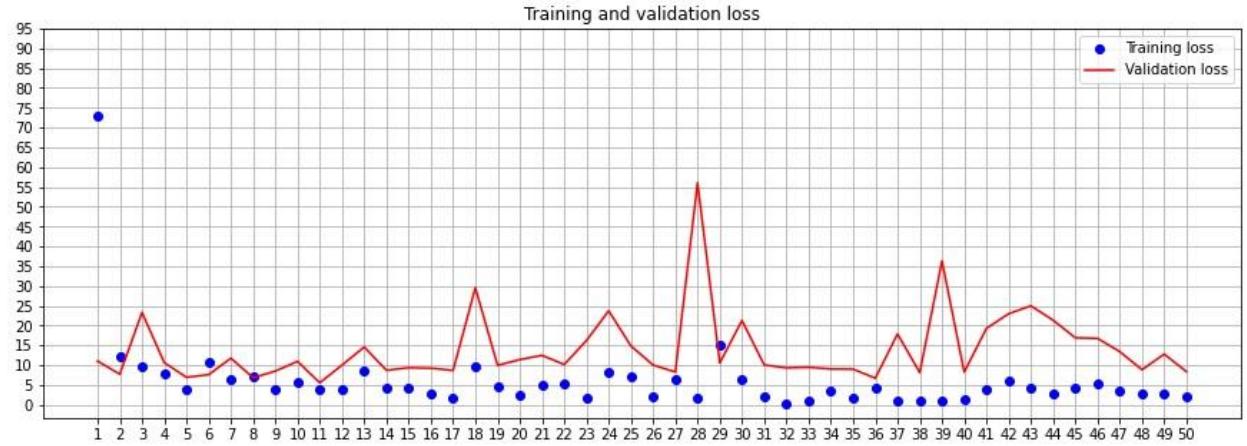
The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
11	0.9784	0.9778	3.9542	5.5825

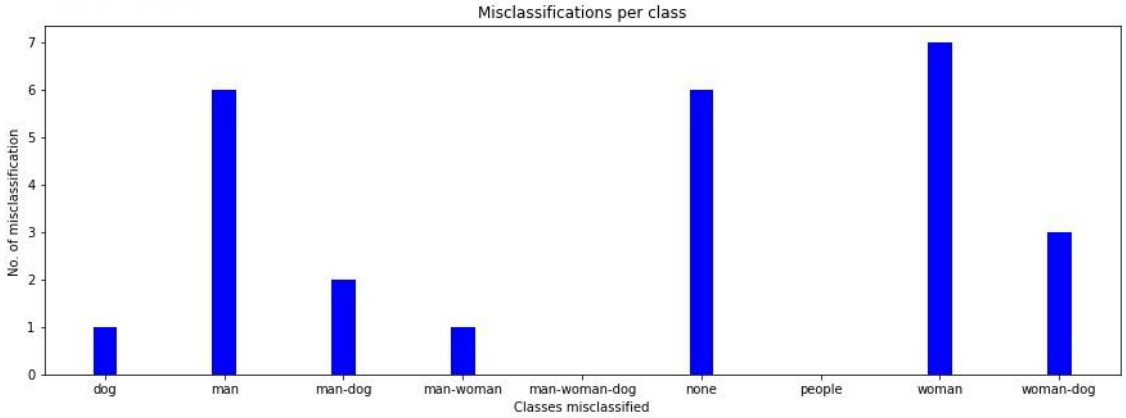
Test Accuracy	Test loss
0.9615	7.9760



<Figure size 432x288 with 0 Axes>



Total errors :26/675



The results evidence that high accuracy can be obtained even with the usage of a pre-trained network. However, the performances obtained are bit worse respect to previous experiments (CNN from scratch). In particular the loss metrics results to be too high. A possible reason of this behaviour is due to mismatch domain of the dataset. Another mismatch can be considered as the dimension of the inputs. In this case, the weights of Xception model have been initialized with the ImageNet dataset (the most common and used), that is trained with input images of (224x224) but the actual input shape here is (640x480).

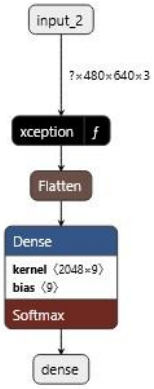
Probably using a different domain, the result could be better. In the next experiment, in order to try improve this result, in particular to reduce the loss value, it is introduced a max pooling layer applied to the final layer. This reduces the dimensionality and the parameters to train.

This can be easily obtained adding a parameter "*pooling"* to the API call for the definition of the pre-trained network and specifying the typology, in this case max-pooling.

```
tf.keras.applications.Xception(
include_top=False,
input_shape=(480, 640, 3),
weights="imagenet",
pooling="max"
)
```

The model structure is the same but the number of trainable parameters is significantly lower:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 480, 640, 3)]     0

 xception (Functional)       (None, 2048)              20861480

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 9)                 18441

=================================================================
Total params: 20,879,921
Trainable params: 18,441
Non-trainable params: 20,861,480
_________________________________________________________________
```

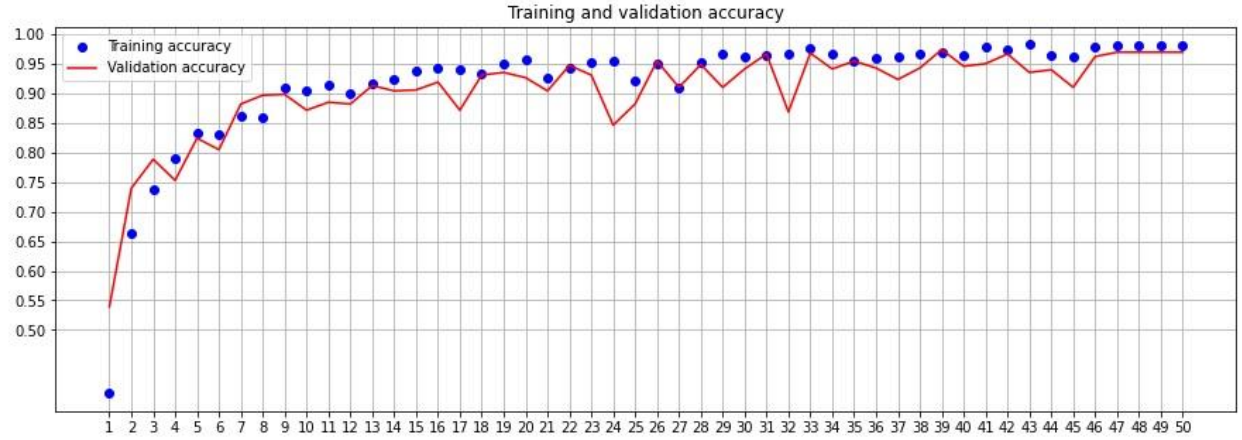


The results obtained are the following:

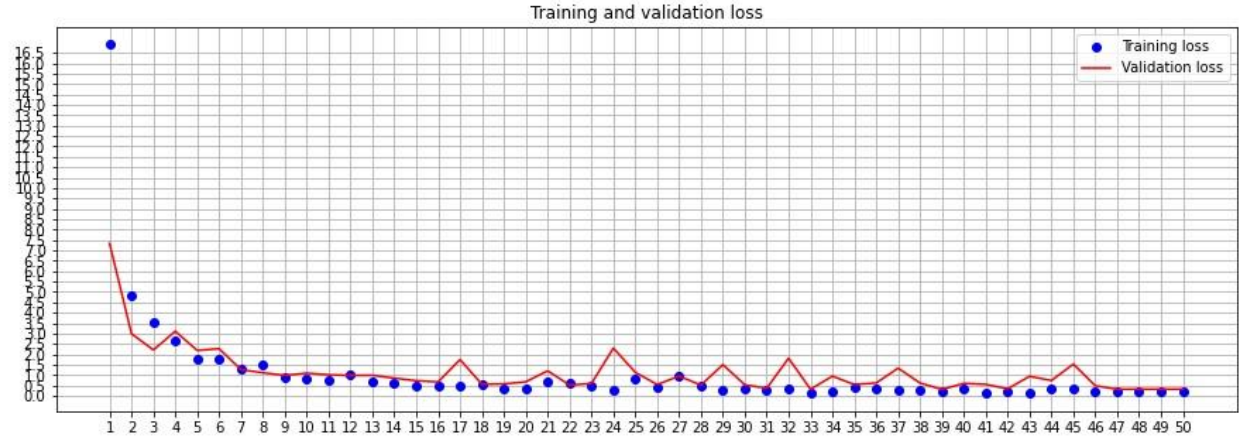
Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
39	0.9692	0.9733	0.2260	0.3093

Test Accuracy	Test loss
0.9437	0.5822

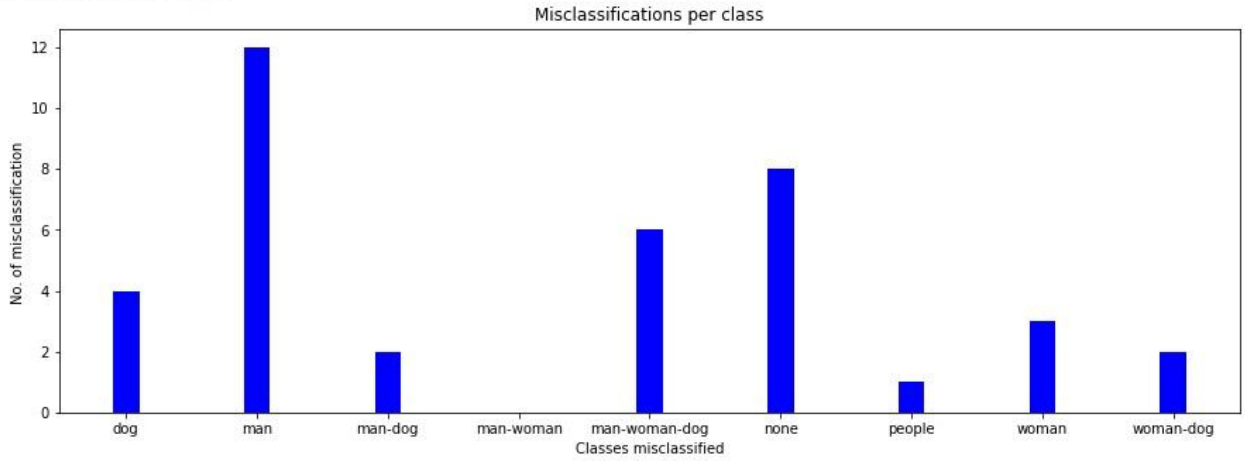
In this second attempt the value of loss is reduced a lot both in validation than in test sets. However, the value of accuracy is lower too. Looking at the training behaviour, there is a non-evident situation of overfitting. Even at iteration 46 the model seems to try to reduce its loss but unfortunately due to reaching the maximum training epoch, the model stops the training early. So, the solutions could be to repeat the training with a greater number of epochs and/or choosing a greater initial learning rate in order to speed up the convergence. If results are same could be interesting to improve the fully connected part, adding more dense layers with more neurons.




```
<Figure size 432x288 with 0 Axes>
```




```
Total errors :38/675
```



Differently from previous models, here the class most misclassified is man. The same class is the second most misclassified in the first experiment of the section. This makes evident that the pre-trained model focuses on different characteristic of the image respect to model built from scratch.

Xception – fine-tuning

In this section the Xception model is used to apply another transfer learning techniques: fine tuning. The idea is to freeze layers of the pre-trained network, like in feature extraction, leaving one or some of the last layers unfrozen in order to train both the remaining layers and the fully connected classifier. The advantage is to reuse the weights of first layers (representing the generic features) computed on a larger dataset and at the same time, training the weights of the last's final layers (representing more specific features) adapting the new network to the custom task.

The approach adopted in the following experiments is to gradually unfreeze layers starting from the bottom, the more specific one, then looking the obtained result and repeat the same operation with the new last remaining frozen layer.

The model used in the first attempt is the same used in previous experiment: Xception with max-pooling layer. In the next picture is present the final part of the Xception 'structure.

```
block14_sepconv1 (SeparableCon  (None, 15, 20, 1536  1582080    ['add_11[0][0]']
v2D)                           )

block14_sepconv1_bn (BatchNorm  (None, 15, 20, 1536  6144       ['block14_sepconv1[0][0]']
alization)                     )

block14_sepconv1_act (Activati  (None, 15, 20, 1536  0          ['block14_sepconv1_bn[0][0]']
on)                            )

block14_sepconv2 (SeparableCon  (None, 15, 20, 2048  3159552    ['block14_sepconv1_act[0][0]']
v2D)                           )

block14_sepconv2_bn (BatchNorm  (None, 15, 20, 2048  8192       ['block14_sepconv2[0][0]']
alization)                     )

block14_sepconv2_act (Activati  (None, 15, 20, 2048  0          ['block14_sepconv2_bn[0][0]']
on)                            )

global_max_pooling2d (GlobalMa  (None, 2048)        0           ['block14_sepconv2_act[0][0]']
xPooling2D)

==================================================================================================
Total params: 20,861,480
Trainable params: 20,806,952
Non-trainable params: 54,528
__________________________________________________________________________________________________
```

In the first attempt the last trainable layer *block14_sepconv2_bn* (Batch Normalization) is unfrozen. Typically, a batch normalization layer should not be unfrozen because during training, it uses the mean and variance of the current batch of data to normalize samples, and during inference, it uses an exponential moving average of the batch-wise mean and variance of the data seen during training. Batch normalization (BN) is a technique used to improve the performance and stability of neural networks by normalizing the activations of a given layer. Batch normalization typically has trainable parameters, such as the scale and shift parameters, which are learned during training. These parameters are used to adjust the distribution of the activations to have a mean of zero and a standard deviation of one. A batch normalization layer allows to accelerating the deep network training by reducing internal covariate shift [16]. When defined the pre-trained model, the parameter *training* is set to false (default value), in order to keep all the BN layers in inference mode. The last BN layer is

trainable that means the trainable weights of the layer should be updated to minimize the loss during training. [17]

The reason why the *block14_sepconv2_act* is not taken in consideration is because is being used as a placeholder layer and it is designed to have no trainable parameters.

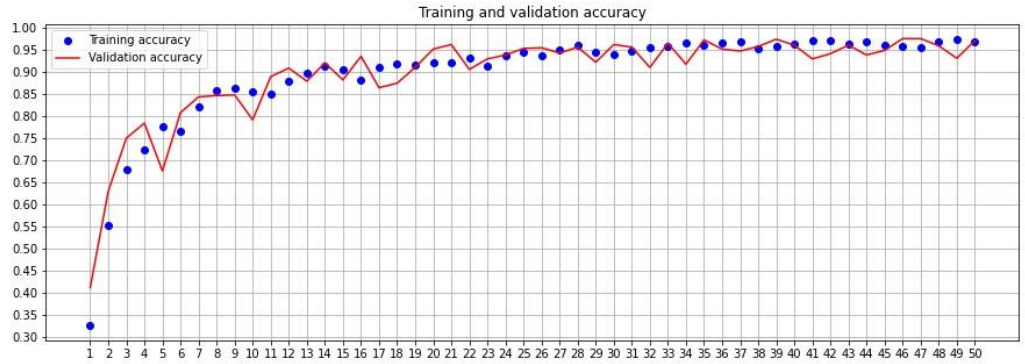
The optimizer used is Adam with initial learning rate of 0.005 (0.005 higher respect to default value). The total number of trainable parameter is 22 537.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 480, 640, 3)]     0

xception (Functional)        (None, 2048)              20861480

flatten_1 (Flatten)          (None, 2048)              0

dense_1 (Dense)              (None, 9)                 18441

=================================================================
Total params: 20,879,921
Trainable params: 22,537
Non-trainable params: 20,857,384
_________________________________________________________________
```

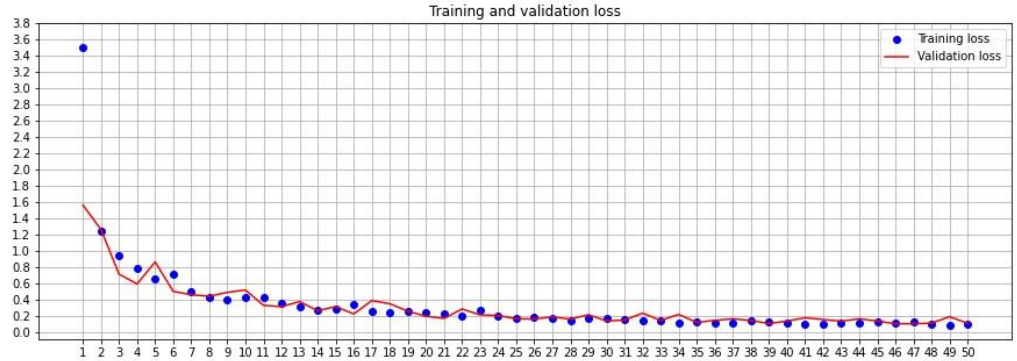
The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
47	0.9546	0.9748	0.1326	0.1060

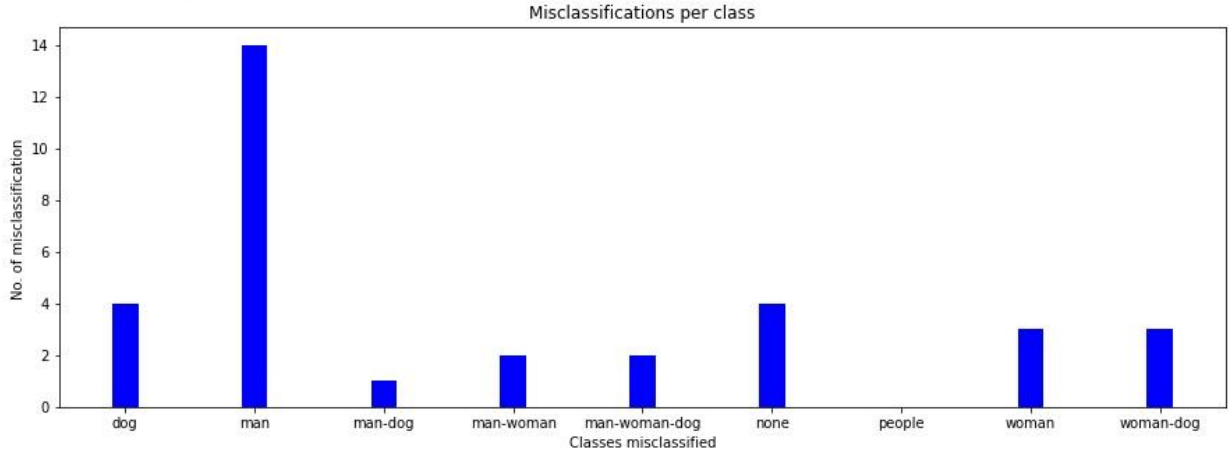
Test Accuracy	Test loss
0.9511	0.1478



<Figure size 432x288 with 0 Axes>



Total errors :33/675



Looking at the training results, the model evidence a slow convergence and an interesting aspect: in most of the epochs the validation accuracy is bigger (a little) then the training accuracy. This effect is probably due to the decision to unfrozen the batch normalization layer. The choice of using a maximum of 50 epochs for training seems to be low even in this case, because the validation loss seems to be better improved.

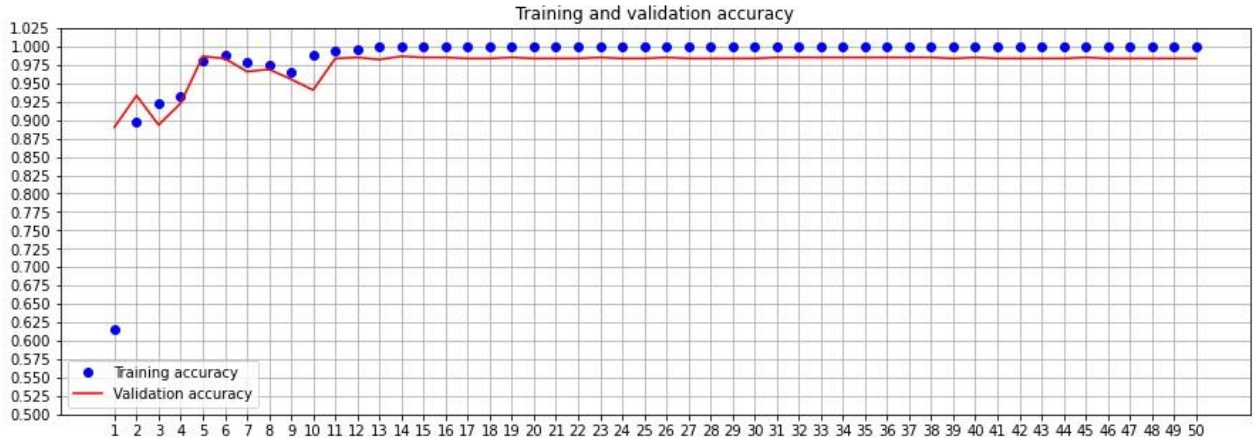
In the second attempt the last two trainable layers are unfrozen: block14_sepconv2 (Separable Convolution 2D) and block14_sepconv2_bn (Batch Normalization). The optimizer used is still Adam with initial learning rate of 0.005. The total number of trainable parameters is 3 182 089, slightly bigger respect to previous attempt but the risk of overfitting is limited due to the presence of big data available.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 480, 640, 3)]     0

xception (Functional)        (None, 2048)              20861480

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 9)                 18441

=================================================================
Total params: 20,879,921
Trainable params: 3,182,089
Non-trainable params: 17,697,832
_________________________________________________________________
```

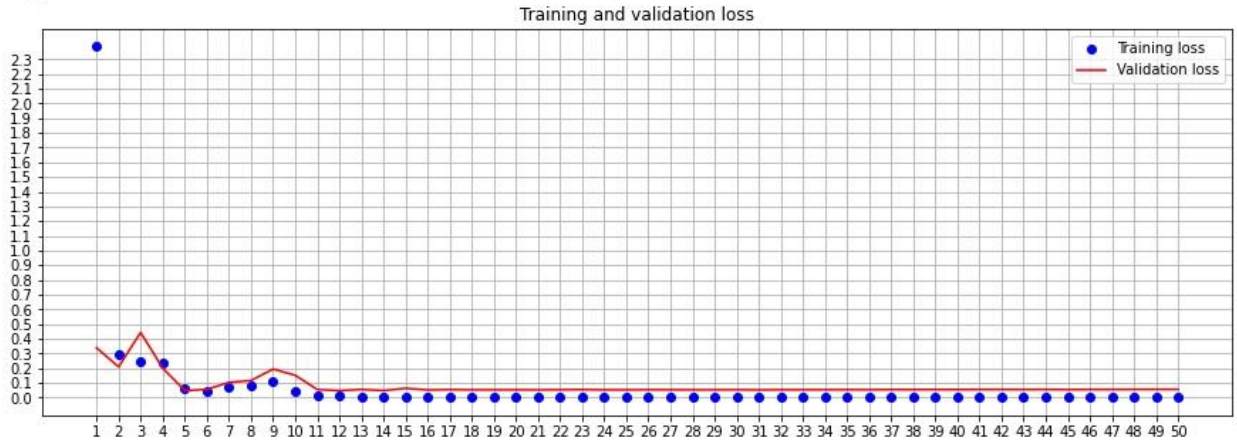
The results obtained are the following:

Epoch best result	Train. accuracy	Valid. Accuracy	Train. loss	Valid. loss
12	0.9962	0.9852	0.0095	0.04767

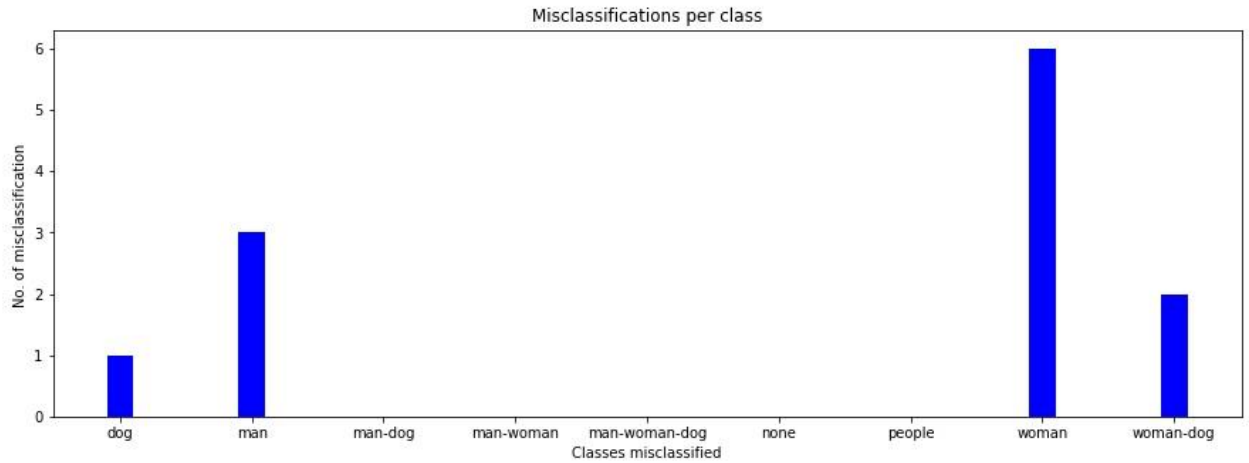
Test Accuracy	Test loss
0.9822	0.0844




```
<Figure size 432x288 with 0 Axes>
```




```
Total errors :12/675
```



In this experiment, the convergence is very fast and reaches very good results. This represents the best result obtained using Xception pre-trained network despite it requires to train more parameters so more training time. Using the fine-tuning technique, the performances result improved respect to feature extraction because unfreezing the last layers the domain problem tends to be resolved making the network adapting to new custom problem.

Evaluation of the results

The results of previous sections are obtained with the help of GPU on Google Colab account, in order to reduce the excessive training time, so reproducibility is not guaranteed. These networks are initialized with random weights, which can lead to different results each time the network is trained. To obtain reproducible results, should be used a deterministic algorithm, setting a random seed, using identical hardware, and using the same hyperparameters.

The following table summarizes all the results obtained on the full dataset:

Stats.			Accuracy			Loss			
CNN_Name	Epoch	#Params	Train	Valid.	Test	Train	Valid.	Test	MissClass
From_scatch RmsProp	17/50	710025	0.997	0.983	0.983	0.017	0.057	0.079	11/675
From_scatch AdaGrad	49/50	710025	0.974	0.967	0.954	0.094	0.115	0.146	31/675
From_scatch Adam	10/50	710025	0.996	0.985	0.977	0.011	0.042	0.072	15/675
From_scatch deep_more_neurons_	8/50	1456905	0.994	0.982	0.977	0.021	0.053	0.082	15/675
From_scatch deep_3_conv_lay.	12/50	173321	0.991	0.989	0.982	0.025	0.053	0.068	12/675
From_scatch deep_3_conv_lay.relu6	10/50	173321	0.995	0.985	0.980	0.018	0.033	0.055	13/675
Pre-Trained_ feat_extr_std	11/50	5529609	0.978	0.977	0.961	3.954	5.582	7.976	26/675
Pre-Trained_ feat_extr_max_pool	39/50	18441	0.969	0.973	0.943	0.226	0.309	0.582	38/675
Pre-Trained_ fine_tuning_1_layer	47/50	22537	0.954	0.974	0.951	0.132	0.106	0.147	33/675
Pre-Trained_ fine_tuning_2_layers	12/50	3182089	0.996	0.985	0.982	0.009	0.047	0.084	12/675

The result obtained are very good complexivly. In general when different models have to be compared multiple ascpects are take into account:

- Model architecture: different architectures, may be better suited for different types of data and tasks; The dataset used in all the experiments is the same but in some cases the results are better than other. This is due to the different model architecure but also for the hyperparameters used and weight inizialization.

- Training data: the quality and quantity of the training data can have a big impact on model performance, like in this case. The worst result obtained in terms of accuracy is 94% on test data. This percentage is quite high because the dataset used contatins enough data and allow to achive a good results, but as it is evident a better result can be obtained with proper network structure.

- Performance metrics: in general the accuracy metric is not the only metric to take in account sometimes could be useful to consider precision, recall, or F1 score. For this simple task that has balanced classes, accuracy is the best way to measure performance.

- Generalization: this is how well the model performs on new, unseen data, compared to its performance on the training data. In all the cases analized, the validation loss is quite similar to test loss and similarly happened for accuracy. Of course better the metric considered is better the generalization will be.

- Speed and memory: training and inference speed, and memory requirements, may be important factors to consider, depending on the application. In this experiments the training time is not considered so much. Instead, the inference speed is quite important because it determines how fast the network will predict a new unseen image. Higher this value is, better the performance of the further real time application (in production) will be. The only limitations considered for memory and resources are imposed from the Google Colab account.

- Explainability and interpretability: for certain types of applications, it may be important to understand how the model is making its predictions. In convolutional neural network context, it has low sense to understand how a network is learning some class, or at least could be useful and intresting how the firsts convolutional layers interpret the generic charateristics but gooing deeper the feauture learned become less human interpretable. For this reason all the models are considered equivalent on this point of view.

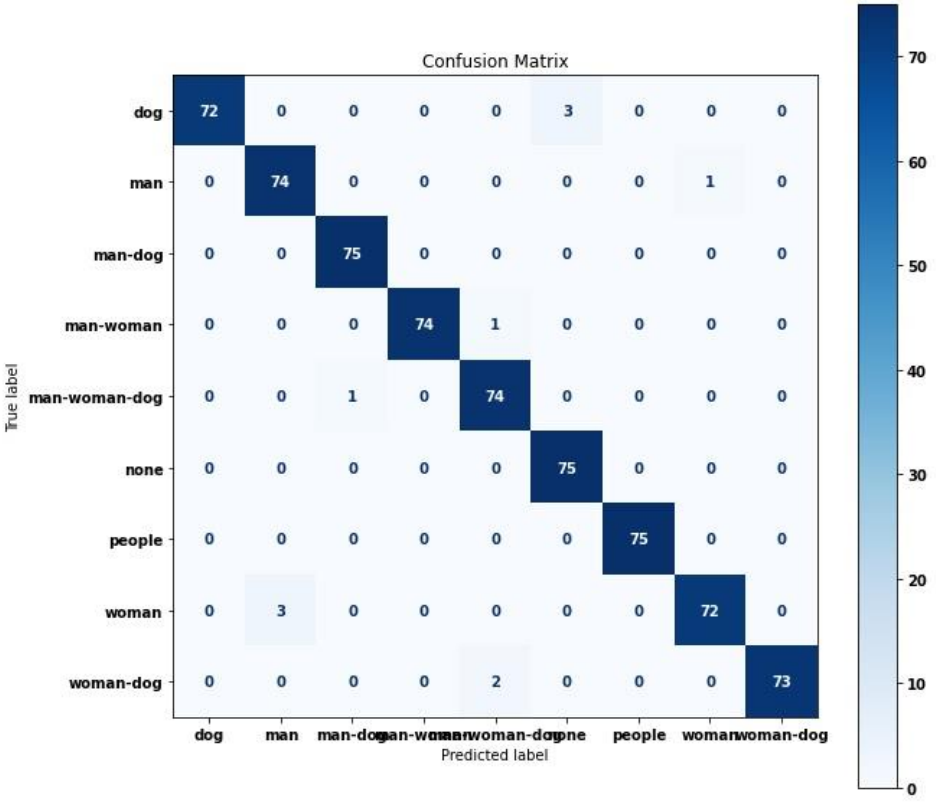
- Model size and complexity: not always a model with too parameters is better than others. With the same accuracy it is generally preferable to choose a model with fewer parameters. A model with too many parameters is prone to the phenomenon of overfitting and therefore less generalizable. A model with fewer parameters is easier to understand and interpret. It also requires less time and memory and can be trained and used faster. In a context of home-made videosurvelliance where the model will be used inside a software runs of a limited hardware like *Raspberry* this aspect has lot of importance.

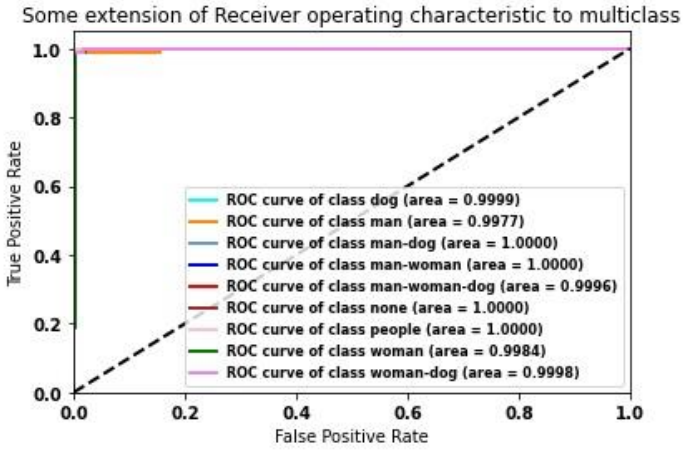
Comparing the results obtained training the network "from scratch" and "using a pre-trained", it is evident how the first typology has achived more important performances because the domain of this task is more specific and the data available are enough to build a model capable of recognize all the classes. It also highlights that the incrementation of the number of parameters does not take a real and considerable benefit in performace. Infact, the best results obtained during the training from scratch is achived in the network *From_scatch deep_3_conv_lay* where accuracy on test set is 98,22% and the loss is 0.0551, although the network *From_scatch RmsProp* has a greater accuracy 98,37% but also a greater loss 0.0797. Considering the number of parameters and the model size the first is preferred.

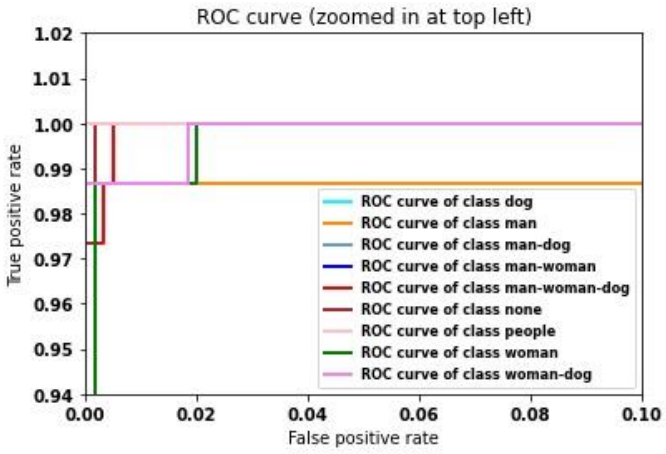
In feature extraction experiments the network is not capable to distinguish all the classes with the same accuracy achived before, instaed in the last experiment of fine-tuning *Pre-Trained_fine_tuning_2_layers* the result is very similar compared to the network trained froms screatch but at cost of training more parameters.

In the next page are reported the confusion matrices and the roc curve of this three models just to have a complete picture of the performance achieved.

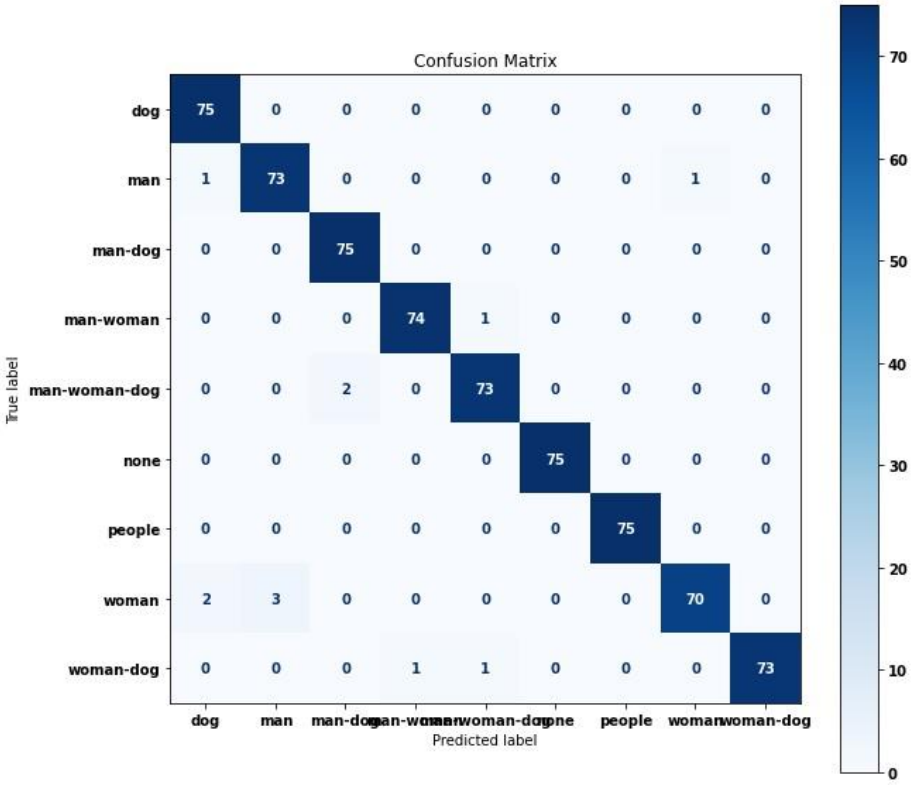
From_scatch: simple model with RmsProp

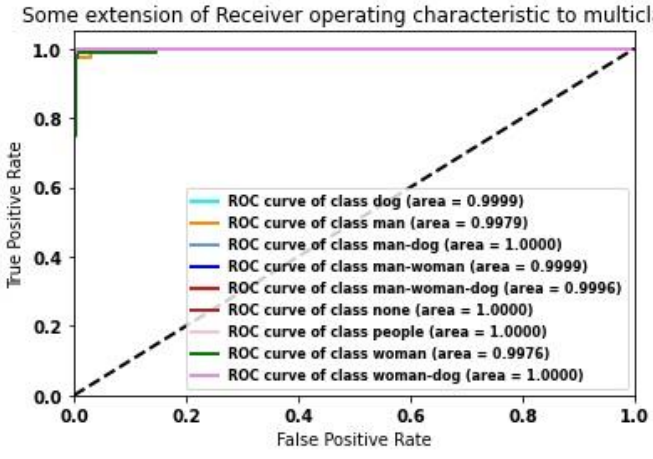


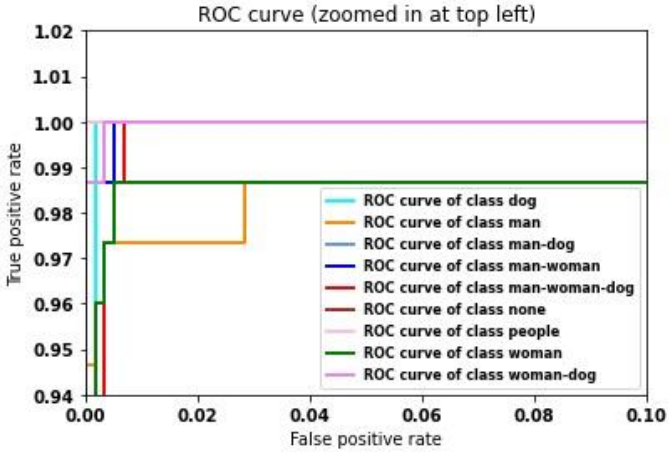




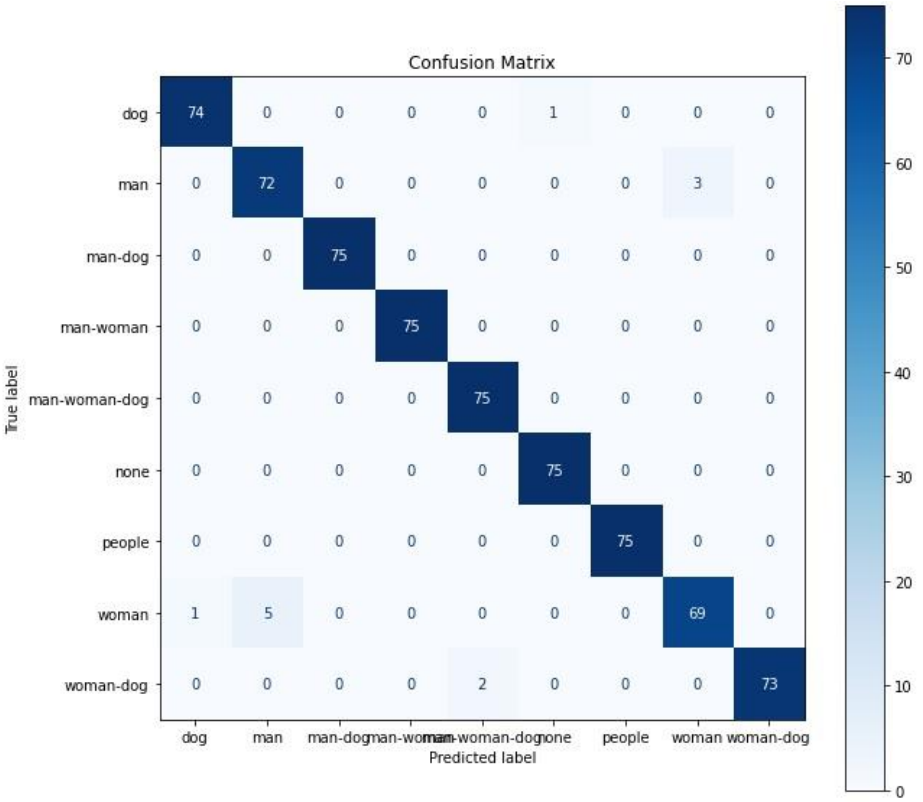
From_scatch deep_3_conv_layers

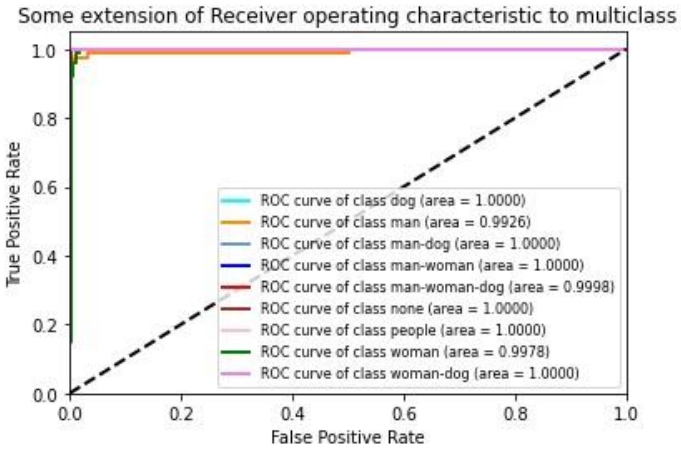


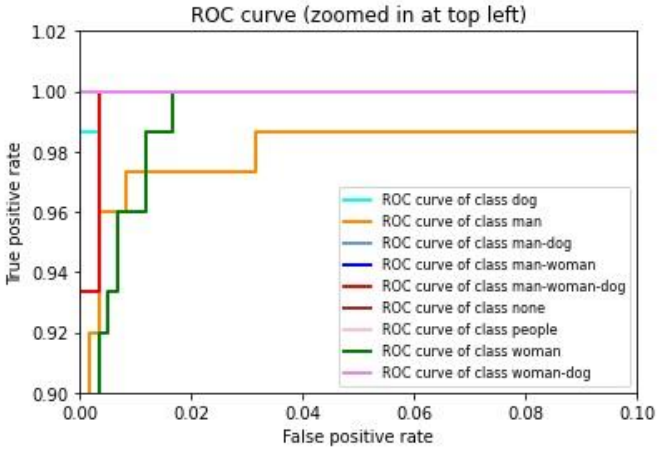




Pre-Trained_fine_tuning_2_layers







Error analysis

This part is dedicated to analyse the miss-classified samples of the previous models in order to understand what are the elements that can make the network mislead in the inference. Although the number of miss-classification is similar, they don't share the same architecture, so their predicting power is different. In previous sections, there are present some bar plots about the classes most miss-classified.

Here a recap of the miss-classification about the three best models (From_scatch RmsProp, From_scatch deep_3_conv_lay , Pre-Trained_fine_tuning_2_lay):

Number of miss classification on 675 samples									
	Class name								
CNN_Name	dog	man	man-dog	man-woman	man-woman-dog	none	people	woman	woman-dog
From_scatch RmsProp	3	1	1	1	1	0	0	3	2
From_scatch deep_3_conv_lay.	0	2	0	1	2	0	0	5	2
Pre-Trained_ fine_tuning_2_lay	1	3	0	0	0	0	0	6	2

The class *woman* results to be the most miss classified and it is often confused with the class *man*. The reason could be because these two individual classes are the only in which a single subject is present and then the network has to distinguish the shape of a man respect to a shape of a woman and depending on the position in which he/she is recorded, there can be some misleading in some circumstances. In order to reduce this miss-classification error more data about these two classes should be added to training set.

Another important result is about the class *none*. Considering these three models on this specific test dataset, there are no cases of miss-classification resolving one of the initial problems: the false alarm. Also, the class *people* is perfectly classified.

One weak point of this project is a no consideration of a situation where two people of the same sex are present in the image. The class people is designed to recognize at least 3 people in the image independently from the sex. So, if two man or two women are present at the same time in the picture, it is not possible to determine what the network will predict. The reason why this aspect is not covered is due to the impossibility to retrieve enough data of people of the same sex present at the same time. This could be a possible improvement for a further upgrade of the project.

The next page shows what are the effective samples miss-classified in these three networks and the relative confidence obtained in miss-prediction. The confidence value refers to the level of certainty that the network has about its predictions. This can be represented as a probability score for each prediction made by the network. The confidence score indicates how likely it is that the prediction is correct, with a higher score indicating higher confidence. Confidence scores is calculated from the last layer before the final prediction (softmax activation).

The samples that are miss-classified in more than one network are highlighted with different letters.

From_scatch: simple model with RmsProp

From_scatch deep_3_conv_layers

Pre-Trained_fine_tuning_2_layers

Ensemble solutions

An ensemble classifier is a machine learning model that combines the predictions of multiple individual classifiers to improve the overall performance and accuracy of the classification task. It is used when the goal is to improve the performance of a single classifier, or when multiple classifiers have complementary strengths and weaknesses. In many cases it performs better but it is not a guarantee.

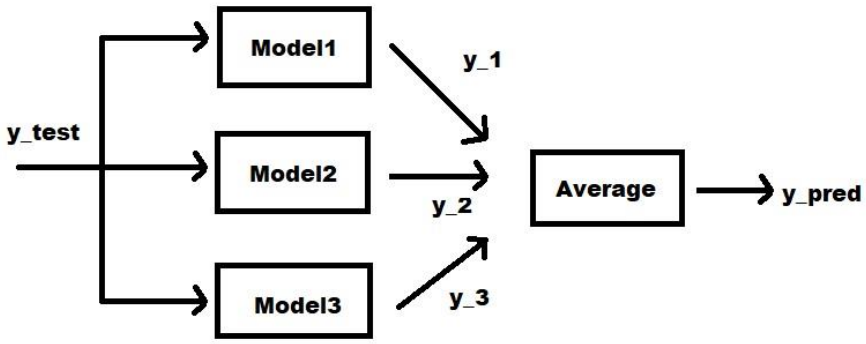
In this section some *basic* ensemble methods are analysed taking into account the best 3 and 5 models:

1) *From_scatch: simple model with RmsProp*
2) *From_scatch deep_3_conv_layers*
3) *Pre-Trained_fine_tuning_2_layers*
4) *From_scatch Adam*
5) *From_scatch deep_3_conv_lay_relu6*

There is the necessity to distinguish which model is used to make a prediction (from scratch or pre-trained typology) because the first requires grey-scale images, tensors of shape (480,640,1) instead the second requires RGB images, tensors of shape (480,640,3), so a proper adaptation is implemented.

The first ensemble method considered is *averaging strategy*. It is mainly used for regression problems. The method consists of building multiple models independently and returning the average of the prediction of all the models. In general, the combined output is better than an individual output because variance is reduced.

The next picture shows schematically how it works.

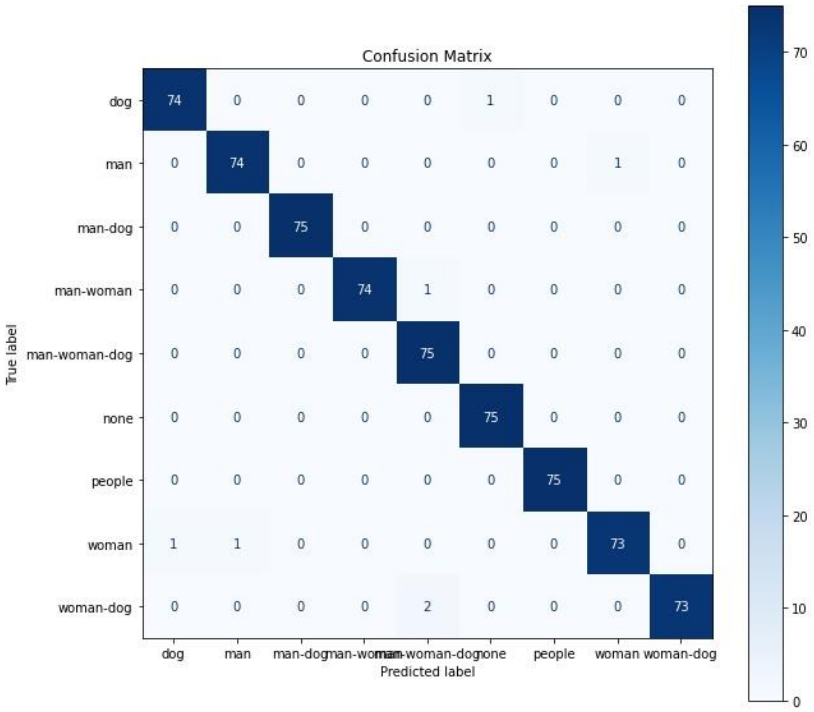


The result obtained are the following:

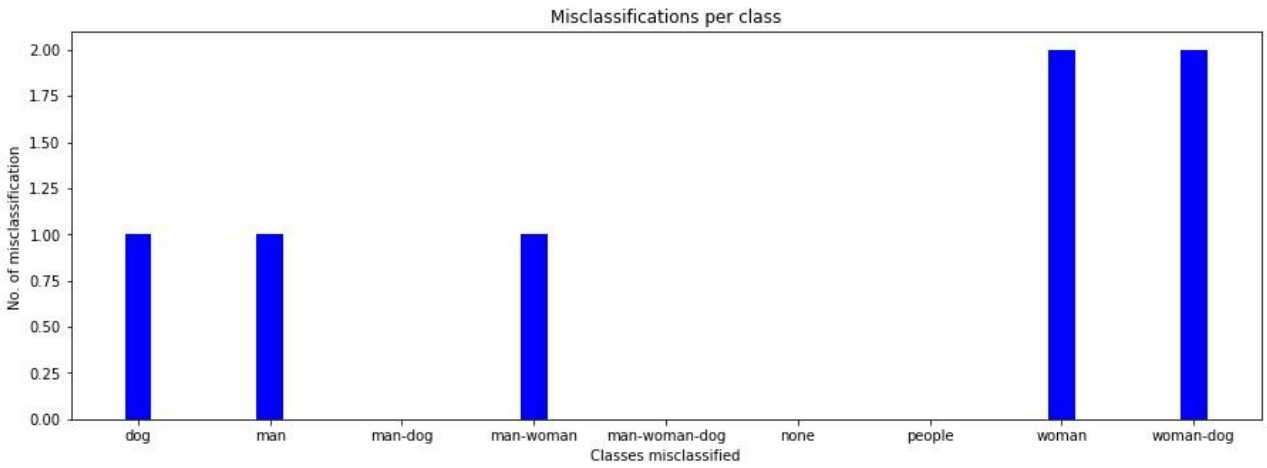
#Models in ensemble	Accuracy	Miss-classification
3	0.9896	7/675
5	0.9852	10/675

It is evident how the combination of the best classifiers has an important impact on the classification accuracy. In the first case the accuracy is almost the 99% on the test set. In the second case, the result is still good because it increases a bit the individual accuracy but the contribution of the 4 and 5 classification models does not take a real benefit as in the first case.

In this page it is present the confusion matrix and the samples incorrectly classified using the three best model in the ensemble.



Total errors :7/675

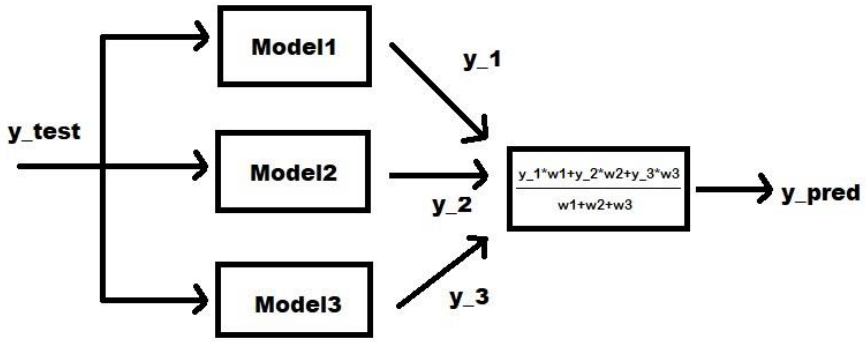


The second ensemble method considered is *weighted average strategy*. The logic is the same as averaging strategy but considering the prediction of each classifier with a different weight. This technique is most used in cases where different classifiers have different performances. The most performing will have a heavier weight respect to the less accurate.

So, the final prediction is given by the formula:

$$Prediction = \frac{\sum_i^n w_i * y_i}{\sum_i^n w_i}$$

where *n* is the number of classifiers in the ensemble, *w* is the vector of weights and *y* is the vector of individual predictions.



In general, the choice of the values of weights vector is important of course. Here are reported the result obtained with different weights:

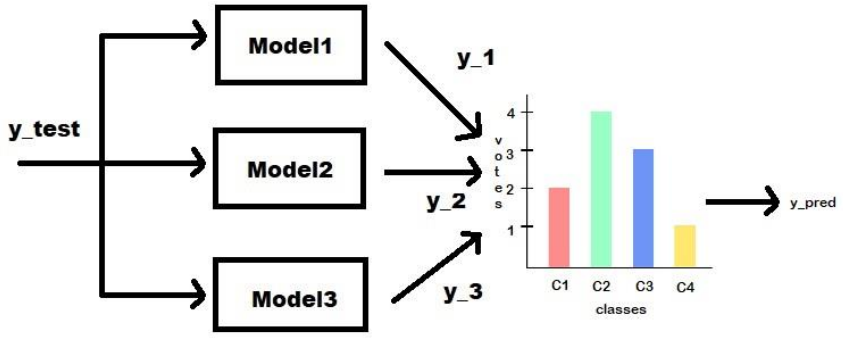
#Models in ensemble	Weights	Accuracy	Miss-classification
3	[0.35,0.40,0.25]	0.9896	7/675
5	[0.25,0.25,0.25,0.125,0.125]	0.9867	9/675
5	[0.05,0.15,0.20,0.30,0.35]	0.9852	10/675

In these trials, there is a no evident advantage to use a weighted average strategy because all the models used have very similar accuracy. In the first case it is almost equivalent to use average strategy because the three best models has almost the same weight importance. Considering 5 models and using a low weight to the last two, the situation is mitigated respect to average strategy. Now the miss-classification are 9 instead of 10.

In the last case, a different combination of weights is voluntarily given to observe how the choice of the weights influence the final decision. In this case given more importance to the last two it is not the best decision.

The confusion matrix and the samples incorrectly classified using the three best model in the ensemble is not reported because it is the same of the average strategy.

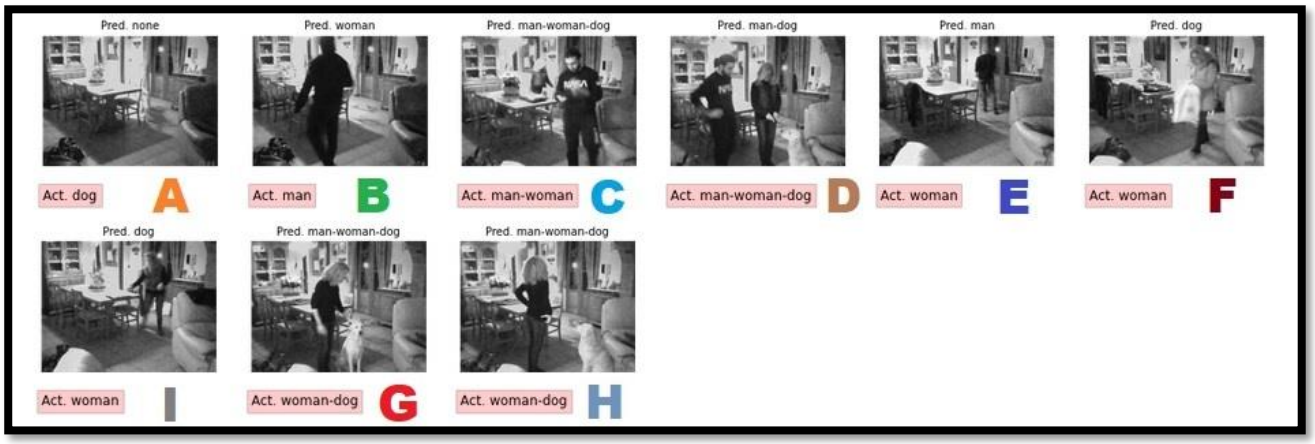
The third and last ensemble method considered is *max-voting strategy*. This is generally used for classification problems and it is one of the simplest ways of combining predictions from multiple learning algorithms. In max-voting, each base model makes a prediction and votes for each sample. Only the sample class with the highest votes is included in the final predictive class.

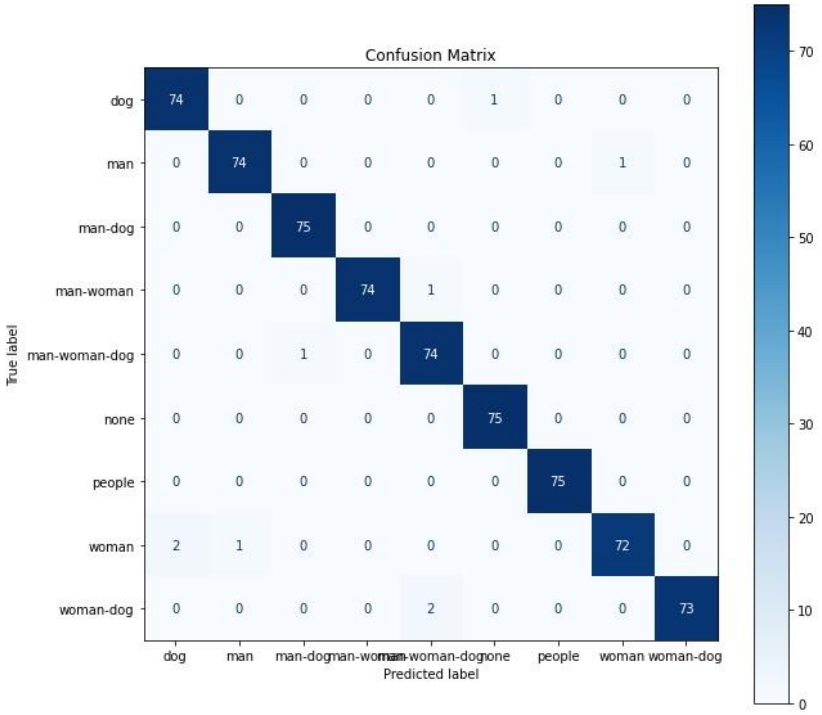


The result obtained are the following:

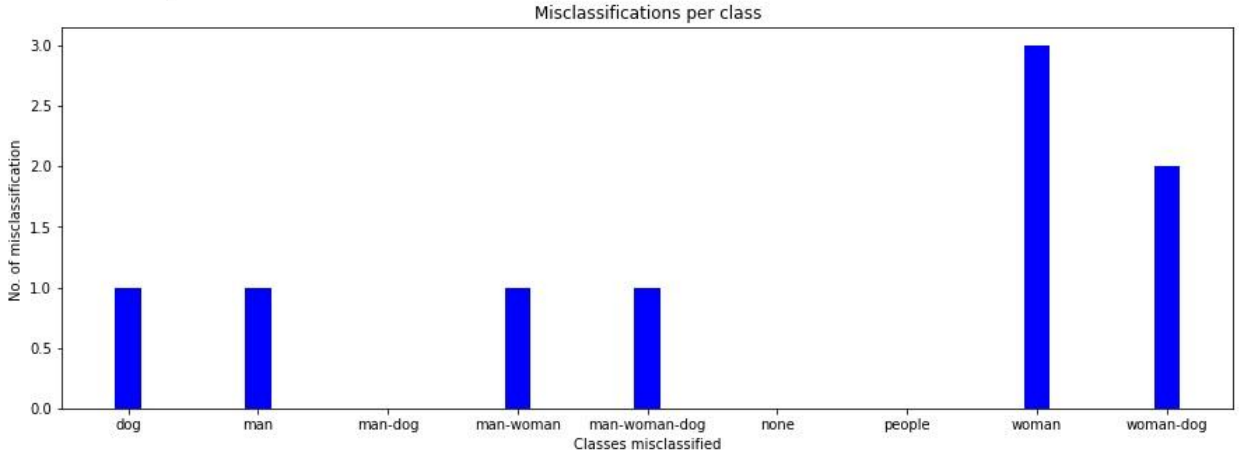
#Models in ensemble	Accuracy	Miss-classification
3	0.9867	9/675
5	0.9822	12/675

The miss-classified images in the first case (3 models considered) are all the images identified with a letter in previous chapter (error analysis) and it confirms that the ensemble method exactly takes as final prediction the class that is most voted. In that case, all the images identified with a letter are miss-classified in at least 2 models, this means that considering 3 classifiers in the ensemble, 2 for sure express a wrong vote (the majority vote). This is also the reason why using 5 models in the ensemble (adding 2 with lower performance) the result is worse.





Total errors :9/675



Deepening: object detection

Object recognition is a bit more complex task than image classification. The ability to detect objects in images and videos enables systems to make informed decisions and respond to real-world situations, making it an essential tool for a wide range of applications. The idea of object detection is to locate the presence of objects with a bounding box and detecting the classes of each one.

Nowadays, there exist two main groups of algorithms:

- Multi Stage Detectors
- Single Stage Detectors

A *multi stage detector* consists of a series of stages designed to perform a specific task in the object detection process. The multiple stages work together to produce the final output, which is a list of bounding boxes around the detected objects in an image. One of the most famous is RCNN (region based convolutional neural network) and it is composed by three stages.

The first stage, also known as a region proposal network, generates a set of candidate regions in the image where objects might be present. Region proposals can be generated using various methods. The most used is *selective search* algorithm. The way the selective search algorithm works is that it applies a segmentation algorithm to find blobs in an image to figure out what could be an object. Selective search recursively combines these groups of regions together into larger ones to create 2000 areas to be investigated. It tries first calculate the similarities between all neighbouring regions, then the two most similar regions are grouped together, and new similarities are calculated between the resulting region and its neighbours. This process is then repeated until the whole object is covered in a single region.

The second stage, known as the classifier, takes these candidate regions and performs more detailed analysis to determine which regions contain objects and what the objects are. This stage often employs a CNN to classify the objects but generally to extract features of interest to use them in the next last stage.

The final stage, also known as the bounding box regression stage, refines the bounding boxes generated by the previous stage to better fit around the objects making use of a SVM (support vector machine).

However, this method has some drawbacks: the selective search algorithm is a complex algorithm, it increases the computational cost and obtaining 2000 region of interest and applying CNN on each one requires too much time. This is not suitable for real-time applications.

Some improvements have been done in Fast RCNN in which a first CNN is applied on the entire image at first then extracting ROI (region of interest). In addition, in the last stage, the SVM layer is replaced with a fully connected layer with a Softmax activation function. This increases the speed by nine-time respect to RCNN. Another improvement has been done in Faster RCNN in which selective search is replaced by a separate neural network to learn region proposal. The main difference between these three models is how they get the regions of interest.

Multi-stage detectors are generally preferred in situations where accuracy is a higher priority, as they tend to produce more accurate results compared to single-stage detectors.

A *single stage detector* combines all the necessary steps in one stage. It takes an input image and directly outputs a set of bounding boxes and class probabilities for the objects present in the image. This makes it faster and more computationally efficient compared to multi-stage detectors.

One famous single stage algorithm is *Single Shot MultiBox Detector* (SSD). This algorithm uses a combination of feature extractor and detector networks to predict the location and class of objects in an image. The feature extractor network is typically a pre-trained CNN, such as VGG or ResNet, that is used to extract rich features from the input image. The detector network is then trained to predict the locations of the objects using the extracted features. The key idea behind SSD is to predict multiple bounding boxes at multiple scales, using anchor boxes, in a single forward pass. Anchor boxes are pre-defined boxes of different aspect ratios and scales that are placed at various positions in the image. The SSD network then predicts the location of the object relative to the anchor box and the class probability for each anchor box. This allows the network to detect objects at different scales in the same image, without the need for multiple resizing and processing steps. A non-maximum suppression mechanism is applied as the last step which decides if there is more than 1 box pointing to the same object, we choose the best fit.

Another famous single stage algorithm is *You Look Only Once* (YOLO). It uses a fully convolutional neural network (CNN) to detect objects in an image and divide the image into a grid of cells, each cell being responsible for detecting objects that fall within that cell. For each cell, the network predicts multiple bounding boxes, along with their class probabilities and the confidence score that indicates the likelihood that the bounding box contains an object. YOLO merges the bounding boxes and removes duplicates to produce the final output, which is a set of boxes with class labels and confidence scores.

It is very similar to SSD but it has some differences :

- it uses a custom network based on the Googlenet architecture name *Darknet*. This Darknet architecture consists of 24 convolutional and 2 fully connected layers.

- the convolutional parts are pre-trained with the Imagenet-1000 dataset and the feature maps that come from the CNN are used to transform to grid cells. Fully connected layers are used to predict bounding box coordinates from these feature map grids.

YOLO is a rapidly evolving object detection algorithm, and new versions are released regularly to improve its accuracy and speed. The main differences between versions regard the architecture, the way to build the bounding box and make predictions, the loss function and the data augmentation techniques used. Each new version of YOLO builds on the previous version, incorporating new ideas and techniques to improve its accuracy and speed.

In general, single-stage detectors are well-suited for real-time applications where speed is a higher priority, such as video processing, but tend to be less accurate compared to multi-stage detectors. Despite this, advances in deep learning have led to the development of single-stage detectors that can achieve high accuracy and are widely used in various applications. [18]

YOLO v.3: usage

In this part it is explored how a pre-trained YOLO v3 works making use of the module OpenCV. The source code is inspired to from a YOLO tutorial at links [19] [20]

DNN (Deep Neural Network) module was initially part of opencv_contrib repository. It has been moved to the master branch of OpenCV repository last year, giving users the ability to *run inference* on pre-trained deep learning models within OpenCV itself. DNN module is not meant be used for training.

Before starting it is necessary to download the following files:

- yolo.config, it contains the network architecture and hyperparameters for the model, including information on the number of layers, filter sizes, and other details that determine how the model processes images to identify objects;
- yolov3.weights, it contains the corresponding learned parameters (weights) trained on a large dataset;
- classes.txt, it contains the classes name to the objects that are going to be predicted.

This particular model is trained on COCO dataset (common objects in context) from Microsoft. It is capable of detecting 80 common objects.

According to the weights and config files, the pre-trained network is instanceted and it is ready to process a new input image. The preparation of the images happens thanks to CV2 API:

```
cv2.dnn.blobFromImage
```

which returns a blob, a multi-dimensional array that can be used as input to a deep neural network. The blob is created by resizing the image to a specified size, converting the image from BGR (Blue Green Red) to RGB, and normalizing the pixel values.

There is the need to define a function: `get_output_layers` to get the name of all the final output layers. Differently from classical CNN, YOLO v3 architecture has multiple output layers containing the predictions. Any output layer is not connected to any next layers.

There is also the need to define a custom function `draw_bounding_box` to draw rectangles over the given predictions. This is done also thanks to some other APIs of CV2 to draw shapes on a pre-existing image.

So once the inference is done, for all the detections from output layers the confidence is observed and if its value is greater the 50% the detection is taken in consideration, saving the information related to width, height, x, y and the class associated to prediction.

Even though the weak detections are ignored, there are lot of duplicate detections with overlapping bounding boxes. So, the idea is to apply a *Non-max suppression* technique to removes boxes with high overlapping. This is obtained thanks to the API:

```
cv2.dnn.NMSBoxes
```

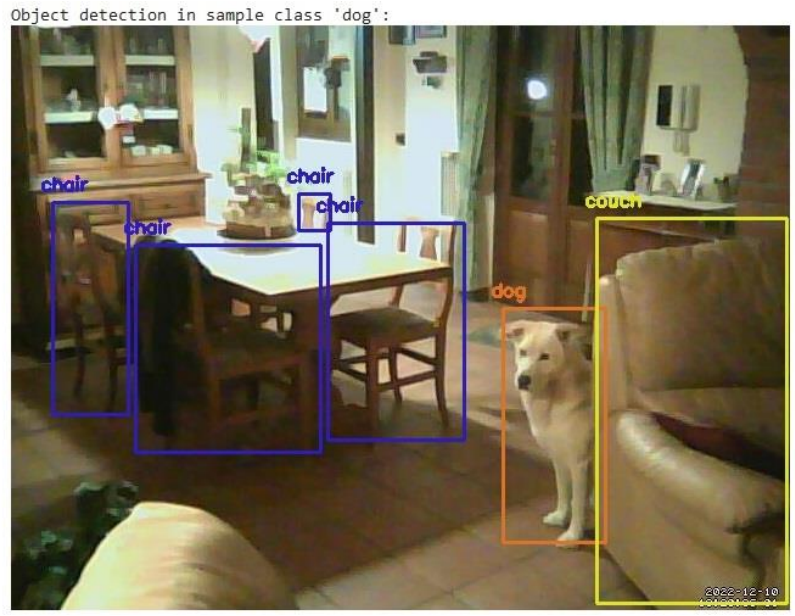
The function takes two arguments:

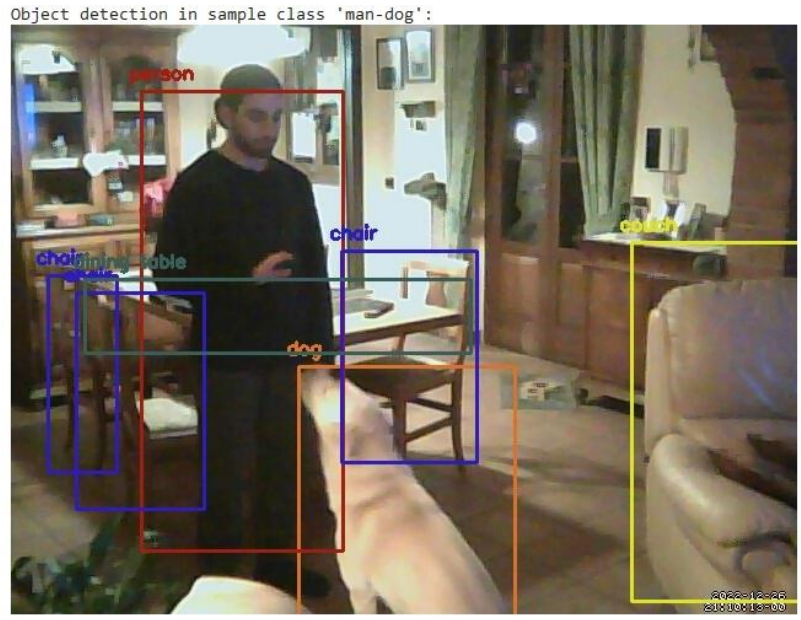
- boxes: a list of bounding boxes, represented as 4-tuples (x, y, w, h), where (x, y) is the top-left corner of the bounding box and (w, h) is its width and height.

- confidences: a list of confidence scores for each bounding box, where the confidence score is a measure of the likelihood that the bounding box contains an object.

The function returns a list of indices of the bounding boxes that should be kept, after applying NMS to eliminate overlapping boxes. This list is finally used for the object detection visualization.

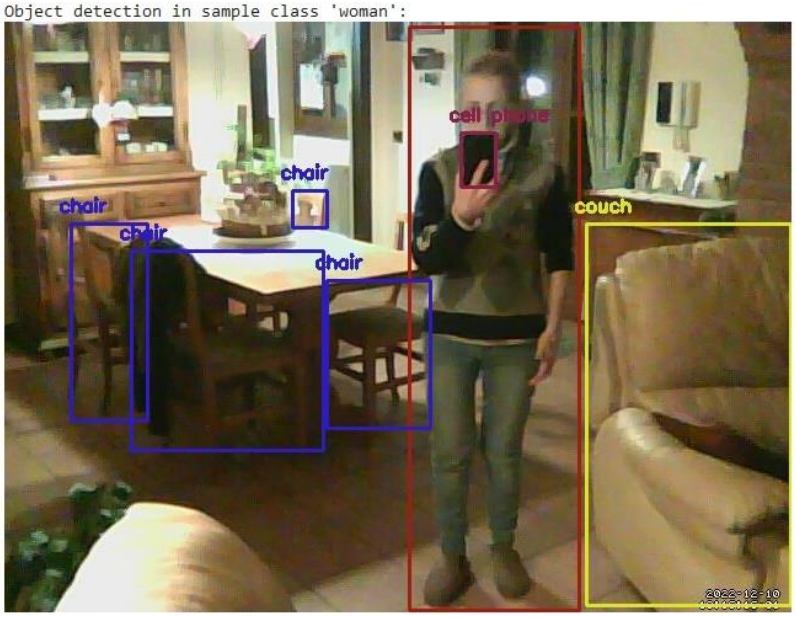
Next are shown some of the results obtained using some samples of the test dataset:





Object detection in sample class 'people':

Object detection in sample class 'woman':



The number of objects detected in these samples is high. In some cases, it is surprisingly able to detect object that are partially covered by other objects like the dining table, in the second picture. Depending on the configuration adopted more region of interest can be considered and detected, so a proper deeper analysis should be done.

Related to this project a further implementation could be developing a custom person detection for each scenario analysed. In particular it could be interesting to try to recognize if the subject that is present in a picture is a specific person who live in that house and if not send an email or alarm.

References

[1] https://viso.ai/deep-learning/object-detection/

[2] https://www.sourcesecurity.com/insights/deep-learning-technology-applications-video-surveillance-co-14319-ga.21460.html

[3] https://www.image-net.org/challenges/LSVRC/

[4] Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. (* = equal contribution) ImageNet Large Scale Visual Recognition Challenge. IJCV, 2015

[5] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9074920

[6] C. V. Amrutha, C. Jyotsna and J. Amudha, "Deep Learning Approach for Suspicious Activity Detection from Surveillance Video," 2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), 2020, pp. 335-339, doi: 10.1109/ICIMIA48430.2020.9074920.

[7] https://www.sciencedirect.com/science/article/abs/pii/S0045790622005419?via%3Dihub#preview-section-abstract

[8] https://gombru.github.io/2018/05/23/cross_entropy_loss

[9] https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e

[10] https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0

[11] https://towardsdatascience.com/how-to-choose-the-right-activation-function-for-neural-networks-3941ff0e6f9c

[12] https://www.tensorflow.org/api_docs/python/tf/nn/relu6

[13] https://www.cs.utoronto.ca/%7Ekriz/conv-cifar10-aug2010.pdf

[14] Xception: Deep Learning with Depthwise Separable Convolutions https://openaccess.thecvf.com/content_cvpr_2017/papers/Chollet_Xception_Deep_Learning_CVPR_2017_paper.pdf

[15] https://iq.opengenus.org/xception-model/

[16] https://arxiv.org/abs/1502.03167

[17] https://keras.io/getting_started/faq/#whats-the-difference-between-the-training-argument-in-call-and-the-trainable-attribute

[18] https://towardsdatascience.com/object-detection-with-convolutional-neural-networks-c9d729eedc18

[19] https://towardsdatascience.com/yolo-object-detection-with-opencv-and-python-21e50ac599e9

[20] https://github.com/arunponnusamy/object-detection-opencv