Github

Prerequisites

This readme implies the following ones:

1. You some some experience in development at all, so question like - let’s install Python and additional packages doesn’t make you to be perplexed
2. You want to feel how such cliche like “neural network” works
3. You have some preliminary knowledge how it works: it’s about having labeled dataset of what we’re going to classify (in this case: dogs and cat), it’s about process of training (frankly speaking it’s enough to understand that it takes some time), it’s about how to test what we get.
4. You have at least 3-4 years old notebook. All experiments which described below been done on Macbook Pro M3 Pro with 18 Gb RAM. This information in order to understand the possible expected configuration.
5. This readme doesn’t makes no claim as being something extraordinary or new. It just reflects the some work of software engineer, which before to hear the academic course about Machine Learning, decided to compile some different texts from Intenet into some manual and get result which might be called as example of Hello World in neural networks.
6. Python is used here.

Result of your work

So after this script will be applied on your local environment you will be able to classify the photos: whether the given image with cat or dog is cat or dog basically. You can download any image from Internet, crop them to the expected size and try to classify it, calling relatively simple Python script.

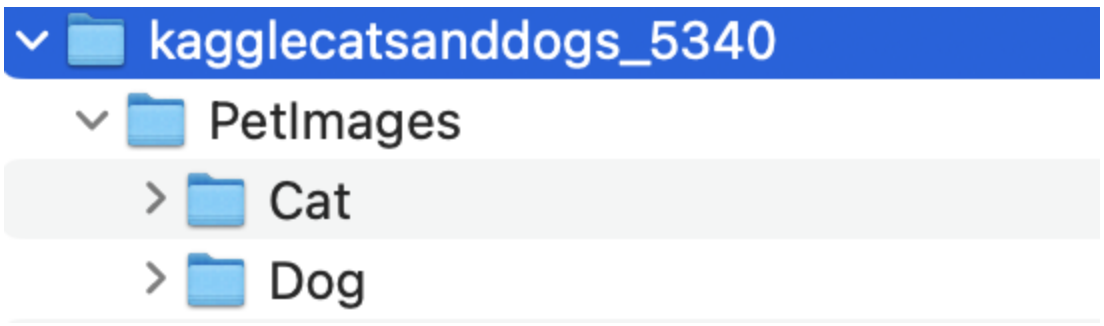
Let’s begin

Dataset

Probably you either know or have intuitive feeling how Machine Learning works. In particular, because we're talking about the image classification, we should have something, using this something our software will learned. Technically under the word "learned" I mean that some set of images will be read by the script, processed in some pretty smart way and a certain combination of pixel, obtained from the sequence of images, will be associated either with cat or dog. It will be possible because someone already did great job: this someone opened the images and put the label to this image, saying this is a cat or dog. Sure, this someone probably is human being (not fully sure, candidly speaking), but this job is drastically important. Our neural network should get to know from somewhere that a certain pixel's combination is a furry cat or grumpy dog. At least till it will not be injected directly in the brain of this neural network (joke).

So we're going to download this dataset from here Cats and Dogs . This resource is maintained by Microsoft, the size of zip-archive is about 800 Mb.

After the archive will be downloaded it makes sense to investigate what we get before using it. So most likely you'll see something like this: two directory with already classified images and due to which some image is put on we have the possibility to judge what depicted on it.



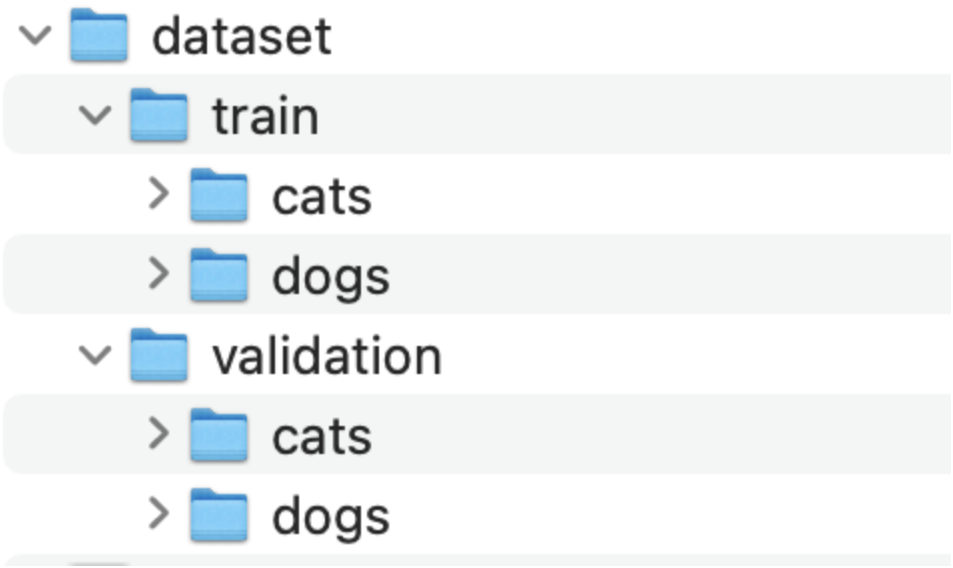
Sure, if you check what every directory contains you will the list of jpg-files. At the moment when I'm writing this archive contains 12502 images of cats and dogs of the each.

The snag is that one of these file is broken apparently, but it will be accounted during the writing code of out classification script.

Train and validation datasets

The reason why we need to split out dataset in two ones might be intuitively acceptable. The core idea of learning is to "**train**" (which means calculate some coefficient which we call as weights as result of the long calculation sequence) and to "**validate**" what we obtained. Since we have the pretty long dataset, which includes approximately 12000 images, we can split this set in two ones on 80% to 20% proportion. It means that 80% files from dataset will be copied in one folder, named "train", and remaining "20%" will be copied in another folder, named "validation".

On my computer the directory's structure looks like this, but the script keeps in mind that directories might not exist, so we will create them.



How are we going to achieve this ? The answer is pretty straightforward: we will write the python-code, which will read the list of files from the downloaded directory, will split them in proportion (80% / 20%) and put them in the corresponding directory, which we either can create on our own or our script will care about this. I will show the snippet and if you think that you still need the explanation - just open the cut.

Please, keep in mind that because of one or several files are corrupted (at least Python was throwing exception during the code's execution on my machine about some files cannot be read) I use the additional function to verify that image, which is going to be used, will not cause problems.

```
from PIL import Image
import os
import shutil
import random

# Function to check if a file is a valid image
def is_image_file(filename):
```

```
    try:
        with Image.open(filename) as img:
            img.verify()
        return True
    except (IOError, SyntaxError):
        return False


# Paths
original_dataset_dir = "kagglecatsanddogs_5340/PetImages"
base_dir = "dataset"

# Create directories
train_dir = os.path.join(base_dir, "train")
validation_dir = os.path.join(base_dir, "validation")
os.makedirs(train_dir, exist_ok=True)
os.makedirs(validation_dir, exist_ok=True)

train_cats_dir = os.path.join(train_dir, "cats")
train_dogs_dir = os.path.join(train_dir, "dogs")
validation_cats_dir = os.path.join(validation_dir, "cats")
validation_dogs_dir = os.path.join(validation_dir, "dogs")

os.makedirs(train_cats_dir, exist_ok=True)
os.makedirs(train_dogs_dir, exist_ok=True)
os.makedirs(validation_cats_dir, exist_ok=True)
os.makedirs(validation_dogs_dir, exist_ok=True)

# List of filenames
cat_dir = os.path.join(original_dataset_dir, "Cat")
dog_dir = os.path.join(original_dataset_dir, "Dog")

cat_filenames = [
    f
    for f in os.listdir(cat_dir)
    if f.endswith(".jpg") and is_image_file(os.path.join(cat_di
```

```
]
dog_filenames = [
    f
    for f in os.listdir(dog_dir)
    if f.endswith(".jpg") and is_image_file(os.path.join(dog_di
]

# Shuffle the data
random.shuffle(cat_filenames)
random.shuffle(dog_filenames)

# Define split sizes
train_size = int(0.8 * len(cat_filenames))  # 80% for training
validation_size = len(cat_filenames) - train_size  # 20% for va

# Copy files to train and validation directories
for i in range(train_size):
    shutil.copyfile(
        os.path.join(cat_dir, cat_filenames[i]),
        os.path.join(train_cats_dir, cat_filenames[i]),
    )
    shutil.copyfile(
        os.path.join(dog_dir, dog_filenames[i]),
        os.path.join(train_dogs_dir, dog_filenames[i]),
    )

for i in range(train_size, len(cat_filenames)):
    shutil.copyfile(
        os.path.join(cat_dir, cat_filenames[i]),
        os.path.join(validation_cats_dir, cat_filenames[i]),
    )
    shutil.copyfile(
        os.path.join(dog_dir, dog_filenames[i]),
        os.path.join(validation_dogs_dir, dog_filenames[i]),
    )
```

Explanation of Python code for file's operation

Let's run through this code if you think that it is needed to be explained. Basically if you are interested in **Image classification** only, you can easily skip it, because it does only files operation: all you need to know that the result is the two folders: **train** and **validation**, where the source image files distributed in the proportion 80/20%. To achieve this you just have to setup two variables, which are self explanatory enough.

```
original_dataset_dir = "kagglecatsanddogs_5340/PetImages"
base_dir = "dataset"
```

So the first function is just to verify that image is valid: it happens, as I've mentioned above already, because some image(s) are broken and I'd like to filter out the corrupted ones.

```
# Function to check if a file is a valid image
def is_image_file(filename):
    try:
        with Image.open(filename) as img:
            img.verify()
        return True
    except (IOError, SyntaxError):
        return False
```

Next block of code just create the two folders with hardcoded names: **train** and **validation,** and it does it carefully, doing nothing if you've already created such folder.

```
# Create directories
train_dir = os.path.join(base_dir, "train")
validation_dir = os.path.join(base_dir, "validation")
os.makedirs(train_dir, exist_ok=True)
os.makedirs(validation_dir, exist_ok=True)
```

The next one code is pretty similar basically: inside every recently created directories it creates the corresponding pair: **cats** and **dogs.**

```
train_cats_dir = os.path.join(train_dir, "cats")
train_dogs_dir = os.path.join(train_dir, "dogs")
validation_cats_dir = os.path.join(validation_dir, "cats")
validation_dogs_dir = os.path.join(validation_dir, "dogs")

os.makedirs(train_cats_dir, exist_ok=True)
os.makedirs(train_dogs_dir, exist_ok=True)
os.makedirs(validation_cats_dir, exist_ok=True)
os.makedirs(validation_dogs_dir, exist_ok=True)
```

And the file's operations again. So let's take the list of files from downloaded and unarchived folder (unarchived by you, sorry for that).

```
cat_filenames = [
    f
    for f in os.listdir(cat_dir)
    if f.endswith(".jpg") and is_image_file(os.path.join(cat_di
]
dog_filenames = [
    f
    for f in os.listdir(dog_dir)
    if f.endswith(".jpg") and is_image_file(os.path.join(dog_di
]
```

... shuffle them

```
# Shuffle the data
random.shuffle(cat_filenames)
random.shuffle(dog_filenames)
```

Here the questions might be arisen - why do we need to shuffle the files before using it, as somebody added the images of animals to folders, keeping some order ? There is the good question. The first and the main reason is the **generalisation** of the data in order to obviate the several factors like **order bias** (basically we have no clue how the dataset has been created, probably there were some circumstances like the sympathy of the creators to the certain type of cats or dogs; or probably the list of images has some correlations - metrics or spatial. For example in our case the images of cats could be captured from videostream and the neighbour images might be pretty similar. In case of the hand-writing classification the case might be even worth: you can imagine that species of writing were gathered from the students of the same school where the order rules of the handwriting exist).

The next one code is pretty straightforward again - it's about having two different it terms of size datasets: 80% images is for training, the remaining 20% is for validation:

```
# Define split sizes
train_size = int(0.8 * len(cat_filenames))  # 80% for training
validation_size = len(cat_filenames) - train_size  # 20% for val
```

And at least:

```
for i in range(train_size):
    shutil.copyfile(os.path.join(cat_dir, cat_filenames[i]), os
    shutil.copyfile(os.path.join(dog_dir, dog_filenames[i]), os

for i in range(train_size, len(cat_filenames)):
```

```
    shutil.copyfile(os.path.join(cat_dir, cat_filenames[i]), os
    shutil.copyfile(os.path.join(dog_dir, dog_filenames[i]), os
```

The copying files to these folders which will be ready to be used for image classification. The result on my laptop looks like this:

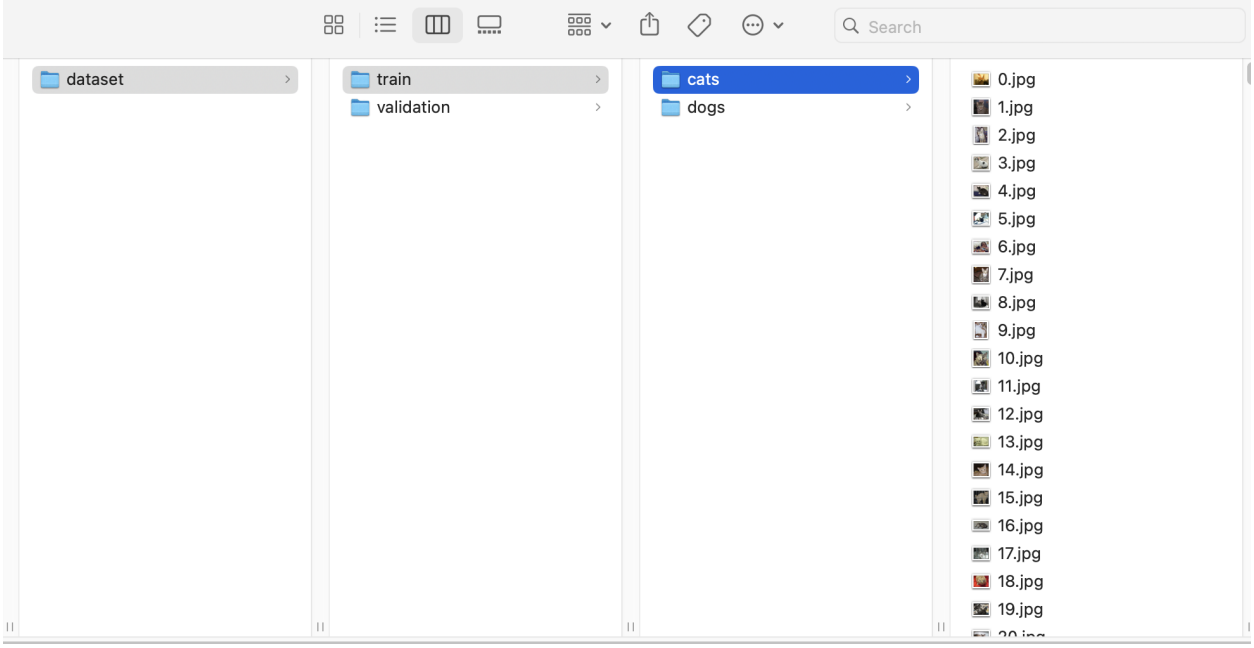


Image classification

There is the plan to describe the process: we will go through the code with the pretty superficial explanation what's happening here and only afterwards will try understand what' s going on underhood.

Preparation of images

So on the next step we're going to prepare our images to be processed. The preparations stands itself in adding the random elements to the images, simultaneously not corrupting them. Every images will be rotated, moved, rescaled, some new pixels will be added in order to prevent the order bias: the reason is the similar to the one, which we discussed the necessity of the shuffling the files. Because our final result is considered to have model which should be able to cope with any unknown at the moment of the learning process images, it makes sense to prepare out model in advance: let's augument our images in the way to assign the elements of uncertainty to the images.

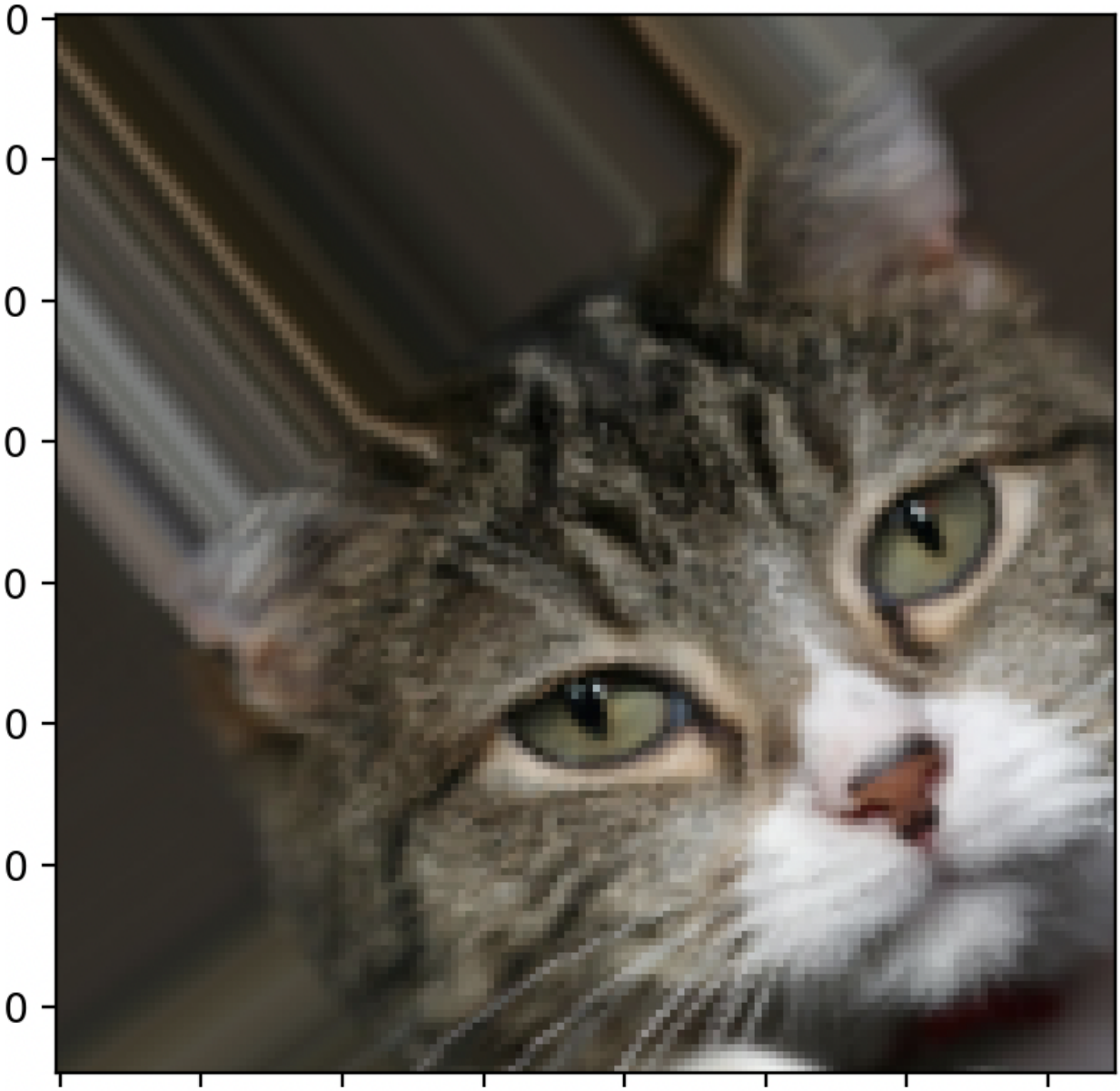
```
# Data augmentation and normalization for training
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

So here we're preparing the object, which is the set of some rules, which in turn will be applied to images a bit further.

I can show you how it works using the augmented image and after we can discuss this step in more details.

Let's see on the original image before augmentation:

And the result of augmentation, using the property’s values, described in object above:



So you can easily detect that the images has been changed pretty considerably, by the applying rules which lead the image to be augmented, but still keeping the recognisable object on that image (I hope the cat still can be identified). Also it's pretty important to mention about that augmentation has random parameters: it's really not mandatory that the same changes will be applied to every image in the same or even similar way. It's because of all numeric parameters in `ImageDataGenerator` has random range of values.

As result we will have more varied dataset and this adds the robustness to our dataset and make it more stable. For academic interest I put the explanation of the parameters below, but, frankly speaking, understanding them is not compulsory for comprehending the next steps. Also please don't ask me about the particular values of the parameters.

ImageDataGenerator Parameters

1. **rescale=1./255**:

- **Purpose**: Normalizes the pixel values of the images.
- **Effect**: Converts pixel values from the range [0, 255] to the range [0, 1]. This normalization helps in speeding up the convergence during training.

2. **rotation_range=40**:

- **Purpose**: Randomly rotates images.
- **Effect**: Each image can be randomly rotated by up to 40 degrees clockwise or counterclockwise.

3. **width_shift_range=0.2**:

- **Purpose**: Randomly shifts images horizontally.
- **Effect**: Each image can be randomly shifted horizontally by up to 20% of the image's width.

4. **height_shift_range=0.2**:

- **Purpose**: Randomly shifts images vertically.
- **Effect**: Each image can be randomly shifted vertically by up to 20% of the image's height.

5. **shear_range=0.2**:

- **Purpose**: Applies random shearing transformations.
- **Effect**: Each image can be sheared (tilted) by an intensity of up to 20%.

6. **zoom_range=0.2**:

- **Purpose**: Randomly zooms in on images.
- **Effect**: Each image can be zoomed in by up to 20%.

7. **horizontal_flip=True**:

- **Purpose**: Randomly flips images horizontally.
- **Effect**: Each image has a 50% chance of being flipped horizontally.

8. **fill_mode='nearest'**:

- **Purpose**: Determines how newly created pixels are filled in after a transformation.
- **Effect**: Pixels newly created by transformations (like shifts and rotations) are filled in with the nearest pixel value from the original image.

We're going to apply the same approach to the validation dataset, but with only one parameters - rescaling - and the reason of that for the validation dataset we'd like to keep our images unchanged to have more realistic images for validation our trained model.

```
# # Only rescaling for validation
validation_datagen = ImageDataGenerator(rescale=1./255)
```

And when our lists of rules how to augmented the images are ready we can start basically to works with images. Please, keep in mind, that albeit now some calculations are already started at this moment, but this calculation pertains to image's processing only, not about neural network's, which will be initialised later.

```
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary'
)
```

```
validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary'
)
```

Since both invocations are basically the same with difference in directory's names, let' s discuss what's happening having the first one as example.

By the name `flow_from_directory` it's pretty easy to suggest that functions reads files from directory - and it's true. Basically this method is used for the long arrays of messages from the storage, when it's nor reasonable to keep all them into memory.

Obviously that `train_dir` is the directory's name from which the method is going to read the images, and `target_size` prescribes to change the size of every methods. It is done in order to have the dataset consistent, what is actually important for the next calculations in terms of neural networks. The sense of `batch_size` is fairly straightforward, so let's stop on the `class_mode` . This specifies the one of the main result of this function - processing images and labelling them. Actually it means that result of this process will be the certain pair - processed image and some label. The second part of this pair - label - will be binary, because we directly asked about that `class_mode='binary'` and the final result will fe the sequence:

`image1 - 0`

`image2 - 0`

`image3 -0`

`….`

`image2000 - 1`

`image2001-1`

You may comprehend this like manual association of some property (which is named as label in Machine Learning) from the space (0;1) to every images and it implies that after the method will complete his job, we will have the labeled dataset.

You way be wondering how it works. So method will run through the `train_dir` and will fetch the names of the subdirectories, obtaining obviously `['cat', 'dogs']` . After that this array will be alphabetically ordered and for the first name **0** will be associated and for the second one, for sure, the label will be **1**. As result we have the binary-labeled dataset. Lucky case that we have only two animals to classify - for sure, if we were have more than two, we would have been do something more complicated.

Coming closer to neural network

So we are pretty close to the most hardcore part of our process - training our model or, better say, neural network. Let me to show you the main actor of this show and we will discuss what it is.

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150,
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
```

```
    Dense(1, activation='sigmoid')
])
```

So you can see exactly what we're going to train. Basically if you familiar with process of developing program, we can pretty conditionally to make the analogy: you see the code, which should be compiled a bit later and be running. So now we are on the first stage: we are "coding" something, which is going to be used as template for the future calculation.

Good news that result of the invocation is the exactly neural networks no matter how corny it sounds. You may be perplexed by this, but let to provide you proof.

If you follow my storyline then you have Python of being installed on your computer, so you can run this code without problem. Let's do this - please, take into account that `model` is the same what we can see above. We're juts going to look into the cover of this `model` .

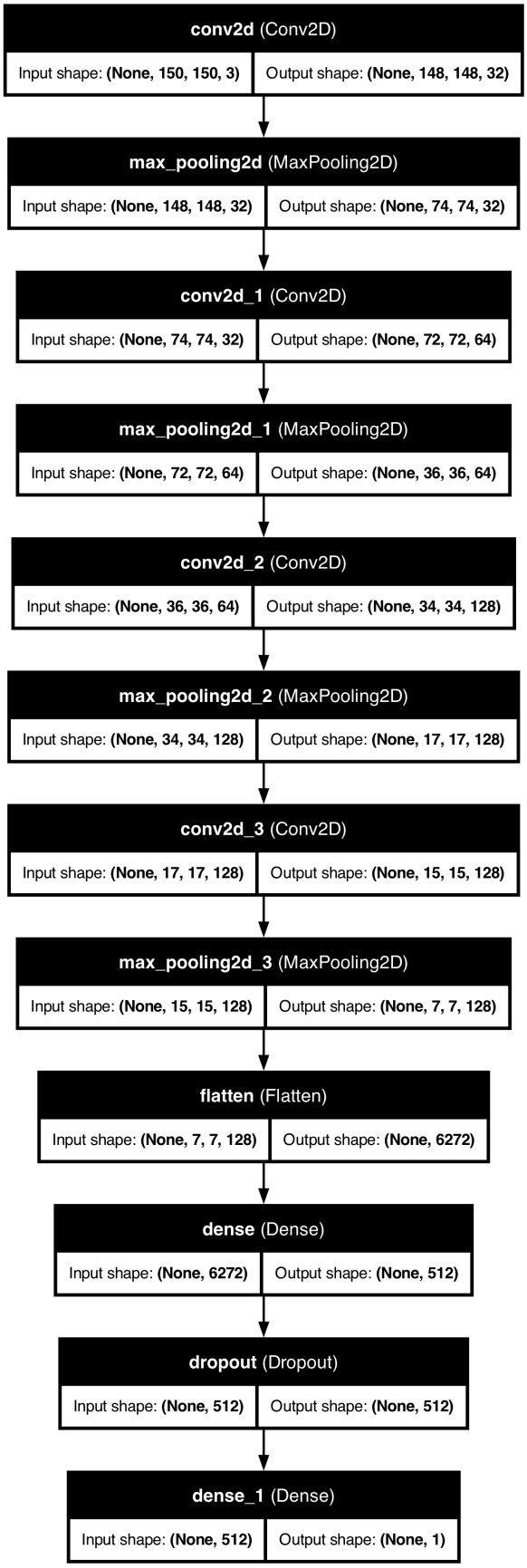
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatte
from tensorflow.keras.utils import plot_model

# Define the model architecture
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150,
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='relu'),
```

```
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Visualize the model architecture
plot_model(model, to_file='cats_vs_dogs_model.png', show_shapes=
```

So if you run this code, then according to the invocation of method

```
plot_model(model, to_file='cats_vs_dogs_model.png', show_shapes=True, show_layer_names=True)
```

in the directory where you script is saved, you will find the png file which looks like this (I quote it in full because it visualise some complicated structure and it might serve as good feedback from the blackbox, which the neural network might seem):

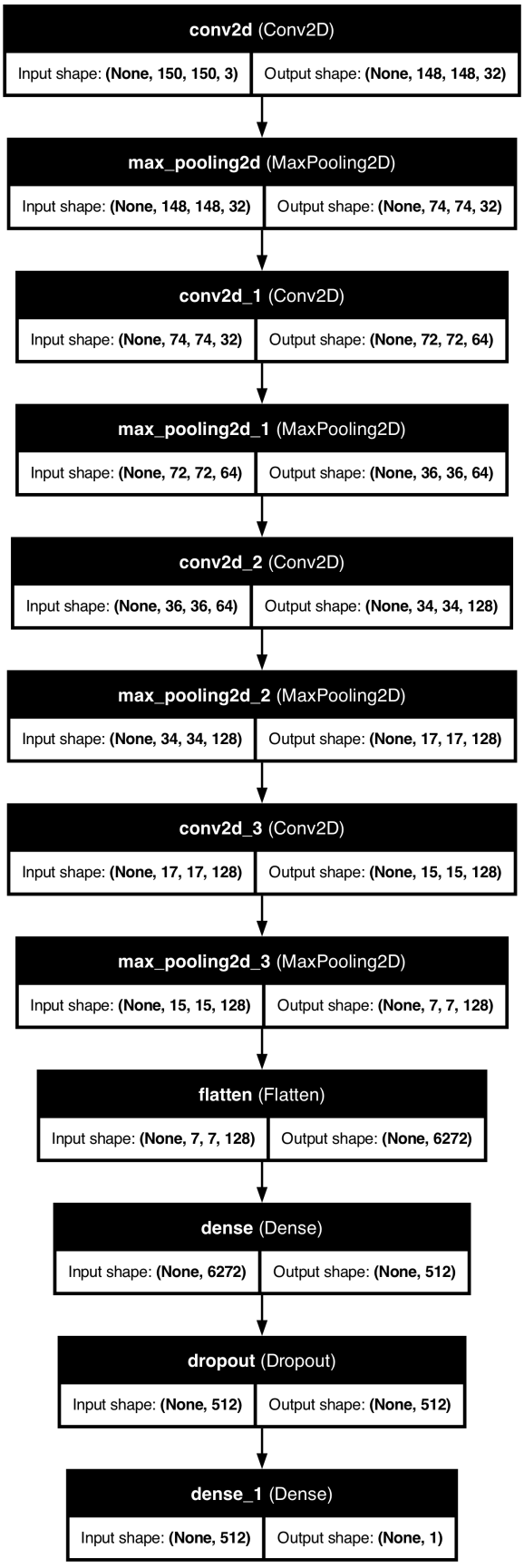

conv2d (Conv2D)
Input shape: (None, 150, 150, 3)
Output shape: (None, 148, 148, 32)
max_pooling2d (MaxPooling2D)
Input shape: (None, 148, 148, 32)
Output shape: (None, 74, 74, 32)
conv2d_1 (Conv2D)
Input shape: (None, 74, 74, 32)
Output shape: (None, 72, 72, 64)
max_pooling2d_1 (MaxPooling2D)
Input shape: (None, 72, 72, 64)
Output shape: (None, 36, 36, 64)
conv2d_2 (Conv2D)
Input shape: (None, 36, 36, 64)
Output shape: (None, 34, 34, 128)
max_pooling2d_2 (MaxPooling2D)
Input shape: (None, 34, 34, 128)
Output shape: (None, 17, 17, 128)
conv2d_3 (Conv2D)
Input shape: (None, 17, 17, 128)
Output shape: (None, 15, 15, 128)
max_pooling2d_3 (MaxPooling2D)
Input shape: (None, 15, 15, 128)
Output shape: (None, 7, 7, 128)
flatten (Flatten)
Input shape: (None, 7, 7, 128)
Output shape: (None, 6272)
dense (Dense)
Input shape: (None, 6272)
Output shape: (None, 512)
dropout (Dropout)
Input shape: (None, 512)
Output shape: (None, 512)
dense_1 (Dense)
Input shape: (None, 512)
Output shape: (None, 1)

We have to discuss it in thoroughly because this is “code” or, if you want, the “skeleten” of our calculation process.

So, using scientific style, we may say that this is visualisation of calculation layers of neural network. You may heard before, that a neural network is the sequence of artefacts, named “neurones”, which are connected in the mesh or network. Seeing on this schema you may notice that there are nodes as well and they are connected sequentially. This gives you some analogue of the out non-precise definition of a neural networks, but it’s good point to start.

Let me to put this image and code, which creates it, together.

```
model = Sequential([
    Conv2D(32, (3, 3), activa
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activa
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activ
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activ
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='re
    Dropout(0.5),
    Dense(1, activation='sigm
])
```


conv2d (Conv2D)
Input shape: (None, 150, 150, 3)
Output shape: (None, 148, 148, 32)
max_pooling2d (MaxPooling2D)
Input shape: (None, 148, 148, 32)
Output shape: (None, 74, 74, 32)
conv2d_1 (Conv2D)
Input shape: (None, 74, 74, 32)
Output shape: (None, 72, 72, 64)
max_pooling2d_1 (MaxPooling2D)
Input shape: (None, 72, 72, 64)
Output shape: (None, 36, 36, 64)
conv2d_2 (Conv2D)
Input shape: (None, 36, 36, 64)
Output shape: (None, 34, 34, 128)
max_pooling2d_2 (MaxPooling2D)
Input shape: (None, 34, 34, 128)
Output shape: (None, 17, 17, 128)
conv2d_3 (Conv2D)
Input shape: (None, 17, 17, 128)
Output shape: (None, 15, 15, 128)
max_pooling2d_3 (MaxPooling2D)
Input shape: (None, 15, 15, 128)
Output shape: (None, 7, 7, 128)
flatten (Flatten)
Input shape: (None, 7, 7, 128)
Output shape: (None, 6272)
dense (Dense)
Input shape: (None, 6272)
Output shape: (None, 512)
dropout (Dropout)
Input shape: (None, 512)
Output shape: (None, 512)
dense_1 (Dense)
Input shape: (None, 512)
Output shape: (None, 1)

You shouldn’t be Sherlock Holmes to find the coincident between the method and his result. If you think that every parameters creates something which is depicted on schema as black-white block - then yes, it’s true, even, some blocks are repeated, but every of them has its own properties.

Convolution

Let’s talk about it in details. And let me to use the term “layer” and “sequence of layers” because both of them correspond as to intuitive and to the real state of things in this abstraction.

So we’re talking about Convolutional Neural Networks, which is perfectly meet the expectation of Image classification, because such networks process the every image not “entirely” as matrix of pixels, but make some convolution for certain part of image, pursuing two targets: dwindling the calculation networks (although it’s not problem in last time) and unification of images.

Virtually it can be images like it is shown on animation below (btw, this gif is created by the neural network as well):

So every movement of the white rectangle over the original image (it our case it will be our cat or dog) covers the some part of pixels, which are argument of some calculation method. This is a convolution without additional details. Of course, there are a lot of computational procedures, which can be applied to solve the certain problem or issue during convolution, and good news here that the majority of cases are already known and you don't need to make the investigation for the concrete one, it's really enough just to see some analogy.

We will discuss the invocation of every parameter in `Sequential()` sequentially, but let me to put at the first place more simple method `MaxPooling2D` . In my opinion, the introducing the logic of this method first will provide some common perspective of computational operation and afterwards we can return back to `Conv2D` .

First invocation of MaxPooling2D(2, 2)

We have this

```
MaxPooling2D(2, 2)
```

Please, believe me that if you read this first time the academic definition gives you nothing, except for probably several details like 2D, which will lead you to the thought that this something is applicable for flat object. The last point is absolutely right, but let me to show how `MaxPooling2D` works using simple example.

So we have some input:

```
Input Feature Map (4x4):
[
[1, 3, 2, 4],
```

```
[5, 6, 7, 8],
[9, 10, 11, 12],
[13, 14, 15, 16]
]
```

Let’s introduce two terms to operate them here.

1. **Pool Size**:

- The pool size specifies the dimensions of the window (e.g., 2×2 - for images you can read “pixels”) that slides over the input feature map.
- For a 2×2 pool size, the window covers 2×2 regions (matrix of pixels 2×2) of the input feature map.

2. **Stride**:

- The stride specifies how much the window moves after each operation. By default, for max pooling, the stride is the same as the pool size, so for a 2×2 pool, the stride is 2.

And basically the operation of pooling looks as you can on animation above.

But example: apply 2×2 max pooling with a stride of 2.

1.
First Window (Top-left 2×2 region)

```
[
    [1, 3],
    [5, 6]
]
```

Do you remember the semantic of our method invocation ?
`Max` .... Yes, max pooling operation's result on this window is 6 because it's maximum value from in this certain window. Go further! Let's shift our window to the right - what do we get ? Correct!

2.
Second Window (Top-right 2×2 region)

```
[
    [2, 4],
    [7, 8]
]
```

Maximum value here is 8. Go further!

3. **Third Window (Bottom-left 2×2 region)**

```
[
    [9, 10],
    [13, 14]
]
```

Maximum here is 14.

4. **Fourth Window (Bottom-right 2×2 region)**

```
[
    [11, 12],
```

```
    [15, 16]
]
```

No needs to say which form of the final result will be:

```
Output Feature Map (2x2):
[
    [6, 8],
    [14, 16]
]
```

So I hope using this example I was able to show you the essential of what pooling is. You can easily think about this like “aggregation” or “convolution” of the matrix pixel in order to reduce dimensions of an every image and, most important, extract the most prominent value from the given window, forming in this way the more informative matrix of data (probably). If you have the insight that this computation procedure is pretty simple and it’s absolutely not obvious how it helps us in our task - image classification - then I can say that most popular result were obtained not because of some light brilliant mind decided that this is sole method to extract data using only pencil and paper, but yes, they tried and it turned out that even such simple method works.

But to be honest my main task here is to eliminate the feeling of magic, which is tangled with the Neural Networks. So far you had a chance to ensure that is was the computation only. Further the same will be.

First invocation of Conv2D

Return back to the code

```
Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)
```

This is the first parameter of invocation of `Sequential` .

We're asking to create the first layer of our convolutional neural network (CNN), which should posses the following properties:

- 32 filters (will be increased in the next invocation)
- Each filter has size 3*3 pixels
- **activation='relu'**: The ReLU (Rectified Linear Unit) activation function is applied, which introduces non-linearity and helps the network learn complex patterns. You can think about this like some computational approach, it should be enough if read this first time.
- **input_shape=(150, 150, 3)**: The input shape of the images is 150×150 pixels with 3 color channels (RGB). This is specified only for the first layer.\

Please stop here and think again what we are trying to do: we get every image from our folder and let him pass through the sequence of layers, where every will apply some computational procedure.

Now I have no choice expect for describing this "computational procedure", hoping that you'll be able to detect some analogy with previous chapter.

Our vocabulary in this case is the following:

1. **Input Shape**:

- **input_shape=(150, 150, 3)**: This specifies that the input to the network is an image of size 150×150 pixels with 3 color channels (RGB).

2. **Convolution Operation**:

- **filters=32**: This means the layer will learn 32 different filters. Each filter will produce a separate output feature map.
- **kernel_size=(3, 3)**: This specifies that each filter is 3×3 pixels. The filter will slide over the input image to compute the output.

3. **Sliding Window**:

- The 3×3 filter (kernel) slides over the input image. At each position, it performs an element-wise multiplication with the part of the image it is currently covering and sums the results to produce a single value in the output feature map.
- This operation is repeated for all positions on the input image to produce a complete feature map.

4. **Activation Function**:

- **activation='relu'**: The Rectified Linear Unit (ReLU) activation function is applied to the output of the convolution operation. ReLU sets all negative values to zero and keeps positive values unchanged, introducing non-linearity into the model.

If you read this without understanding, let me to show you the process by example again. As example we're going to use the grayscale image (it means that the value of each color will be restricted by the fixed amount of colors, but it doesn't matter for our demonstration of principle how it works) 5×5 pixels. Such image can be presented in terms of Python (or linear algebra if you want)like this:

Input image

```
[
    [1, 2, 3, 0, 1],
    [0, 1, 2, 3, 1],
    [3, 2, 1, 0, 2],
    [1, 0, 3, 2, 1],
    [2, 1, 0, 1, 0]
]
```

Also we're going to use **filter** as we described in during the method invocation.

```
[
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
]
```

If you still remember my clumsy animation, showing the movement of the filter over image, we're going to simulate every step this movement, but using the certain amount values of the image and filter.

So we're at the left-top corner. Out filter covers the some part of our image. Extract the values and it will be:

```
[
    [1, 2, 3],
    [0, 1, 2],
    [3, 2, 1]
]
```

Let's make element-wise multiplication of our extracted values of pixels (don't forget we're operating by the grayscale image) and our filter. I hope either Excel or Python or whatever you prefer in this case (I don't use multiplication signs between the digits of the same sign):

```
(11 + 20 + 3*(-1)) + (01 + 10 + 2*(-1)) + (31 + 20 + 1*(-1))
= (1 + 0 - 3) + (0 + 0 - 2) + (3 + 0 - 1)
= -2 - 2 + 2
= -2
```

And obviously the first result of applying our filter is 2.

Next shift to the right by our filter over our image and we have the extracted values of pixels:

```
[
    [2, 3, 0],
    [1, 2, 3],
    [2, 1, 0]
]
```

We already know how out filter looks like, so let me to make the element-wise multiplication again:

```
(21 + 30 + 0*(-1)) + (11 + 20 + 3*(-1)) + (21 + 10 + 0*(-1))
= (2 + 0 + 0) + (1 + 0 - 3) + (2 + 0 + 0)
= 2 - 2 + 2
= 2
```

The result is 2.

I will let myself not continue the process because I hope it is already should be pretty clear. So I will put the intermediate state of filters like this:

```
[
[-2, 2, ...],
[..., ...],
...
]
```

ReLU activation is telling us that all negative values in this matrix should be a zero, so:

```
[
[0, 2, ...],
[..., ...],
...
]
```

This is our skimped result, but I hope the gist of computational procedure should be clear.

Let’s repeat one more time what we did: we load every available image and pass it through the sequence of layers, each of them does some part of work. And the amount of such layer and certain values, which we use during invocation, this is separated and fairly big topic, which I let myself to skip in this text.

Next invocations of Conv2D and MaxPooling2D

I will put the code which we’re keep under our magnitude here again.

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150,
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

Seeing on this code with some part of attentions we can notice that method Conv2D() is invoked several times (four times for precise’s sake) and with every invocation the filter’s amount increases: **32, 64, 128 and 128** again.And basically nothing can stop us from doing the same.

The reason of such sequential invocation is that after the previous one we extract more complex and concurrently abstract data from image. As it is considered it brings to our models more levels of complexity and decreases the spatial complexity of images in turn. If you read this and feel that this explanation is not satisfying you, then believe, you are not the one who thinks the same. This computational approach - sequence of layers - are result of "learning" models by infinite times before a researcher obtained some meaningful result. If you going to delve deeper in this, you should be ready to accept some things which seem not obvious, straightforward or intuitively clear. All this is the result of some experience and before you get the same you have to pass through this on your own.

Let me to prove you again that the sequence invocation of the pair

`Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)), MaxPooling2D(2, 2)`

brings fairly big benefit to our computational procedure.

First pair

```
Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)
MaxPooling2D(2, 2)
```

gives us (remembering that 150×150 is the size of our images and the gist of MaxPooling is the reducing of out matrix of pixels by some simple computational method described above):

Conv2D: Applies 32 filters to the input image, producing 32 feature maps.

MaxPooling2D: Reduces the spatial dimensions of each feature map from 150×150 to 75×75.

Second pair

```
Conv2D(64, (3, 3), activation='relu')
MaxPooling2D(2, 2)
```

gives us:

Conv2D: Applies 64 filters to the 32 (the result of the previous pair's invocation) feature maps, producing 64 feature maps.

MaxPooling2D: Reduces the spatial dimensions of each feature map from 75×75 (the result of the previous pair's invocation) to 37×37.

Third pair

```
Conv2D(128, (3, 3), activation='relu')
MaxPooling2D(2, 2)
```

Conv2D: Applies 128 filters to the 64 feature maps, producing 128 feature maps.

MaxPooling2D: Reduces the spatial dimensions of each feature map from 37×37 to 18×18.
And at least:

Fourth pair

```
Conv2D(128, (3, 3), activation='relu')
MaxPooling2D(2, 2)
```

Conv2D: Applies 128 filters to the 128 feature maps, producing 128 feature maps.

MaxPooling2D: Reduces the spatial dimensions of each feature map from 18×18 to 9×9.

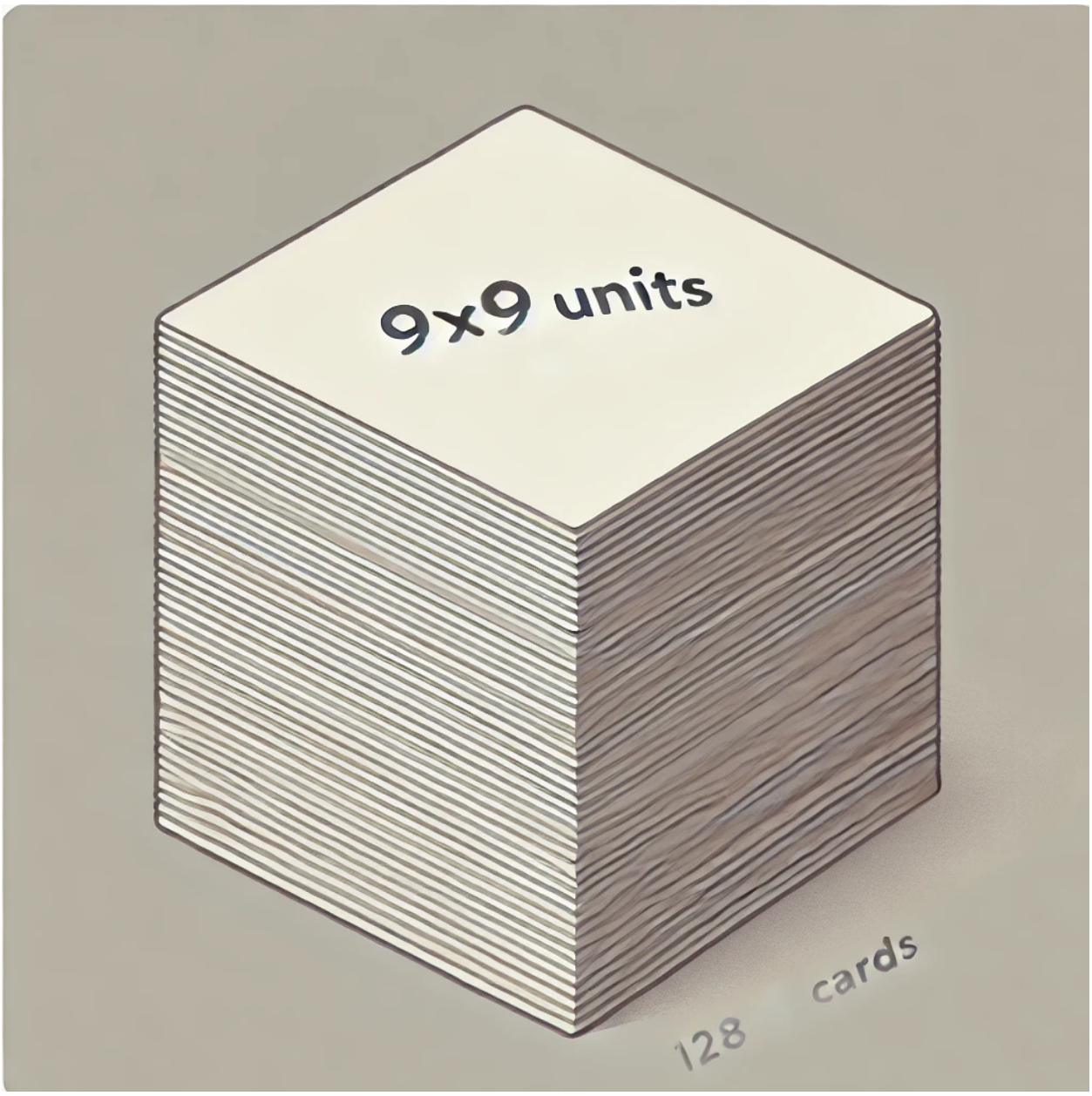
If you keep looking on the decreasing the size of parameters, which were outputs of `MaxPooling2D` , you can easily detect that it was dwindled drastically, which in turn gives to us the meaningful benefit - the decreasing the size of maps which we process during out learning of neural network.

Flatten() invocation

There is good case when the name of the method fully corresponding to what this method does. It really flattens our 2D input (basically images which have been passed through the previous layers) into the 1D output literally. So it means that out next computational procedure will work not with matrix, but with single-row vector. Mathematically we're talking about concatenation of elements of 2D input to 1D.

For example in our case of image classification before the invocation of `flatten()` our input will be the **tensor** (9, 9, 128). You can understand this like the following - tensor is the mathematical object, which has **width** and **height,** which are 9 and 9 correspondingly and has third parameter, conditionally named **depth,** which can be attributed like results of computations from the previous steps and this results are the sequence of maps with certain width and height. You can imagine this like **stack** of matrixes with size 9×9 and count of this matrixes in this stack is 128. Or more figural - every 9×9 matrix from these 128 is some "snapshot" of the original image, and every snapshot keeps some unique characteristic of image, which will be applied during image classification.

Next image of this visualisation is created by the neural network, by the way, but this is absolutely another story.



So before the `flatten()` will be invoked we have such structure, which is named tensor, and after `flatten()` invoked we have 1D-structure, which is the result of concatenation of 81 items of evert feature map (card on the picture above) and such concatenation block will be repeated 128 times. So it's easily to calculate that total amount of element in this structure will be

```
Flattened Shape: (9 * 9 * 128) = (10,368)
```

Let me repeat again that all magic of this pretty long image classification procedure in reality is just the sequence of trivial (relatively) calculations. There is absolutely another question **why** this sequence lead us to the required result, but let me leave this out of the scope of this text.

Dense() invocation

More precisely

```
Dense(512, activation='relu'),
```

So let's investigate the anatomy of what is happening here as well.

Here we have to introduce several new terms to operate them further. We need **weights** and **biases**.

But before let's define the formal values what we can see in the method's invocation:

1. `512` (Neurons):
   - **Description**: The 512 specifies the number of neurons in this dense layer.
2. `activation='relu':`
   - **Description**: The activation function applied to the output of each neuron. This function is

$$f(x) = \max(0, x)$$

which as it's easy to notice that it sets all negative values to zero and leaves positive values unchanged.

Keeping in mind that our input for this case is 1D-structure which contains `10368` items, we will define the **weights** as some numerical characteristic which should be associated with **every** neurons, declared as first parameter of our method. Because we're talking about every neuron, it means that whole amount of weights will be `10368*512 = 5308416`

Also every neuron has his own **bias**. Bias is the characteristic of every neuron as well, so we will be having 512 biases here.

Now we have add more complexity to our computational process, because we have to introduce the weighted sums for every neuron. And the formula using which we will be calculation this parameter will be the following:

$$z_j = \sum_{i=1}^{10,368} w_{ij}x_i + b_j$$

What do we have here ?

z_j — The result of our calculation

w_{ij} — This is the weight connecting i and j neuron. Don't forget that our input is the one-dimensional structure, with some values in amount 10368 and with every neuron we have the array of values, named weights, associated with the certain neuron.

x_i — Our input. Basically this is every certain element from out vector with 10368 items.

b_j Bias for j-th neuron

Here we have the except for our input x_i two arrays as well - weights and biases. Concretely for this step of calculation both of them will be initialised by some random values - weights by values which are close to zero, biases by the zero. I'm going to illustrate this using Python code, which you can try execute to get the full comprehension of what is going on. I will show the screenshots to keep you in context anyway and you will be able to compare what I do have and what you do have.

The following example for the such invocation `Dense(1, activation='relu')` - for sure, it has no technical sense and used only for demonstration purposes.

So we have the only one neuron. Let's get the arrays of weight for it - 3 only for simplicity:

```
np.random.seed(0)  # For reproducibility
weight_vector = np.random.uniform(-0.05, 0.05, 10368)
```

My output looks like that from Google Colab:

```
import numpy as np
```

```
np.random.seed(0)  # For reproducibility
weight_vector = np.random.uniform(-0.05, 0.05, 10368)
```

```
print(weight_vector)
```

```
[ 0.00488135  0.02151894  0.01027634 ... -0.03281221 -0.02656741
  0.03966115]
```

For sure, Python prints only small part from this array. The ranges of initialisation are chosen randomly, but pretty close to the real ones - small values, close to zero.

The bias is just zero.

```
bias = 0.0
```

The input data for the procedure - pls, remember, that our real input is from the previous one step. Here is the demonstration.

```
input_vector = np.random.rand(10368)  # Example input vector
```

```
[4]  input_vector = np.random.rand(10368)
```

```
print(input_vector)
```

```
[0.03187045 0.14748451 0.51816987 ... 0.0332969  0.91009476 0.59239908]
```

Next step according to our formula, given before:

```
z = np.dot(weight_vector, input_vector) + bias
```

And his result is (sum of the pair plus bias, which is the zero for a while):

```
[8]  z = np.dot(weight_vector, input_vector) + bias
```

```
print(z)
```

```
-3.846068718579327
```

And the last step - ReLU activation

```
a = max(0, z)  # ReLU activation
```

Basically it's about getting either result of our computation if it is positive, or getting zero if not. You can consider this as getting absolute value of number or getting module of number in mathematic.

So this is kind of "dense" happened, when we from the pretty long input get something convoluted into the scalar value instead of vector. Also, pls, don't forget that all just is only for one neuron, but according to the parameter in our invocation we have the 512 such ones.

If at this moment you catch the though that you don't understand why exactly we do all these things and what the essentials of weights and biases are and why do we initialise them either zero or random small values - I have to confess I understand you very well. Please, read further and I hope you'll get the answer below.

Why do we do this ? Preparation for the training - open the carts

Speaking about the final purpose of our calculation, we're talking about Image Classification. Or we can rephrase this in the following way - we have ordered our set of pixels of image in some canny way (literally we have 1D-sequence of pixel's value) and we have to compare what we have as result with what we expect for. So since we have the simple case in terms of classification - binary classification

or dogs-cat classification - we can say more certainly our purpose: let's calculate something overall our pixels of some unknown image and the result we have to compare with the dog. Or, more concretely, let's check whether this value is close to dog or whether this value is close to 1.

So in every case for every neuron (having these things with the weights and biases) we are able to make such comparison - comparison with 1.

Due to this fact - fact that we the calculated value and some expected value - we get the possibility to talk about bias not only in the common way that such bias exists, but we even can calculate such bias relatively easily.

Now let's consider another one moment. Why, basically, if we have such formula $z_j = \sum_{i=1}^{10,368} w_{ij}x_i + b_j$ to calculate the output parameter z_j - why do we think at all that some synthetic calculation result being multiplied and summed up using the set of weights (which are, let me to remind, just small randomly initialised values) plus bias which is zero at the stage of start - may give us something, which we should be able to compare with 1 or 0 ?

If you feel perplexity - then you are not the one. If we leave our model as is $z_j = \sum_{i=1}^{10,368} w_{ij}x_i + b_j$ and we will do the calculation only once time - it would be the most futile action in the world. Really. But secret here is that our model has some variable parameters which are weights and some parameter which allow us to say how far or close our result to what we are looking for - I mean bias. And due to this amazing fact we can do something - we can hone our variable parameters in such way - then it will be so close to 1 or zero as far as you wish. Basically this honing is that exactly what the second word in the "Machine Learning" expresses. We're going to learn our model to make what we'd like to get - we're going to change weights!

So no magic at all!

Let’s return back to the example in the previous chapter. We did the example using the sole neuron. But according to our Python’s invocation we have the 512 such neurons and it means that we have to repeat this at least 512 times.

We have to keep in mind that for every neuron the new set of weights and new values of bias will be generated. And further - in the process of learning - these weights will be changed in the way that our z_j will be relatively close to 1 as long as we’re trying to identify dogs on our process of binary image classification.

Ok, we’re moving closer and closer to the final stage of our performance. And the one of our stop is `Dropout(0.5)`.

Invocation Dropout(0.5)

Let me to formulate what this invocation does (buy the way, this is good case of naming of functions) and after that we discuss why do we need this.

As you remember in the previous step we get 512 neurons, where with each of them we have associated the set of weights and bias. In this step - it may sound pretty strange, but nevertheless - we will drop put randomly neurons, which we obtained from the previous step. Literally the parameter in invocation of `Dropout(0.5)` tells that 50% of neurons have a chance to be deleted from the calculation process. Let me to visualise this in this way.

For simplicity we can consider that from the previous step - when we do the dense of our neurons - we get only six ones in the layer #1 (in reality it was 512)

```
Layer 1 (Dense):
[Neuron1] [Neuron2] [Neuron3] [Neuron4] [Neuron5] [Neuron6]
```

So for the following three successive iteration we will get the following states of out first layer:

```
Iteration 1:
[Neuron1] [Neuron2] [0] [0] [Neuron5] [Neuron6]
```

```
Iteration 2:
[0] [Neuron2] [Neuron3] [0] [Neuron5] [0]
```

```
Iteration 3:
[Neuron1] [0] [Neuron3] [Neuron4] [0] [Neuron6]
```

So if you observe on the behaviour which is presented here, you may notice that on every iteration approximately the half of the neurons have a chance to be removed from training and being replaced by the zero, which is indicated in my schema like `[0]`

If you are wondering why we need to remove some calculation parts from our calculation steps - I can give you list of pretty unobvious reasons, which basically will explain you nothing if you read this first time. The main reason is the preventing overfitting - the behaviour of our model when it works fairly good on the training data (I hope you still remember that above we separated our dataset in two parts - training and validation), but fails on the validation data. Don't ask how somebody discovered that exactly drop out of some computation segments from our chain works in this way - intuitively you can accept this like making the conditions of our model worthier than it might be. It's considered that it increases the duration model and how good it predicts because it adds the element of randomicity to the process of learning. It makes sense to add that dropout works only while the learning happening, but it's switched off when the validation takes place.

Invocation Dense(1, activation='sigmoid')

You may remember that we have already deal with this method, when we discussed 512 neurons, weights and biases. This is the final layer of our model and this single one neuron represents the output of our neural network. Simply speaking, this item gives us some value which will give us the final answer - is this set of pixel presents a dog or cat at least ? More literally you can imagine this like the bunch of the connection lines which go from the every neurons of the previous step to the single one neuron, which is final output of our layers.

The `activation='sigmoid'` needs to be explained. Basically for this neuron we have the same story as the previous ones - we have the weighted sum of outputs (and all stories about weights to be randomly initialised and bias to be equal zero in the first iteration are the same as for previous `Dense` invocation)

$$z = \sum_{i=1}^{N} w_i x_i + b$$

and some activation function which is in this case not so trivial as it was in previous chapter, when we kept observing `activation='relu'` .

So this activation function looks like this:

$$y = \sigma(z) = \frac{1}{1+e^{-z}}$$

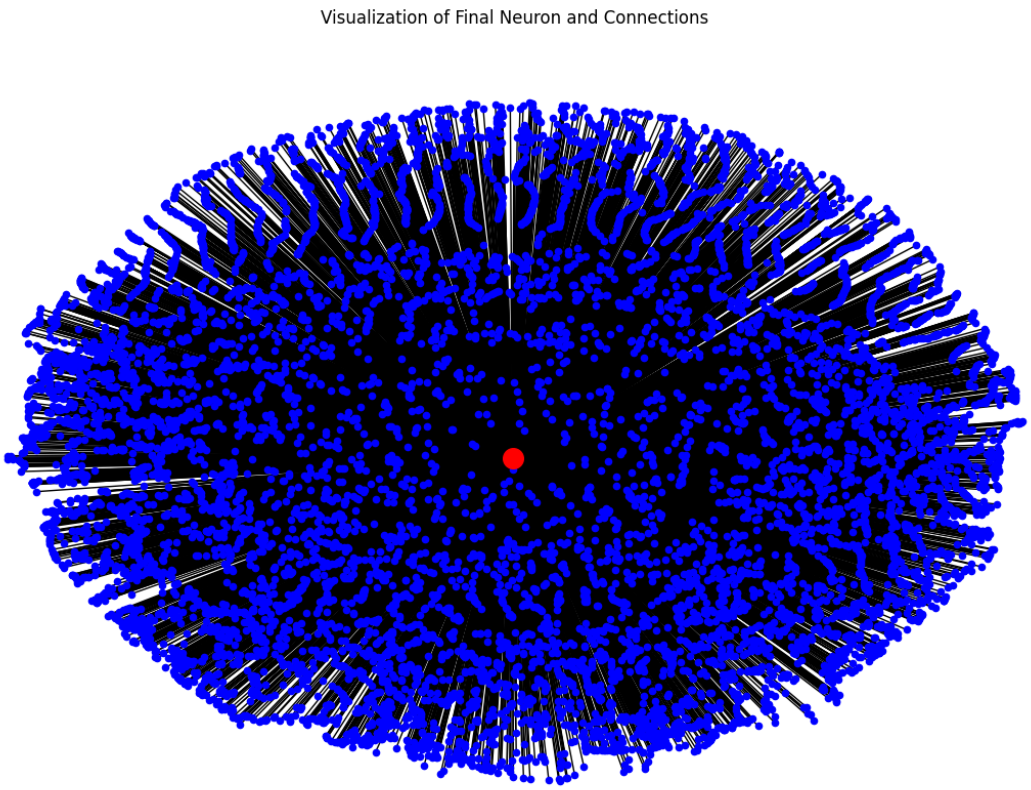
As you can the the calculated value if y is always less then 1, so we can speak about probability, showing us the likelihood that given image is dog.

If you want to see how the visualisation of this last final layer on you own - you can try to execute this code (be careful and tolerant - execution uses real number of neurons - 10368 - so process is not flashlight):

```
import matplotlib.pyplot as plt
import networkx as nx

# Create a graph
```

```
G = nx.Graph()

# Number of neurons in the previous layer
previous_layer_neurons = 10368

# Add nodes for the previous layer
for i in range(previous_layer_neurons):
    G.add_node(f'Prev_{i}')

# Add node for the final neuron
G.add_node('Final')

# Add edges between previous layer neurons and the final neuron
for i in range(previous_layer_neurons):
    G.add_edge(f'Prev_{i}', 'Final')

# Draw the graph
pos = nx.spring_layout(G, seed=42)
plt.figure(figsize=(12, 8))
nx.draw(G, pos, with_labels=False, node_size=20, node_color='blu
nx.draw_networkx_nodes(G, pos, nodelist=['Final'], node_size=200
plt.title('Visualization of Final Neuron and Connections')
plt.show()
```

but I can show you anyway. The legend for this schema is pretty simple: big red dot in the center is our final neuron, created by `Dense(1, activation='sigmoid')` and the plethora of the blue points are out connections of neurons which comes from the previous step.



And okey - let me to congratulate you. Now we achieved pretty big milestone in understanding what this big piece of code means

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150,
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
```

```
    Dense(1, activation='sigmoid')
])
```

So let me to give you small resume what we obtained and which memory milestones you may keep in your memory.

Resume for creation the model

1. We build model in order to make Image classification
2. We calculate nothing now yet, calculation will be done later, that's only recipe how calculation will be done
3. Our model is the sequence of layers.
4. Big part of our layers are convolutional. It means that not all image entirely will be used for learning or what is the same is calculation process. It means that the image will be splitted for some parts, every parts will be featured by the some numeric value characteristic - so the whole image is as be convoluted or wrapped.
5. The future calculation will be going through the first layer to the last layer gradually.
6. Big part of our computation is the definition of the weights and biases vectors in order to make them having such values, that the final output should be pretty enough close to the 1 (if we learn using dogs) or to the 0 (if we learn using cats)
7. In order to achieve this we will go through a plethora of iteration during which the weights and biases will be adjusted.

So, let's go further!

Preparation our model to do something useful

If you can see our code you can notice that we have to go to the following strings:

```
model.compile(loss='binary_crossentropy',
              optimizer=Adam(learning_rate=0.001),
              metrics=['accuracy'])
```

By and large if the previous step was about definition the model in general like a sequence of layers, this step is to define the certain numeric parameter which should make impact on our model. Simply speaking we define the rules how good our model will be learning and how fast it will happen.

For sure we have to consider the parameters of this invocation:

`loss='binary_crossentropy'` - this loss defines how well our model matches to the labels. This function is developed certainly for such type of image classification - I mean binary classification - and the formula of this loss looks like that (to be honest if you read this first time this part is kind of useless):

$$\text{Binary Crossentropy} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i)\right]$$

`optimizer=Adam(learning_rate=0.001)` - basically that is what defines the parameters of changing the weights. If you remember, we had been talking about the arrays of weights and biases, which are initialised by some random values, close to zero, or just zero, if we talk about bias exactly. This parameter defines the step of changing weights during training process in order to bring the value of

$$z_j = \sum_{i=1}^{10,368} w_{ij}x_i + b_j$$

to be so close to 1 as possible. Adam is **Adaptive Moment Estimation** is an optimization algorithm that combines the advantages of two other extensions of

stochastic gradient descent: AdaGrad and RMSProp. It adapts the learning rate for each parameter.

`metrics=['accuracy']` - this is some kind of feedback system which allow us to understand what is happening inside model during training process. You can consider this as metric which can be obtained from the model and using which we can judge how well all is going.

Do a training!

We are swiftly getting to the final stage of our goal and in the end we can see the main stage of our process:

```
# # Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=validation_steps
)
```

This is exactly the point when calculation begins. In this point our model starts to be changed (what is literally means - weights and biases are changing) and it happens not immediately of course, but during some count of iterations, which are in terms of Machine Learning named as epochs. You can notice that this is single parameter which is hardcoded here and that's exactly the count of iteration, which through our model has to come to obtain something, which should be saved to file in oder not to repeat the learning process again and again.

Let me to remind you what `train_generator` is:

```
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary'
)
```

`validation_generator` is very similar:

```
validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary'
)
```

So both of them are objects which have our images prepared for training by the reading then from directories.

We need to achieve the comprehension what `steps_per_epoch=steps_per_epoch` and `validation_steps=validation_steps` are.

Let me to add the snapshot:

```
# # Calculate steps_per_epoch and validation_steps
steps_per_epoch = train_generator.samples // train_generator.bat
validation_steps = validation_generator.samples // validation_ge
```

Checking this code we may to notice, that the values, which we are interesting in, are the result of division the some properties of the corresponding objects.

Actually we may leave this as is, but because we'd like delve in all this things, it makes sense to open this secret as well. Earlier we defined two objects:

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

and more simple:

```
validation_datagen = ImageDataGenerator(rescale=1./255)
```

If you run out script you may to see that in console two strings which on my laptop look like that:

```
Found 24998 images belonging to 2 classes.
Found 23652 images belonging to 2 classes.
```

The number, which we can see in the first line, is the amount of files, loaded by `train_datagen` and which are stored in `train_generator.samples` . Absolutely the same we can see about the second line, but only for `validation_datagen` . You can put some efforts and count if you want, but in more literal sense it is the number of files on out dataset, in the training and validation datasets correspondingly. And because the our training process will be not the one by one files, but by the some batch, we can easily conclude that the default value of this batch is 20, what exactly we can see in our logs.

So as result we can print out the following values:

```
train_generator.samples : 24998
train_generator.batch_size : 20
Steps per epoch: 1249
Validation steps: 1182
```

and move further.

The learning and the its results

First of all, be patient. This process is not fast. Depending of your computer - I'm doing this code and article on Macbook Pro M3 Pro - it really takes time. Moreover, we have to care about the results of our computational process, really because this costs some time and if before image classification we would start the training process every time, we easily would have been identified as useless resource consumer. So we're going to save something which is result of our work to the dedicated file with extension `h5` in order to you will be able to use the results of learning process in another script.

So when our weights and biases will be pretty close to 1 or 0 after the learning will be finished, we do such thing in our code:

```
model.save('cats_vs_dogs.h5')
```

Let me repeat - this is exactly what we need. This is the sequence of some maps with averaged images of either dogs or cats, with which the image which has to be classified in some particular case, will be compared.

So that's time to launch our script and see what's is happening. I really hope if you read till this moment, you are able to launch Python code, so let me to skip this

step. One thing which has to be mentioned here, if you have never done such things before - it takes time and looks like that.

```
Epoch 1/30
/Users/eugene/IdeaProjects/mine/neural networks/lib/python3.12/s
self._warn_if_super_not_called()
1101/1249 ━━━━━━━━━━━━━━━━━━━━ 15s 102ms/step - accuracy: 0.500
warnings.warn(str(msg))
1249/1249 ━━━━━━━━━━━━━━━━━━━━ 164s 130ms/step - accuracy: 0.50
Epoch 2/30
1/1249 ━━━━━━━━━━━━━━━━━━━━ 3:22 163ms/step - accuracy: 0.6500
[[{{node IteratorGetNext}}]]
/opt/homebrew/Cellar/python@3.12/3.12.4/Frameworks/Python.framew
self.gen.throw(value)
2024-07-30 21:59:00.769309: I tensorflow/core/framework/local_re
[[{{node IteratorGetNext}}]]
1249/1249 ━━━━━━━━━━━━━━━━━━━━ 0s 80us/step - accuracy: 0.6500
Epoch 3/30
644/1249 ━━━━━━━━━━━━━━━━━━━━ 1:01 102ms/step - accuracy: 0.494
```

(*Depending on your configuration you may see the warning in console, at least in my case I see the following:*

```
lib/python3.12/site-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121:
UserWarning: Your PyDataset class should call super().__init__(**kwargs) in its constructor.
**kwargs can include workers, use_multiprocessing, max_queue_size. Do not pass these arguments
to fit(), as they will be ignored.
self._warn_if_super_not_called())
```

)

If you read your output meticulously you may to notice that your calculation process is separated in some iterations, named `epochs` and count of such epochs is `30` as we did this on our own in our code above. Also you can see warnings: actually process of training is pretty fragile and can be interrupted by the several

reasons, but because of our case is relatively simple, I dare it won't happen. By the way, this is one of the few amount of tasks when I can hear the fans of my Macbook Pro M3 Pro.

But there is another interesting moment which we need to discuss - in logs you can find the string like that

```
accuracy: 0.5008 - loss: 0.6947 - val_accuracy: 0.4991 - val_loss: 0.6931
```

And it really makes sense to talk about this, but believe me, it would be much better if you wait till the end of your training process and you can extract in some way this string to have the all output with this data. Observing the values we can extract several very alluring details.

Let's discuss what do we have here. First of all there are pairs: accuracy and loss for two types of data - training and validation. The prefix `val_` in this case indicates that we are dealing with validation data.

So having this example - what do we have ?

accuracy: 0.5008 - accuracy of training data for the current epoch, in my case this is first epoch, so the first one iteration of our learning process. The value 0.5008 means that our model is able to classify correctly approximately 50.08% of training samples. Not so good yet, frankly speaking.

loss: 0.6947 - The loss of our model for current epoch, in my case for the first one as well of course. This is basically metric which shows us how well our model match to actual labels. Lower loss indicates better state of things. In our case we'd like to expect more since it's only the first step - or the epoch 1.

Another part of logs has the same sense, but only applicable to validation data.

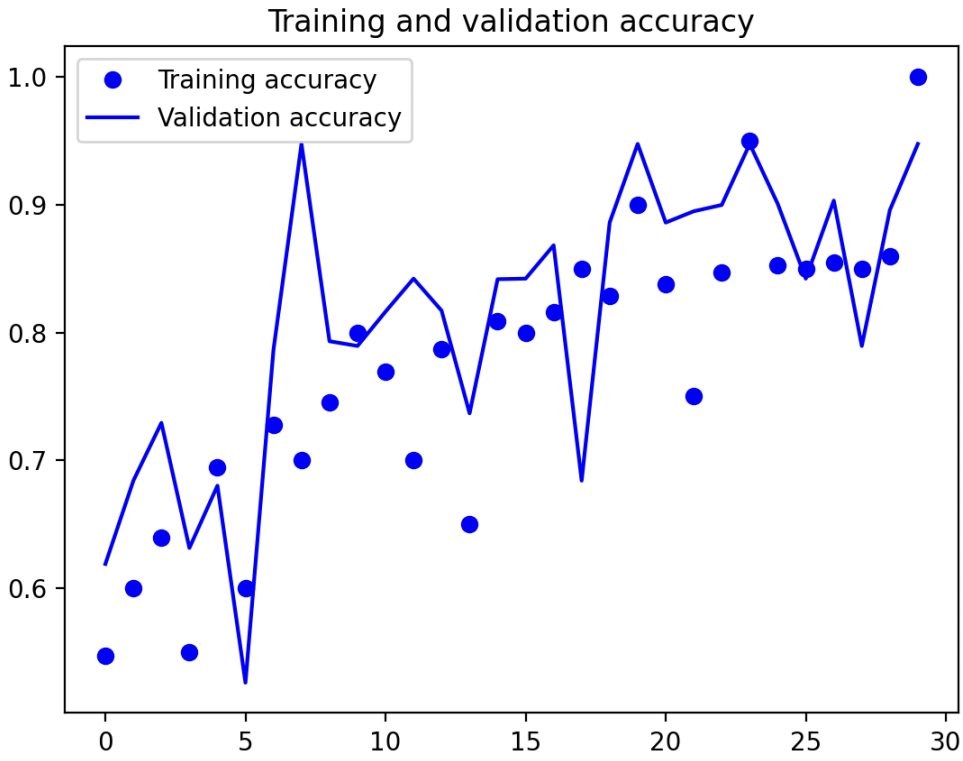
It doesn’t make sense to check out logs of our script to understand the evaluation of our model, because of after the model will be saved, we will try to build the plot to understand how well our model works.

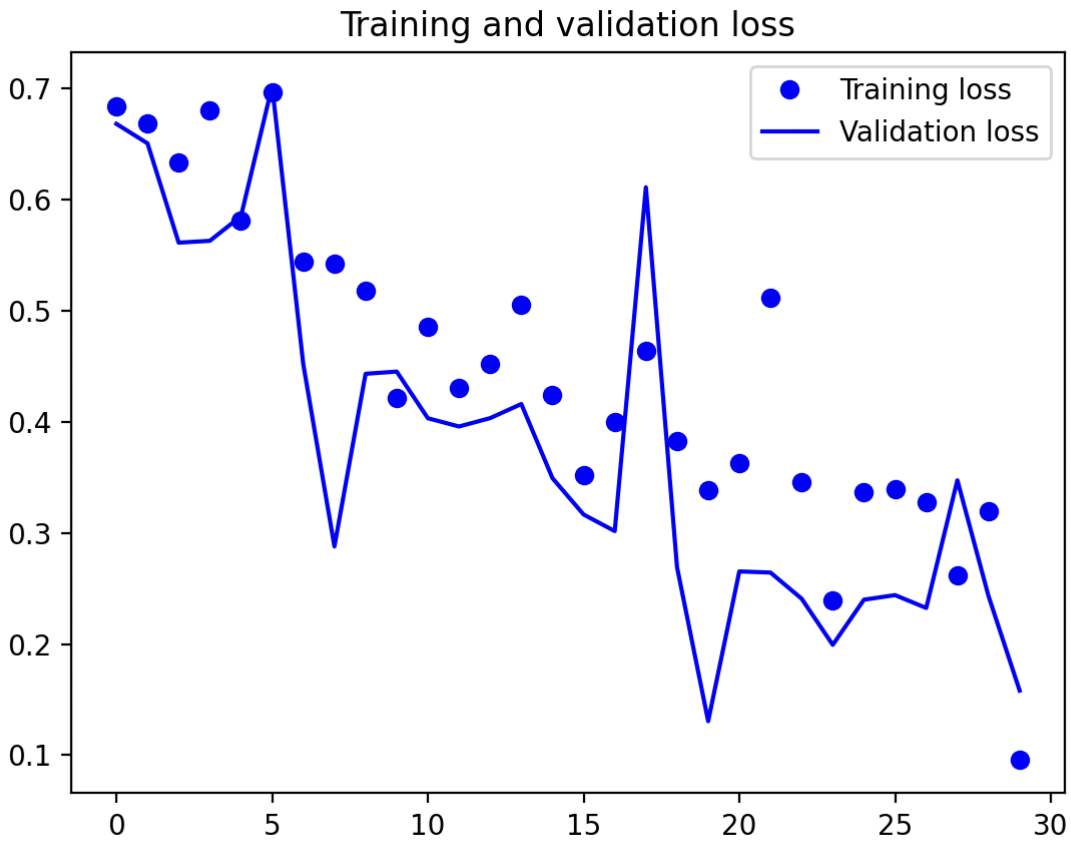
The plot will be built by this code, which is pretty universal (in terms that this will be regular 2D-plot) and doesn’t related directly to machine learning, so I will let myself not to describe it in details.

```
# Evaluate the model
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

So when you learning process will be finished you have to see something like that - two plots will opened automatically. The first one is describing the accuracy of

classification, the second one describe the loss, with which this classification happens.





It makes sense to discuss the both plots in details.

1. **X-Axis**: Epochs (from 0 to 30) Each point on the X-axis represents one epoch, which is one complete pass through the entire training dataset.
2. **Y-Axis**: Accuracy (ranging from 0.5 to 1.0) This measures how accurately the model is able to classify the training and validation data.

First of all, if you can see the first plot with accuracy, you may notice that training accuracy slightly increases, right ? Of course, there are several very distinguished fluctuations (I mean the blue points which are laying pretty far away from the curve, which can be drawn through the majority of points), but by and large we can make the conclusion that, in spite of our dataset is fairly primitive and count of

training epoch are restricted by 30, our accuracy increases and at the last position we can evaluate her value which is pretty close to 0.9. Shortly speaking, it’s well.

The situation with validation accuracy is more difficult. The fluctuations are more prominent, but in general we can conditionally to say that validation accuracy increases as well, but with much less quality.

The combination of these two observations (relatively good training accuracy and much less quality of validation accuracy with prominent fluctuation) allows to suspect that the model is overfitted by the training data. It means that model works relatively well with training data, even if it contains the noise, but when the model tries to apply herself to the validation data the thing are getting worth. In general in well-trained model we should not to observe such state of things and in ideal world we have to apply the additional steps to enhance it. The enhancing may include more files of dogs and cats for training, the manipulation with layer’s count of our neural network or/and manipulation with parameters of these layers. Simply speaking, it’s about changing the count of layers and numeric parameters here:

```
model = Sequential([
Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3),
MaxPooling2D(2, 2),
Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.00
MaxPooling2D(2, 2),
Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(0.0
MaxPooling2D(2, 2),
Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(0.0
MaxPooling2D(2, 2),
Flatten(),
Dense(512, activation='relu', kernel_regularizer=l2(0.001)),
Dropout(0.5),
```

```
Dense(1, activation='sigmoid')
])
```

Verify how well we are

Of course, when we have trained our model and even we have the results of our training process, you would like to test at least what we get. We can do this in separated script, which implements several goals:

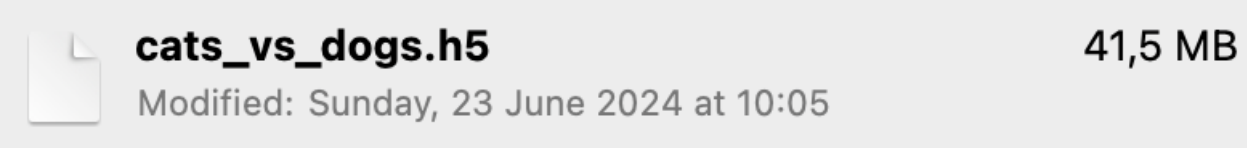
1. accepts file to be classified whether it is cat or dog
2. loads our trained model
3. actually classification

Let me to emphasise one more time that learning process will be done once time only (if you don't want to make experiments with epoch counts for example or any another parameters) and for every new invocation of our classifier we will be using the already trained model, which will be saved in file by this code:

```
model.save('cats_vs_dogs.h5')
```

In order to let you feel what do we have, saying "trained model", I can show you how it looks like. If you check out the size of our dataset (you downloaded him from Microsoft's site in the beginning of our trip), you can find out that it will be around `1 673 718 106 bytes (1,77 GB on disk)` - of course, in my case.

So the file of trained model has the following size:



You can effortlessly notice that sizes of data set (image files) and trained models are different drastically. The one of the reason of that we applied the convolutional approach which is able to wrap our data significantly and of course we averaged and remove a lot of extra information from files, which are not needed for image classification.

This file is binary to be maximally efficient, but basically you can open it and check the content if you want, but, frankly speaking, you cannot extract something useful from this data:

So our prediction.py looks like this:

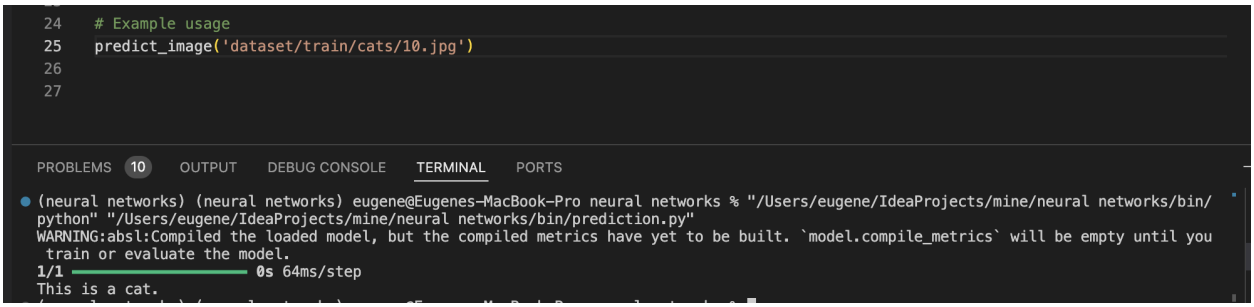
```
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np

# Load the trained model
```

```
model = load_model('cats_vs_dogs.h5')

def predict_image(img_path):
    # Load and preprocess the image
    img = image.load_img(img_path, target_size=(150, 150))
    img_tensor = image.img_to_array(img)  # Convert image to ar
    img_tensor = np.expand_dims(img_tensor, axis=0)  # Add batc
    img_tensor /= 255.0  # Normalize the image

    # Make a prediction
    prediction = model.predict(img_tensor)

    # Interpret the result
    if prediction[0] > 0.5:
        print("This is a dog.")
    else:
        print("This is a cat.")

# Example usage
predict_image('KITTEN-WITH-A-MOUSE-150x150.jpg')
```

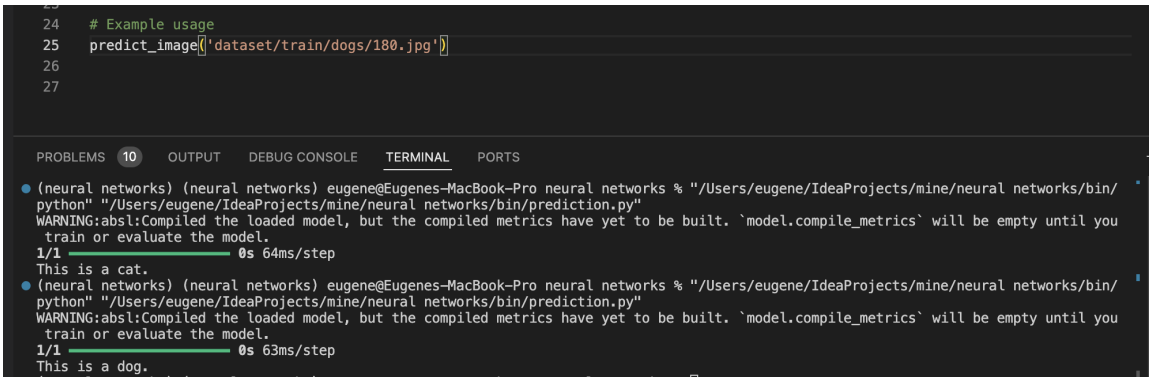
Since I believe if you read this and moreover you read till this place, you have a least small experience in developing and I don't need to explain you what is happening in this code.

Let me just describe my scenario of verification, which can be applied to verify how well we learned our model.

First of all I will take a image from training data folder and of course I will put the name of this file as parameter of method `predict_image()` . So what do I have, putting the image from the /cat folder - small nondescriptive label "This is cat" is telling me that, it looks like, our weights and biases works relatively well in this case.

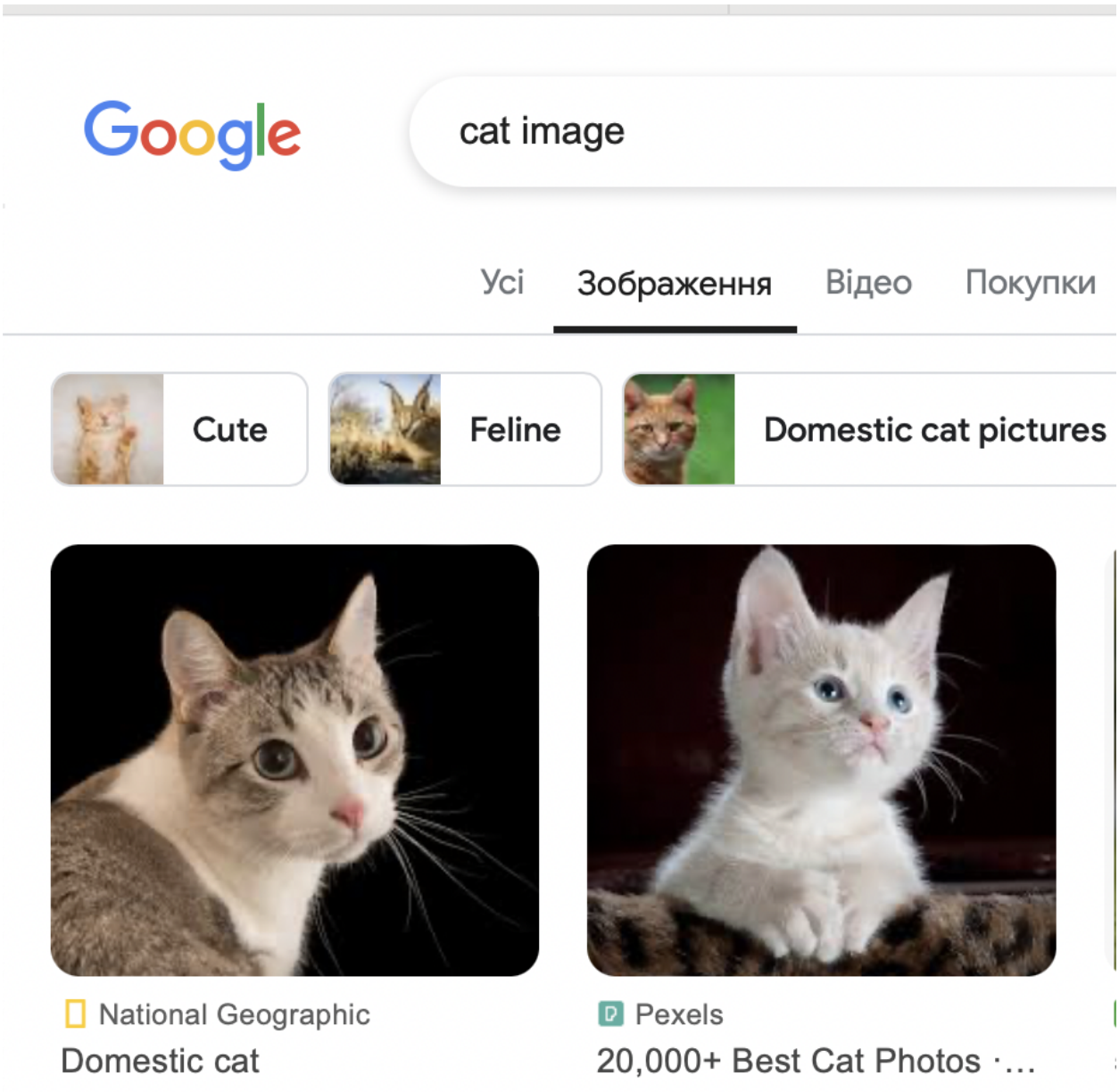


But let's go further and suggest to identify whether it is dog - and it will be, sure. Let me to show the result on my side:



As you can see the model was right again - the file really represents dogs. So it seems again that .... No, let's have a real test, using real image from internet. I'm going to show whole process on my side.

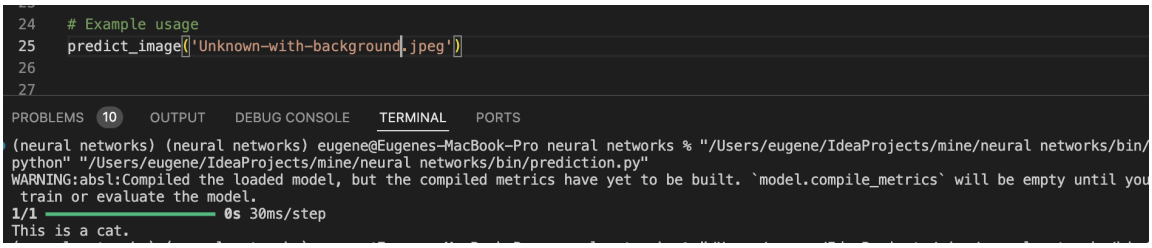
This is my result of Google Images search and I'm suggesting to our model to identify what is on the first image, without any edit with downloaded images.



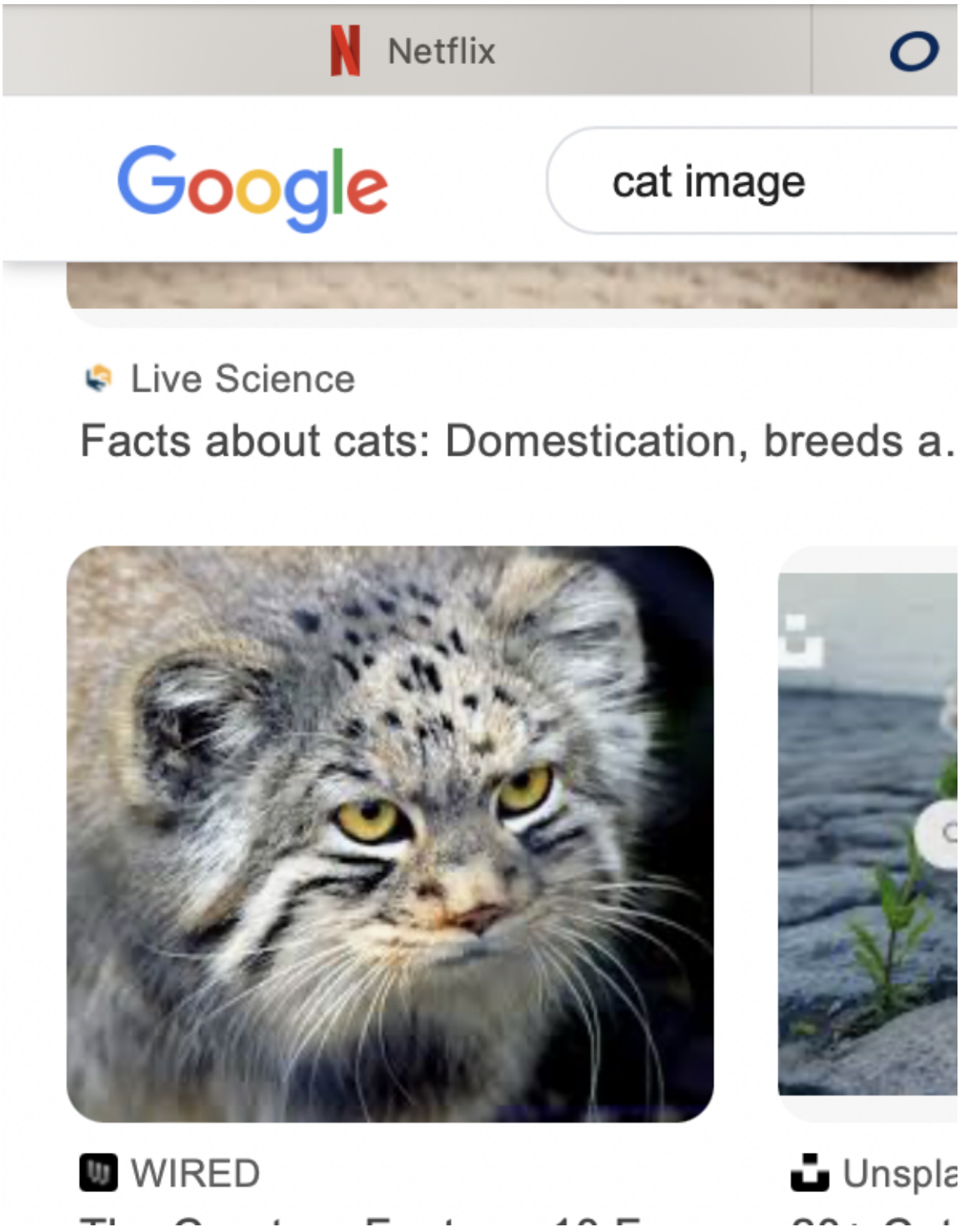
As you can see the model is right again!

```
25  predict_image('Unknown.jpeg')
26
27

PROBLEMS 10  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

(neural networks) (neural networks) eugene@Eugenes-MacBook-Pro neural networks % "/Users/eugene/IdeaProjects/mine/neural networ
python" "/Users/eugene/IdeaProjects/mine/neural networks/bin/prediction.py"
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty un
 train or evaluate the model.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
This is a cat.
```

Let’s try something new - with another background or just noice. I’m going to use cat from Google Search again and which exactly I bordered it by the red box.



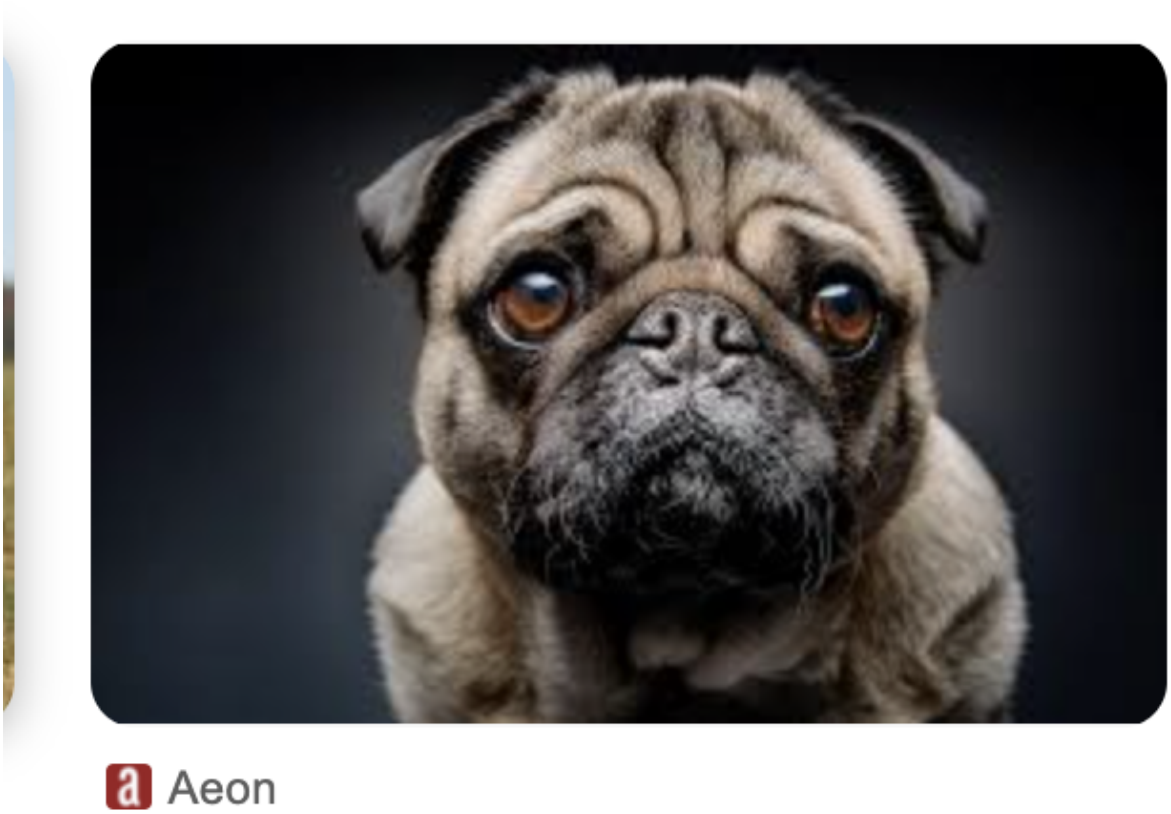
Interesting! It still works! So now is the time to propose something really strange:



So these combination of pixels, despite of out plots, which may bring doubt, still can be identified as a cat by our model.

```
24  # Example usage
25  predict_image('Unknown-fat-furry-cat.jpeg')
26
27

PROBLEMS 10  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

(neural networks) (neural networks) eugene@Eugenes-MacBook-Pro neural networks % "/Users/eugene/IdeaProjects/mine/neural networks/bin/
python" "/Users/eugene/IdeaProjects/mine/neural networks/bin/prediction.py"
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you
 train or evaluate the model.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
This is a cat.
```

So that’s good time to check what about dogs ?



So .... it might seem strange, but it still works!

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

/eugene/IdeaProjects/mine/neural networks/bin/prediction.py"
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until
you train or evaluate the model.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
This is a dog.
```

OK, I hope it’s pretty understandable how you can test your trained model to make the simplest but really working image classification.

Conclusion

Thank you if you read till this point, it means that this trip was interesting as it was for me. Personally for me, when I was thinking about whether should I put some attention to such hype topic as machine learning, it seemed to me like some sort of magic, but it reality it’s been realised that approximately 80% of what called “machine learning” is part of almost every course of discrete math and relatively small part is really founded on probabilistic methods, which are hardly learned in non-mathematical university courses.

So small resume what it make sense to have as milestones in you memory:

1. We need to have the datasets if we want to “learn” our model.
2. This dataset should be separated in two parts - for training and validation.
3. Neural networks is layered structure which process the set of pixel of particular image in some canny way.
4. The learning process is the iteration process, because the set of coefficients and errors, names the weights and biases correspondingly, should be adjusted in order to meet the expectation of our model.
5. Main idea of learning is the generalisation of images to abstract structure, which keeps the combinations of pixel, which, after learning, identify the objects which we’re trying to classify.
6. How all this was discovered and was realized - it is absolutely another story.