UNIVERSITY OF PISA

Multimedia Information Retrieval and Computer Vision

Year 2021/22

https://github.com/rosselladedo/DOG_BREEDS_SEARCH_ENGINE

DOG_BREEDS_SEARCH_ENGINE

Rossella DE DOMINICIS

Giacomo PIACENTINI

Gaetano Niccolò TERRANOVA

Project Description

1. Implement index “**LSH**” to allow fast similarity search on deep features and create an image search engine on top of it
2. Use the pre trained Deep Neural Network “**DenseNet121**” to extract features from the dataset “**Dog_Breeds**” and the distractor *“MirFlickr”*
3. Index the extracted features using your search engine
4. Measure the retrieval performance of the image search engine
    - Pick a fixed set of queries, of your choice, taken from you test set
    - Measure the *mAP* we have seen in Lab Session 4 or the sklearn version
    - Measure the *average query time*
5. Finetune the “**DenseNet121**” for “**Dog_Breeds**”
6. Use the finetuned “**DenseNet121**” to extract features from “**Dog_Breeds**” and “MirFlickr”
7. Index the new extracted features using your search engine
8. Measure the retrieval performance of the image search engine using the new features
9. Compare the performance of the two features
10. *(Optional)* Build a web based user interface for your web search engine

Introduction

A search engine is an automatic system that analyzes a set of data, returns an index of the available contents automatically classifying them according to statistical-mathematical formulas that indicate the degree of relevance given a query.

“DOG_BREEDS_SEARCH_ENGINE” is a search engine that recognize dog breeds image inserted by the user, from a set of 120 different dog breeds.
We used a pre trained Deep Neural Network “DenseNet121” to extract features from the dataset “Dog_Breeds” and the distractor “MirFlickr”. We decided to extract the features using the finetuned model, we implemented our version of LSH-Index to enable fast similarity search on deep features. We measured the performance with the use of LSH index and with a brute force approach, to compare the results. To explore our solutions, we create a simple web-interface that after the insertion of an image, gives the 10 most similar results.

Notebook

- *FeatureExtraction*: extraction of features from normative model and our model finetuned;
- *FineTuning:* fine Tuning means taking weights of a pre-trained neural network (DenseNet121) and using them as initialization for our model;
- *LSH:* implementation of our version of the LSH index (binary and not binary)
- *maPPerformance:* includes the comparison of all the performance for every experiment;
- *NoIndexSearch:* we perfomed the query on a normative model without index;
- Server: code containing the development of web application;

Dataset:

Our dataset is composed by 2 different datasets:

- ***Dog_breeds*** has 120 classes and 20K images, divided by breeds of dog;
- ***MirFlickr*** has 1 class and 25k images; contains random images.

Figure 1--Example of image from dog_breeds

Figure 2- Example of image from mirflickr

DenseNet121

Convolutional neural networks (CNNs) have become the dominant machine learning approach for visual object recognition: it is a Deep Learning algorithm which can take in an input image, assign importance to various aspects/objects in the image and be able to differentiate one from the other. DenseNet is a convolutional network where each layer is connected to all other layers that are deeper in the network.

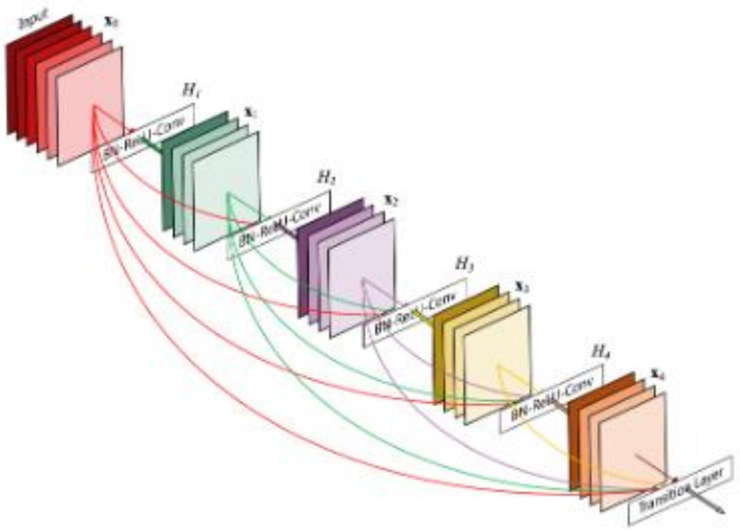


Figure 3- DenseNet with 5 layers

DenseNets simplify the connectivity pattern between layers ensuring maximum information (and gradient) flow; connecting every layer directly with each other. DenseNets layers are very narrow and they just add a small set of new feature-maps.
DenseNets are divided into DenseBlocks, where the dimensions of the feature maps remains constant within a block, but the number of filters changes between them.
For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers.

Features extraction

We extracted the features with the normative model first, on the *Dog_breeds* dataset and *Mirflickr,* with the functions "`extract_features()`" and "`extract_features_noise()`", which includes the same operations, and then we concatenated them to confront them with the features extracted from the query.
We extracted the features also with our model finetuned with the functions "`extract_finetuned_features()`" and "`extract_finetuned_features_noise()`" and then we did that the same operations as before.

Index LSH

LSH is a randomized algorithms that hashes similar input items into the same "buckets" with high probability. A randomized algorithm does not guarantee an exact answer but instead provides a high probability guarantee that it will return the correct answer or one close to it.
LSH is based on the simple idea that, if two points are close together, then after a "projection" operation these two points will remain close together. Two points that are close together on the sphere are also close together when the sphere is projected onto the two-dimensional page. At the core of LSH is the scalar projection (or dot product), given by:

- $g(p)=<h_1(p), h_2(p),\ldots, h_k(p),>$

Where

- $h_i(p)=\lfloor(p*X_i+b_i)/w)\rfloor$

- h() function is the definition of the projection
- w is the size of the intervals (we decided to fix w=4)
- Xi is a vector containing the elements of the line
- bi is a random scalar value between 0 and w.
- P is the features array of the object we want insert into buckets

When we define one hash function g(), we have to define k different projections h(), generating k vectors X as well as k random values for b.
If we consider the binary implementation of the LSH index, we consider a binary array as features array and we create hash functions h() as a position of the binary features array (so an hash family g() is composed by several h() each one corresponding to a position of the binary features array)

Our implementation

We implemented our index LSH in two forms: binary and not, we differentiated them with a Boolean value: True-> binary, False->non binary.

Not binary:

- The function `hashfunctions_generator(x,g,h, binary = False, w=4 ):`
- extract random values for x and b for each hash function h for each hash family g. We tried different combinations of g and h values, to optimize the buckets generation. The best solutions we found were g =10 and h=2. In fact if h is too high buckets contains only one or none images, if h = 1 buckets contains too many images and the index is useless.
- `take_features(path):` it loads the features previously extracted, return variables *features;*
- `make_ranking(q_features, features):` the first parameters is the query features extracted; it orders the features from the closest to the furthest;
- `insert(p, nomefoto, g, binary, MEAN_VECTOR, w = 4):` insertion of the image in the right bucket and calculate the given function;
- `insert_execution(g, h, binary = False):` execute the previous function;
- same operations are applied for each query inserted;
- `search_and_print(g_dim,h_dim,image_name, binary = False):` execution of the query and display results.

Binary:

- Same functions of the not binary case (the hash_functions generator and the insert of course generate different kinds of buckets), In this case we used g = 10 and h = 10, because if h is too low there are too many image for each bucket and the index would be usless.
- `Generate_mean_vector(features_vector)` : take the features vector of all images and generate a vector which every element is the mean value of the same elements of features vector

- `Generate_mean_vector(mean_vector, features_vector)`: compare every element of the features vector of an image with the mean vector, setting the same element of the binary vector to 1 if features_vector[i] > mean_vector[i], 0 instead. Return the binary vector.

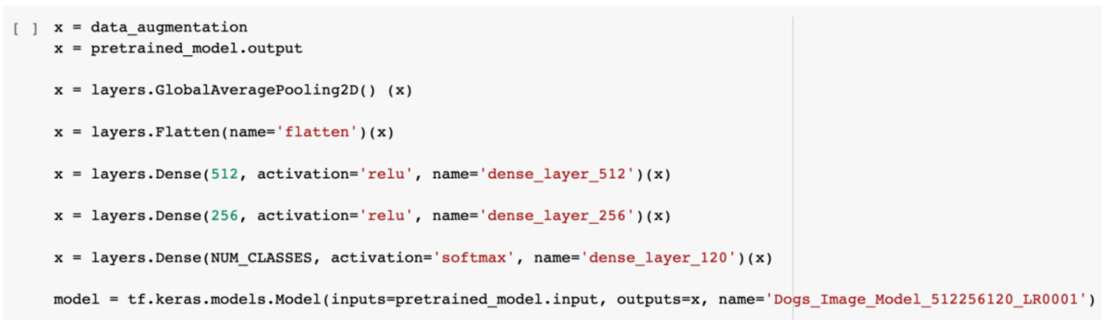
Fine-Tuning

Fine Tuning means taking weights of a pre-trained neural network and using them as initialization for a new model being trained on data from the same domain (often e.g. images).

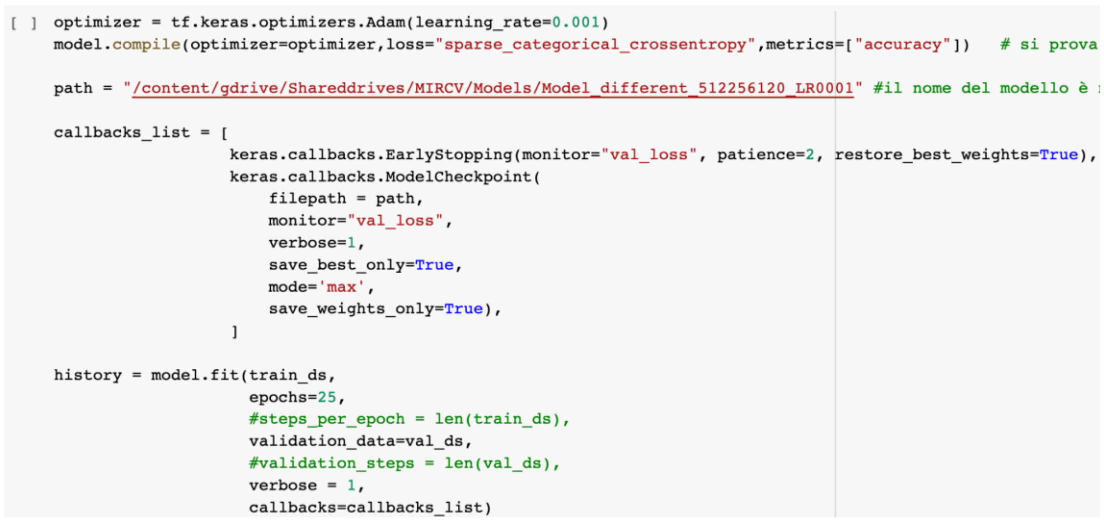
The CNN that we choose as the base network is DenseNet-121, and we are going to obtain the fine-tuned model starting from it.
To avoid big gradients coming from the new layers to sabotage the weights of the pretrained model, we decided to freeze the layers of the base model and to train only the new layers that we added.
After this initial stage, we are going to unlock the previously frozen layers and fine tune the weights of the base model, but only the last two layers, since they are the interesting ones for our case.

In order to build the new model, first we removed the fully connected layer on top and then we made various experiments, changing the number of hidden layers, the activation function, the learning rate, adding and removing flatten.
Here are some of the results that we obtain:

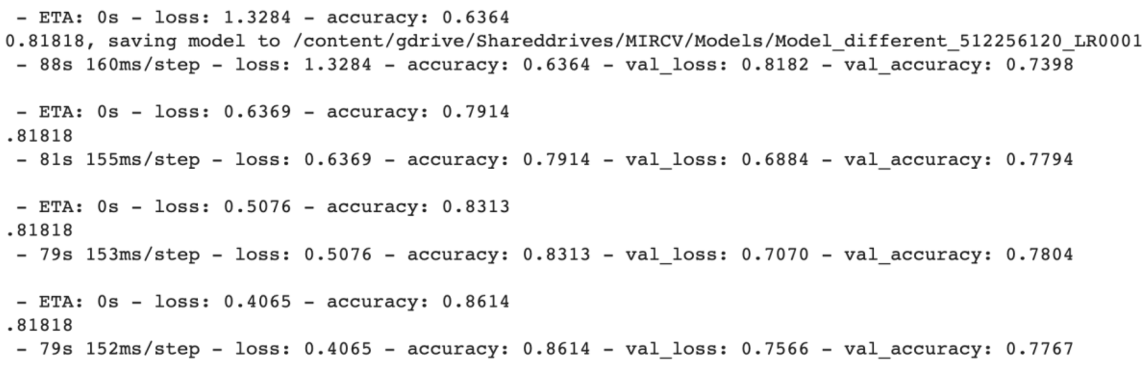
EXPERIMENT 3



```
[ ] x = data_augmentation
    x = pretrained_model.output

    x = layers.GlobalAveragePooling2D() (x)

    x = layers.Flatten(name='flatten')(x)

    x = layers.Dense(512, activation='relu', name='dense_layer_512')(x)

    x = layers.Dense(256, activation='relu', name='dense_layer_256')(x)

    x = layers.Dense(NUM_CLASSES, activation='softmax', name='dense_layer_120')(x)

    model = tf.keras.models.Model(inputs=pretrained_model.input, outputs=x, name='Dogs_Image_Model_512256120_LR0001')
```



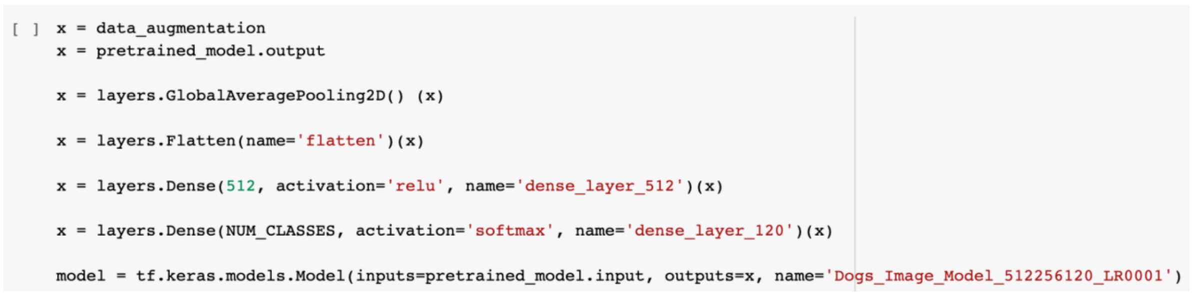
```
[ ] optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer,loss="sparse_categorical_crossentropy",metrics=["accuracy"])   # si prova

    path = "/content/gdrive/Shareddrives/MIRCV/Models/Model_different_512256120_LR0001" #il nome del modello è :

    callbacks_list = [
                      keras.callbacks.EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True),
                      keras.callbacks.ModelCheckpoint(
                          filepath = path,
                          monitor="val_loss",
                          verbose=1,
                          save_best_only=True,
                          mode='max',
                          save_weights_only=True),
                      ]

    history = model.fit(train_ds,
                        epochs=25,
                        #steps_per_epoch = len(train_ds),
                        validation_data=val_ds,
                        #validation_steps = len(val_ds),
                        verbose = 1,
                        callbacks=callbacks_list)
```



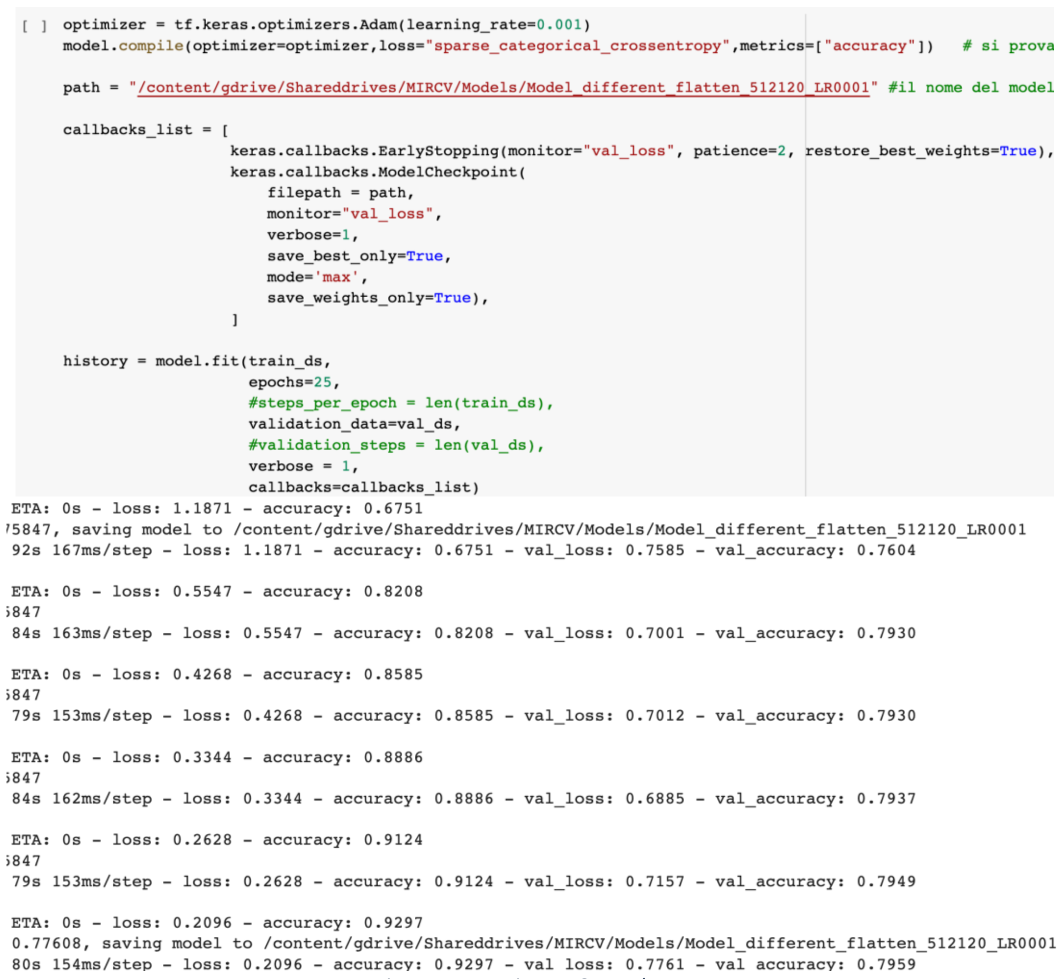
```
 - ETA: 0s - loss: 1.3284 - accuracy: 0.6364
0.81818, saving model to /content/gdrive/Shareddrives/MIRCV/Models/Model_different_512256120_LR0001
 - 88s 160ms/step - loss: 1.3284 - accuracy: 0.6364 - val_loss: 0.8182 - val_accuracy: 0.7398

 - ETA: 0s - loss: 0.6369 - accuracy: 0.7914
.81818
 - 81s 155ms/step - loss: 0.6369 - accuracy: 0.7914 - val_loss: 0.6884 - val_accuracy: 0.7794

 - ETA: 0s - loss: 0.5076 - accuracy: 0.8313
.81818
 - 79s 153ms/step - loss: 0.5076 - accuracy: 0.8313 - val_loss: 0.7070 - val_accuracy: 0.7804

 - ETA: 0s - loss: 0.4065 - accuracy: 0.8614
.81818
 - 79s 152ms/step - loss: 0.4065 - accuracy: 0.8614 - val_loss: 0.7566 - val_accuracy: 0.7767
```

Figure 4- experiment 3 new layers

EXPERIMENT 6



```
[ ] x = data_augmentation
    x = pretrained_model.output

    x = layers.GlobalAveragePooling2D() (x)

    x = layers.Flatten(name='flatten')(x)

    x = layers.Dense(512, activation='relu', name='dense_layer_512')(x)

    x = layers.Dense(NUM_CLASSES, activation='softmax', name='dense_layer_120')(x)

    model = tf.keras.models.Model(inputs=pretrained_model.input, outputs=x, name='Dogs_Image_Model_512256120_LR0001')
```



```
[ ] optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer,loss="sparse_categorical_crossentropy",metrics=["accuracy"])   # si prova

    path = "/content/gdrive/Shareddrives/MIRCV/Models/Model_different_flatten_512120_LR0001" #il nome del model

    callbacks_list = [
                      keras.callbacks.EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True),
                      keras.callbacks.ModelCheckpoint(
                          filepath = path,
                          monitor="val_loss",
                          verbose=1,
                          save_best_only=True,
                          mode='max',
                          save_weights_only=True),
                      ]

    history = model.fit(train_ds,
                        epochs=25,
                        #steps_per_epoch = len(train_ds),
                        validation_data=val_ds,
                        #validation_steps = len(val_ds),
                        verbose = 1,
                        callbacks=callbacks_list)
```

```
 ETA: 0s - loss: 1.1871 - accuracy: 0.6751
75847, saving model to /content/gdrive/Shareddrives/MIRCV/Models/Model_different_flatten_512120_LR0001
 92s 167ms/step - loss: 1.1871 - accuracy: 0.6751 - val_loss: 0.7585 - val_accuracy: 0.7604

 ETA: 0s - loss: 0.5547 - accuracy: 0.8208
5847
 84s 163ms/step - loss: 0.5547 - accuracy: 0.8208 - val_loss: 0.7001 - val_accuracy: 0.7930

 ETA: 0s - loss: 0.4268 - accuracy: 0.8585
5847
 79s 153ms/step - loss: 0.4268 - accuracy: 0.8585 - val_loss: 0.7012 - val_accuracy: 0.7930

 ETA: 0s - loss: 0.3344 - accuracy: 0.8886
5847
 84s 162ms/step - loss: 0.3344 - accuracy: 0.8886 - val_loss: 0.6885 - val_accuracy: 0.7937

 ETA: 0s - loss: 0.2628 - accuracy: 0.9124
5847
 79s 153ms/step - loss: 0.2628 - accuracy: 0.9124 - val_loss: 0.7157 - val_accuracy: 0.7949

 ETA: 0s - loss: 0.2096 - accuracy: 0.9297
 0.77608, saving model to /content/gdrive/Shareddrives/MIRCV/Models/Model_different_flatten_512120_LR0001
 80s 154ms/step - loss: 0.2096 - accuracy: 0.9297 - val_loss: 0.7761 - val_accuracy: 0.7959
```

Figure 5-Experiment 6 new layers

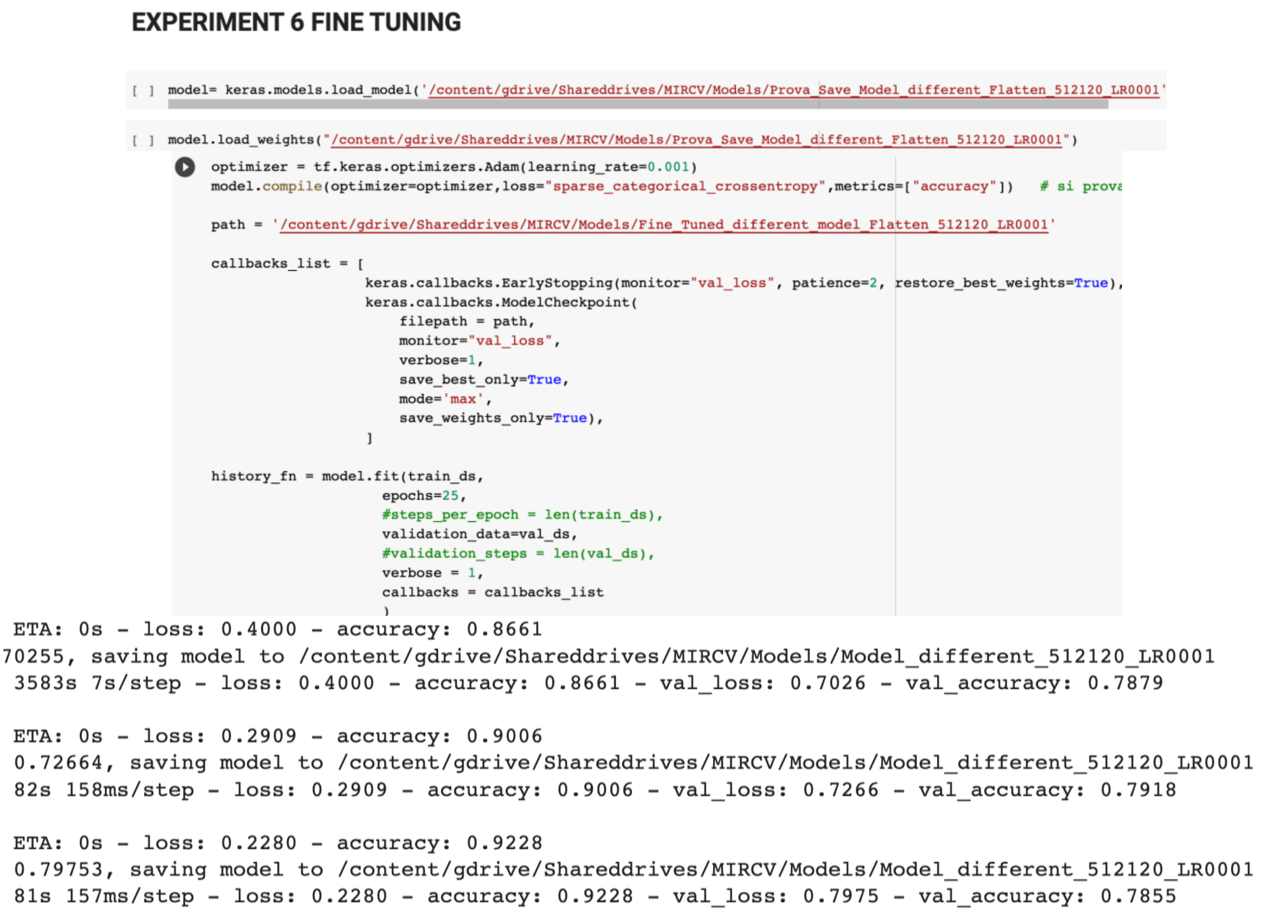
The experiment 6 resulted as the best one out of them all, with an *accuracy* of 0.9297 and a *val_accuracy* of 0.7959

In order to obtain the results, we applied data augmentation to have more samples for the training phase.
The flatten function which flattens the input without affecting the batch size, global average pooling that takes a tensor of size (width x height) and computes the average value of all values across the entire matrix of each input channel.
As for the layers, we added a hidden classifier layer of 512 neurons with 'relu' as activation function.
Finally, we added a hidden classifier with 120 neurons, one for each class of the dataset, and SoftMax as activation function, to be accurate on the classification of the breeds.

The Dataset was divided into the Train set, Validation set and Test set.
The test set is composed of 1000 images, chosen randomly from all the images, the training was 80% of the remaining images and the validation set was the last 20%.
The size of the input image (as required by DenseNet) is 224x224 pixels.
The Batch size used was 32, since bigger size led to crash google colab due to RAM usage, but probably with a bigger batch size, we would have obtained better results.

The optimizer that we chose to use was Adam, the value of the Learning rate was 0.001 and the number of epochs was set to be 25, but we used an early stopping technique to prevent overfitting. The patience parameter for the early stopping was 2, so the training is stopped when for two consecutive epochs the accuracy of the validation doesn't improve.

To fine-tune the network, we unfrozen the last two blocks of DenseNet before the training, and we re-performed training with the same configuration as experiment 6.
We decided to unfreeze only the last two layers because the time grows exponentially with more layers and our results are already good enough.
The results of the training are the following:



EXPERIMENT 6 FINE TUNING

```
[ ] model= keras.models.load_model('/content/gdrive/Shareddrives/MIRCV/Models/Prova_Save_Model_different_Flatten_512120_LR0001'

[ ] model.load_weights("/content/gdrive/Shareddrives/MIRCV/Models/Prova_Save_Model_different_Flatten_512120_LR0001")

    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer,loss="sparse_categorical_crossentropy",metrics=["accuracy"])   # si prova

    path = '/content/gdrive/Shareddrives/MIRCV/Models/Fine_Tuned_different_model_Flatten_512120_LR0001'

    callbacks_list = [
                      keras.callbacks.EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True),
                      keras.callbacks.ModelCheckpoint(
                          filepath = path,
                          monitor="val_loss",
                          verbose=1,
                          save_best_only=True,
                          mode='max',
                          save_weights_only=True),
                      ]

    history_fn = model.fit(train_ds,
                        epochs=25,
                        #steps_per_epoch = len(train_ds),
                        validation_data=val_ds,
                        #validation_steps = len(val_ds),
                        verbose = 1,
                        callbacks = callbacks_list
                        )
```

```
 ETA: 0s - loss: 0.4000 - accuracy: 0.8661
70255, saving model to /content/gdrive/Shareddrives/MIRCV/Models/Model_different_512120_LR0001
 3583s 7s/step - loss: 0.4000 - accuracy: 0.8661 - val_loss: 0.7026 - val_accuracy: 0.7879

 ETA: 0s - loss: 0.2909 - accuracy: 0.9006
 0.72664, saving model to /content/gdrive/Shareddrives/MIRCV/Models/Model_different_512120_LR0001
 82s 158ms/step - loss: 0.2909 - accuracy: 0.9006 - val_loss: 0.7266 - val_accuracy: 0.7918

 ETA: 0s - loss: 0.2280 - accuracy: 0.9228
 0.79753, saving model to /content/gdrive/Shareddrives/MIRCV/Models/Model_different_512120_LR0001
 81s 157ms/step - loss: 0.2280 - accuracy: 0.9228 - val_loss: 0.7975 - val_accuracy: 0.7855
```

Figure 6-Experiment 6 fine-tuning

As we can see, the accuracy is really high since it is 0.9228 after only 3 epochs, the early stopping prevents overfitting and stops at epoch three.
The accuracy of the model doesn't change that much since it was already really high and difficult to improve.
The value of the loss is **0.228,** it is not perfect but it's good enough.

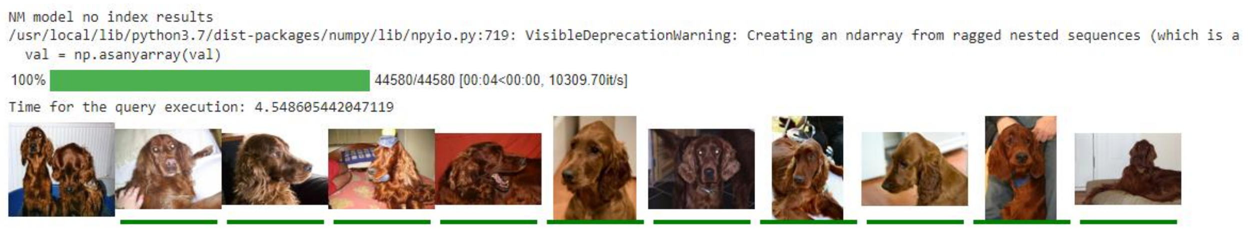
Performance evaluation

Dog breeds search engine operates with the support of the pre-trained DenseNet Neural Network, that assure better results in term of accuracy, and the LSH index that speeds up the process. We execute 4 different type of queries and evaluated the results thanks to different tools.

Query

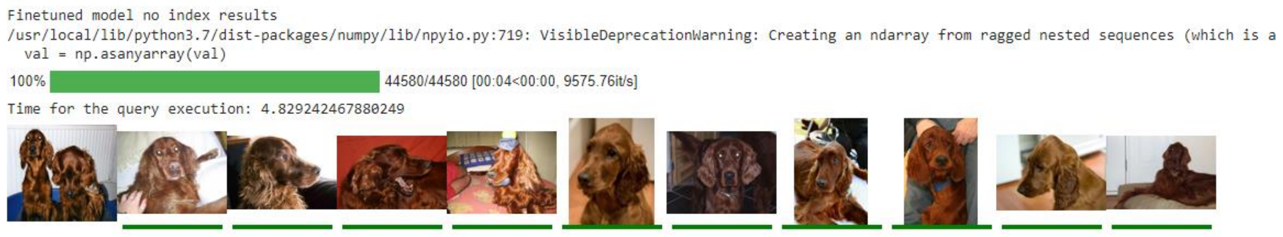
We extracted the features with the normative and the finetuned model, in order to compare retrieval performance of the image search engine between four different approaches:
Example of query: "insert query image, print the 10 most similar images containing similar dogs"
Following there are the results from different extraction:

BRUTE FORCE NORMATIVE MODEL



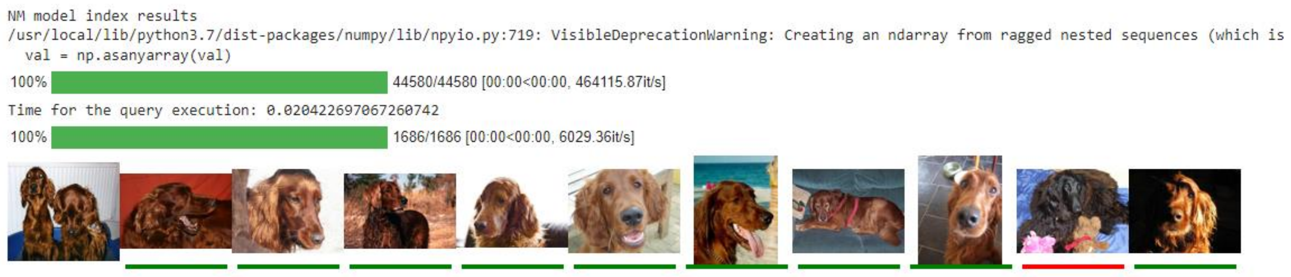
```
NM model no index results
/usr/local/lib/python3.7/dist-packages/numpy/lib/npyio.py:719: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a
  val = np.asanyarray(val)
100% 44580/44580 [00:04<00:00, 10309.70it/s]
Time for the query execution: 4.548605442047119
```

BRUTE FORCE FINE TUNED MODEL



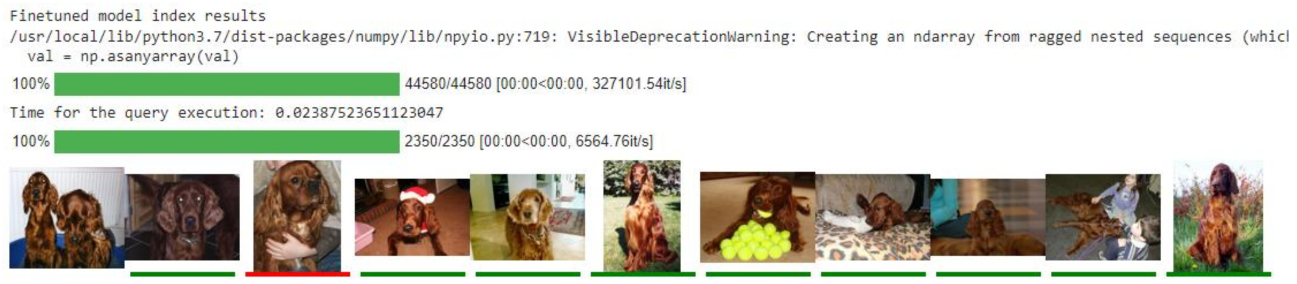
```
Finetuned model no index results
/usr/local/lib/python3.7/dist-packages/numpy/lib/npyio.py:719: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a
  val = np.asanyarray(val)
100% 44580/44580 [00:04<00:00, 9575.76it/s]
Time for the query execution: 4.829242467880249
```

LSH INDEX NORMATIVE MODEL



```
NM model index results
/usr/local/lib/python3.7/dist-packages/numpy/lib/npyio.py:719: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is
  val = np.asanyarray(val)
100% 44580/44580 [00:00<00:00, 464115.87it/s]
Time for the query execution: 0.020422697067260742
100% 1686/1686 [00:00<00:00, 6029.36it/s]
```

LSH INDEX FINE TUNED MODEL



```
Finetuned model index results
/usr/local/lib/python3.7/dist-packages/numpy/lib/npyio.py:719: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (whicl
  val = np.asanyarray(val)
100%                                      44580/44580 [00:00<00:00, 327101.54it/s]
Time for the query execution: 0.02387523651123047
100%                                      2350/2350 [00:00<00:00, 6564.76it/s]
```

The elements underlined with a green line are the images that correspond to the same breed of dogs as the image used as query.

mAp

To evaluate the mean average precision, we execute 10 random queries checking the first k=50 images.
Dogs that have the same breed as the query dog are worth 1 and the others are worth 0, then we added all the images which value was 1 and divide for the number of results (Average). The mAP tells you how many dogs of the right breed are, in percentage.
A higher mAP value indicates a better performance of your detector, given your ground-truth and set of classes.

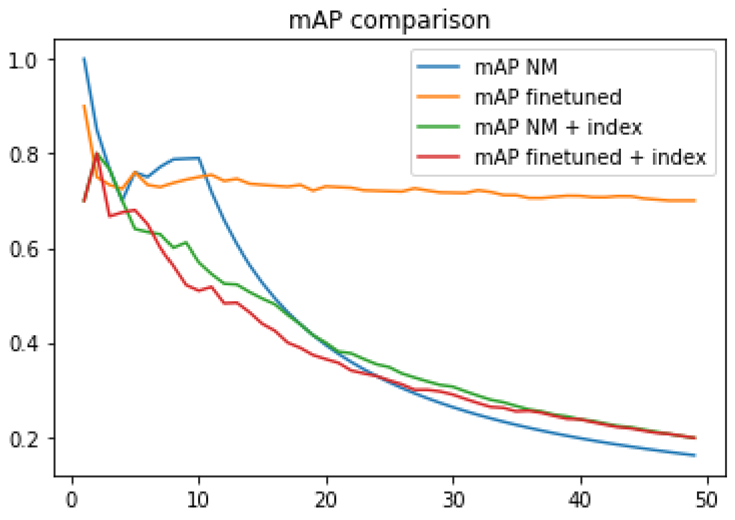


Figure 7- mAP

As we can see, the mAP, decreases as k increases, furthermore we see how the mAP values for the two features are different, and are less in the index because, we know that with index we reduce the time of execution, but the accuracy becomes lower.

Average query time

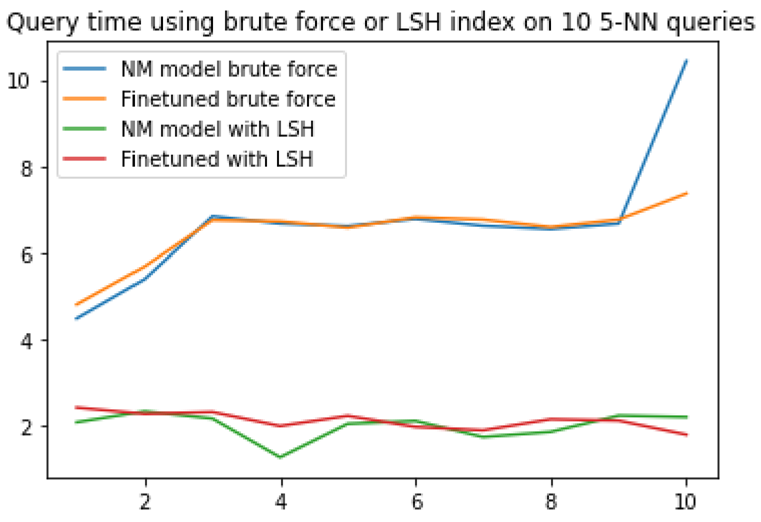


Figure 8- Average query time

As we can see from the graph the average query time with index is half of the brute force time.

Web interface

We decided to exploit the Anvil framework to create our web interface built in python. The Anvil App Server is a standalone, open-source server that can host an Anvil app on any computer – from a beefy server to a Raspberry Pi.
An Anvil app is made up of:

- A User Interface, which you design with a drag-and-drop designer
- Client-side Python code, which runs in the web browser
- Server-side Python code, which runs on Anvil's servers

The server is written in the Notebook "Server" on github link, every component of the group has his own activation key. We insert both models, but we decide to run the fine tuned model with LSH. There are two functions that allows client and server to communicate, once the server is available, from the web you can start the client: inserting the image from a directory and then display the query solutions.

Conclusions

At the end of this project we understand how a CNN works and how important the time of response of the query search is.
Regarding the DenseNet we can say that it works fine, sometimes it focuses more on the colors and the shape of the dog, so it returns very similar dogs even if they are from different breeds.

Regarding the fine-tuning, we were able to obtain a very high accuracy and a good enough value for val_accuracy, since the normative model was already really good.
The LSH index helped us with the velocity for images retrieval on behalf of the accuracy.