5. Deep learning with Autoencoders

INTRODUCTION

Autoencoder (AE) network is another class of ANN that has been extensively studied in the domain of deep learning to address some issues intrinsic to convolutional NN especially related to the training. One of the advantage of AE units is that the training is unsupervised that is there is no need of specifying a label.

AUTOENCODER NETWORKS

One of the most challenging techniques to overcome troubles with supervised learning based on backpropagation in multi-layer FFNN makes use of autoencoder networks. An autoencoder network is a FFNN, endowed with 3 layers, that recreates the input data vector **x** in the vector **z**, which is the network output, by means of an intermediate processing step performed by the internal hidden layer (Fig. 1). The hidden layer is composed by a number *H* of neurons lower than the number *N* of neurons in the input/output layers, allowing a compressed and distributed representation of the input dataset **x** into the vector **y**. The training for such type of network corresponds to learn an approximation to the identity function by means of one processing step, performed by the hidden neuron weight matrix $\mathbf{W}^{(h)} = \left\{ w_{ij}^{(h)} \right\}$, and one processing step, performed by the output neuron weight matrix $\mathbf{W}^{(o)} = \left\{ w_{ij}^{(o)} \right\}$. More specifically, the hidden layer is said to perform an encoding of **x** in the vector $\mathbf{y} = \left\{ y_i \right\}$, where $y_i = \varphi\left(\sum_j w_{ij}^{(h)} x_j + b_i \right) = \varphi_i^{(h)}\left(\mathbf{x} \right)$ is the activation of the hidden neuron *i*. The output layer is said to perform a decoding of hidden layer, where $z_i = \varphi\left(\sum_j^H w_{ij}^{(o)} y_j + c_i \right) = \varphi_i^{(o)}\left(\mathbf{y} \right)$ is the activation of the output neuron *i*. We are assuming again here that φ is the sigmoid function.

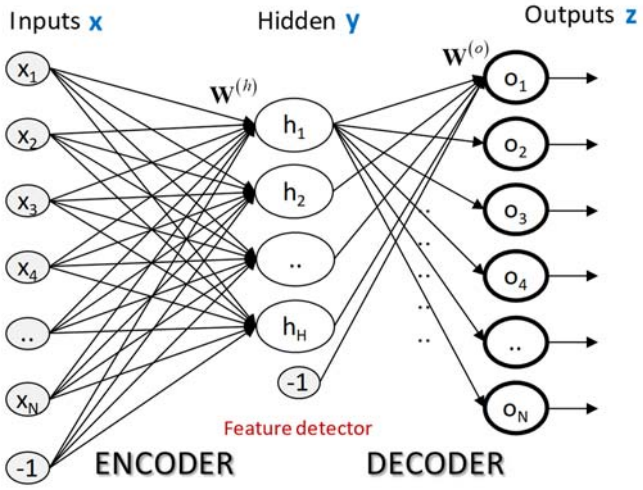


Figure 1. Simplest Autoencoder network.

One simple example of using autoencoder networks can be appreciated in Fig. 2. We are entailing a gray-scale image dataset representing a set of four geometric shapes (circle, triangle, rectangle and star). The shapes can be in different locations and orientations, affected by blurring or even incomplete. We are aiming at discovering the intrinsic features that discriminate among the four shapes trying to remove such unsettling effects. This is a typical case of recognizing visual items into images. Applying the autoencoder-based processing means to learn an encoding from the input dimension (scalar image size) to a reduced dimension expressed by the number of the hidden neurons. The decoding stage sets back the encoded signals (feature space) into the image space. Considering a 30×30 pixels image, the autoencoder input consists of the gray values of the 900 pixels. A hidden layer of 100 neurons (this is an arbitrary size) is attempting to extract the most relevant 100 descriptive features. Taking one image in input, the network is generating an encoded version of such image represented by 100 elements y_i. The practical question is yet what visual features are the encoded by the signal set $\left\{ y_i \right\}$.

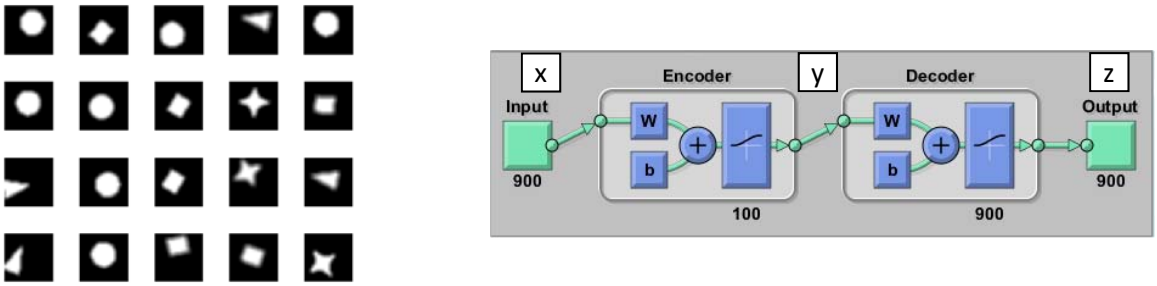


Figure 2. Shape analysis problem addressed by an autoencoder network.

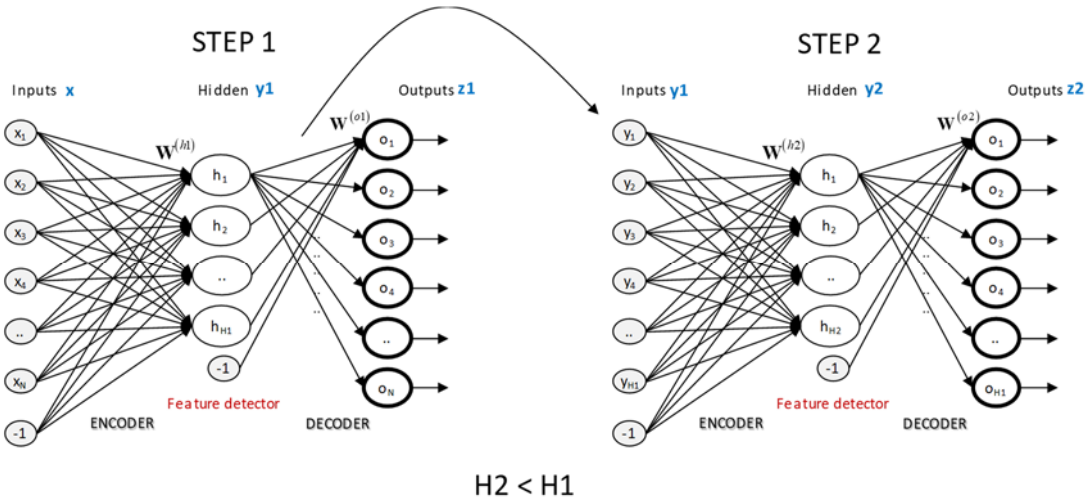


Figure 3. A cascade of two autoencoders.

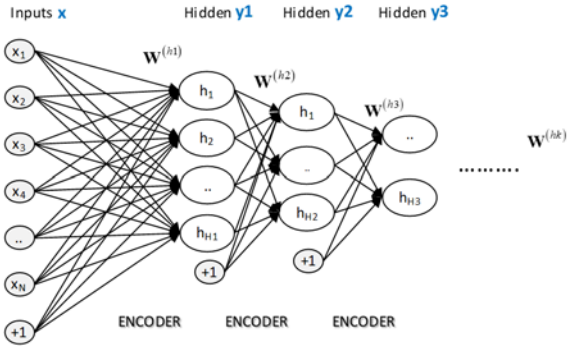


Figure 4. Resulting multi-layer FFNN.

Aiming at modeling the differences among the four shapes, we can assume that a single autoencoder stage is not sufficient and further encoding stages are necessary leading to a multi-layer FFNN. Using autoencoders to address multi-layer FFNN learning consists of splitting the FFNN, layer by layer, and build a corresponding set of autoencoders, being the output of the hidden layer of the encoder *k* the input of the encoder *k*+1 (Fig. 3). After training each single autoencoder network, the resulting multi-layer FFNN is attained by cascading the hidden layers of all the autoencoders (Fig. 4).

TRAINING AN AUTOENCODER

We have just anticipated that training one autoencoder network means learning the weights of the hidden and output layers so that **x**≡**z**, provided that a training dataset $\left\{\mathbf{x}^{(1)},\mathbf{x}^{(2)},...,\mathbf{x}^{(R)}\right\}$ is available. While, from the point of view of the supervised learning paradigm, we have a reference value **x** for the network output **z**, the

learning technique for autoencoders is traditionally named unsupervised to remark the difference between this case and typical function approximation and classification applications, where the reference output consists of codomain or label values. In principle, it can be argued that the backpropagation could work for the autoencoder but learning the identity function is ill-posed problem when the number of hidden units is lower than the number of inputs.
In order to avoid learning instability and making the weight estimation feasible when using backpropagation, a modification of the error function *E* is mandatory. The first step consists of adding to *E* a regularization term that should prevent the weights to grow indefinitely. Thus, in agreement to this consideration, the traditional error function of the Delta rule is rearranged as:

$$E=\sum_{k=1}^{R}\left(\sum_{i=1}^{N}\frac{1}{2}\left(x_i^{(k)}-z_i^{(k)}\right)^2\right)+\frac{\alpha}{2}\sum_{l=2}^{L}\sum_{i}^{M_l}\sum_{j}^{M_{l-1}}\left(w_{ij}^{(l)}\right)^2$$

where α (0<α<<1) and M_l are a regularization weight and the number of neurons in the layer *l*, respectively. Essentially, α tends to decrease the magnitude of the weights, and helps prevent overfitting. For the autoencoder with 3 layers, in the 2nd and 3rd layers *l=h* and *l=o* hold, respectively. Solving the minimization problem leads to attain the following updating rule for the weights:

$$\Delta w_{ij}^{(l)}=\eta\left(\sum_{k=1}^{R}\delta_i^{(l),(k)}e_j^{(l-1),(k)}+\alpha w_{ij}^{(l)}\right)$$

where $\delta_i^{(l),(k)}=\left(x_i^{(k)}-z_i^{(k)}\right)\varphi'\left(P_i^{(k)}\right)$ *if* $l=L$ and $\delta_i^{(l),(k)}=\varphi'\left(P_i^{(k)}\right)\sum_{r=1}^{M_{l+1}}\left(\delta_r^{(l+1),(k)}w_{ri}^{(l+1)}\right)$ *if* $l<L$.

with P_i the action potential of neuron *i*. However, nothing ensures about either the priority or the specificity (with respect to features) of any particular neuron in the hidden layer with respect to the others in the same layer. One of the main approaches to further address the constraint on the weights of the hidden neurons is represented by the so called “sparsity paradigm”. This paradigm rests on the concept that one neuron in the hidden layer should be active only in correspondence of homogenous input data, that is for data embedding similar features. In the learning stage, this is verified by opportunely constraining the average activation of the hidden neurons over the training dataset. We are assuming that when the neuron is active (high firing rate) its output approaches 1 whereas the neuron is inactive (low firing rate) its output approaches 0 (-1). Provided that the activation of the hidden neuron *i*-th in correspondence of the input pattern $\mathbf{x}^{(k)}$ is:

$$y_i^{(k)}=\varphi\left(\sum_{j}^{N}w_{ij}{}^{(h)}x_j^{(k)}+b_i\right)=\varphi_i^{(h)}\left(\mathbf{x}^{(k)}\right)$$

the average activation $\hat{\rho}_i$, of hidden unit *i* over the *R* input data in the training dataset, can be computed as:

$$\hat{\rho}_i=\frac{1}{R}\sum_{k}^{R}y_i^{(k)}=\frac{1}{R}\sum_{k}^{R}\varphi_i^{(h)}\left(\mathbf{x}^{(k)}\right)=\frac{1}{R}\sum_{k}^{R}\left(\sum_{j}^{N}w_{ij}^{(h)}x_j^{(k)}\right)$$

The sparsity assumption should constrain the hidden neurons to be inactive most of the time, which is ensured whether

$$\hat{\rho}_i=\rho$$

where ρ, defined a priori, is the sparsity parameter. For instance setting ρ=0.01 shall constrain every single hidden neuron to be active only to 1% of the input patterns. This means that it is likely representative of particular features contained in that specific subgroup of the training dataset.
In order to include the sparsity constrain into the learning process based on backpropagation, we need to modify further the error function by including explicitly the sparsity constraint, which shall penalize $\hat{\rho}_i$ deviating significantly from ρ,as:

$$E = \sum_{k=1}^{R}\sum_{i=1}^{N}\frac{1}{2}\left(x_i^{(k)} - z_i^{(k)}\right)^2 + \frac{\alpha}{2}\sum_{l=2}^{L=3}\sum_{i}^{N_l}\sum_{j}^{N_{l-1}}\left(w_{ij}^{(l)}\right)^2 + \beta G\left(\{\hat{\rho}_i\},\rho\right)$$

where β (β >0) and G are the regularization weight and the penalty term, respectively. As $\hat{\rho}_i$ is a function of $w_{ij}^{(h)}$ (*h* standing for hidden layer), the minimization of the error function means to compute the derivative $\partial G / \partial w_{ij}^{(h)}$. Amongst several choices of penalty term that will provide reasonable results, we indicate the following:

$$G\left(\{\hat{\rho}_i\},\rho\right) = \sum_{i}^{H}\left(\rho\log\frac{\rho}{\hat{\rho}_i} + (1-\rho)\log\frac{(1-\rho)}{(1-\hat{\rho}_i)}\right) = \sum_{i}^{H} KL\left(\rho\|\hat{\rho}_i\right)$$

where *H* is the number of neurons in the hidden layer, and *i* is the index summing over the hidden units of the network.
This function represents the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_i$. It is a classical function for measuring how different two different distributions are. This penalty function has the property that $KL\left(\rho\|\hat{\rho}_i\right)=0$ whether $\hat{\rho}_i=\rho$ and otherwise it increases monotonically as $\hat{\rho}_i$ deviates from ρ (Fig. 5). Synthesizing, we can therefore write:

$$\hat{\rho}_i -> \rho \;\Rightarrow\; KL -> 0$$
$$\hat{\rho}_i -> 1 \;\Rightarrow\; KL -> \infty$$
$$\hat{\rho}_i -> 0 \;\Rightarrow\; KL -> \infty$$

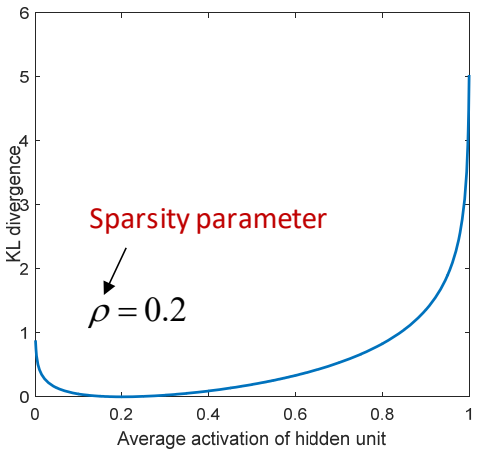


Figure 5. Kullback-Leibler (KL) divergence function representing the divergence between $\hat{\rho}_i$ and ρ, when $\rho = 0.2$.

Let us get back to the autoencoder and remember that the 2nd and the 3rd layers are the hidden layer (*h*) and the output (o) layers, respectively (Fig. 6). Being *k* one input data into the training dataset, the minimization of E wrt $w_{ij}^{(l)}$ leads to write (disregarding here the weight norm regularization contribute):

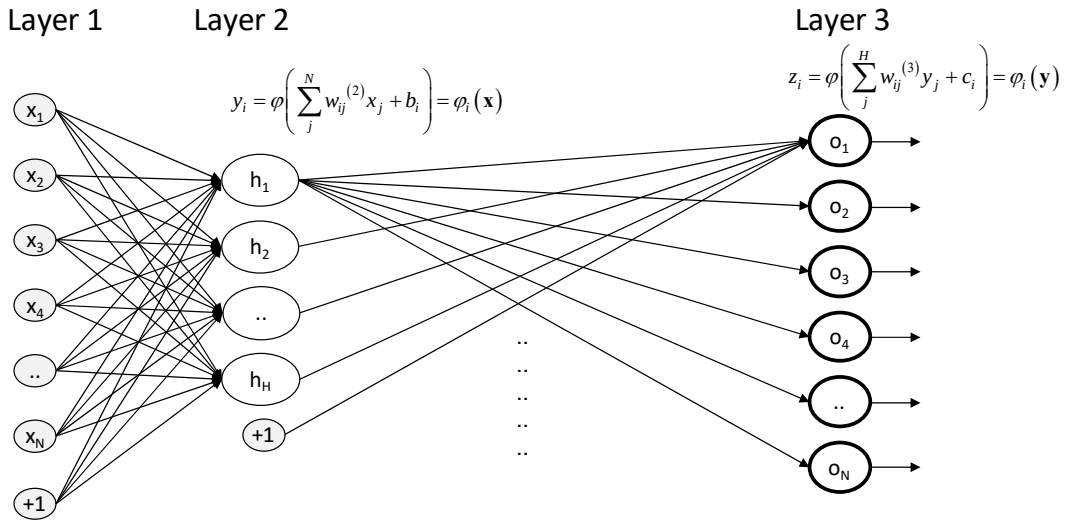


Figure 6. Autoencoder.

$$\delta_i^{(3),(k)} = \left(x_i^{(k)} - z_i^{(k)}\right)\varphi'\left(P_i^{(k)}\right)$$

$$\Delta w_{ij}^{(3)} = \eta \sum_{k=1}^{R} \delta_i^{(3),(k)} y_j^{(k)}$$

for the 3rd layer (output) and

$$\delta_i^{(2),(k)} = \varphi'\left(P_i^{(k)}\right)\sum_{r=1}^{N}\left(\delta_r^{(3),(k)} w_{ri}^{(3)}\right) + \beta g\left(\hat{\rho}_i, \rho\right)$$

$$\Delta w_{ij}^{(2)} = \eta \sum_{k=1}^{R} \delta_i^{(2),(k)} x_j^{(k)}$$

for the 2nd layer (hidden), being $g\left(\hat{\rho}_i, \rho\right)$ the derivative of divergence function $KL\left(\rho \| \hat{\rho}_i\right)$.

Recalling that $\hat{\rho}_i = \frac{1}{R}\sum_k^R\left(\sum_j^N w_{ij}^{(2)} x_j^{(k)}\right)$ the derivative of $G\left(\hat{\rho}_i, \rho\right)$ is computed as:

$$\frac{\partial G\left(\hat{\rho}_i, \rho\right)}{\partial w_{ij}} = g\left(\hat{\rho}_i, \rho\right) = \frac{\partial\left(\rho \log\frac{\rho}{\hat{\rho}_i} + (1-\rho)\log\frac{(1-\rho)}{(1-\hat{\rho}_i)}\right)}{\partial w_{ij}} = \frac{\partial\left(\rho \log\frac{\rho}{\hat{\rho}_i} + (1-\rho)\log\frac{(1-\rho)}{(1-\hat{\rho}_i)}\right)}{\partial \hat{\rho}_i}\frac{\partial \hat{\rho}_i}{\partial w_{ij}}$$

which can be developed into $\left(\frac{(1-\rho)}{(1-\hat{\rho}_i)} - \frac{\rho}{\hat{\rho}_i}\right)\frac{\partial \hat{\rho}_i}{\partial w_{ij}} = \left(\frac{(1-\rho)}{(1-\hat{\rho}_i)} - \frac{\rho}{\hat{\rho}_i}\right)\frac{1}{R}\sum_k^R \varphi'\left(P_i\right)$.

The contribute to the $\delta_i^{(2),(k)}$ factor, over a single input data *k*, is therefore

$\delta_i^{(2),(k)} = \varphi'\left(P_i^{(k)}\right)\left(\sum_{r=1}^{N}\left(\delta_r^{(3),(k)} w_{ri}^{(3)}\right) + \beta\left(\frac{(1-\rho)}{(1-\hat{\rho}_i)} - \frac{\rho}{\hat{\rho}_i}\right)\right)$ and the corresponding weight update (batch update) can be written as:

$$\Delta w_{ij}^{(2)} = \eta \sum_{k=1}^{R} \delta_i^{(2),(k)} x_j^{(k)}$$

The overall learning procedures is synthesized in the following chart (Fig. 7).

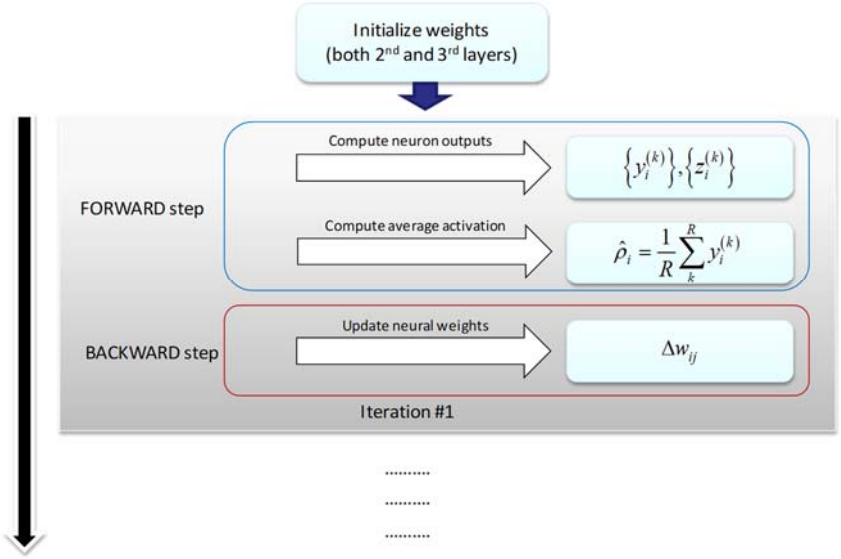


Figure 7. Constrained backpropagation for an autoencoder.

After initializing the weights of both layers, the first forward pass computes all the neuron output signals for all the input data in the training dataset, which is the same procedure we described for the traditional backpropagation. In a second forward pass, all the $\hat{\rho}_i$ are computed. In the backward pass, the weight updates are computed according to the earlier equations. These three passes are replicated in the next iterations up to convergence.

WHAT IS ACTUALLY THE AUTOENCODER LEARNING?

After training a (sparse) autoencoder, we can attempt to visualize the function learned by the algorithm, to try to understand what it has learned. Consider the case of training an autoencoder on 30×30pixel images (geometric shapes), so that *N*=900. Each hidden unit *i* computes a function of the input:

$$y_i = \varphi\left(\sum_{j}^{900} w_{ij}^{(2)} x_j + b_i\right)$$

We will visualize the function computed by hidden unit *i*, which depends on the parameters $w_{ij}^{(2)}$ (ignoring the bias term for now) using a 2D image. In particular, we think of y_i as some non-linear feature of the input **x**. The questions is therefore: what input image **x** would cause y_i to be maximally activated? Less formally, what is the feature that hidden unit *i* is looking for? For this question to have a non-trivial answer, we must impose some constraints on **x**. If we suppose that the input is norm constrained by

$$\|\mathbf{x}\|^2 = \sum_{j}^{900} x_j^2 \le 1$$

then one can show that the input which maximally activates hidden unit *i* is given by setting pixel x_j (for all 900 pixels, $j = 1,\ldots,900$) to

$$x_j = \frac{w_{ij}^{(2)}}{\sqrt{\sum_{j}^{900}\left(w_{ij}^{(2)}\right)^2}}$$

By displaying the image formed by these pixel intensity values, we can begin to understand what feature hidden unit *i* is looking for. If we have an autoencoder with 100 hidden units, then the visualization consists of 100 such images, one for each hidden unit. By examining these 100 images, we can try to understand what the ensemble of hidden units is learning. When we do this for a sparse autoencoder (trained with 100 hidden units on 30×30pixel inputs we get the result depicted in the next figure (Fig. 8).

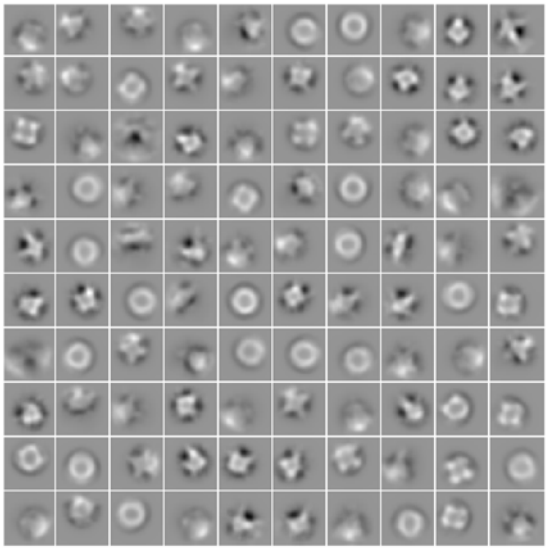
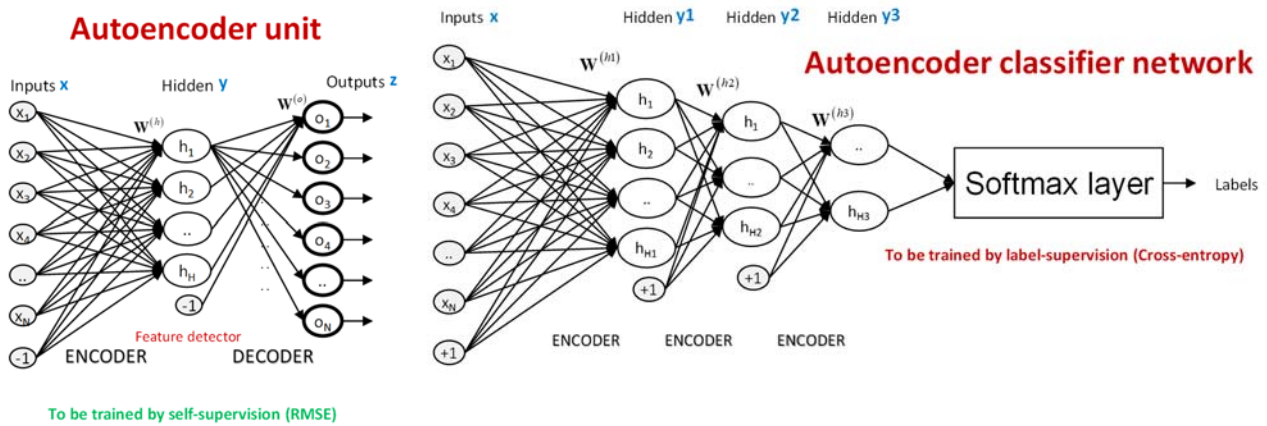


Figure 8. Normalized weights of the hidden neurons displayed as images.

Each square in the figure above shows the (norm bounded) input image that maximally actives one of 100 hidden units. We see that the different hidden units have learned to detect shape features at different positions and orientations in the image. Interestingly, the circle is the shape that activates more neurons, this probably due to rotation invariance. Let notice however that both the rectangle and the cross shapes have specific encoding neurons whereas it is evident graphically whether any specific neuron is encoding for the triangle.



In detail, each autoencoder unit, into an autoencoder network for classification, can be trained separately using self-supervision and RMSE as the loss function. As soon as all the encoder layers are trained, the classification layer, for instance a Softmax layer, will be trained using traditional supervision with reference label for each class and the Cross-entropy as the loss function.

SYNTHESIS

In this note, we have been discussing about the use of convolutional and autoencoder neural networks to compute features from unlabeled input patterns. CNNs were shown to exploit explicitly the physiological concept of local receptive fields and shared weights. This allows representing the processing as a convolution of a filter with the input pattern (image) to extract features according to the number and the characteristics of used feature maps. Similarly, the working principle of the autoencoder consists of reconstructing the input patterns by encoding the input dimensionality into a lower dimensional space, extracting the main information content distributed in the data. The difference between CNN and autoencoder rests on the data processing performed by the hidden layers. The locality of the processing is not predetermined as in CNN but arises automatically during learning from the distribution of information content into the training dataset. An unsupervised learning mechanism has been presented, which exploits the sparse activation of hidden neurons. In neuroscience, the term neural coding is adopted to denote the patterns of electrical activity of neurons induced by a stimulus. Sparse coding in its turn is one kind of pattern. A code is said to be sparse when a stimulus (like an image) yields the activation of just a relatively small number of neurons, that combined represent it in a sparse way. In machine learning, the same optimization constraint used to create a sparse code model can be used to implement sparse autoencoders, which are regular autoencoders trained with a sparsity constraint. Sparse autoencoders candidate to be machine learning tools to feature detection and smart data compression.

Summary of notation

$\mathbf{x}$	Input vector of the autoencoder network (training example), $\mathbf{x} \in \Re^N$
x_i	Feature of a training example **x** (*i*-th input component)
$\mathbf{z}$	Output vector of the autoencoder network, $\mathbf{z} \in \Re^N$
z_i	Feature of an output vector **z** (*i*-th output component)
$\mathbf{y}$	Output vector of the hidden layer, $\mathbf{y} \in \Re^H$
y_i	Feature of an output vector **y** (*i*-th hidden neuron component)
$\mathbf{W}^{(l)}$	Weight matrix of the layer *l*
$w_{ij}^{(l)}$	Weight factor associated with the connection between unit *j* in the layer *l*, and unit *i* in the layer *l*+1
ρ	Sparsity parameter, which specifies the desired level of sparsity of the activation in the hidden layer
$\hat{\rho}_i$	The average activation of hidden unit *i*
α	Weight (in the error function) of the norm constraint of the neural weights
β	Weight (in the sparse autoencoder error function) of the sparsity penalty term