Silesian University of Technology
Institute of Informatics

Politechnika
Śląska

Biologically Inspired Artificial Iintelligence

Program for lossy compressing images with implementation of self-organizng map(Kohonen SOM)

author	Marek Żabiałowicz
e-mail	marekzabialowicz@gmail.com
teacher	D.E. Grzegorz Baron
academic year	2018/2019
study type	SSI
major	Computer Science
semester	6
group	GkiO4
section	9
github	https://github.com/marqustd/ImageSOMCompressorCSharp

1 Introduction

The aim of my project was to write a program for lossy compresing images with implementation of self-organizng map(Kohonen SOM) with implementaion one or a couple of alghoritims: Winner Takes All, Winner takes Most, Soft Competition Scheme, Stochastic Relaxation, Neuron Gas.

2 External specification

The program uses graphical window. With usage of buttons and sliders user can upload an image, set up lattice width and height, compress the image and save it.

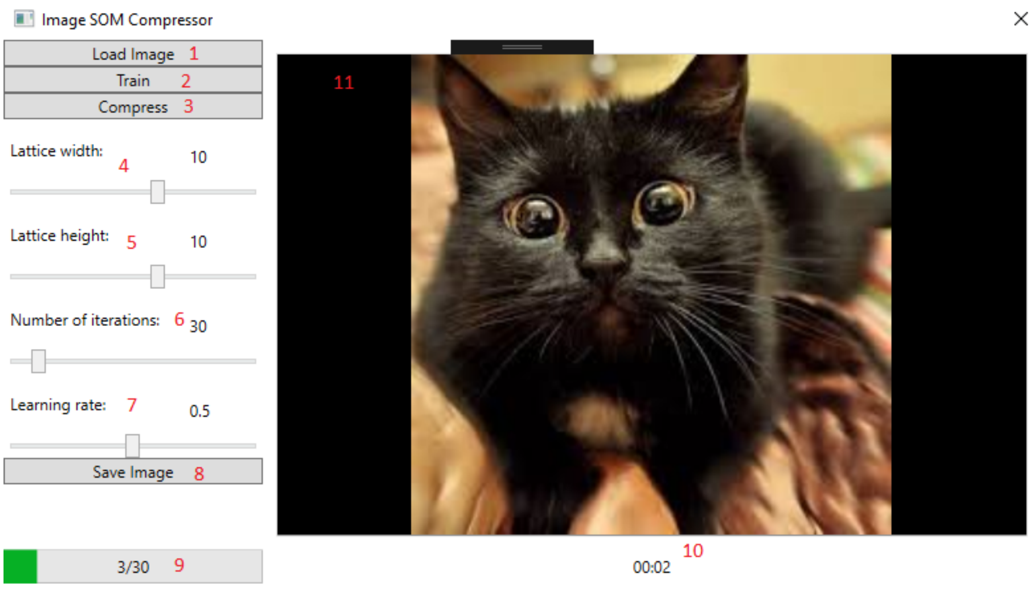


Figure 1: User interface

1 - load the image
2 - train self-organizng map
3 - compress the image
4 - setting latice width
5 - setting lattice height
6 - setting number of iterations
7 - setting a learning rate
8 - save the image
9 - progeress bar with number of current iteration

10 - elapsed time to learn
11 - preview of the image(before and after compressing)

3 Internal specification

The first step in the learning process of self-organizing map is the initialization of all weights(three because of 3 colors RGB) on connections. After that, a first pixel from a picture is used as an input to the network. The network then calculates weights of which neuron are most like the input data (input vector). For this purpose, this formula is used:

$$Distance^2 = \sum_{i=0}^{n}(input_i - weight_i)$$

where n is the number of connection (weights). The map neuron with the best result is called Best Matching Unit or BMU. In an essence, this means that the pixel can be simpled to this neuron.

This means that weights on this connection are updated in a manner that calculated distance is even smaller.

The weights of neighbors of BMU are also modified so they are closer to this pixel too. This is how the whole map is 'pulled' toward this point. For this purpose, we have to know the radius of the neighbors that will be updated. This radius is initially large, but it is reduced in every iteration (epoch). The next step in training self-organizing maps is actually calculating mentioned radius value. Following formula is applied:

$$\sigma(t) = \sigma_0 e^{\frac{t}{\lambda}}$$

where t is the current iteration, σ_0 is the radius of the map. The λ in the formula is defined like this:

$$\lambda = \frac{k}{\sigma_0}$$

where k is the number of iterations. This formula utilizes exponential decay, making radius smaller as the training goes on, which was the initial goal.

When the radius of the current iteration is calculated weights of all neurons within the radius are updated. The closer the neuron is to the BMU the more its weights are changed. This is achieved by using this formula:

$$weight(t+1) = weight(t) + \Theta(t)L(t)(input(t) - weight(t))$$

This is the main learning formula, and it has a few important points that should be discussed. The first one is L(t) which represents the learning rate.

Similarly to the radius formula, it is utilizing exponential decay and it is getting smaller in every iteration:

$$L(t) = L_0 e^{-\frac{t}{\lambda}}$$

Apart from that, we mentioned that the weight of the neuron will be more modified if that neuron is closer to the BMU. In the formula, that is handled with the $\Theta(t)$. This value is calculated like this:

$$\Theta(t) = e^{-distBMU/2\sigma(t)^2}$$

The whole alghoritim we can write in 7 steps:
1. Weight initialization.
2. The pixel is selected from the picture and used as an input for the network.
3. BMU is calculated.
4. The radius of neighbors that will be updated is calculated.
5. Each weight of the neurons within the radius are adjusted to make them more like the input vector.
6. Steps from 2 to 5 are repeated for each input vector of the dataset.
7. Steps from 2 to 6 are repeated till the end.

But after simpilying picture to 256 pixels(lattice 16x16) we need to save the picture.

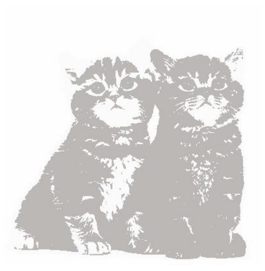
```

private void SaveToFile(string filename, SomFileModel model)
        {
            using (var outputFile = new BinaryWriter(new FileStream(filename, FileMode.OpenOrCreate)))
            {
                outputFile.Write((byte) (model.Neurons.Count - 1));
                foreach (var neuron in model.Neurons)
                {
                    outputFile.Write(neuron.R);
                    outputFile.Write(neuron.G);
                    outputFile.Write(neuron.B);
                }

                outputFile.Write(model.Width);
                outputFile.Write(model.Height);

                var former = model.Input[0];
                byte count = 0;

                for (var i = 0; i < model.Input.Count; i++)
                {
                    if (model.Input[i] == former && count < byte.MaxValue)
                    {
                        count++;

                        if (i == model.Input.Count - 1)
                        {
                            outputFile.Write(former);
                            outputFile.Write(count);
                        }
                    }
                    else
                    {
                        outputFile.Write(former);
                        outputFile.Write(count);

                        former = model.Input[i];
                        count = 1;
                    }
                }
            }
        }
```

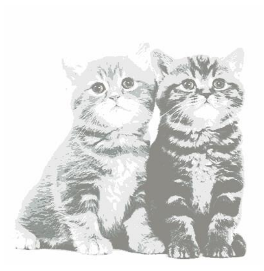
Firstly I write a number of neurons, then describe all neurons each with 3 bytes, each byte for one color(RGB). Next I write picture's width and height and all pixels in format [neuron index—number].

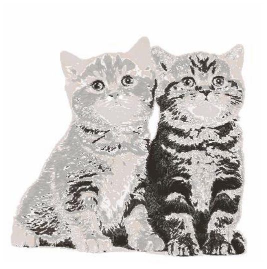
4 Testing the program

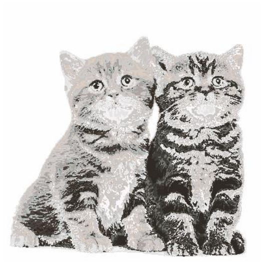
I've tested the program with 3 kind of files - small(255x255), big(400x400) and cartoon and several setups.

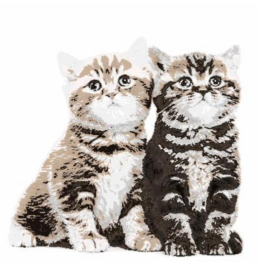
(a) original

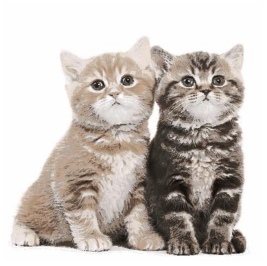

(b) 3x3 5 0.5

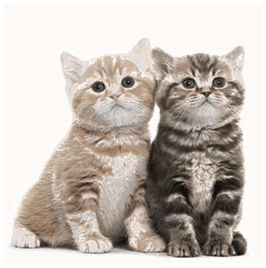

(c) 3x3 30 0.5

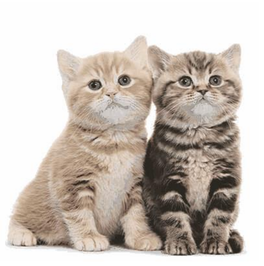

(d) 5x5 50 0.01

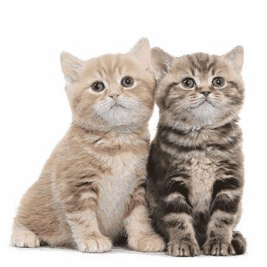

(e) 5x5 255 0.5

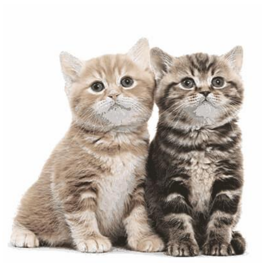

(f) 6x6 30 0.5


(g) 12x12 30 0.5


(h) 16x16 30 0.5


(i) 16x16 50 0.01


(j) 16x16 50 1


(k) 16x16 100 0.5

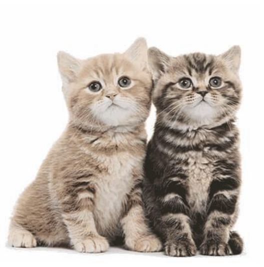

(l) 16x16 255 0.5

Figure 2: big picture - 400x400

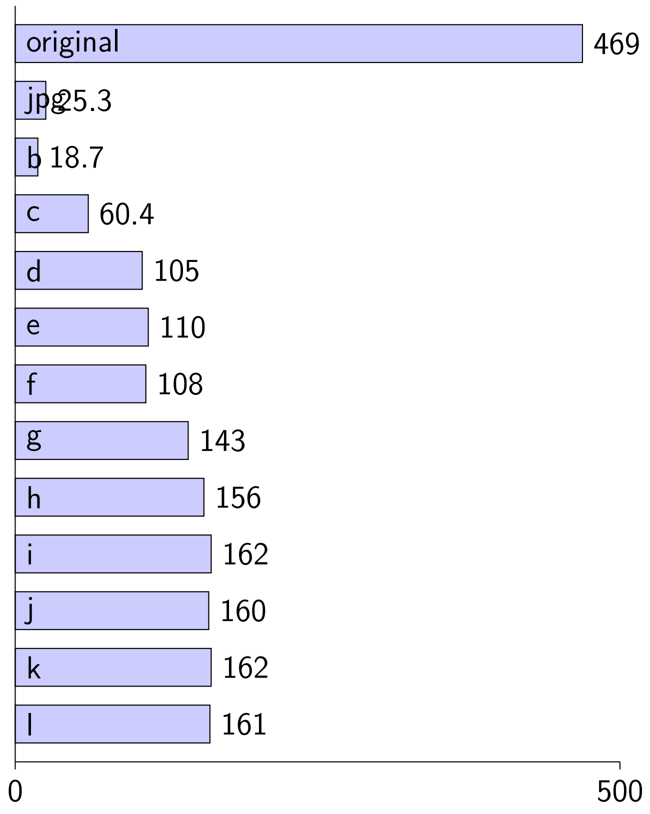
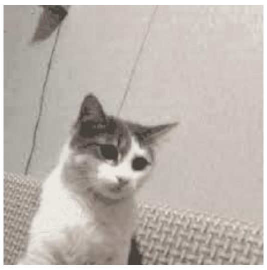
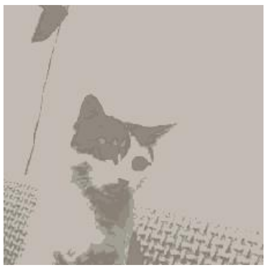
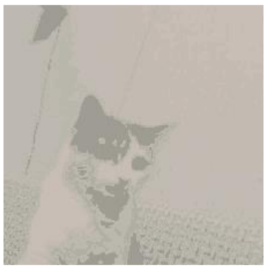


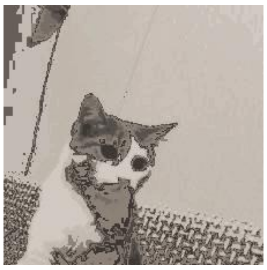
Figure 3: Diference in size in big picture files in kB

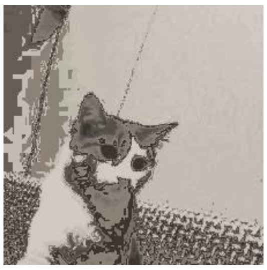
We can see that there's a huge difference between first 5 pictures and the other ones in Fig.2. The quality has improved but also the size of files has increased(Fig.3). Last row is almost the same as original file.

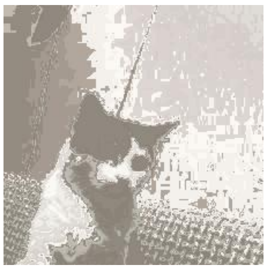

(a) original

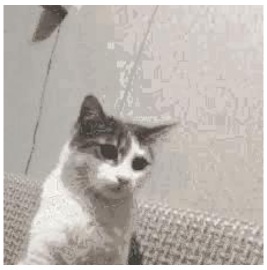

(b) 3x3 5 0.5

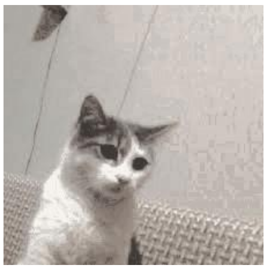

(c) 3x3 30 0.5

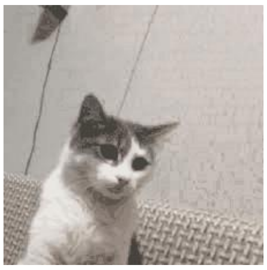

(d) 5x5 50 0.01

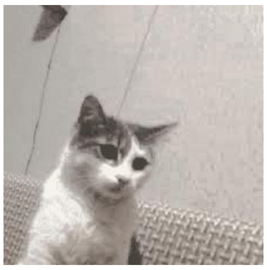

(e) 5x5 255 0.5

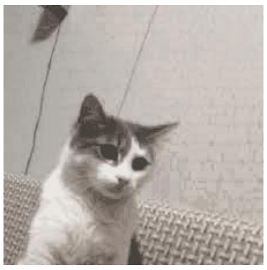

(f) 6x6 30 0.5


(g) 12x12 30 0.5


(h) 16x16 30 0.5


(i) 16x16 50 0.01


(j) 16x16 50 1


(k) 16x16 100 0.5

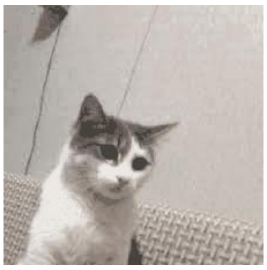

(l) 16x16 255 0.5

Figure 4: small picture - 255x255

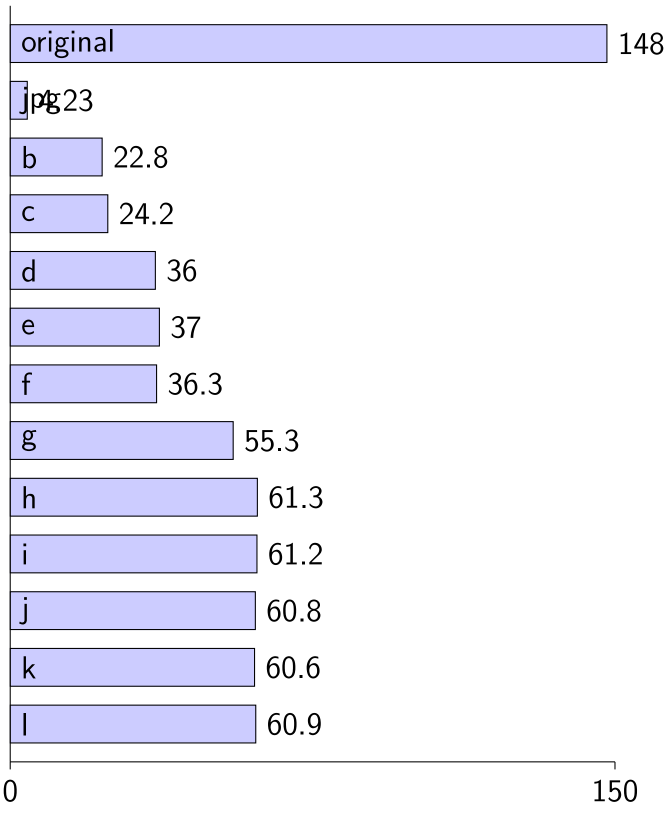
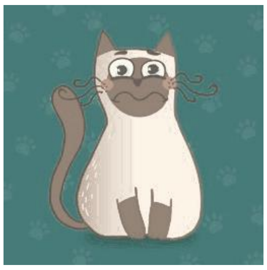
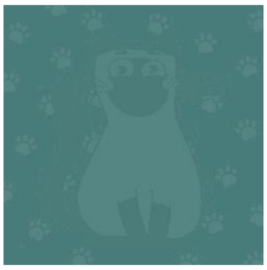
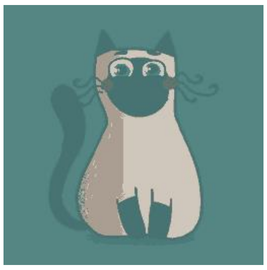
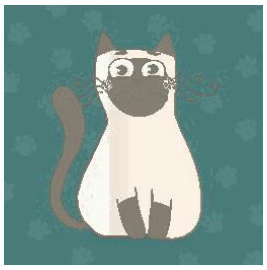


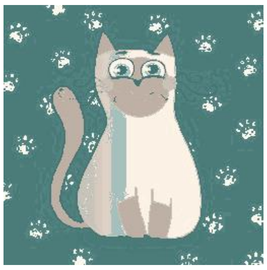
Figure 5: Diference in size in small picture files in kB

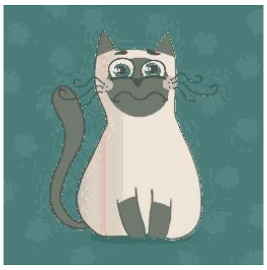

(a) original

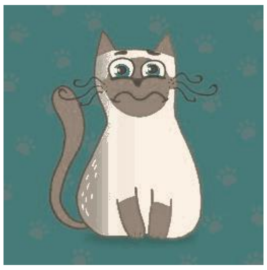

(b) 3x3 5 0.5

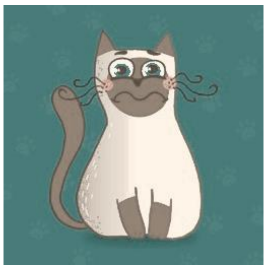

(c) 3x3 30 0.5

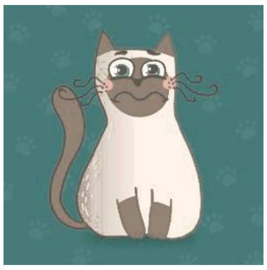

(d) 5x5 50 0.01

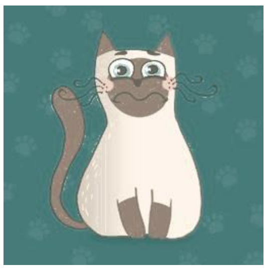

(e) 5x5 255 0.5

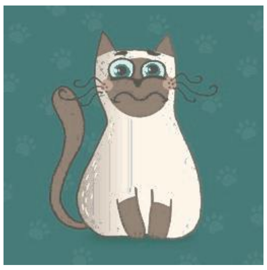

(f) 6x6 30 0.5


(g) 12x12 30 0.5


(h) 16x16 30 0.5


(i) 16x16 50 0.01


(j) 16x16 50 1


(k) 16x16 100 0.5

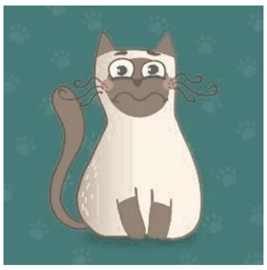

(l) 16x16 255 0.5

Figure 6: cartoon picture

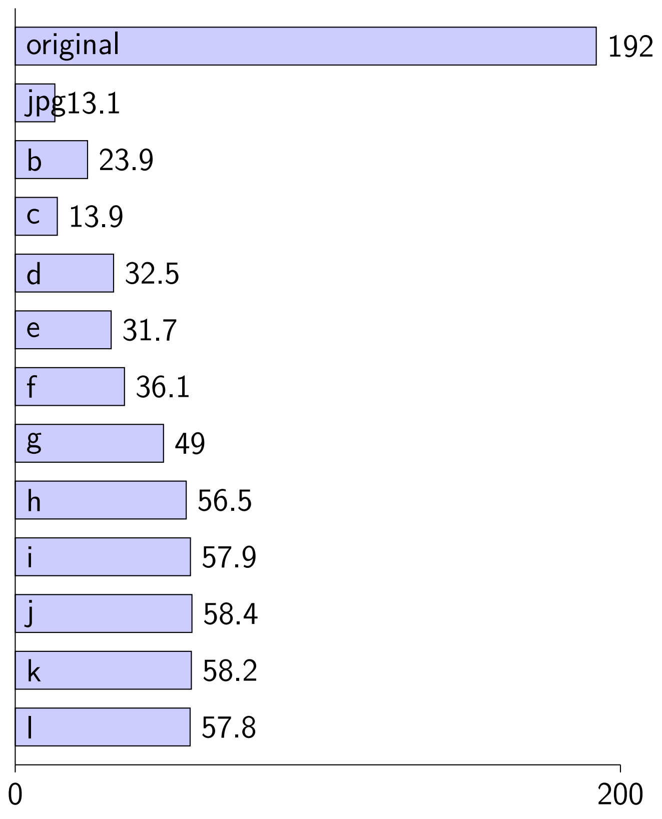


Figure 7: Diference in size in cartoon picture files in kB

The same situation we can see in a small file and cartoon(Fig.4 and Fig.6). There is a visible difference between second and third row.

The best improvement in quality is made by increasing the lattice size. Incresing number of iterations increases contrast of the image. Changing learing rate seems to have no impact.

I managed only once to create a file lower than jpg, but the quality is a lot worse.

5 Further remarks

The self-organizing map alghoritim should be implemented with usage of GPU and CUDA. I developed a code in C# and currently, there is no fully supported way to create a program for GPU with .NET. I tried to use library Hybrydizer, but after a couple days Igave up.

The proper way is to implement alghoritim in C++ and then with usage NSight and CUDA made it for GPU.

6 Conclusions

Taking into consideration size of compressed pictures, their quality and also time needed to compress one image, my alghorithm will not revolutionize image proccessing industry. But the project thought me principals behind using a self-organizing kohonen map.

I also learned that it's better to not use high-level programming laungage such as C# to implement artificial iintelligence.