Neural networks for cats dogs classification

Experimental project for ”Statistical methods for machine learning” course

Mattia Paravisi, registration number: 08395A

Contents

1 Introduction

I have chosen the neural network project because i never worked with neural network for image classification and because i believe this field is extremely fascinating: there are a lot of different structures, layers and ideas to test. In this paper i will report almost every step i have done during this months; i faced a lot of close-end roads, false wins and difficulties but i loved the project: i used technology i never used, i rode a lot of articles and learnt a lot in general.

2 Data analysis

The given data-set contains a folder and two sub-folder: one for cats image and the other for dogs image. In each sub-folder there are 12500 image in JPEG format. Each image represent a cat or a dog in a different situation or context. In the data-set are present images with combinations of humans and cats/dogs (for example Dogs/647) and images with writings and cats/dogs (for example Cats/149); i report this aspect because those images will possibly fool the network in both training and testing phase. One project requirement is "Images must be transformed from JPG to RGB (or grayscale) pixel values and scaled down"; in order to accomplish this i wrote a simple python script (in appendix) 1.

With this script, for each image i checked the correctness and the possibility of conversion to RGB as required (the grayscale and down scaling will be done at a later time). The results of this script are:

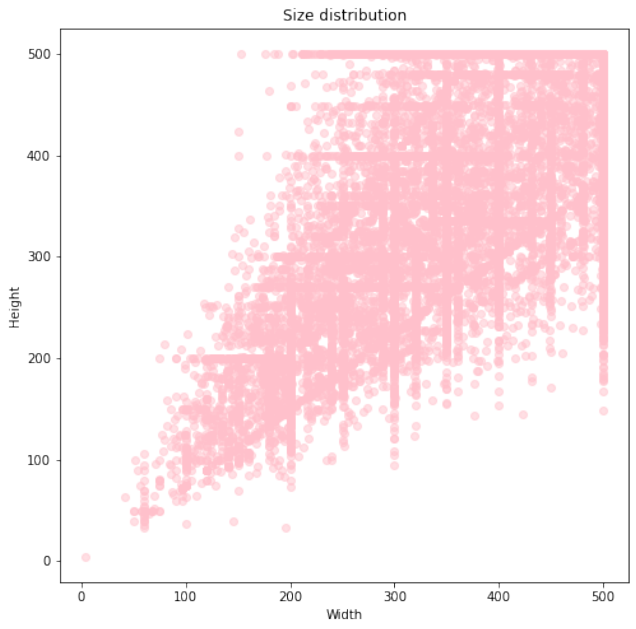
1. There are 2 broken images: CatsDogs/Cats/666.jpg and CatsDogs/-Dogs/11702.jpg
2. Every other image can be converted to RGB

i proceeded deleting the two broken images. The result is: 2 new folders, one for cats and one for dogs, each one with 12499 images. After this step i proceeded with the analysis in particular focusing on the images' sizes; to accomplish this i used the following script (in appendix): 2. For every image i got its size: some of them gave size less than zero so i had to re-get the size with another package. I plotted a scatter plot with the height and the width of every image (so every point is an image) as one can see in figure 1. The distribution is not balanced against image height and width: there is a predominance of images with height/weight values grater than 300px x 300px. Then, in order to accomplish the "downscaling request" i checked for the smallest image in the data-set.

```
img_meta_df[img_meta_df.Width == img_meta_df.Width.min()]
img_meta_df[img_meta_df.Height == img_meta_df.Height.min()]
```

.

Figure 1: Sizes distribution for images in data-set

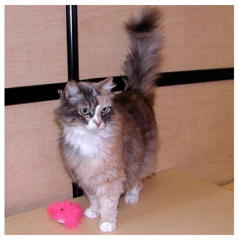


The result of both query is the same image: CatsDogs/Cats/5673.jpg which size is 4px x 4px, i considered this image as an outlier: it is far too small. I then checked for the second smallest image:

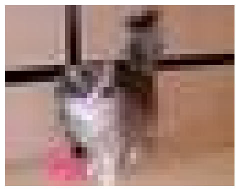
```
img_meta_df.loc[(img_meta_df['Width'] == min(img_meta_df['Width
    ↪ '].where(img_meta_df['Width'] != min(img_meta_df['Width'])
    ↪ )))]

img_meta_df.loc[(img_meta_df['Height'] == min(img_meta_df['Height
    ↪ '].where(img_meta_df['Height'] != min(img_meta_df['Height
    ↪ ']))))]
```

The result are three different images: CatsDogs/Dogs/10733.jpg with the smallest width of 42px and CatsDogs/Cats/6402.jpg, CatsDogs/Dogs/4367.jpg with the same smallest height of 33px. I thought that downscaling all the images at 42px x 33px was not a great idea due to the information loss in the image. For example a 500px x 500px image would began like in figure 2.

I decided to find an acceptable resolution in order to avoid the information loss problem on the image itself and to avoid also to delete a lot of images from data-set(i have to delete images with lower resolution with respect to the one



(a) Original cat

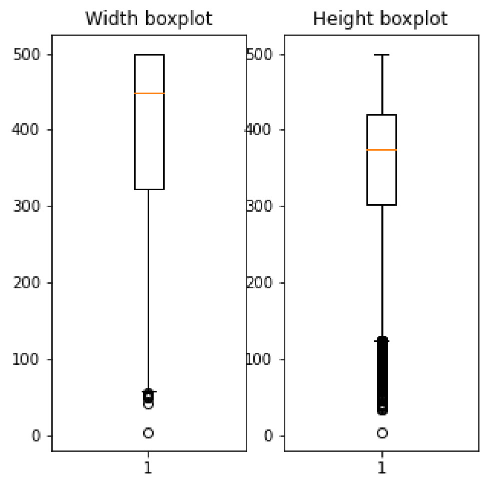


(b) Resized cat

Figure 2: Loss of information in resizing

i chose); for example if the mean acceptable resolution will be 200px x 200px i will have to delete all the images which resolution is smaller than 200px x 200px. If the set of those image is the 10% of entire data-set i consider that resolution a good trade off, on the other hand if the set is the 40% of the entire data-set i don't consider that resolution a good trade off. In order to do this i checked for the quantile of width and height as shown in figure 3.

Figure 3: Height width boxplots



It's easy to find the 0.1 quantile for both the features separately in order to exclude more or less the 10% of the total images; it's not exactly the 10% because i get separately the two quantiles and then i merge them together into a single size. So if the quantile .1 for the height is 250 and for the width is 270 maybe i get more than 10% if i consider the image which size is less than 270px x 250px.

I used the following code and got (240.0, 225.0).

```
img_meta_df.Width.quantile(0.1), img_meta_df.Height.quantile(0.1)
```

So i had to exclude all the images having a width less than 240px or an height less than 225px:

```
df2 = img_meta_df.loc[(img_meta_df['Width'] < 240.0) | (
    ↪ img_meta_df['Height'] < 225.0)]
df2 = df2.reset_index(drop=True)
len(df2)
```

.

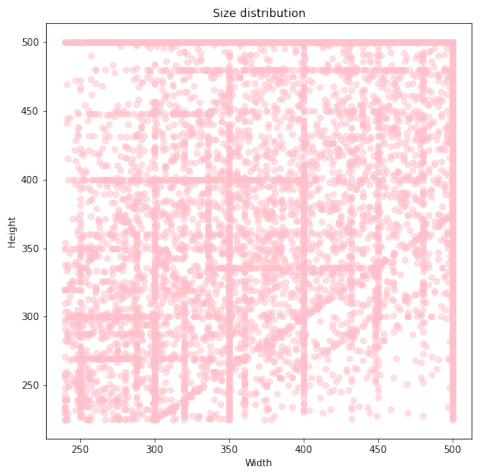
The images involved are 3204, the 12.81% of the starting data-set. I considered this reduction factor a good factor: the initial data-set has 24998 images, a reduction of this kind will not effect the training process. I then created a new dataframe excluding the image in df2

```
cond = img_meta_df['FileName'].isin(df2['FileName'])
df3 = img_meta_df.drop(img_meta_df[cond].index, inplace = False)
df3 = df3.reset_index(drop=True)
```

.

And i plotted the new size distribution to check if it was more uniform than before, the result is shown in figure 4.

Figure 4: Sizes distribution for images in data-set after resizing



I then proceeded with the resizing of the images: i used the script (in appendix) 3. The result is a new folder "CatsDogs_resized", this folder contains two other folders: "Cats_resized", "Dogs_resized". I checked for errors occurred during the resizing with the script (in appendix) 4 and i checked if all the files are present and have all the same size with the script (in appendix) 5. The output is: ”Total Nr of Images in the dataset: 21794 ↵ (240.0, 225.0)”, that is what i have expected. I lastly used a tool called SplitFolder to split the new folder in the train, validation and test folders using parameters train = 80% test = 10% and validation = 10% obtaining the following structure:

```
NN_data
├── test
│   ├── Cats_resized
│   └── Dogs_resized
├── train
│   ├── Cats_resized
│   └── Dogs_resized
└── val
    ├── Cats_resized
    └── Dogs_resized
```

I decided to do a train/test/validation split because during the training process in TensorFlow there is the possibility to use a validation set to have a validation loss during the training process: this is useful to have a loss history during the training to check for overfitting and visualizing it. In fact if one splits in train test and uses the evaluate method on the test part only gets a single value that is not useful if one wants to plot the loss during the entire training process.

3 Neural Network

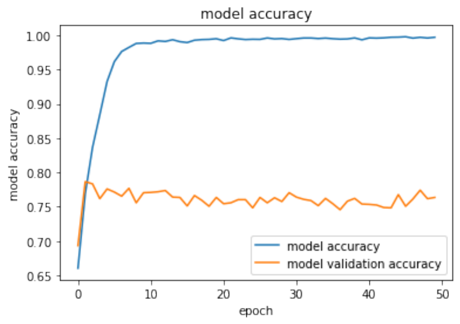
As i mentioned in introduction before this project i never practically used neural networks of any kind so i decided to apply a ”little step” approach in order to have a deeper comprehension (and a correct comprehension, i hope) of some aspects. I wanted to put a base line to my project, in order to do this i built the easiest network i could came up with. The type of neural network i chose is a convolutional neural network; i decided to use this type of network because in a lot of article and online pages this type of network is recommended to deal with images classification problems. In particular in this paper i analyzed the effect of:

1. Image augmentation in training phase
2. Epochs number
3. Batch size
4. Layers

on the networks i used. The first network i have implemented is reported in 6: i defined the model as sequential, added two convolutional layers with increasing filters number, added a ReLU activation function (i found that ReLU is a great function in CNN as explained in chapter 2, section 2.1 in [1]) on all layers except for the last one that is a sigmoid, this is because the last activation give the probability for an image to be a dog or a cat (for a multi label classification i would have used a softmax). I then chose a dense layer with size 64 in order to have a faster training time and added the last layer with a single neuron to get the classification. The loss i'm using is the binary cross entropy, i chose this loss for no particular reason, i rode of it in some articles. In this first attempt i used the images in the data-set re-scaling their pixel's value between 0 and 1 in order to obtain a gray scale image as input to the network (in RGB a pixel can have values in the range 0-255, if one operate a division by 255 gets a 0-1 range which is gray scale).

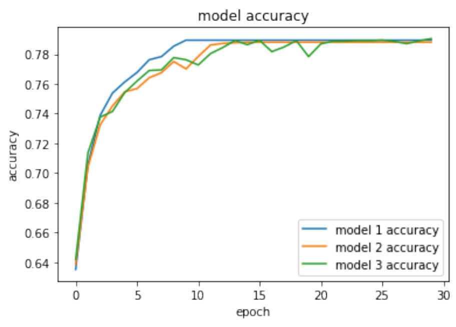
Overfitting Without any other operation on the images the first model's training accuracy was very high ~ 0.9969 with a very low loss ~ 0.017; on the other hand the validation accuracy was ~ 0.7637 and loss ~ 4.7741: the model has overfitted the data as shown in image 5.

Figure 5: Overfitting in first model

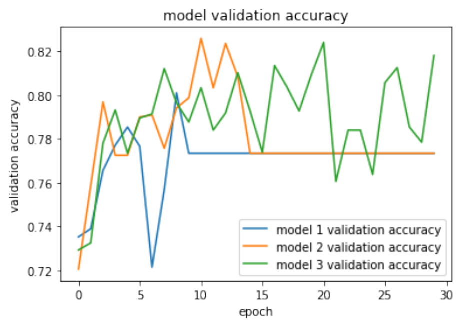


In order to solve this issue i could have decreased the number of epochs until the accuracy and the validation accuracy had been similar; using this method i should have set the epochs number to 3: after this number of epochs the accuracy starts increase and the validation accuracy remains the same, this method is clearly unfeasible. Alternatively i found a methodology and a layer to get rid of this problem: images augmentation and dropout layer. Image augmentation consists in applying transformations like rotations, zoom, flips to the images before feeding them into the network; one could also use generative adversarial network in order to generate new images as reported in [2] (chapter 3, section 2) but i think this is a bit out of the scope of this report. The dropout layer turns off some inputs in training as reported in Keras documentation: "randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting". I applied both the methods: image augmentation for training and dropout layer starting from the following phase.

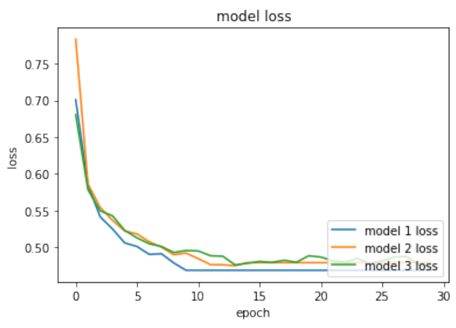
Epochs number analysis The second phase was the epochs number analysis on the network reported in 8 (as you can see i added a dropout layer, the images augmentation is done in training/validation/testing generator). The number of epochs is an hyperparameter defining the number of times the algorithm will go through the entire data set, i suppose that the more epochs we use, the more accurate the algorithm is due to the fact that it sees a lot of times the examples and has the possibility to adapt the trainable parameters. On the other hand, the more the algorithm go through the data set, the more it tends to overfit: it will learn the specific examples in the data set, as reported in previous section. I trained the same model on the same training set (and evaluated on the same validation set) using a different epochs number: 10, 15, 30. I chose this range of values for both hardware limitations and time consumption, i know that giving more epochs is better. The results are reported in figure 6 (straight line are due to the fact that i had different results length and i had to repeat the last value of the shortest array to have a precise visual comparison). It's interesting



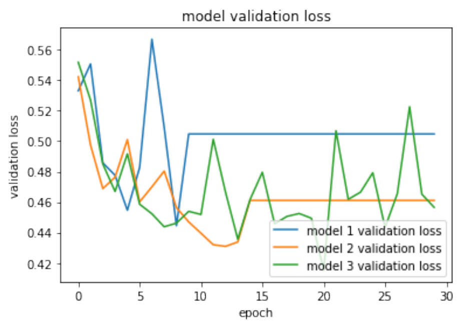
(a) Accuracy



(b) Validation accuracy



(c) Loss



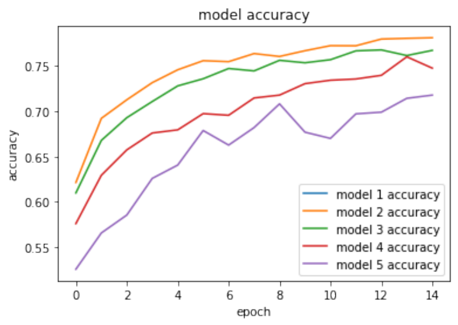
(d) Validation loss

Figure 6: Epochs number analysis. Model 1: 10 epochs, model 2: 15 epochs, model 3: 30 epochs

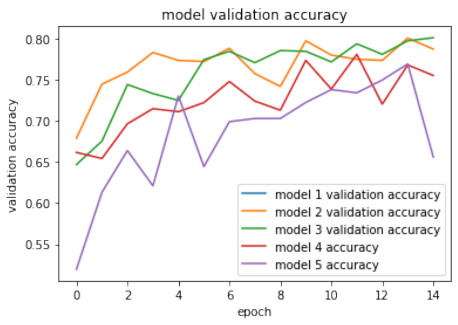
to point that in the case of training accuracy and training loss the values are similar for all models, the differences are more clear in validation: if we use more epochs the model with the maximum number of epochs is more accurate and achieve less loss with respect to other models in which we used less epochs. Another thing to notice is that the model is very simple and maybe its learning potential is limited, so it's impossible for this model to achieve a better accuracy even if we let the model train for a lot of epochs. Giving more epochs to the models it's useful and helps to increase accuracy and decrease loss in validation and testing but it uses more time in training the model; on the other hand more epochs means more overfitting possibility, to avoid this it's possible to use image augmentation as pre processing and dropout layers in the model (or use an early stopping callback). **In the following section i will use a number of epochs equal to 15, this is only because my hardware couldn't handle extended training session. I know that giving a lot of epochs to train the model is better!**.

Batch size analysis The third phase was the batch size analysis, i used the same network reported in 8. The batch size is the number of images (samples) fed in the network before doing the weights adjustments. For example if i have 2000 images and a batch size of 100 the algorithm takes the first 100 (from 0 to 99) and trains the network; then it adjust the weights using gradient descent and then takes the following 100 examples (from 100 to 199) and so on. Using a little batch size allows the network to adjust the weights more frequently, for this reason i supposed (before training the network) that a network which uses a small batch size would have been more accurate than a network which uses a large batch size. I trained 5 models using same training/validation/test image generator with different batch sizes: 16, 32, 64, 128, 256. The results of accuracy and loss for all the models are reported in 7 (model 1 is not present due to an error: "Your input ran out of data" apparently i was not able to generate the correct number of batches). One can see that the models which batch size was smaller (model 2 and 3) are able to achieve a smaller loss and an higher accuracy both in training and validation. This can be a clue to confirm what i said: the more a model have a chance to adapt its weights the more it began accurate and the less loss it achieve.

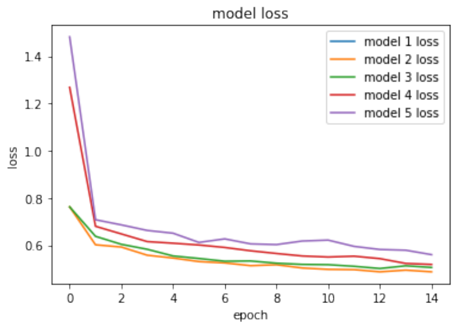
Increasing-decreasing filters number After all the analysis reported before i decided to focus on the effect of layers in the network; for example if one looks at the model in appendix will notice that i used naturally an increasing number of filters in convolutional layers, but maybe is better to use a decreasing number of filters. The network uses a window of $N \times N$ pixels (in my case is a 3×3 window) to scan the image, the window's weights are randomly assigned at the beginning and are adjusted during the training process. In this way the network obtains the filters that it will use during the test phase. Every layer of filter is used in the network to recognize a certain pattern, i imagine that if one use an increasing number of filters firstly they recognize elementary patterns as corners, shapes ... subsequent filters will combine those elementary patterns to create more complex patterns. As we move forward in convolutional layers



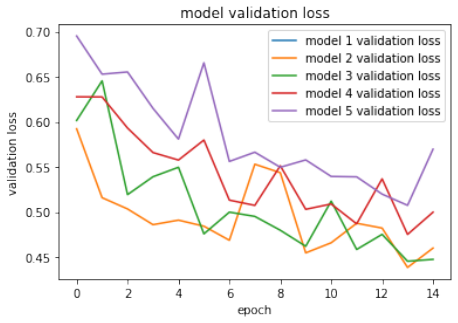
(a) Accuracy



(b) Validation accuracy



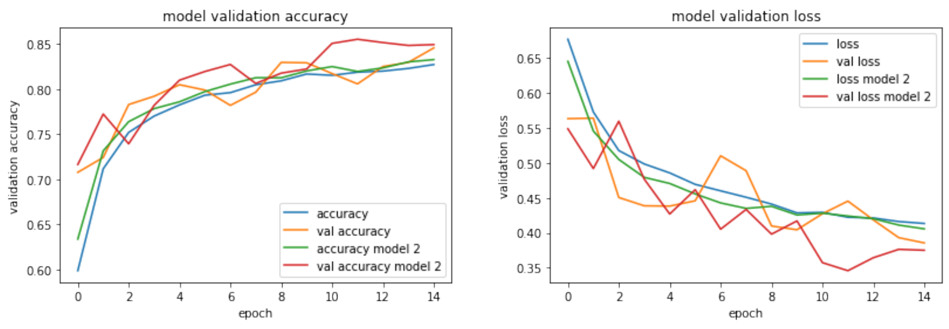
(c) Loss



(d) Validation loss

Figure 7: Batch size analysis. Model 1: batch size 16, model 2: batch size 32, model 3: batch size 64, model 4: batch size 128, model 5: batch size 256

as we'll need more filters because the patterns are every time more complex (more combinations of them), for this reason we need to capture every time more features and maybe for this same reason is better to choose an increasing filters number order. Another clue for choosing the increasing order is that if use a decreasing order and we learn a lot of patterns in input image but, when we apply a max pooling layer to the image, the number of information will be reduced and maybe a lot of the initial patterns are gonna be lost; furthermore if we use a decreasing order the last convolution layer will have less filters to apply to the image in order to classify it and maybe this will result in a higher loss. For this reason i expect that a network with increasing number of filters will perform better than a network with decreasing number of filters. I trained 2 networks, the first is reported in 7, the second is the same network with increasing number of filters, i report the results in the image 8. The difference is not huge but one can see that if is used an increasing number of filters the loss in validation and training is lesser with respect to the other network and the accuracy in validation is higher in both validation and training. This analysis confirm my initial hypothesis: using an increasing number of filters leads the network to better results.



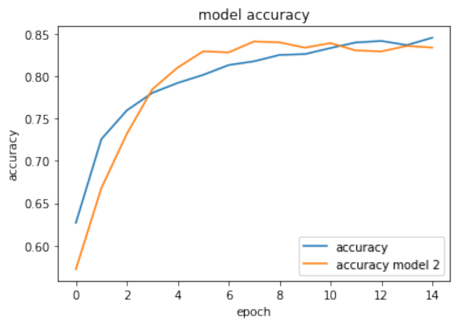
(a) Validation Accuracy (b) Validation Loss

Figure 8: model 2: increasing number of filters, the other is the network with decreasing number of filters

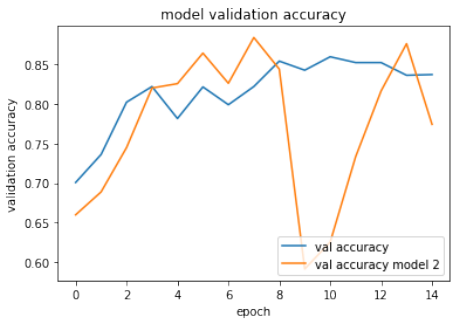
Number of filters Once established the best filter's number order i tried to understand if using a big number of filters was useful for the dogs/cats classification, for this purpose i trained two networks: one with six convolutional layers (32 filters . . . 1024 filters) and one with three layers (32 filters . . . 128 filters). The images i am using in this project are not in high resolution so my hypothesis was that adding a lot of layers would be useless because for every layer the network apply a max pool which halves the dimension of the input in the next layer (using 1024 filters on few pixel is useless). The results of the accuracy and the loss are reported in figure 9, it can be noticed that no improvements are given by the model with a high number of filters; furthermore one can point that the model with 1024 filters is probably less stable: in the graph is present an high drop of accuracy and a high loss in the middle of the training, but this can be caused by other factors. From this analysis and for the current project settings one can deduce that using a high number of filters is useless.

A deeper analysis is useful in order to understand the best number of filters to use; i have carried out this analysis too, the results are reported in figure 10, from the graphs is easy to deduce the optimal maximum number for the filters; in this case the best number of filters is 512: it achieves better results in every metric; there's a little drop in validation accuracy and validation loss but as one can see the green line was almost always above the orange line in accuracy and under in loss.

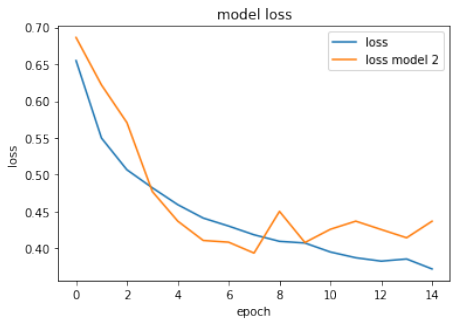
Dropout layer Dropout layer are very useful to avoid overfitting but also for model generalization. Dropout layer randomly sets the outgoing edge of neurons to 0 at each training phase update, for this reason the model doesn't use its full potential during the training phase but use it during the test phase (or validation). Removing edges during the training is used, as i mentioned, for



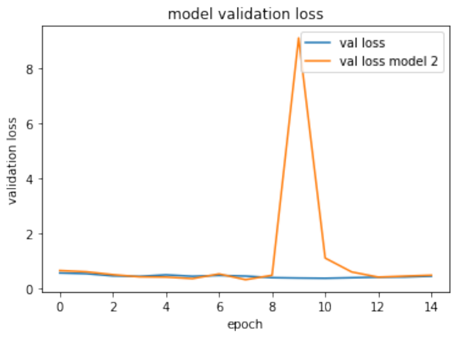
(a) Accuracy



(b) Validation accuracy



(c) Loss

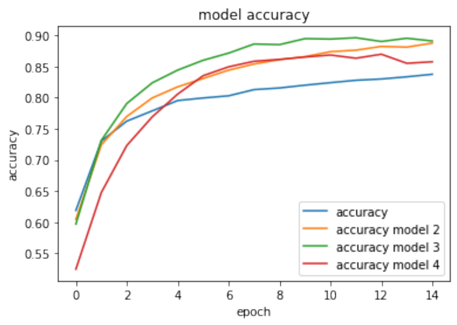


(d) Validation loss

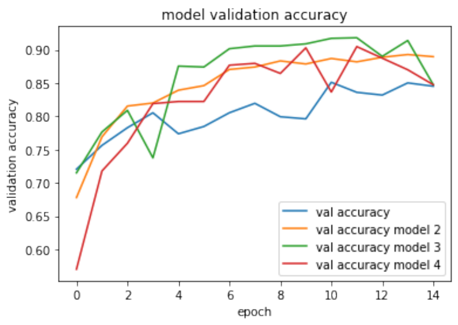
Figure 9: model 1: three layers, model 2: six layers

generalization. In the code provided till now i only use dropout before the last dense layer; i asked myself if it's useful to use dropout layer in every layer and if it's better to increase or decrease the dropout value in every layer. Starting from the last question, i have trained (as usual) four networks each one with a different dropout architecture, one example is reported in 9, other networks are the same with different dropout values: decreasing dropout by a factor 0.1 from a starting dropout value of 0.5, increasing dropout from value 0.3 by a factor 0.05 (to check if starting with an high dropout is useful), increasing dropout from value 0.1 with a factor 0.1. My hypothesis here is: is better to start with a low dropout value and then increase it, this is because if we set an high dropout since the first layer the network will lose a lot of information before even starting learning; on the contrary leaving a low dropout at the beginning will permit to the network to learn with (almost) full information and generalize in the end where the dropout is higher. Results are reported in 11; one can see that model 4, after an adequate number of iterations, performs better in every value. Starting with a low dropout value and increasing it is better than all the other options.

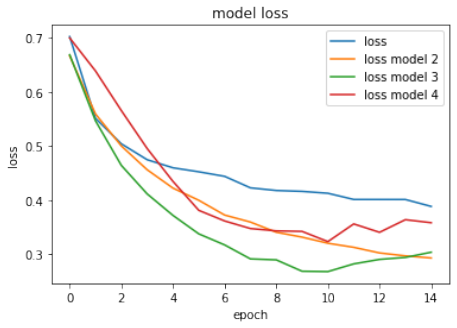
At this point i tested (but i don't report the images or the paper will be too



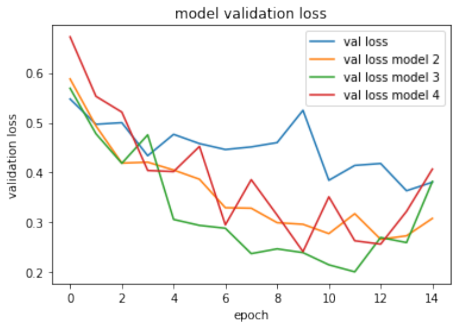
(a) Accuracy



(b) Validation accuracy



(c) Loss



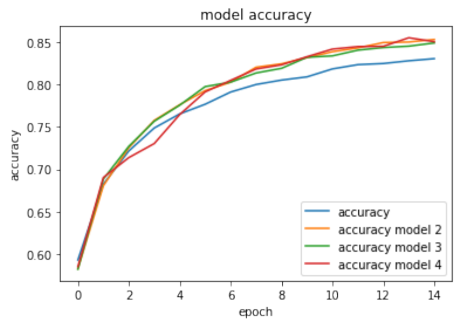
(d) Validation loss

Figure 10: model 1: max 128, model 2: max 256, model 3: max 512, model 4: max 1024

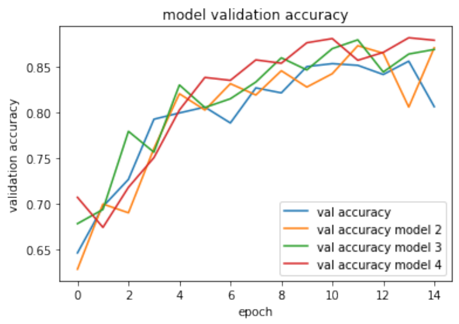
long) what the best dense layer size could be and i got that 256 is a good choice among other one $64, 128, 512$ because it keep small loss and high accuracy with respect to the other models. In particular the network with 256 as dense layer size obtains, after 15 epochs, a training accuracy of ~ 0.87 with a loss of ~ 0.31 and a validation accuracy of ~ 0.90 with a loss of ~ 0.25 while the second best model obtains in training an accuracy of ~ 0.87 with a loss of ~ 0.32 and a validation accuracy of ~ 0.88 with a loss of ~ 0.28. Then i tried how much deep i could go with the dense layer starting from a size equal to 256; i tried with the following dense layer concatenations (every number is a dense layer size. I've applied the following choice: keep adding layers till the loss starts to increase with respect to the previous layer):

- $256 \rightarrow 128 \rightarrow 1$
- $256 \rightarrow 128 \rightarrow 64 \rightarrow 1$
- $256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 1$

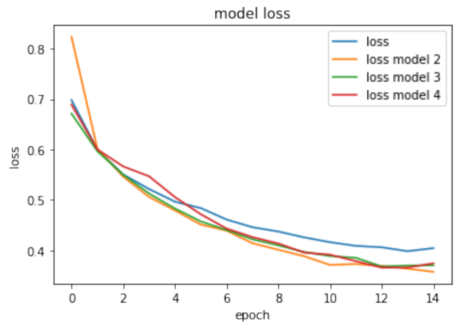
i also tried using dropout after every layer or only before the one sized dense



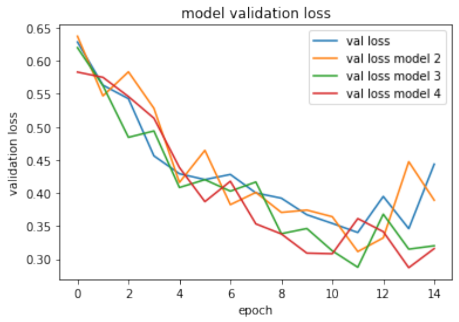
(a) Accuracy



(b) Validation accuracy



(c) Loss



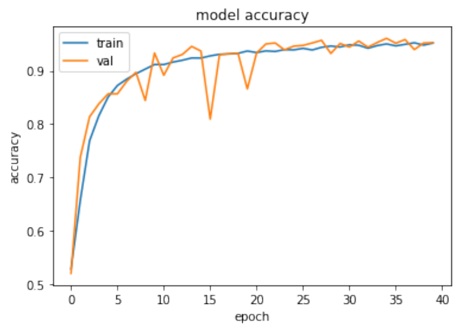
(d) Validation loss

Figure 11: model 1: Decreasing from dropout 0.5 (decreasing factor: 0.05), model 2: Decreasing from dropout 0.5 (decreasing factor: 0.1), model 3: Increasing from dropout 0.3 (increasing factor: 0.05), model 4: Increasing from dropout 0.1 (increasing factor: 0.1)

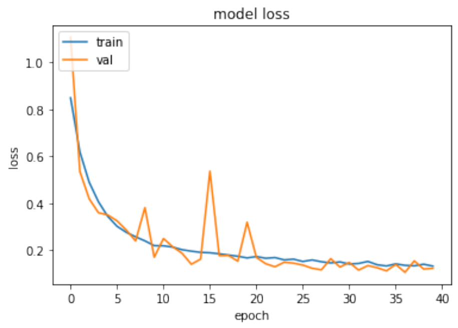
layer.

The best model Following the modus operandi reported in this paper i came up with a model which follows all the hints received from my previous analysis, i report my final model on 10. I trained, validated and tested the model to be sure of the results (i used an 300 epochs for this last step with an early termination rules to avoid useless training phases), as reported in image 12:

- Training accuracy after 40 epochs (early terminated): 0.9520, loss: 0.1328
- Validation accuracy after 40 epochs (early terminated): 0.9527, loss: 0.1234
- Testing accuracy after 40 epochs: 0.9569, loss: 0.1110



(a) Best model training and validation accuracy



(b) Best model training and validation loss

Figure 12: Best model performance

4 Cross validation

I will keep this section relatively small because cross validation is just a technique to estimate model error and i don't have done any other experiment. Despite the previous part, in which the model loss function could be freely chosen, in this part the required loss function is a zero-one loss. To handle this requirement i just encoded the zero one loss: this loss assigns 0 loss to a correct guess and 1 loss to an incorrect guess; in symbols:

$$L(i,j) = \begin{cases} 1 & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

in keras one can give to the compile method a user defined loss with signature *myloss(y_true, y_pred)*, so it's easy, given two tensors and the Tensorflow's operations, to write the loss; if one suppose to have the following true array (tensor) [1, 0, 0, 1] and the predicted array (tensor) [1, 0, 1, 0] with the same size and which every place refers to the same input to the net, applying a subtraction and squaring the subtracted result gives the desired output:

1. $[1, 0, 0, 1] - [1, 0, 1, 0]$
2. $[1 - 1, 0 - 0, 0 - 1, 1 - 0]$
3. $[0, 0, -1, 1]$
4. $[0, 0, -1, 1]^2$
5. $[0, 0, 1, 1]$

in code:

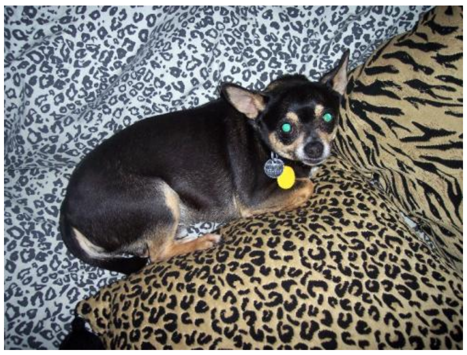
```
def my_loss_fn(y_true, y_pred):
    casted = tf.subtract(y_true, y_pred)
    squared = tf.square(casted)
    return squared
```

the only attention one has to put is on the operation type, they all must be differentiable in Tensorflow. To handle the split in 5 parts (as required) i created a dataframe with the full path to every image in the dataset with their true label, i then performed the split with scikit *KFold* method and i gave my loss to the compile method as i reported, i trained the best model for five times with 4 folds and tested with the remaining fold, i then collected all test losses and performed the mean. I report the error estimation (test error mean): 0.10 with 10 epochs of training (with more epochs the result could be lower and more precise, i invite you to consider this as a proof of understanding of the kfold concept; in a typical situation i would let the network trains for an high number of epochs using a early termination callback). I also add the complete series of testing error obtained: 0.08521, 0.08964, 0.1499, 0.09831, 0.08235.

5 Conclusions

I really enjoyed this project. I also think that a lot of improvements can be done to my network: i wanted to test (but i didn't) the effect of stacking conv layer with same size, use grid search to find best hyperparameters, use other layers in my network, other optimizers, change activation functions ... in general i am proud of this results: i started with almost zero knowledge of CNN and was able to build a pretty accurate network (in my opinion). I know i did not used sophisticated techniques to find parameters to use, but the focus on this project for me was to learn something new doing things and looking at the effect of those changes, not to achieve the maximum accuracy and the minimum loss. To conclude in a funny way: i checked for the images my network was more uncertain of and, considered that a probability ~ 0 means absolute certainty for the network that an image is a cat and probability ~ 1 to be a dog, i found an image 13 which probability is 0.49996376: my network was fooled by a Chihuahua.

Figure 13: It's a dog or a cat?



A Code snippets

Listing 1: Image format check

```
folder_path = 'CatsDogs'
extensions = []
file = open('out.txt', 'w')
for fldr in os.listdir(folder_path):
    sub_folder_path = os.path.join(folder_path, fldr)
    for filee in os.listdir(sub_folder_path):
        file_path = os.path.join(sub_folder_path, filee)
        try:
            img = Image.open(file_path)
        except:
            print(f'image {file_path} broken')
        try:
            if img.mode in ("RGBA", "P"):
                img = img.convert("RGB")
            file.write(f'{file_path}: can be converted from {img.
                ↪ mode} to RGB\n')
        except Exception as e:
            print(f'Format not recognized: {e}')
            continue
```

Listing 2: Data set creation

```
folder_path = 'CatsDogs'
img_meta = {}
for fldr in os.listdir(folder_path):
    sub_folder_path = os.path.join(folder_path, fldr)
    imgs = [str(img) for img in Path(sub_folder_path).iterdir() if
        ↪  img.suffix == ".jpg"]
    for f in imgs:
        width, height = imagesize.get(f)
        if width == -1 or height == -1:
            print(f"broken size {f}")
            image = Image.open(f)
            width, height = image.size
            print(f"now {width}, {height}")
        img_meta[f] = (width, height)

img_meta_df = pd.DataFrame.from_dict([img_meta]).T.reset_index().
    ↪ set_axis(['FileName', 'Size'], axis='columns', inplace=
    ↪ False)
img_meta_df[["Width", "Height"]] = pd.DataFrame(img_meta_df["Size
    ↪ "].tolist(), index=img_meta_df.index)
```

```
img_meta_df["Aspect Ratio"] = round(img_meta_df["Width"] /
    ↪ img_meta_df["Height"], 2)
```

Listing 3: Images resizing

```
folder_path = 'CatsDogs'

basewidth = 240
baseheight = 225
i = 0

for fldr in os.listdir(folder_path):
    sub_folder_path = os.path.join(folder_path, fldr)
    for filee in os.listdir(sub_folder_path):
        file_path = os.path.join(sub_folder_path, filee)
        if file_path.endswith(".jpg"):
            img = Image.open(file_path)
            img = img.convert('RGB')
            img_resized = tf.image.resize(img, (baseheight,
                ↪ basewidth), preserve_aspect_ratio=False)
            arr_specs = file_path.split("/")
            #0 -> CatsDogs
            #1 -> Cats / Dogs
            #2 -> name
            tf.keras.utils.save_img(f'./{arr_specs[1]}_resized/{
                ↪ arr_specs[2]}', img_resized.numpy(), data_format
                ↪ ='channels_last', scale=True)
```

Listing 4: Resized image format checking

```
folder_path = 'CatsDogs_resized'
extensions = []
file = open('out.txt', 'w')
for fldr in os.listdir(folder_path):
    sub_folder_path = os.path.join(folder_path, fldr)
    for filee in os.listdir(sub_folder_path):
        file_path = os.path.join(sub_folder_path, filee)
        #print('** Path: {} **'.format(file_path), end="\r", flush
            ↪ =True)
        try:
            img = Image.open(file_path)
        except:
            print(f'image {file_path} broken')
        try:
            if img.mode in ("RGBA", "P"):
                img = img.convert("RGB")
            file.write(f'{file_path}: can be converted from {img.
                ↪ mode} to RGB\n')
        except Exception as e:
            print(f'Format not recognized: {e}')
            continue
        if img.format != "JPG" and img.format != "JPEG":
            print(f'format error {file_path}: {img.format}')
```

Listing 5: Resized image size checking

```
folder_path = 'CatsDogs_resized'
img_meta = {}
for fldr in os.listdir(folder_path):
    sub_folder_path = os.path.join(folder_path, fldr)
    imgs = [str(img) for img in Path(sub_folder_path).iterdir() if
        ↪  img.suffix == ".jpg"]
    for f in imgs:
        width, height = imagesize.get(f)
        if width == -1 or height == -1:
            print(f"broken size {f}")
            image = Image.open(f)
            width, height = image.size
            print(f"now {width}, {height}")
        img_meta[f] = (width, height)
img_meta_df = pd.DataFrame.from_dict([img_meta]).T.reset_index().
    ↪ set_axis(['FileName', 'Size'], axis='columns', inplace=
    ↪ False)
img_meta_df[["Width", "Height"]] = pd.DataFrame(img_meta_df["Size
    ↪ "].tolist(), index=img_meta_df.index)
img_meta_df["Aspect Ratio"] = round(img_meta_df["Width"] /
    ↪ img_meta_df["Height"], 2)
print(f'Total Nr of Images in the dataset: {len(img_meta_df)}')
print(img_meta_df.Width.mean(), img_meta_df.Height.mean())
```

B Neural network code

Listing 6: First network

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(image_height,
    ↪ image_width, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 7: Filter analysis network

```
model = Sequential()
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), input_shape=(image_height,
    ↪ image_width, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.50))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 8: Base line network

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(image_height,
    ↪ image_width, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.50))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 9: Network for dropout analysis

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(image_height,
    ↪ image_width, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.45))
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.40))
model.add(Conv2D(256, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.35))
model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.30))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 10: Best network

```
model=Sequential()
model.add(Conv2D(32,(3,3),activation='relu',input_shape=(
    ↪ image_height, image_width, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.1))
model.add(Conv2D(64,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.2))
model.add(Conv2D(128,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))
model.add(Conv2D(256,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))
model.add(Conv2D(512,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(256,activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.55))
model.add(Dense(128,activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.60))
model.add(Dense(64,activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.65))
model.add(Dense(1,activation='sigmoid'))
```

References

[1] Zhi Chen and Pin-Han Ho. Global-connected network with generalized relu activation. *Pattern Recognition*, 96:106961, 2019.

[2] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.