


Cat Breeds Classification

-Phạm Thiên Nhật, Trần Giang Anh, Nguyễn Minh Tùng-

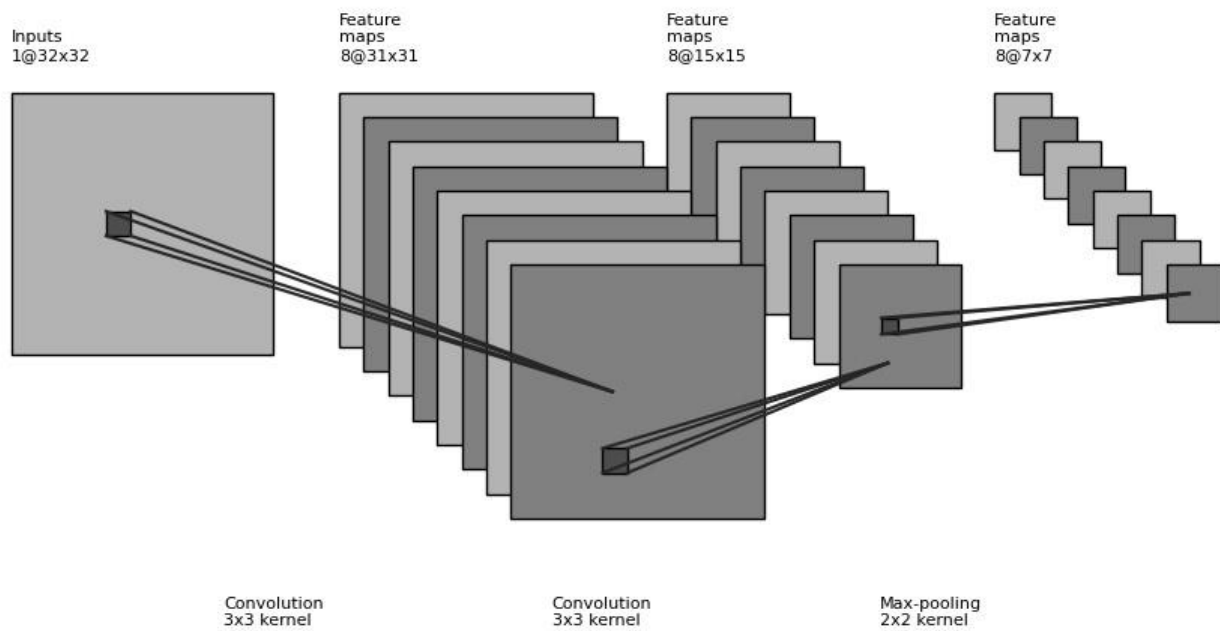
1, Problem:

Given images of cats, the model is expected to give a prediction of the cat's breed in 1 of the 4 breeds below:

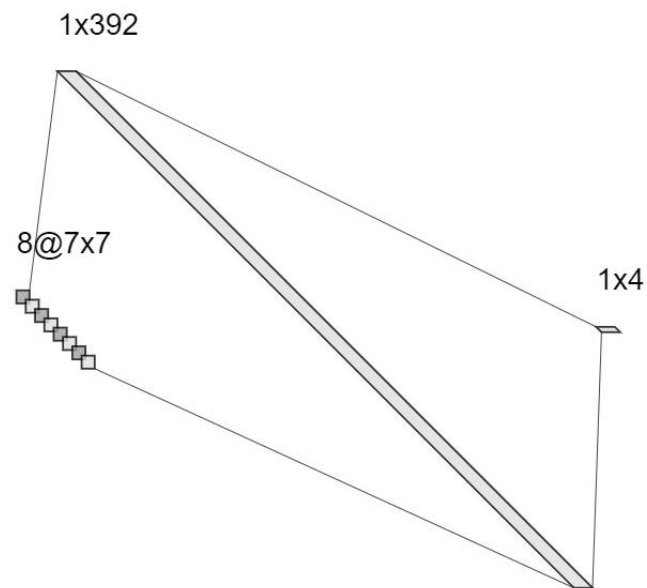
| Munchkin | Persian |
|---|--|
|  |  |
| Siamese | Sphynx |
|  |  |

2, Approach:

In order to solve the problem, we decided to use a ConvNet as it's much more efficient than usual Neural Network at the same task of classification. An illustration an architecture we came up with and designed can be seen below:



(Convolutional Layers Stacking)



| Layer | Filter Size | Stride | Activation | Input Shape | Output Shape |
|---------------|-------------|--------|------------|-------------|--------------|
| Input | - | - | - | 64x64 | 1x64x64 |
| Convolutional | 3 | 2 | Leaky ReLU | 1x64x64 | 8x31x31 |
| Convolutional | 3 | 2 | Leaky ReLU | 8x31x31 | 8x15x15 |
| Max Pooling | 2 | 2 | Leaky ReLU | 8x15x15 | 8x7x7 |
| Dense | - | - | - | 1x392 | 1x4 |
| Output | - | - | - | 1x4 | 1x4 |

The formula used to calculate the Shape of convolutional output is as below:

$$\left\lfloor \frac{(W - K + 2P)}{S} \right\rfloor + 1$$

W = Input size

K = Filter size

S = Stride

P = Padding (=0 in our case)

We decided to implement Leaky ReLU instead as normal ReLU usually suffer from the “dying ReLU” problem which may kill potential neurons, preventing it from activating again. This is a huge problem as our architecture has a lot neuron.

We implemented each layer type as a class, all has 2 common functions:

1. **forward()** : Perform forward propagate used when predicting, training, returning the output after processing the input.
2. **backward()** : Perform backward propagate which is used mostly in training, it update the weights of the layer based on the loss value returned to the layer, then return the loss value to the layer before it.

The choice of fixing the size of the kernel to be 3x3 is to reduce the computational costs, while choosing a stride equal to 2 let us downsample the image smartly, making the architecture more expressive than abusing the use of pooling layers to downsample.

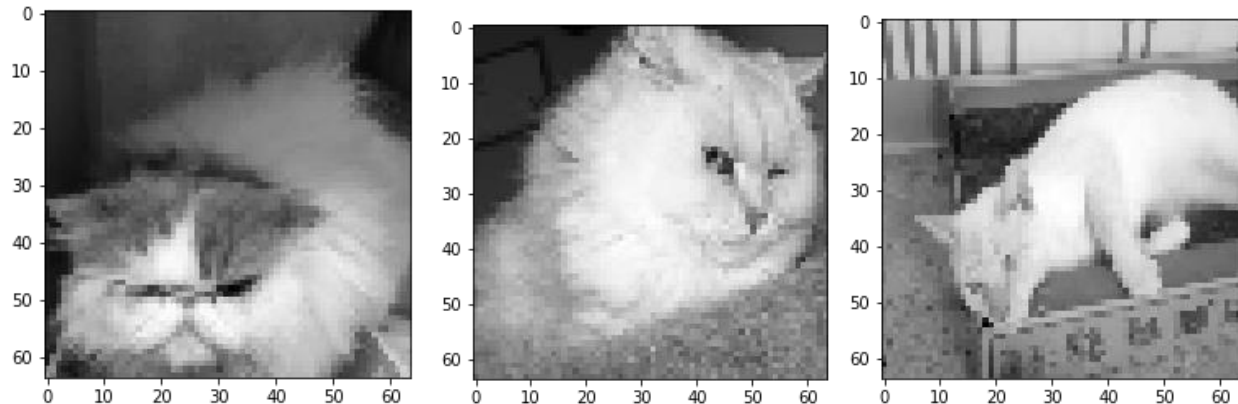
We also implemented L2 regularization, although it is not used yet as there are still bugs (?). The basic idea of L2 can be described as below:

$$Cost\ function = Loss + \frac{\lambda}{2m} \sum \|w\|^2$$

L2 regularization basically forces the weights to decay towards zero (but not exactly zero), hence the term “weight decay”.

3, Dataset:

The data we used was mined from petfinder.com using their APIs. The data contained a total of 7296 images total, divided into 4 breeds of cat as mentioned earlier. The data are fitted and cropped into a size of 64x64, and then grayscaled to reduce the dimension to 1. This is to reduce the data complexity and speed up computation task.



(Example of images transformed)

Each cat breeds are then labelled numerically, indexing from 0 to 4.

Then, we grouped the data into arrays, which is then through the usage of [Pickle](#) library, will be transformed into a single file for caching, which reduce the disk usage tremendously as it tends to become a bottleneck when training.

We divided the data into 2 files separately, one contains the images, and one contains the labels. The final shape of each array stored in each file is as below:

- Images: (7296, 64, 64)
- Labels: (7296,)