Autoencoder and k-Sparse Autoencoder with Caffe Libraries

Guido Borghi

Università di Modena e Reggio Emilia
80036@studenti.unimore.it

Abstract

In this paper first we talk about neural network, or rather their links with human brain and how they work. Then we focus about Caffe Libraries, a deep learning framework, created by Yangqing Jia during his PhD at UC Berkeley: we analyze libraries' anatomy and then we see some neural network examples built with Caffe. Finally we use Caffe Libraries to create an Autoencoder and an Autoencoder k-Sparse, that is an autoencoder with linear activation function, where in hidden layers only the k highest activities are kept: this process encourages sparsity of the codes learned by autoencoder k-Sparse; this because sparsity improves classification tasks. After that we train and test theese two neural networks with two different dataset: The Olivetti Faces and Faces in the Wild. We report graphical results (feature maps and autoencoder codes for distinct levels of sparsity) and loss function results.

I. Introduction

In early years neural networks had a great improvement. There were two main problems that obstructed the development of this discipline: the learning algorithm in multilayers feedforward neural network and then the learning algorithm in (very) deep neural network. These two problems were solved with backpropagation algorithm and restricted Boltzmann machines (RBMs). Also the increasing power of electronic calculator helped the development of neural networks.
In this paper we focus about a particoular type of neural networks, autoencoders, networks used to convert high-dimensional data in low-dimensional codes. An autoencoder is a neural network that has a small central layer and that is trained to reconstruct high-dimensional input vector.
With Caffe Libraries we realize a specific type of autoencoders called autoencoder k-Sparse: it is a networks trained in a way that encourages sparsity in order to improve performance on classification tasks. An autoencoder k-Sparse is an autoencoder with linear activation function, where in hidden layers only the k highest activities are kept.
We test it with two dataset: *The Olivetti Faces* and *Faces in the Wild*.

I. Neural Networks

Neural networks were inspired by the examinations about human's central nervous system. In an artificial neural network, are connected togheter simple artificial nodes, known as *neurons*, *neurodes*, *processing elements* or *units*, to form a network which mimics a biological neural network.

Human brain is a very complex and affascinating machine and it is very difficult to try to imitate it: after all we don't have a good awareness about human brain yet. Generally, in animal brains there are complex organisations of neural cell: their work is, for example, to recognise input parameters that come from external world, the memorisation and the reaction that control body.

We can say a neuron, also known as a neurone or nerve cell, is composed by three principal parts:

- **soma**: the cell body;
- **axon**: a long projection of a nerve cell that conducts electrical impulses away from the neuron's cell body;
- **dendrites**: the branched projections that act to propagate the electrochemical stimulation received from other neural cells to the cell body.

Scientific studies reveal that neurones communicate with electric impulse, and each neuron does a weighted sum of these impulses. In plain terms we can see a neuron as a *black box* that has weighted data input from other neurons and produces weighted data output for others neurons.

II. Evolution of Neural Networks

The first computational model for neural networks was created by Warren McCulloch and Walter Pitts in 1943, based on mathematics and algorithms: they called this model threshold logic. This model proposed two distinct approaches to future research about Neural Networks: one approach focused on biological processes in the brain and the other focused on the application of neural networks to artificial intelligence.

The invention and develop of electronic calculators during the WWII allowed the use of computational machines to simulate neural networks. In 1958 Frank Rosenblatt created the Perceptron at the Cornell Aeronautical Laboratory: it is an algorithm for pattern recognition based on a two-layer learning computer network. It was a forerunner of the actual neural networks.
The main difference between McCulloch and Pitts' computational model and Rosenblatt's perceptron is that the second one has variable synaptic weights and so it can learn.
From a mathematical point of view we can describe Rosenblatt's Perceptron as shown below:

$$y_i = \Phi(A_i) = \Phi(\sum(w_{i,j}, x_j) - \vartheta_i))$$

Where Φ is called *Activation Function*, $\sum$ is the *Transfer Function*, $w_{i,j}$ are the weights, ϑ the threshold and x_j the inputs.
We can note that output correspond of a sum of weighted inputs and the threshold's single neuron value decides about its activation.

For about ten years there were a lot of reasearches about neural networks but in 1969 Marvin Minsky and Seymour A. Papert, in [5], demonstrated that Perceptron, but in general a simply two-layer neural networks, can only solve problems based on a linear separability solution. For example perceptron is not able to compute XOR function but only AND function. Besides in the 70's computers were not sophisticated enough to effectively handle the long run time required by large neural networks.

The problem about computers'power can be simply solved by techological progress in the 80's and 90's and later, in accordance with Moore's Law (1965): the number of transistors in a dense integrated circuit doubles approximately every two years.
The problem about linear separability solution can be solved adopting a multi-layer netowrk, also called MLP (Multi-Layers Perceptron), but it would introduce another relevant problem about the learning: until 1986 doesn't exist an algorithm that was able to do training for MLP. In 1965 David E. Rumelhart, G. Hinton e R. J. Williams proposed the *Error Backpropagation*, also know only as *Backpropagation*: it is used in conjunction with an optimization method, such as gradient descent, and it calculates the gradient of a *loss function* with respects of all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.
The steps of Backpropagation algorithm are:

- **Forward Propagation**: a training pattern's input is propagated through the neural network in order to generate the propagation's output activations.
- **Backward Propagation**: the propaga-

tion's output activations is propagated through the neural network using the training pattern target in order to generate the deltas of all output and hidden neurons.

- **Weight Updates**: weights are updated in this way:

$$\Delta w_{i,j}(t) = -\eta \frac{\delta E}{\delta w_{i,j}(t)} + \alpha \Delta w_{i,j}(t-1)$$

where η is the *learning rate*, otherwise learning's rate of the network, E the *loss function*, α the *momentum*, a constant that is used to avoid oscillation in weight update, $\Delta w_{i,j}(t)$ the actual matrix of weights and $\Delta w_{i,j}(t-1)$ the previous matrix of weights.

In recent years the relation between neural networks and brain biological architecture is debated, as it is not clear to what degree artificial neural networks mirror brain function.
In the 1990s, neural networks were overtaken in popularity in machine learning by support vector machines and other, much simpler methods such as linear classifiers. Renewed interest in neural nets was sparked in the 2000s by the advent of deep learning.

III. Topologies of Neural Networks

There are three main topologies for neural network. The single-layer feedforward Neural Network (Fig. 1) is the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward, from the input nodes to the output nodes. There are no hidden nodes and there are no cycles or loops in the network.

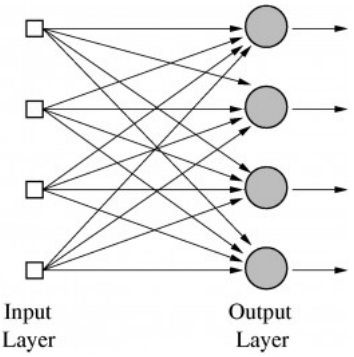

Figure 1: *Single-layer Feedforward*

The multi-layer feedforward Neural Network (Fig. 2) are similar to previous, but in this case we have one or more hidden layers that connect input to output layers. Even here there are no cycles or loops in the network.

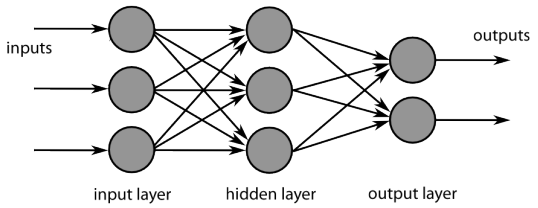

Figure 2: *Multi-layer Feedforward*

Finally, a recurrent neural network (Fig. 3) is a class of artificial Neural Network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. They can use their internal memory to process arbitrary sequences of inputs. In this paper we don't spend a lot of time about them.

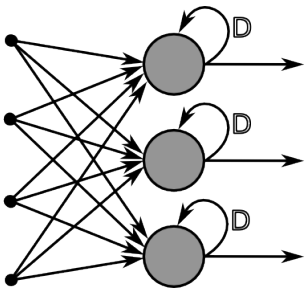

Figure 3: *Recurrent Net*

II. Caffe Libraries

Caffe is a deep learning framework, but in general it can be used to work with neural network. It was created by *Yangqing Jia* during his PhD at UC Berkeley, and is in active development by the Berkeley Vision and Learning Center (BVLC) and by community contributors.
Caffe libraries were created in order to achieve:

- **Clean architecture**: libraries use simple config files and it is very simple switch CPU to GPU computation to use a cluster of machines.
- **Readable and modifiable implementation**: a lot of developers contribuited to Caffe's evolution (about 600 fork in the first year).
- **Speed**: Caffe developers say that Caffe is the fastest CNN implementation available (2014).
- **Community**: there is a support comunity on Github (https://github.com/BVLC/caffe).

I. Anatomy of Libraries

Deep networks are network that have a lot of hidden layers and they typically work on chunks of data. Caffe defines a net layer-by-layer in its own model schema, bottom-to-top from input data to loss. The information go through layers as *blobs*: the blob is the standard array and unified memory interface for the framework. The details of blob describe how information is stored and communicated in and across layers and nets.

I.1 Net

The net is a set of layers connected in a graph without cycles, called Direct Acyclic Graph (DAG). Caffe controls the forward and backward passes to ensure correctness.
A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

I.2 Blobs

As mentioned, a blob is a wrapper over the data that are passed along by Caffe; blobs also provide synchronization capability between the CPU and the GPU.
From a computation point of view blob stores and communicates data in 4-dimensional arrays in the order of (Num, Channels, Height and Width), from major to minor. In this way blobs provide a unified memory interface, holding data, model parameters, and derivatives for optimization.

I.3 Solver

The solver coordinates the network's forward inference and backward gradients to form parameter updatesin order to improve the loss. There are three Caffe solvers: *Stochastic Gradient Descent* (SGD), *Adaptive Gradient* (ADAGRAD), and *Nesterov's Accelerated Gradie*nt (NAG). In our work we use only SGD.
The solver:

1. Creates the training network for learning and test network(s) for evaluation;
2. Iteratively it calls forward and backward computation and it updates parameters;
3. Snapshots the model and solver state throughout the optimization.

where each iteration

1. Calls network forward to compute the output and loss
2. Calls network backward to compute the gradients
3. Incorporates the gradients into parameter updates according to the solver method updates the solver state according to learning rate, history, and method to take the weights all the way from initialization to learned model.

I.4 Loss

In Caffe, as in most of machine learning, learning is driven by a loss function (also known as an *error*, *cost*, or *objective function*). A loss function specifies the goal of learning by mapping current network weights to a scalar value specifying the "badness" of these parameter settings. The goal of learning is to find a setting of the weights that minimizes the loss function.

I.5 Layers

The layer is the fundamental unit of computation in Caffe Libraries. A layer takes input through bottom connections and makes output through top connections.
Here a small list of possible layers:

- **Convolution**: it performs the convolution between an image input and a filter (kernel). The output are *k* convolved features, where k is the number of kernels.
- **Pooling**: an aggregation operation to reduce the number of convolved features, on order to avoid overfitting and to increase the robustness for variance.
- **Local Response Normalization** (LRN): it performs a kind of "lateral inhibition" by normalizing over local input regions.
- **Softmax**: a loss layer
- **Sum-of-Squares / Euclidean**: another loss layer

III. Related Works

I. Autoencoders

In [1] is presented the aim of autoencoders: they are used to convert high-dimensional data in low-dimensional codes. Infact an autoencoder is a neural network that has a small central layer and that is trained to reconstruct high-dimensional input vector.
The gradient descent technique is used for network's weight update; it is suggested to do a pretraining phase that consists in learning a stack of restricted Boltzmann machines (RBMs) to initialize weights in network and then do the real train phase: that why in [1] is used a deep autoencoder and it is difficult to adopt backpropagation without weight inizialization in deep neural network: pretraining greatly reduces their total training time. In generale in deep networks initial weights must be close enough to a good solution.
Besides codes learnt by autoencodes are plotted (Figure 11): we can see that they are clustered in function of the input class.

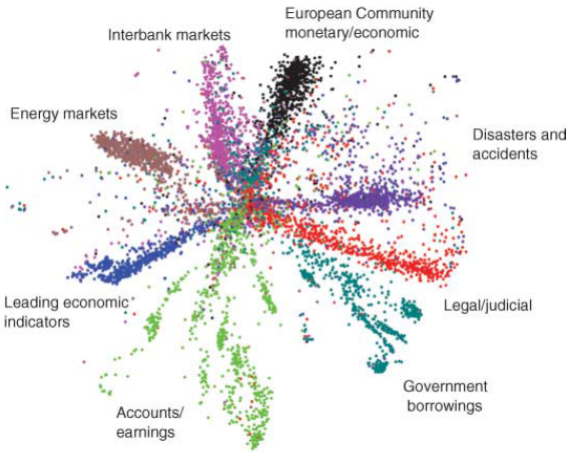

Figure 4: *An autoencoder learns clustered codes*

IV. Autoencoders k-Sparse

In [4] is presented the idea of autoencoders k-sparse: autoencoders are trained in a way that encourages sparsity in order to improve performance on classification tasks. An autoencoder k-Sparse is an autoencoder with linear activation function, where in hidden layers only the k highest activities are kept. This type of autoencoder is better on classification than denoising autoencoders, networks trainedwith dropout and RBMs.
Makhzani et al. demonstrate that k-sparse autoencoders are suitable for pre-training and achieve results comparable to state-of-the-art on MNIST and NORB datasets.

V. Implementation with Caffe Libraries

In this section, we evaluate the behaviour and the performance of autoencoder and k-sparse autoencoders created with Caffe Libraries.

I. Datasets

We use two datasets for the implementation.
The first we used is *The Olivetti Faces* dataset: it contains about 400 images (92x112) taken between April 1992 and April 1994 at AT&T Laboratories Cambridge.
As described on the original website: "There are ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions and facial details. All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement)".
Olivetti's images are quantized to 256 grey levels and stored as unsigned 8-bit integers.

Figure 5: *An example of The Olivetti Faces dataset*

The second dataset is *Faces in the Wild* dataset. It consists of 30,281 faces collected from News Photographs. These faces have been automatically labeled using a particoular algorithm. Unlike Olivetti dataset here we have color images (RGB) with a resolution of 85x85.

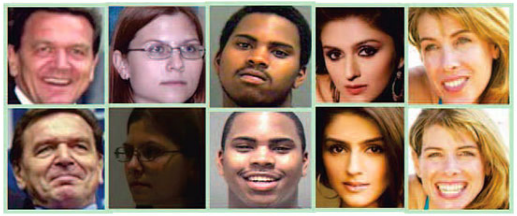

Figure 6: *An example of Faces in the Wild dataset*

II. Autoencoder with Olivetti

With Caffe libraries we created a 10304-1000-10304 (input-hidden-output) autoencoder. The value 10304 comes from image's resolution. The high number of neurons in the first layer makes computation heavy. We used a machine with an i7-950 @3.07 CPU and train and test phases occupy about 10 hours for 50000 iterations.
In fig. 7 we plot feature maps where we can see that each neuron learns faces in a very general way, infact we can't see any part or detail of faces.

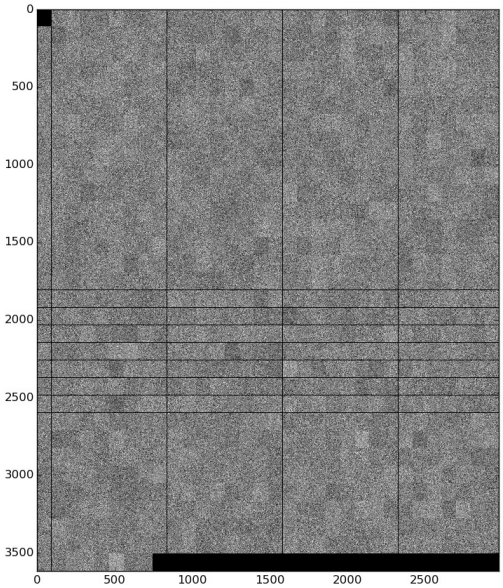

Figure 7: *Olivetti codes without sparsity*

III. Autoencoder k-Sparse with Olivetti

We created a 10304-1000-10304 autoencoder k-sparse. First, we pretrained network using a dropout autoencoder with a dropout rate of 50%, in order to have initial weights close enough to the good solution. The pretrain phase lasts 20000 iterations.
Then we trained the net with different levels of sparsity (k) and we took the best results. In particoular in fig. 8 we used k=70 and we did 50000 iterations.
With k=70 we encouraged autoencoder to learn sparse codes: in other words some neurons learn even for other neurons that have not a k hightest activation.
From a graphical point of view faces, or parts of theese, appears in some neuron: this result depends on the level of sparsity we have chosen.

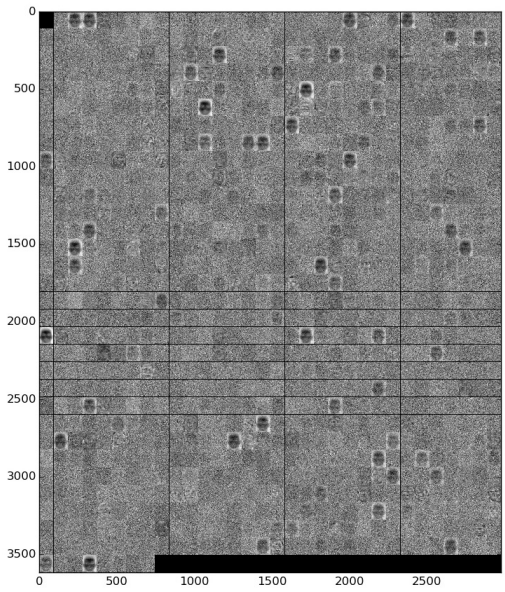

Figure 8: *Olivetti codes with k=70*

However faces in fig. 8 are too similar: in this way autoencoder learns only frontal faces with no particoular details (for example glasses, smiles, long hair...). The learning process of autoencoder k-sparse doesn't generalize enough and that it is a negative condition for classification tasks.
Probably this appens because dataset has a small number of images and images are too similar: we don't have, for example, different points of view for faces. This is confirmed by the plot of loss during training phase (fig. 9): we can note that after about 5000 iterations the loss value doesn't change but rather the graphic has some peaks. Autoencoder stops to learn soon: after all the dataset is very small and there are not a lot of faces to learn.

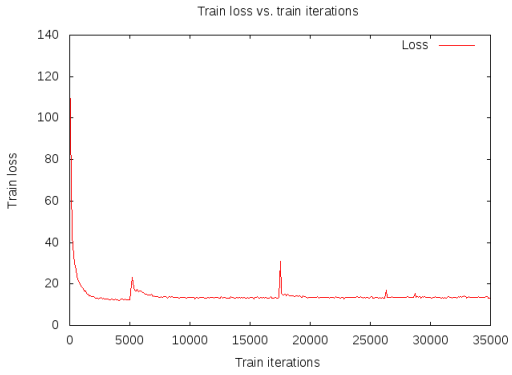

Figure 9: *Train phase: loss with k=70*

Also in fig. 10 we have the same trend of the loss function: after about 7000 iterations the value of the loss doesn't change and it has a high value.
For this reason we focused on another dataset: *Faces in the Wild*.

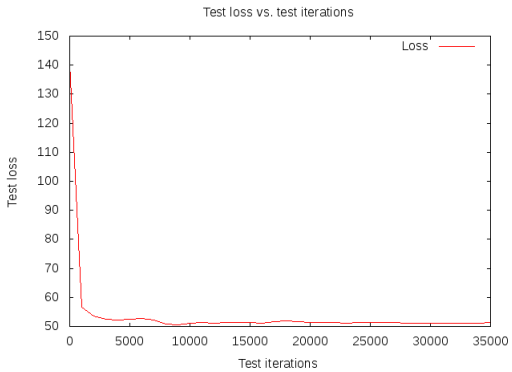

Figure 10: *Test phase: loss with k=70*

IV. Autoencoder with Faces in the Wild dataset

In this case we created a 3025-1000-3025 autoencoder. We resized images from 85x85 to 55x55, and so the number of the input neurons is 3025.
For each image we create mirror image and cropped image to increase the number of total images and to augment variability to avoid *overfitting* problem.
We used a machine with an *i7-950 @3.07GHz* CPU and train and test phases occupy about 8 hours for 50000 iterations instead of the 10 hours for Olivetti dataset, because of we have a smaller number of input neurons than Olivetti case.
Also in this case without any level of sparsity neurons learn very general codes and no face appears in feature maps.

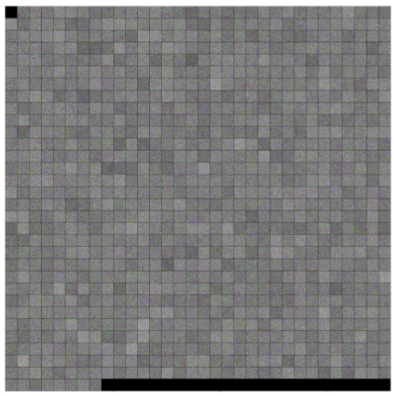

Figure 11: *Faces in the Wild codes without sparsity*

V. Autoencoder k-Sparse with Faces in the Wild dataset

We created a 3025-1000-3025 autoencoder k-sparse. We train and test the net with three different levels of sparsity: k=25, k=60 and k=200.

First, we pretrained network using a dropout autoencoder with a dropout rate of 50%, in order to have initial weights close enough to the good solution. The pretrain phase lasts 20000 iterations. In fig. 12 we can see the decreasing of the loss function's value.

Figure 12: *Loss in pretraining phase*

In fig. 13 we plot a selection of six faces obtained with k=25. We can see some details of faces. With k=25 we forced too much sparsity and results in features are too global and do not factor the input into parts.

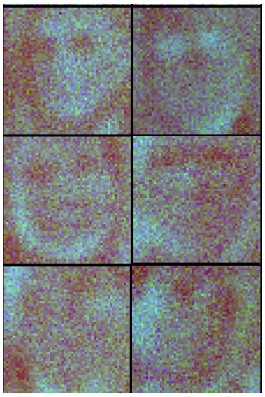

Figure 13: *Faces in the Wild: k=25*

When we increased sparsity level (k=60) we obtained the best result. Neurons learn faces with very significative details as is shown in fig. 14. The output is reconstructed using a higher number of hidden units and thus the features tend to be less global and its can be utilized in classification tasks with relevant results.

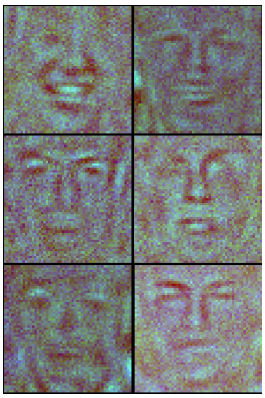

Figure 14: *Faces in the Wild: k=60*

With k=200 autoencoder tends to learn very local features: these features are too primitive to be used for classification using a shallow architecture since a naive linear classifier does not have enough capacity to combine these features and achieve a good classification rate. However, these features could be used for pre-training deep neural nets.

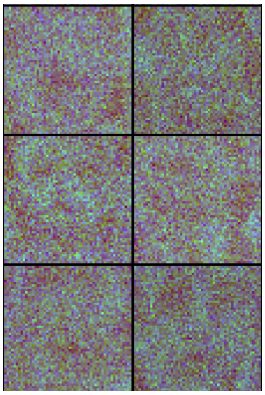

Figure 15: *Faces in the Wild: k=200*

For a complete documentation we report also the features of the output neurons; we can see that autoencoder can reconstruct input images with a good precision.

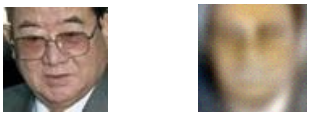

Figure 16: *Faces reconstructed by autoencoder k-Sparse*

Unlike Olivetti case, the loss with *Faces in the Wild* dataset constantly decreases. The loss function is the object function that we try to minimize in autoencoder.

Figure 17: *Train phase: loss with k=60*

Also in test phase the value of the loss function costantly decreases.

Figure 18: *Test phase: loss with k=60*

VI. Final considerations

There are three main considerations.
First, considerations about sparsity levels. The learning of an autoencoder k-sparse depends on the value of k.
In general we can say that:

- With a **large value of k** autoencoder k-sparse learns very local features: theese features are too primitive to be used in classification tasks, but they could be used to pretraining deep neural networks.
- With **middle values of k** we obtained best features, no too global and no too local.
- With too much sparsity (**small value of k**) we obtained too global features that do not factor the input into parts.

The second consideration is about Gabor filters. In some images with k=60 there are some parts that remember this filter. Gabor filter is a linear filter used for edge detection.
Frequency and orientation representations of Gabor filters are similar to those of the human visual system, and they have been found to be particularly appropriate for texture representation and discrimination.
Simple cells in the visual cortex of mammalian brains can be modeled by Gabor functions. Thus, image analysis with Gabor filters is thought to be similar to perception in the human visual system.
Our neural network, that is built to reproduce human brain, produces something like Gabor filter and this can supports the idea that in brain we have processes like Gabor filters.

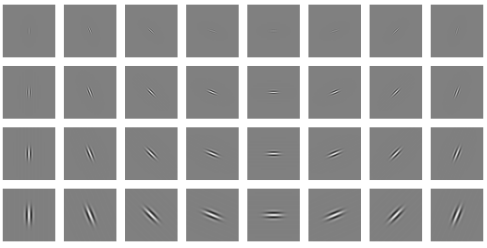

Figure 19: *Gabor filters*

The third consideration is about Viola-Jones algorithm used in face detection. As reported in [6] the algorithm is based on simple black and white patterns that are used to detect faces in images. The base idea is to identificate faces with its significative features. In our results we see some faces that have dark areas in constrast with bright areas: we can say that also autoencoders k-sparse focus its learning on something like Viola-Jones patterns.

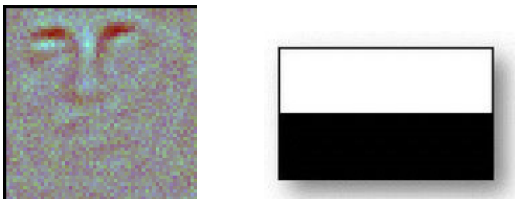

Figure 20: *Viola-Jones patterns*

References

[1] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313 no. 5786, 2006.

[2] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[3] Christian K. Machens. Building the human brain. *Science*, 338 no. 6111, 2012.

[4] Alireza Makhzani and Brendan Frey. k-sparse autoencoders. *CoRR*, abs/1312.5663, 2013.

[5] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[6] Paul Viola and Michael Jones. Robust real-time object detection. *International Journal of Computer Vision*, 2001.

[7] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *Arxiv*, abs/1311.2901, 2013.