DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, SWEDEN 2015

A study on the similarities of Deep Belief Networks and Stacked Autoencoders

ANDREA DE GIORGIO

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

KTH, Royal Institute of Technology

A study on the similarities of Deep Belief Networks and Stacked Autoencoders

Degree Project in Computer Science, Second Cycle (DD221X)

Master's Program in Machine Learning

Supervisor
Anders Holst

Examiner
Anders Lansner

Master student
Andrea de Giorgio
andreadg@kth.se

October 3rd, 2015

Abstract

Restricted Boltzmann Machines (RBMs) and autoencoders have been used - in several variants - for similar tasks, such as reducing dimensionality or extracting features from signals. Even though their structures are quite similar, they rely on different training theories. Lately, they have been largely used as building blocks in deep learning architectures that are called deep belief networks (instead of stacked RBMs) and stacked autoencoders.

In light of this, the student has worked on this thesis with the aim to understand the extent of the similarities and the overall pros and cons of using either RBMs, autoencoders or denoising autoencoders in deep networks. Important characteristics are tested, such as the robustness to noise, the influence on training of the availability of data and the tendency to overtrain. The author has then dedicated part of the thesis to study how the three deep networks in exam form their deep internal representations and how similar these can be to each other. In result of this, a novel approach for the evaluation of internal representations is presented with the name of F-Mapping. Results are reported and discussed.

Contents

Abbreviations

AE *Autoencoder*

ANN *Artificial Neural Network*

BM *Boltzmann Machine*

CD *Contrastive Divergence*

CNN *Convolutional Neural Network*

DBN *Deep Belief Network*

DN *Deep Network*

FNN *Feedforward Neural Network*

GPU *Graphics Processing Unit*

MLP *Multi-Layer Perceptron*

MRF *Markov Random Field*

MSE *Mean Squared Error*

NLL *Negative Log Likelihood*

PCA *Principal Component Analysis*

PCD *Persistent Contrastive Divergence*

RBM *Restricted Boltzmann Machine*

SA *Stacked Autoencoder*

SdA *Stacked denoising Autoencoder*

SE *Squared Error*

Acknowledgments

This thesis was possible thanks to:

Anders Holst, for his great ideas and support.

Erik Ylipää, for his generous contributions and corrections.

Morgan Svensson and Vanya Avramova, for their precious friendship.

Carlo de Giorgio and Roberta Eller Vainicher, per il loro supporto e amore infinito.

Chapter 1

Introduction

1.1 Context and purpose

Deep Belief Networks (DBNs), Stacked Autoencoders (SAs) and Convolutional Neural Networks (CNNs) are the three main networks used by a field of Machine Learning called **deep learning** (Bengio, 2007).

CNNs are biologically-inspired variants of the Multi-Layer Perceptron (MLP), from Hubel and Wiesel's early work on the cat's visual cortex (Hubel & Wiesel, 1968). CNNs are deeply different w.r.t the other two architectures, DBNs and SAs, which are composed respectively by Restricted Boltzmann Machines (RBMs) and autoencoders as building blocks stacked to form a Deep Network (DN). In literature there are many different implementations of similar stacked architectures that have been proposed, but always arbitrarily selecting the use of RBMs or autoencoders without stating a precise reason for the choice.

By a quick glance the architectures look similar and it seems that their selection is totally interchangeable. Therefore, the purpose given to this thesis is to compare DBNs, SAs and SdAs with a focus on:

- implementation;
- robustness to noise;
- influence on training of the availability of more or less training data;
- tendency to overtrain;
- analysis of the internal representations formed at different layers;
- similarity of the internal representations.

When the task is to extract features automatically, invariance plays a key role in the way those features are represented. Having invariance in the internal representations means that the

features extracted are independent of some details inherent to the data such as, for example, temporal scale or spacial scale, rotation and translation. CNNs contains a complex arrangement of cells, inspired by the visual cortex, that are sensitive to small regions of the visual field and are tiled to fully cover it. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images. There are two basic cell types, the simple cells that respond maximally to specific edge-like patterns within their receptive field and the complex cells that have larger receptive fields and are locally invariant to the exact position of the pattern. This means that CNNs have a built-in structure to obtain invariance in the internal representations.

On the other hand, DBNs, SAs and SdAs are not able to produce invariant internal representations without improving the algorithms (see subsection 1.2.2 for related work), so an analysis of this property can not be included in the comparison presented in this thesis.

1.2 Related work

1.2.1 Performance of basic architectures

Since the introduction of unsupervised pre-training (Hinton et al., 2006) and Deep Networks (DNs), many new building blocks and overall schemas for stacking layers have been proposed. Most are focused on creating new training algorithms to build single-layer models which performs better. Among the possible architectures considered in literature are RBMs (Hinton et al., 2006; Krizhevsky & Hinton, 2009), sparse-coding (Olshausen et al., 1996; Lee et al., 2006; Yang et al., 2009), sparse RBMs (Lee et al., 2008), sparse autoencoders (Goodfellow et al., 2009; Poultney et al., 2006), denoising autoencoders (Vincent et al., 2008), marginalized denoising autoencoders (Chen et al., 2014), "factored" (Krizhevsky et al., 2010) and mean-covariance (Ranzato & Hinton, 2010) RBMs, as well as many others. Even though the learning module appears to be the most heavily scrutinized component of DNs, the literature focus mostly on incremental implementation of the same models.

In Tan & Eswaran (2008), Stacked Autoencoders are compared to stacked RBMs, i.e. DBNs, and the results clearly show that the stacked RBMs perform better. This comparison is the closest one to the purpose of this thesis. However they don't compare the internal representations.

In Coates et al. (2011) there is an analysis of the compared performance of sparse autoencoders and sparse RBMs, as state-of-the-art architectures, with a third newly-introduced architecture, to use as building blocks in DNs. Even in that case, the performance of sparse autoencoders and sparse RBMs are shown to to be similar.

There is little or no work aimed to show that autoencoders and RBMs perform in the same way, in spite of their training algorithms differences.

1.2.2 Invariance in internal representations

Goodfellow et al. (2009) presented a general set of tests for measuring invariances in the internal representations of Deep Networks. A finding in their experiments with visual data is that Stacked Autoencoders yield only modest improvements in invariance as depth increases. From this comes the suggestion that a mere stacking of shallow architectures may not be sufficient to exploit the full potential of deep architectures to learn invariant features.

Other authors preferred to introduce novel algorithms, such as in Sohn & Lee (2012), where Transformation-Invariant RBMs (TIRBMs) and Sparse TIRBMs, that can achieve invariance to a set of predefined transformations, are presented and compared to RBMs.

In Rasmus et al. (2014) the invariant representations in a denoising autoencoder are obtained with modulated lateral connections. They show that invariance increases towards the higher layers but significantly so only if the decoder has a suitable structure, i.e. the lateral connections.

The overall idea is that invariance in internal representations is highly desired but non-achievable with shallow architectures.

1.3 Overview of the thesis

In chapter 2 all the building blocks are presented and their theory explained.

In chapter 3, the deep learning methodology is introduced and the blocks are put together to form the examined deep architectures.

All the details regarding the implementation, the training, the test and the methodology behind all the operations executed to analyze the three architectures in exam are accurately described in chapter 4.

Chapter 5 contains all the results, thoroughly labeled, but only partially commented.

Finally, in chapter 6, the results are discussed in their wholeness and some conclusions are made.

Chapter 2

Autoencoders and Restricted Boltzmann Machines

2.1 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is an artificial neural network composed of an input layer, one or more hidden layers and an output layer (see figure 2.1.1).

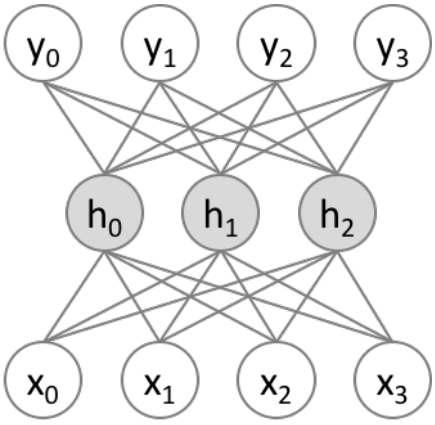

Figure 2.1.1: Graph of a multi-layer perceptron with one hidden layer. From bottom to top, respectively, the input layer (white), the hidden layer (gray) and the output layer (white). x_k is an input unit, h_j is a hidden unit and y_k is an output unit. Connections are exclusively drawn between different layers.

For a MLP with one hidden layer, the output vector $\boldsymbol{y}$ is obtained as a function of the input vector $\boldsymbol{x}$:

$$\boldsymbol{y} = f(\boldsymbol{x}) = G\left(\boldsymbol{b^{(2)}} + \boldsymbol{W^{(2)}} \cdot \boldsymbol{h}\left(\boldsymbol{x}\right)\right) = G\left(\boldsymbol{b^{(2)}} + \boldsymbol{W^{(2)}} \cdot s\left(\boldsymbol{b^{(1)}} + \boldsymbol{W^{(1)}} \cdot \boldsymbol{x}\right)\right)$$

with bias vectors $\boldsymbol{b^{(1)}}$, $\boldsymbol{b^{(2)}}$, weight matrices $\boldsymbol{W^{(1)}}$, $\boldsymbol{W^{(2)}}$ and activation functions G and s.

The hidden layer is the vector $\boldsymbol{h}(\boldsymbol{x})$.

Typical choices for s include $tanh$, with $tanh(a) = (e^a - e^{-a})/(e^a + e^{-a})$, that has been used in this thesis, or the logistic sigmoid function, with $sigmoid(a) = 1/(1 + e^{-a})$. A choice for G can be the $softmax$ function, especially when the MLP is used to identify multi-class membership probabilities, such as in this work.

Using one hidden layer is sufficient to obtain from an MLP an universal approximator (i.e. the neural network can represent a wide variety of interesting functions when given appropriate parameters), but substantial benefits come from using many hidden layers, that is the very premise of deep learning (LeCun et al., 2015).

2.1.1 Training

A typical training of a MLP is executed with **backpropagation**, an abbreviation for "backward propagation of errors", that is a common method of training artificial neural networks, used in conjunction with an optimization method such as **gradient descent**. This method calculates the gradient of a cost function, e.g. Negative Log Likelihood (NLL) or Mean Squared Error (MSE), with respect to all the weights in the network. The gradient is then used to update the weights, in an attempt to minimize the cost function. A local minimum of the cost function is found taking steps proportional to the negative of the gradient (or of the approximate gradient) of the cost function at the current point.

The backpropagation learning algorithm can be divided into two phases: propagation and weight update.

The propagation involves the following steps:

- Forward propagation of a training pattern's input through the neural network in order to generate the network's output activations.
- Backward propagation of the network's output activations through the neural network using the training pattern target in order to generate the deltas (the difference between the input and output values) of all output and hidden neurons.

The weight update is performed in the following steps for each neuron:

- Multiply the neural output delta and input activation to get the gradient of the weight.
- Subtract a ratio (percentage) of the gradient from the weight. This ratio influences the speed and quality of learning and it is called the **learning rate**. The greater the ratio, the faster the neuron trains; the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.

The two phases are repeated until the performance of the network is satisfactory.

2.1.2 Logistic regression

Logistic regression is a training algorithm inspired by a MLP with no hidden layers, so that the input is directly connected to the output through a sigmoid or softmax function. It is used as output layer in deep networks.

2.2 Autoencoders

An Autoencoder (AE), also called autoassociator or diabolo network, is an auto-associative neural network derived from the multi-layer perceptron which aim to transform inputs into outputs with the least possible amount of distortion (Bourlard & Kamp, 1988; Hinton & Zemel, 1994; Rumelhart et al., 1986). Autoencoders have been used as building blocks to train deep networks, where each level is associated with an AE that can be trained separately. An almost equal but more efficient implementation of the autoencoder is called denoising autoencoder (dA) and it also used as building block of one of the deep network architectures, so it is independently presented in section 2.3.

2.2.1 Structure

An autoencoder is respectively composed by an input layer, a hidden layer and an output layer (see figure 2.2.1). It takes an input $\mathbf{x} \in [0,1]^d$ and first maps it (with an encoder) to a hidden representation $\mathbf{y} \in [0,1]^{d'}$ through a deterministic mapping, e.g.:

$$\mathbf{y} = s(\mathbf{W}\mathbf{x} + \mathbf{b})$$

where s is a non-linearity such as a sigmoid function, e.g. $s(t) = \frac{1}{1+e^{-t}}$. The latent representation $\mathbf{y}$, or code, is then mapped back (with a decoder) into a reconstruction $\mathbf{z}$ of the same shape as $\mathbf{x}$. The mapping happens through a similar transformation, e.g.:

$$\mathbf{z} = s(\bar{\mathbf{W}}\mathbf{y} + \bar{\mathbf{b}})$$

where $\mathbf{z}$ should be seen as a prediction of $\mathbf{x}$, given the code $\mathbf{y}$.

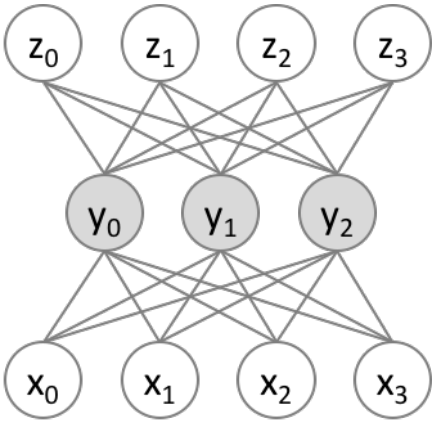

Figure 2.2.1: Graph of an Autoencoder. From bottom to top, respectively, the input layer (white), the hidden layer (gray) and the output layer (white). x_k is an input unit, y_j is a hidden unit and z_k is an output unit. Connections are exclusively drawn between different layers.

It is a good approach to constrain the weight matrix $\bar{\mathbf{W}}$ of the reverse mapping to be the transpose of the forward mapping: $\bar{\mathbf{W}} = \mathbf{W}^T$ because the number of free parameters is reduced and it simplifies the training. This is referred to as tied weights. The parameters of this model (namely $\mathbf{W}$, $\mathbf{b}$, $\bar{\mathbf{b}}$ and, if one doesn't use tied weights, also $\bar{\mathbf{W}}$) are optimized such that the average reconstruction error is minimized.

The reconstruction quality can be quantified with an error function, the traditional Squared Error can do the trick:

$$L(\mathbf{x}, \mathbf{z}) = ||\mathbf{x} - \mathbf{z}||^2.$$

If the input is interpreted as either bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

$$L_H(\mathbf{x}, \mathbf{z}) = -\sum_{k=1}^{d} [\mathbf{x}_k \log \mathbf{z}_k + (1 - \mathbf{x}_k) \log(1 - \mathbf{z}_k)]$$

2.2.2 Training

Autoencoders are supervised neural networks which are trained using the gradient descent method (see section 2.1).

If the number of hidden neurons is less than the number of input/output neurons, then the network will be forced to learn some sort of compression.

Even when the number of hidden neurons is large, there are various tricks in cost functions one can use to ensure that the network does something more interesting than learning the identity function. In fact, a common problem is that with no constraint besides minimizing the reconstruction error the autoencoder merely maps an input to its copy.

In Bengio (2007) there is the interesting discovery that, when trained with stochastic gradient descent, non-linear autoencoders with more hidden units than inputs (called over-complete) yield good representations, in the sense that those representations can be fed to a next network for classification, and it will have a low classification error.

Other ways by which an autoencoder with more hidden units than inputs could be prevented from learning the identity function, capturing something useful about the input in its hidden representation are the addition of sparsity (Poultney et al., 2006; Lee et al., 2008), i.e. forcing many of the hidden units to be zero or near-zero, or the addition of noise in the transformation from input to reconstruction (Vincent et al., 2008), a technique that is used for the denoising autoencoders, discussed in section 2.3. .

2.3 Denoising Autoencoders

A denoising Autoencoder (dA) is an improved version of the basic autoencoders (Vincent et al., 2008), the simple idea behind it is to force the Autoencoders to not learn the identity function, but more robust features, by reconstructing the input from a corrupted version of it. The only way for an Autoencoder to reconstruct a distorted input is to capture the statistical dependencies between the inputs, and that is exactly what one would like. Note that being able to predict any subset of variables from the rest is a sufficient condition for completely capturing the joint distribution between a set of variables and this is also how Gibbs sampling works in the Restricted Boltzmann Machines (RBMs) (see section 2.4 and subsection 2.4.2).

2.4 Restricted Boltzmann Machines

A Boltzmann Machine (BM) is a particular form of Markov Random Field (MRF) for which the energy function is linear in its free parameters. Some of its variables, called hidden units, are never observed, but allow the machine to represent complicated distributions internally.

A Restricted Boltzmann Machine (RBM) was originally presented by Smolensky (1986) with the name of Harmonium, and further restrict the BM by eliminating all the visible-visible and hidden-hidden connections. A graphical depiction of an RBM is shown in figure 2.4.1. The RBMs rose to prominence only after that Geoffrey Hinton and his collaborators invented fast learning algorithms (see subsection 2.4.3) for RBMs in the mid-2000s.

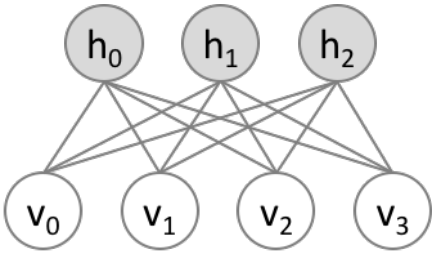

Figure 2.4.1: Graph of a Restricted Boltzmann Machine. v_j is a visible unit and h_j is a hidden unit. In an RBM, connections are exclusively drawn between visible units and hidden units.

2.4.1 Structure

The energy function of an RBM is defined as:

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b^T v} - \boldsymbol{c^T h} - \boldsymbol{h^T W v}, \tag{2.4.1}$$

where $\boldsymbol{W}$ represents the weights connecting visible and hidden units and $\boldsymbol{b}$ and $\boldsymbol{c}$ are biases, respectively, of the visible and hidden units

The free energy can also be expressed in the following form:

$$F(\boldsymbol{v}) = -\boldsymbol{b^T v} - \sum_i log \sum_{h_i} e^{h_i(\mathbf{c_i} + \boldsymbol{W_i v})}. \tag{2.4.2}$$

By the property that visible and hidden units are conditionally independent of one-another, results:

$$p(\boldsymbol{h}|\boldsymbol{v}) = \prod_i p(h_i|\boldsymbol{v});$$

$$p(\boldsymbol{v}|\boldsymbol{h}) = \prod_j p(v_j|\boldsymbol{h}).$$

Each sample $p(\boldsymbol{x})$ can be obtained by running a Markov chain to convergence, using Gibbs sampling (see subsection 2.4.2).

A common studied case is when binary units are used, so that v_j and $h_i \in \{0, 1\}$, and a probabilistic version of the usual neural activation is obtained:

$$P(h_i = 1|\boldsymbol{v}) = sigm(c_i + \boldsymbol{W_i v})$$

$$P(v_j = 1|\boldsymbol{h}) = sigm\left(b_j + \boldsymbol{W^T}_{\boldsymbol{j}} \boldsymbol{h}\right).$$

The free energy of an RBM with binary units becomes:

$$F(\boldsymbol{v}) = -\boldsymbol{b^T v} - \sum_i log\left(1 + e^{(c_i + \boldsymbol{W_i v})}\right).$$

2.4.2 Gibbs sampling

Gibbs sampling is used to obtain a sample $p(\boldsymbol{x})$ by running a Markov chain to convergence, the sampling of the joint of N random variables $\boldsymbol{S} = (S_1, \ldots, S_N)$ is done through a sequence of N sampling sub-steps of the form $S_i \backsimeq p(S_i|S_{-i})$ where S_{-i} contains the $N-1$ other random variables in $\boldsymbol{S}$, excluding S_i.

In RBMs the Gibbs sampling is a way to obtain values for visible or hidden units, starting from the ones that are given.

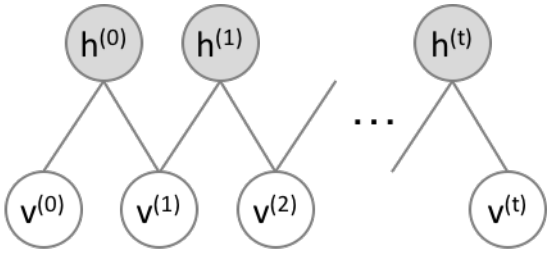

Figure 2.4.2: Steps of Gibbs sampling in a Restricted Boltzmann Machine. $\boldsymbol{v}$ are visible units and $\boldsymbol{h}$ are hidden units. Each step is performed from a visible unit to hidden unit, or vice versa.

A step in the Markov chain (see figure 2.4.2) estimates the hidden units from the visible units (encoding of the input):

$$\boldsymbol{h^{(n+1)}} \backsimeq sigm(\boldsymbol{W^T v^{(n)}} + \boldsymbol{c}),$$

or the visible units from the hidden units (reconstruction of the input):

$$\boldsymbol{v^{(n+1)}} \backsimeq sigm(\boldsymbol{W h^{(n+1)}} + \boldsymbol{b}),$$

where $\boldsymbol{v}^{(n)}$ and $\boldsymbol{h}^{(n)}$, respectively, refer to the set of all visible and hidden units at the n-th step of the Markov chain.

When the number of steps tends to infinity, it is possible to consider $p(\boldsymbol{v}, \boldsymbol{h})$ as accurate samples, but to get sufficiently close to this limit there is a need for immense computational power.

An algorithm called **contrastive divergence** is often used instead (see subsection 2.4.3).

2.4.3 Contrastive Divergence (CD-k)

Contrastive Divergence (CD-k) is based on two simple ideas.

The first algorithm was presented by Hinton (2002). It proposes that, since the true underlying distribution of the data $p(\boldsymbol{v}) \backsimeq p_{train}(\boldsymbol{v})$ is desired, the Markov chain can be initialized with a training sample, so that the first step can be already close to the convergence to the final distribution.

The second idea for CD is to not wait for the chain to converge. It has been demonstrated that, with the first modification, for only k steps of Gibbs sampling, especially with just one sample ($k = 1$), the samples obtained are extremely good (Bengio & Delalleau, 2008).

In order to improve the contrastive divergence, another technique called **persistent CD** has been proposed and it is presented in subsection 2.4.4.

2.4.4 Persistent Contrastive Divergence

Persistent Contrastive Divergence (PCD), presented by Tieleman (2008), uses another approximation for sampling from $p(\boldsymbol{v}, \boldsymbol{h})$. It relies on a single Markov chain which has a persistent state, i.e. the chain is not restarted for each observed example. This allows to extract new samples by simply running the Markov chain for k-steps and the state is always preserved for subsequent updates.

2.4.5 Training

Since RBMs are energy based models, i.e. associate a scalar energy to each configuration of the variables of interest, learning corresponds to modifying that energy function so that its shape has desirable properties, such as low energy configurations.

Energy-based probabilistic models define a probability distribution through an energy function, as follows:

$$p(x) = \frac{e^{-E(x)}}{Z}. \tag{2.4.3}$$

The normalizing factor Z is called the **partition function** by analogy with physical systems:

$$Z = \sum_{x} e^{-E(x)}. \tag{2.4.4}$$

An energy-based model can be learnt by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data. The log-likelihood is defined first:

$$L(\theta, D) = \frac{1}{N} \sum_{x^{(i)} \in D} log\ p(x^{(i)}) \tag{2.4.5}$$

and then the loss function as being the negative log-likelihood:

$$l(\theta, D) = -L(\theta, D). \tag{2.4.6}$$

A problem that regards the training of an RBM is about the impossibility to estimate the log-likelihood $log(P(x))$ during the training because of the partition function in equation 2.4.3 that is unknown.

To solve this, a **pseudo-likelihood** (PL) is calculated as follows:

$$PL(\boldsymbol{x}) = \prod_i P(\boldsymbol{x_i}|\boldsymbol{x_{-i}}) \tag{2.4.7}$$

and

$$logPL(\boldsymbol{x}) = \sum_i logP(\boldsymbol{x_i}|\boldsymbol{x_{-i}}), \tag{2.4.8}$$

where $\boldsymbol{x_{-i}}$ denotes the set of all bits of $\boldsymbol{x}$ except the i-th bit. The log-PL is therefore a sum of the log-probabilities of each bit $\boldsymbol{x_i}$ conditioned on the state of all other bits.

A stochastic approximation of log-PL can be also introduced:

$$g = N \cdot logP(\boldsymbol{x_i}|\boldsymbol{x_{-i}}), \tag{2.4.9}$$

where $i \backsimeq U(0, N)$, the uniform distribution, and

$$E[g] = logPL(\boldsymbol{x}) \tag{2.4.10}$$

is the expectation taken over the uniform random choice of index i, with N as number of visible units.

For binary units it becomes:

$$logPL(\boldsymbol{x}) \backsimeq N \cdot log\frac{e^{-FE(\boldsymbol{x})}}{e^{-FE(\boldsymbol{x})} + e^{-FE(\bar{\boldsymbol{x}})}} \backsimeq N \cdot log\left[sigm\left(FE(\bar{\boldsymbol{x}_i}) - FE(\boldsymbol{x})\right)\right] \tag{2.4.11}$$

where $\bar{\boldsymbol{x}_i}$ refers to $\boldsymbol{x}$ with the i-th bit flipped.

The pseudo-likelihood is then used as a cost function for the training of the RBM with gradient descent.

Chapter 3

Deep Learning

Deep Learning (deep machine learning, or deep structured learning, or hierarchical learning, or sometimes DL) is an approach to machine learning that refers to a set of algorithms that aim to model high-level abstractions in the input data by using complex model architectures, mainly based on multiple non-linear transformations (Deng & Yu, 2014; Bengio et al., 2013; Bengio, 2009; Schmidhuber, 2015). Various deep learning architectures such as stacked autoencoders, convolutional deep neural networks, deep belief networks (referring to stacked restricted Boltzmann machines and rarely also extended to stacked autoencoders) and recurrent neural networks have been applied to fields such as computer vision, automatic speech recognition, natural language processing, audio recognition and bioinformatics where they have been shown to produce state-of-the-art results on various tasks.

3.1 Deep Belief Networks

Deep Belief Networks (DBN) have been introduced by Hinton & Salakhutdinov (2006) as stacked restricted Boltzmann machines with a fast-learning algorithm (Hinton et al., 2006) that allowed the structure to achieve better results with less computational effort. It refers to stacked RBMs, but rarely also to its counter-part realized with stacked autoencoders (Bengio et al., 2007). In this thesis the acronym is referred exclusively to stacked RBMs.

A DBN is a graphical model which learns to extract a deep hierarchical representation of the training data. It models the joint distribution between an observed vector x and l hidden layers h^k as follows:

$$P\left(\boldsymbol{x}, \boldsymbol{h^1}, \ldots, \boldsymbol{h^l}\right) = \left(\prod_{k=0}^{l-2} P\left(\boldsymbol{h^k} | \boldsymbol{h^{k+1}}\right)\right) P\left(\boldsymbol{h^{l-1}} | \boldsymbol{h^l}\right) \tag{3.1.1}$$

where $\boldsymbol{x} = \boldsymbol{h^0}$, $P\left(\boldsymbol{h^k} | \boldsymbol{h^{k+1}}\right)$ is a conditional distribution for the visible units conditioned on the hidden units of the RBM at level k, and $P\left(\boldsymbol{h^{l-1}} | \boldsymbol{h^l}\right)$ is the visible-hidden joint distribution

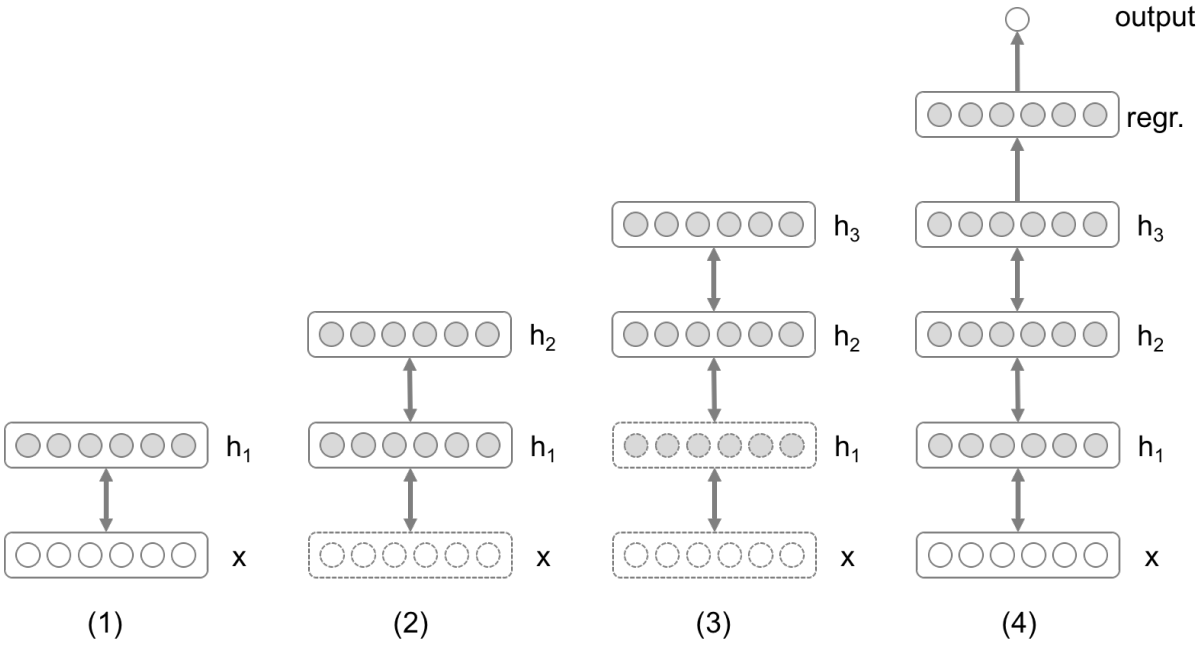

Figure 3.1.1: Layer-wise training of a Deep Belief Network, composed by stacked RBMs. From the bottom, $\boldsymbol{x}$ is the input and $\boldsymbol{h_k}$ are hidden layers. (1) The first layer is trained. (2) The second layer is trained using the first hidden layer as visible units. (3) The third layer is trained using the second hidden layer as visible units. (4) A regression layer is added and the resultant DBN is ready for the fine-tuning.

in the top-level RBM (see figure 3.1.1).

Hinton et al. (2006) and Bengio et al. (2007) introduced a greedy layer-wise unsupervised training that can be applied to DBNs with RBMs as building blocks for each layer. The algorithm is the following:

1. Train the first layer as an RBM that models the raw input $\boldsymbol{x} = \boldsymbol{h^0}$ as its visible layer.

2. The first layer internal representation is then used as input data for the second layer. This representation can be either the mean activations $P\left(\boldsymbol{h^{(1)}} = 1|\boldsymbol{h^{(0)}}\right)$ or samples of $P\left(\boldsymbol{h^{(1)}}|\boldsymbol{h^{(0)}}\right)$. The second layer is thus trained.

3. Iterate 2 for the desired number of layers, each time propagating upward either samples or mean values.

4. Fine-tune all the parameters of the deep network with respect to a supervised training criterion, after having added a further layer to convert the learned deep representation into supervised predictions. Such layer can be a simple linear classifier, e.g. a regression layer.

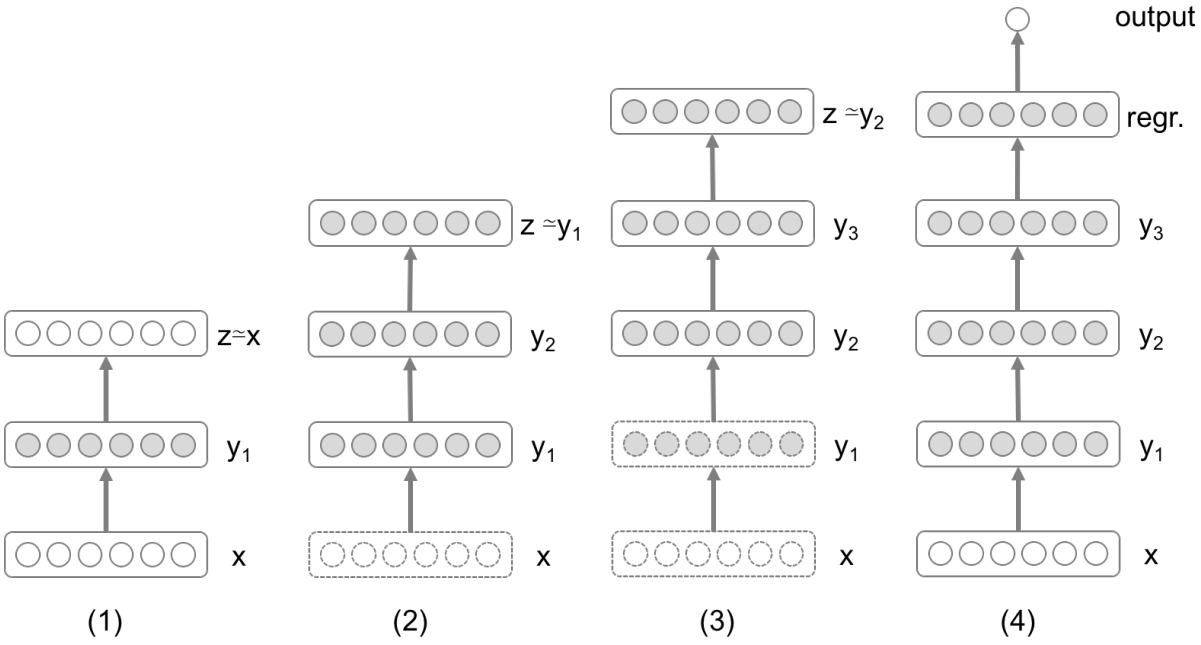

Figure 3.2.1: Layer-wise training of a Stacked Autoencoder. From the bottom, $\boldsymbol{x}$ is the input and $\boldsymbol{y_k}$ are hidden layers and $\boldsymbol{z}$ is the output (reconstructed input). (1) The first layer is trained. (2) The second layer is trained using the first hidden layer as input to reconstruct. (3) The third layer is trained using the second hidden layer as input to reconstruct. (4) A regression layer is added and the resultant Stacked Autoencoder is ready for the fine-tuning.

3.2 Stacked Autoencoders

Autoencoders can be stacked to form a deep network by feeding the internal representation (output code) of the Autoencoder at the layer below as input to the considered layer. The unsupervised pre-training of the architecture is done one layer at a time (see figure 3.2.1). Once the first k layers are trained, it is possible to train the $(k+1)$-th layer using the internal representation of the k-th layer.

When all the layers are pre-trained, a classification layer is added and the deep network can be fine-tuned, exactly how it is done in a Deep Belief Network (see section 3.1).

The advantage of this architecture is that by using more hidden layers than a single autoencoder, a high-dimensional input data can be reduced to a much smaller code representing the important features.

3.3 Stacked denoising Autoencoders

A Stacked denoising Autoencoder (SdA) is an extension of the stacked autoencoder and was introduced by Vincent et al. (2008). The building blocks of this deep network are denoising autoencoders (see section 2.3). The SdA is layer-wise trained and fine-tuned exactly as described

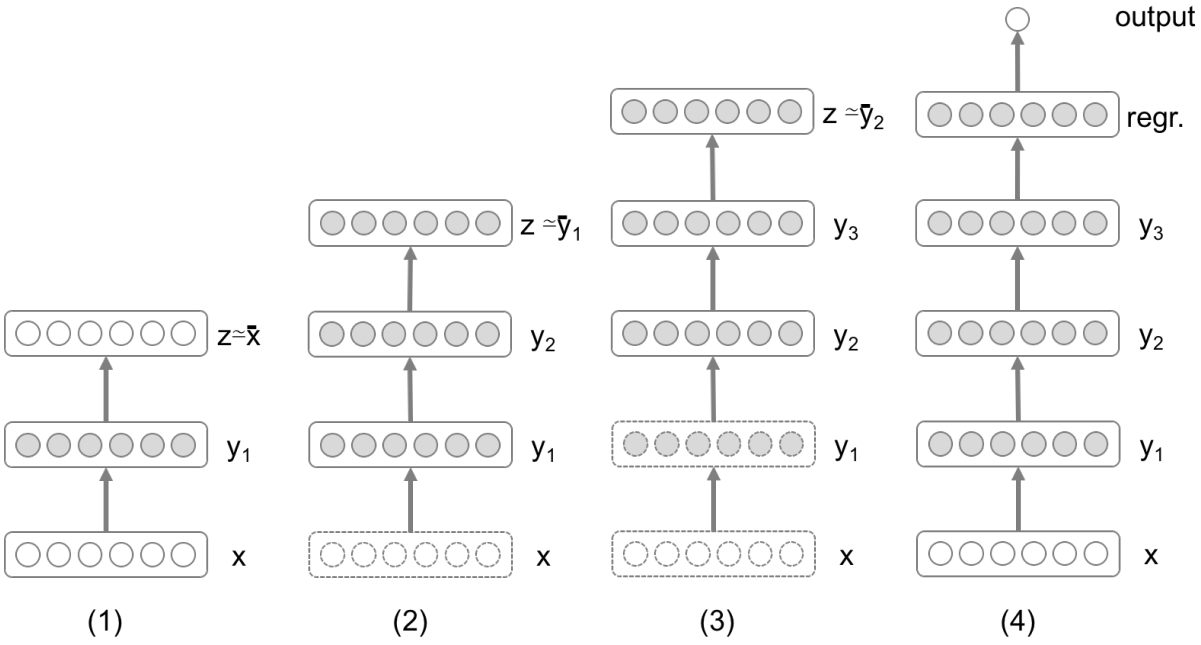

Figure 3.3.1: Layer-wise training of a Stacked denoising Autoencoder. From the bottom, $\boldsymbol{x}$ is the input and $\boldsymbol{y_k}$ are hidden layers and $\boldsymbol{z}$ is the output (reconstructed input). Each autoencoder, given its input $\boldsymbol{x}$, reconstructs a noisy version of it $\bar{\boldsymbol{x}}$. (1) The first layer is trained. (2) The second layer is trained using the first hidden layer as input to reconstruct. (3) The third layer is trained using the second hidden layer as input to reconstruct. (4) A regression layer is added and the resultant Stacked denoising Autoencoder is ready for the fine-tuning.

in section 3.2. In figure 3.3.1 it is shown how the training is performed, but there is basically no difference w.r.t. the SA, with the only exception that each denoising autoencoder, given an input $\boldsymbol{x}$, reconstructs a noisy version of it, namely $\bar{\boldsymbol{x}}$, and not the original input.

3.4 Relation between models

Comparing the structure of either a stacked autoencoder (SA) or a stacked denoising autoencoder (SdA) (they are equal) with the structure of a deep belief network (DBN), one can see that the fundamental differences consist in the output layer and the direction of the connections between layers. In the first architecture the output layer is separate from the input layer but in the second architecture it coincides with it. This is also remarked by the arrows connecting the layers. In a SA or SdA the information flows unidirectionally from the input layer, through the hidden layer, up to the output layer. In a DBN the information flows both ways between the visible (input/output) layer and the hidden layer.

By the point of view of the data flowing into the three structures, this suggests that they may all behave in the same way.

If the input data given to the three structures is the same, a legit question is: what are the similarities in the hidden layers, i.e. the internal representations, especially if the outputs turned out to be very similar after the separate trainings?

The difference in the (layer-wise) training algorithms has already been delineated in this chapter, a similarity in the structures has been hereby noted, in the following chapters the focus is moved to the data.

Chapter 4

Methods

4.1 Dataset

The dataset chosen for the analysis of the deep networks is MNIST which contains a set of hand-written digit samples from 0 to 9. Each digit is in the form of a grey-scale image, of size 28x28 pixels, with values in range $[0, 1]$. Background is represented by low (0) values and the digits have high values (around 1).

The dataset is composed by a total of 50'000 training samples, 10'000 validation samples and 10'000 test samples.

The MNIST dataset has been distributed in three new datasets named MNIST-10, MNIST-50 and MNIST-100 which contain, respectively, the 10%, 50% and 100% of the original dataset training, validation and test samples. The composition of each dataset is summarized in table 4.1.

Table 4.1: Composition of the MNIST-10, MNIST-50 and MNIST-100 datasets.

Dataset	**Percentage of MNIST**	**Number of samples**		
		Training	**Validation**	**Test**
MNIST-10	10%	5'000	1'000	1'000
MNIST-50	50%	25'000	5'000	5'000
MNIST-100	100%	50'000	10'000	10'000

4.2 Implementation of the Deep Networks

This section contains the details concerning the implementation of the deep learning architectures.

4.2.1 Programming language and framework

All the code used in this work is written in the Python language, using Theano which is a Python library that is used to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays. Important features of Theano are the possibility to share parameters between different networks and use the Graphics Processing Units (GPUs) to speed up simulations. Theano has been employed for large-scale computationally intensive scientific investigations since 2007 (Bergstra et al., 2010; Bastien et al., 2012).

4.2.2 Hardware and experiments

The analysis of the deep architectures, namely Deep Belief Network (DBN), Stacked Autoencoders (SA) and Stacked denoising Autoencoders (SdA), is done through three different trainings, validations and tests each, using the datasets MNIST-10, MNIST-50 and MNIST-100. All the experiments are executed on a MacBook Pro with NVIDIA GeForce GT 650M 512MB GPU.

4.2.3 Training parameters

The training of deep architectures consists of two steps:

- a layer-wise pretraining of each building block;
- the fine-tuning of all the three architectures composed of all the layers, including the last regression layer, with a backpropagation (see section 2.1.1) method applied considering the whole architecture a multi-layer perceptron (see section 2.1).

In order to achieve a comparable performance for all the architectures, the training and fine-tuning parameters have been set equally.

In this thesis a logistic regression classifier (see section 2.1.2) is used to classify the input $\boldsymbol{x}$ based on the output of the last hidden layer, $\boldsymbol{h^{(l)}}$ for a DBN (see figure 3.1.1) or $\boldsymbol{y^{(l)}}$ for SA (see figure 3.2.1) and SdA (see figure 3.3.1). Fine-tuning is performed via supervised gradient descent (see section 2.1.1) of the negative log-likelihood cost function.

4.2.3.1 Number of layers

All the architectures have a same number of hidden layers, that is three, and for each hidden layer the same size. The first layer has 900 hidden units, the second has 400 hidden units and the third has 100 hidden units.

4.2.3.2 Layer-wise training

Every layer, built with either an RBM, an autoencoder (AE) or a denoising autoencoder (dA), is trained for 20 epochs, where one epoch corresponds to 500 iterations on MNIST-10, 2'500 iterations on MNIST-50 and 5'000 iterations on MNIST-100, given the different sizes of the three datasets used and a mini-batch corresponding to 10 samples. This means that the gradient is computed against 10 instead one training examples at each step of the gradient descent (see section 2.1.1). This size for the mini-batches was chosen, after several try and error attempts, and based on the fact that the databases contain 10 different digits, so this size of mini-batch should carry at least a few different digits in the error evaluation per weights update. It also allows for more training speed than in 1 sample mini-batches and still a good accuracy. For layer-wise training times refer to table 4.2.

The training learning rate is set to 0.01, always after several try and error attempts with the aim of maximizing the error reduction w.r.t. a shorter training time.

When training a denoising autoencoder, the parameter for the reconstruction noise is also set to 30%, for the quantity of salt and pepper noise added to the samples before the input reconstruction.

4.2.3.3 Fine-tuning of the deep networks

The fine-tuning parameters include a learning rate, that is again set to 0.01, and a number of maximum epochs at which the fine-tuning needs to be stopped in case the early-stopping had not done it yet. This maximum value is set to 30 epochs, where one epoch corresponds to 500 iterations on MNIST-10, 2'500 iterations on MNIST-50 and 5'000 iterations on MNIST-100, given the different sizes of the three datasets used and a mini-batch corresponding to 10 samples.

The early stopping has a patience parameter which varies at each epoch, increasing if a best model is found, or decreasing if no best model is found. For fine-tuning times refer to table 4.2.

Table 4.2: Layer-wise training and fine-tuning times for the three architectures. The models are evaluated on the validation and test sets at the end of every epoch, that is 500 iterations on MNIST-10, 2'500 iterations on MNIST-50 and 5'000 iterations on MNIST-100. Whenever the model performs better than the previous it is chosen as best model. The best model is found with early-stopping or a forced-stop at epoch 30 in all cases.

Architecture	**Dataset**	**Training time**		**Best model on iteration**
		Layer-wise	**Fine-tuning**	
DBN	MNIST-10	1.49m	1.56m	14'500 (epoch 29)
	MNIST-50	7.95m	6.78m	75'000 (epoch 30)
	MNIST-100	14.45m	13.03m	150'000 (epoch 30)
SA	MNIST-10	1.42m	0.57m	3'000 (epoch 6)
	MNIST-50	7.94m	6.56m	70'000 (epoch 28)
	MNIST-100	14.33m	12.37m	150'000 (epoch 30)
SdA	MNIST-10	1.51m	1.50m	10'500 (epoch 21)
	MNIST-50	7.72m	6.47m	62'500 (epoch 25)
	MNIST-100	14.51m	12.32m	150'000 (epoch 30)

4.3 Evaluation of the Deep Networks

This section contains all the methods used for the evaluation of the deep learning architectures.

The robustness to noise of the three architectures is evaluated with standard methods in subsection 4.3.1. Filters are created and evaluated following an approach discussed in the latest literature and presented in subsection 4.3.2.

About the similarities in the internal representations, a numeric standard evaluation criterion is applied (see subsection 4.3.3), but the author has also found a way to compare the internal representations with a technique that does not seem to exist in literature. The author has chosen to call it **F-Mapping** (from Filter Mapping) and presents it in subsection 4.3.4.

Finally, the inputs are reconstructed using the internal representations to show the success of the overall training and testing of the architectures and their learnt internal representations. Numerical analysis is also applied to the reconstructed input as a final test (see subsection 4.3.6).

4.3.1 Test robustness to noise

In order to test the robustness of each architecture to noisy inputs, all the samples in the test sets are corrupted with salt and pepper noise in three different levels: 10%, 20% and 50%. The three architectures are then trained and validated, respectively, with the uncorrupted training and validation sets, then tested over the corrupted test sets. Note that the denoising autoencoder continues to apply a 30% of salt and pepper noise to the training data to reconstruct by default.

It is to be noted that the salt and pepper noise is the same one used in the denoising autoencoder as an addition to the input during the layer-wise training (for further details read section 2.3).

4.3.2 Create filters

A way to visualize how the deep learning architectures are encoding the input data into their internal representations is to exploit the information stored in the weight matrices, which can be seen as filters used to transform the input data. As each architecture is composed by more than one layer which means more than one $\mathbf{W}$ matrix, it is possible to project the $\mathbf{W}$ matrix of the last layer down to the $\mathbf{W}$ matrix of the first layer, as described by Erhan et al. (2010), mainly with three techniques:

1. Linear combination of previous units;
2. Output unit sampling;
3. Activation Maximization (AM).

In this experimental set it is chosen to use neither the activation maximization nor the output unit sampling (can only be used for RBMs), but only the linear combination of previous units, for it is a more general approach and can be immediately applied to the weight matrices to show the filters.

In the first layer, the matrix $\mathbf{W_1}$ can be immediately shaped as filters:

$$\mathbf{F_1} = (\mathbf{W_1})^T$$

From the second layer on, it is necessary to calculate the matrix product of that layer's $\mathbf{W_i}$ matrix up to $\mathbf{W_0}$:

$$\mathbf{F_2} = (\mathbf{W_1} \times \mathbf{W_2})^T$$

$$\mathbf{F_2} = (\mathbf{W_1} \times \mathbf{W_2} \times \mathbf{W_3})^T$$

Where $\boldsymbol{F_k}$ represents a set of filters at layer k. Each row is a filter that can be shaped as an image and should show what details are relevant in the input image in order to be represented as the activation of the corresponding unit in the hidden layer. While the first layer or even the second one's filters are not significantly good for understanding how the deep architectures encode the input data, in the third layer's filters a high-level representation of the input images details is expected.

Note that both the bias $\mathbf{b_i}$ and the sigmoid activation function - that would make the equation non-linear - are obviously excluded from the filters. Also, the filters do not depend on any input values.

The **W** matrices used in the three architectures here analyzed have the same dimensions for the respective layers. Each input from the MNIST dataset is a digit image of 28x28 pixels that is vectorized as 784 values. The first hidden layer encodes a vector of 900 values (visualized as a tile of 30x30 pixels), the second hidden layer encodes a vector of 400 values (visualized a tile of 20x20 pixels) and the third hidden layer encodes a vector of 100 values (visualized as a tile of 10x10 pixels). So, the first **W** matrix has dimensions 784x900, the second 900x400, the third 400x100.

A filter is a 28x28 image that contains the corresponding values of a single hidden unit per layer. So, there are 900 filters for the first layer, 400 for the second and 100 for the third.

4.3.3 Exploit the internal representations

An internal representation is formed by all the hidden units activations at a given input. One layer of hidden units can already encode the input data as features represented in a lower dimensional space. In deep learning, adding more layers produces better low dimensional data representations. In fact, stacked autoencoders have been used for dimensionality reduction purposes (Hinton & Salakhutdinov, 2006). In general, deeper layers should correspond to higher level features.

The only disadvantage of the internal representations for humans is that, being merely an encoding, they are very difficult to understand in an intuitive way, e.g. as an image (see figure 5.4.1), so it is difficult to attest the quality of the information carried in the internal representations. A way to check the composition of an internal representation consists of analyzing its informational content and its properties with a numerical approach.

Sparseness is one desired property because, as reported by Olshausen & Field (2004):

- "Early work on associative memory models, for example, showed that sparse representations are most effective for storing patterns, as they maximize memory capacity because of the fact that there are fewer collisions (less cross-talk) between patterns."

- "Later work has similarly showed that sparse representations would be advantageous for learning associations in neural networks, as they enable associations to be formed effectively using local (neurobiologically plausible) learning rules, such as Hebbian learning."

- "In many sensory nervous systems neurons at later stages of processing are generally less active than those at earlier stages. [...] The nervous system was attempting to form neural representations with higher degrees of specificity. For example, a neuron in the retina

> responds simply to whatever contrast is present at that point in space, whereas a neuron in the cortex would respond only to a specific spatial configuration of light intensities (e.g. an edge of a particular orientation. [...] Several computational studies conducted have demonstrated concretely the relationship between sparsity and the statistics of natural scenes."

Since the architectures in exam perform a reduction of dimensionality when encoding the data in the deep layers, a trade-off between density, i.e. the compressed data, and sparsity is expected. If the training is successful, all architectures should entangle the main features in the data through their internal representations.

4.3.3.1 Histograms

One way to understand the numerical composition of an internal representation is to compute an histogram, which allows to count the activations and to visualize if their distribution is sparse or not. If an internal representation is sparse, only a few hidden units have high (1) values and the rest have no activations, i.e. low (0) values. The information carried by an internal representation can thus be studied w.r.t the number and intensity of its activations.

4.3.3.2 The Gini coefficients: a measure of sparseness

A second numerical attempt can be made with a method presented in recent literature: the Gini coefficients (Hurley & Rickard, 2009).

The Gini coefficients are a mean to quantify the sparseness of the high (1) and low (0) activations in a sequence, by comparison with the Lorenz curve (Lorenz, 1905) generated from the sequence to test. By looking at a Lorenz curve (see figure 4.3.1), greater equality means that the line based on actual data is closer to a 45-degree line that shows a perfectly equal distribution. Greater inequality means that the line based on actual data will be more bowed away from the 45-degree line. The Gini coefficient is based on the ratio of the area between the 45-degree line and the actual data line, and the area under the data line.

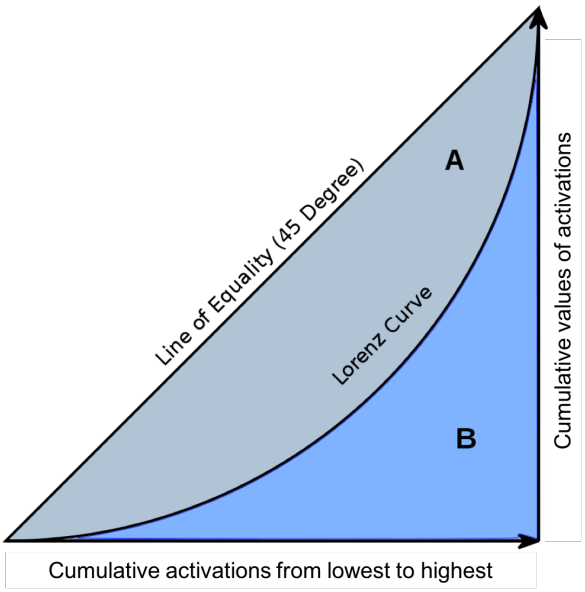

Figure 4.3.1: A Lorenz curve for an internal representation. A is the area between the equality 45-degree line and the Lorenz curve generated by the sequence; B is the area under the Lorenz curve. The Gini coefficient is proportional to the ratio of the areas A/B.

In order to obtain a Gini coefficient, a regularization of the sequence is made, so that the sum of all the values is one. Then, a normalization of the Gini coefficients is executed by dividing the obtained value for the Gini coefficient of the sparsest sequence (one activation, all zeros) of the same length.

As a result of the described procedure, a Gini value of 1 means that a sequence is sparse, a Gini value of 0 means that the sequence is non-sparse.

A high number of random with uniform distributions in $[0, 1]$ sequences have also been generated and tested, their Gini coefficient tends to $1/3$.

4.3.4 Map the internal representations (F-Mapping)

A simple numerical method can be used to compare the internal representations of the three architectures with the Mean Squared Error (MSE). What is tricky is to obtain two internal representations of the same input, created by two different architectures, that can be compared to each other. In fact, there is no guarantee that two different representations may be able to store the same information (common features created from the input) in the same shape.

This because each hidden unit that works on a different feature can be located at any position of the internal representation for a given architecture.

But there is a way out. The filters, that have been shown before, represent a trace of the information stored in each hidden unit of an hidden layer. The vector with all the activations of the hidden units in a hidden layer is called an internal representation. So, for each element

of an internal representation vector there is a filter image telling what sort of information that element carries.

Therefore, by sorting the filters $\mathbf{F^i}$ and $\mathbf{F^j}$ of the same layer of two different architectures i and j, one can obtain a mapping function f_M of their internal representations. This procedure has been hereby referred as **F-Mapping** (from Filter Mapping) and can be described by the following steps:

1. Two sets of filters $\mathbf{F^i}$ and $\mathbf{F^j}$, corresponding to the same layer of architectures i and j, are first normalized in range $[-2, 2]$. For each row, namely one filter, the mean of the first three values and last three values is used as background value. This allows to center the background at value zero.

2. Two index tables are created for the filter set $\mathbf{F^i}$ and $\mathbf{F^j}$, so that N indices, corresponding to N filter positions are stored in each index table. k is initialized to 1.

3. Each filter of the set $\mathbf{F^i}$ present in its index table is compared to each filter of the set $\mathbf{F^j}$ present in its index table through mean squared error. The MSE gives more importance to the values in ranges $[-2, -1]$ and $[1, 2]$, that are the filters' white and black marks, leaving what is closer to the background, range $[-1, 1]$, less relevant. Also, positive and negative values are treated equally. The couple of filters that minimize the sum of MSEs of their values compared one by one, is chosen and mapped by their respective positions in $f_M^i(k)$ and $f_M^j(k)$ and k is incremented of 1.

4. The position of each chosen filter is then removed from the respective set of filter index table, now composed of N-1 elements each.

5. Steps 3 and 4 are repeated other N-1 times until the two sets $\mathbf{F^i}$ and $\mathbf{F^j}$ index tables have no elements left and all the mapping values $f_M^i(1), \ldots, f_M^i(N)$ and $f_M^j(1), \ldots, f_M^j(N)$ are obtained.

At the end of this process an F-Mapping function f_M for the two sets of filters is obtained, the original filter sets can be compared using their F-Mappings. Since the idea for the F-Mapping technique came out during the test of the architectures, images of the normalized filters used during the F-Mapping are also presented, together with the results in section 5.5.1.

4.3.5 Comparison of sorted internal representations with F-Mapping

Once an F-Mapping f_M is defined for each architecture pair, two internal representations $\mathbf{v^i}$ and $\mathbf{v^j}$, of the architectures i and j, can be sorted so that $\bar{\mathbf{v^i}} = f_M^i\left(\mathbf{v^i}\right)$ can be supposed to represent information in the same shape of $\bar{\mathbf{v^j}} = f_M^j\left(\mathbf{v^j}\right)$. Using again the Mean Squared

Error to compare first $\mathbf{v^i}$ and $\mathbf{v^j}$ and then $\bar{\mathbf{v}}^\mathbf{i}$ and $\bar{\mathbf{v}}^\mathbf{j}$ allows to note if the F-Mapped internal representations look more similar or not before and after this procedure.

The aim of this procedure is to align features in the internal representations, so that the meaning of the whole internal representation vector can be comparable for different architectures. For a numerical clue on the validity of this technique, refer to the results in subsection 5.5.2.

4.3.6 Verify the input reconstructions

One internal representation can also be fed to each previous layer in order to obtain the reconstructed input of that layer, using its reconstruction function, i.e. sampling the visible units from the hidden units in RBMs or reconstructing the input (output) from the hidden units in the autoencoders. Each reconstruction corresponds to the previous layer's hidden representation, and the final one is the input of the deep architecture, or better a reconstruction of what it would have been in order to trigger the highest layer internal representation considered. This process is hereby referred to as **input reconstruction**.

It is possible to reconstruct different kind of internal representations from any layer. In particular: single activations, digits, mean-digits and filtered mean-digits. Each of them is further explained in the relative subsection.

4.3.6.1 Single activations

When an internal representation - for any given architecture, at any given layer - is reconstructed as input, it shows the information that the architecture stores about the real sample it was trained with. One question then is: Can one single hidden unit alone contain an useful piece of the whole information carried by the hidden layer?

An easy experiment can help answering this question. A number of handmade internal representations equal to the number of the hidden units for that hidden layer is created. Then, a very high activation (~ 1000) is added only to one hidden unit per internal representation, and the remaining values are all unactivated (0).

4.3.6.2 Reconstruction of digits

A quick reconstruction of the first-10-digits samples in the test set can visually show the validity of the internal representations created by each architecture. If the reconstruction fails it is a clue that the architecture cannot decode the internal representation anymore.

4.3.6.3 Reconstruction of the mean-digits

An interesting way to test the generalizations made by the internal representations in a high layer is to try to reconstruct something that is not a valid internal representation, i.e. anything that has not been produced by the architecture training, but contains some information derived from other valid representations. What immediately appears to fit perfectly for this purpose are the mean-digits, previously created for the histograms, as the mean of the internal representations (the mean activations) of all the correctly classified test samples per digit are used as input for reconstruction.

4.3.6.4 Filtering the mean-digits

A problem tied to the use of the mean-digits as internal representations is that they are composed of real values in range $[0, 1]$, as expected, while an internal representation - from all the given architectures - tends to discriminate more between high (1) and low (0) activations. A solution consists of filtering the mean-digits in order to obtain clearer high or low activations. For this purpose it's useful to create the **variance-digits** using the same procedure adopted for the mean, and then set a criterion using both: a high activation in a filtered mean-digit is set for each value that is α times greater than that mean-digit's mean value, and its corresponding variance-digit value is β times less than that variance-digit's mean value. α and β are parameters chosen with a trial and error approach, in order to get a number of high activations close to the average number of high activations of a typical internal representation (all the digits have a comparable number of high activations in the histograms).

4.3.7 Feeding internal representations to other architectures

Lastly, a way to check the similarity of an architecture's internal representations with another's is to feed each internal representation to the other architecture and let it reconstruct an input digit. Other than visually, the quality of the reconstructions can be attested with the MSE calculated on the original input and the reconstructed input from an internal representation. This is not always possible, in fact, the mean-digits and the filtered mean-digits have been produced in the third layer and have no original input. In this case the first-10-digits original input can be used to sample the quality of the reconstructions.

4.3.7.1 Inverse F-Mapping

The F-Mapped internal representations described before can also be fed to any architectures, once a reverse F-mapping is provided. In particular, if it's true that, given two internal representations $\mathbf{v^i}$ and $\mathbf{v^j}$ for two different architectures i and j:

$$\bar{\mathbf{v}^{\mathbf{i}}} = f_M^i\left(\mathbf{v}^{\mathbf{i}}\right) \approx \bar{\mathbf{v}^{\mathbf{j}}} = f_M^j\left(\mathbf{v}^{\mathbf{j}}\right),$$

then it is also possible to use the inverse F-Mappings f_M^{-1} and feed the following internal representations to the architecture i:

$$\mathbf{v}^{\mathbf{i}} \text{ and } \bar{\bar{\mathbf{v}^{\mathbf{i}}}} = f_M^{i,-1}\left(\bar{\mathbf{v}^{\mathbf{i}}}\right) \approx f_M^{i,-1}\left(\bar{\mathbf{v}^{\mathbf{j}}}\right) = f_M^{i,-1}\left(\mathbf{f_M^{j,-1}}\left(\mathbf{v}^{\mathbf{j}}\right)\right).$$

It is then possible to reconstruct the F-Mapped internal representations of the first-10-digits, the mean-digits and the filtered mean-digits, in order to evaluate if there is a gain in the reconstruction. Again, a comparison with MSE is the best way to attests results.

Chapter 5

Results

5.1 Training

The three architectures - Deep Belief Networks (DBNs), Stacked Autoencoders (SAs) and Stacked denoising Autoencoders (SdAs) - have been trained, validated and tested with three different percentages of samples in the same dataset: 10% for MNIST-10 (see figure 5.1.1), 50% MNIST-50 (see figure 5.1.2) and 100% for MNIST-100 (see figure 5.1.3). The MNIST-100 dataset is composed by a total of 50'000 training samples, 10'000 validation samples and 10'000 test samples. The training is successfully stopped when the validation error touches its minimum, so the best model is chosen before the maximum epoch. For training methods and times refer to table 4.2 in subsection 4.2.3. Note that for all the plots, one epoch corresponds to 500 iterations on MNIST-10, 2'500 iterations on MNIST-50 and 5'000 iterations on MNIST-100, given the different sizes of the three datasets used and a mini-batch corresponding to 10 samples.

All the training curves show that the training error immediately becomes very low, while the validation and test errors keep oscillating around an acceptable value, which is minimized by the repeated training steps, until last epoch or early stopping. Overtrain may happen when the training error is so low w.r.t. the validation and test errors, but all the architectures show a good success rate on the test set (low error), and the validation error continue to decrease until the training is ended with no significant increments in the test error. The training is successful in all cases.

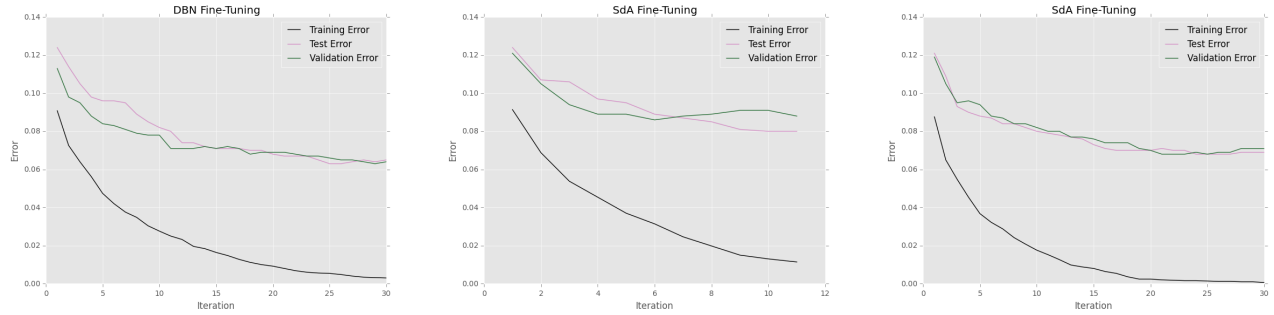

Figure 5.1.1: Training (black), validation (green) and test (pink), respectively, for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

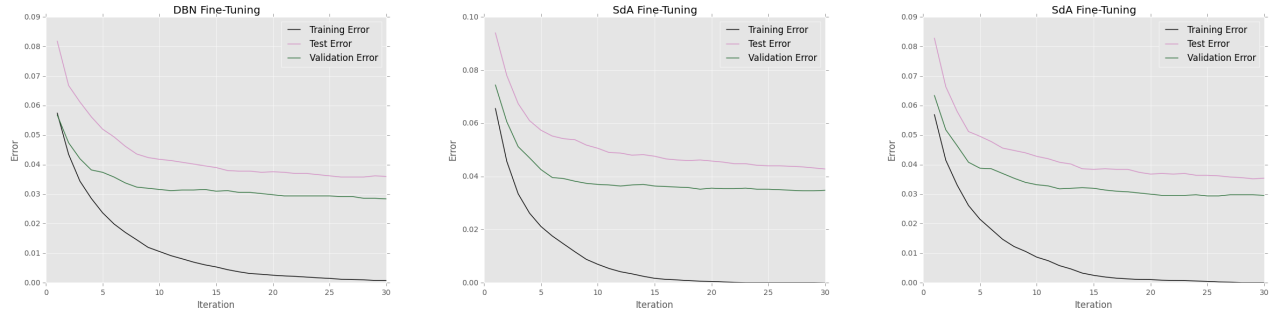

Figure 5.1.2: Training (black), validation (green) and test (pink), respectively, for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

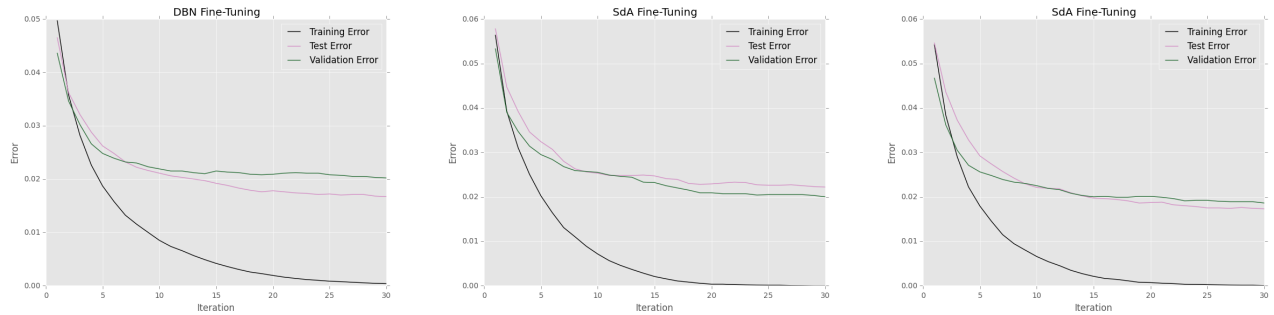

Figure 5.1.3: Training (black), validation (green) and test (pink), respectively, for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

5.2 Robustness to noise

In order to test the robustness of each architecture to noisy inputs, all the samples in the test sets have been corrupted with salt and pepper noise in three different levels: 10%, 20% and

50%. The architectures have been trained and validated, respectively, with the uncorrupted training and validation sets, then tested over the corrupted test sets. Results are showed in table 5.1.

For highly corrupted inputs and not much training data available, the stacked denoising autoencoder and the stacked autoencoder perform both better than the deep belief network. For highly corrupted inputs and more training data available, unintuitively - note that it happens just in this one case so cannot be significant - the stacked autoencoder performs better than the stacked denoising autoencoder, but both are still better than the deep belief network. Otherwise the DBN has the best performance, but does not show significant results w.r.t. the other two architectures.

Table 5.1: Classification errors of the three architectures trained on different percentages of the MNIST dataset and tested with three different levels of salt and pepper noise applied to the test sets.

Architecture	**Dataset**	**Classification Errors**			
		Original	**Noise 10%**	**Noise 20%**	**Noise 50%**
DBN	MNIST-10	64/1000	76/1000 (+119%)	92/1000 (+144%)	271/1000 (+423%)
	MNIST-50	180/5000	198/5000 (+110%)	254/5000 (+141%)	1157/5000 (+643%)
	MNIST-100	167/10000	200/10000 (+120%)	261/10000 (+156%)	1422/10000 (+851%)
SA	MNIST-10	89/1000	103/1000 (+116%)	110/1000 (+124%)	269/1000 (+302%)
	MNIST-50	218/5000	254/5000 (+117%)	290/5000 (+133%)	881/5000 (+404%)
	MNIST-100	222/10000	263/10000 (+118%)	326/10000 (+147%)	1205/10000 (+543%)
SdA	MNIST-10	71/1000	78/1000 (+110%)	83/1000 (+117%)	225/1000 (+317%)
	MNIST-50	182/5000	207/5000 (+114%)	236/5000 (+130%)	802/5000 (+441%)
	MNIST-100	173/10000	215/10000 (+124%)	298/10000 (+172%)	1288/10000 (+745%)

5.3 Filters

A filter is a 28x28 image that contains the corresponding values of a single hidden unit per layer. There are 900 filters for the first layer, 400 for the second and 100 for the third. All the results are reported for a maximum of 100 filters per layer, where the exceeding ones are randomly pruned.

It is possible to notice that the overall quality of a filter increases when more training samples are given to the architectures (see figures from 5.3.1 to 5.3.9). At a quick glance it is possible to note that the SdA and DBN filters are quite similar. Also the DBN seems to include more of a digit-shaped figures distinguishable from the background, while all filters have some clear white or black marks concentrated in a specific location of the filter. As expected the deeper layers show more localized filters, and the quality of details of the shapes in the filters increases with the quantity of training data used (from MNIST-10 to MNIST-100).

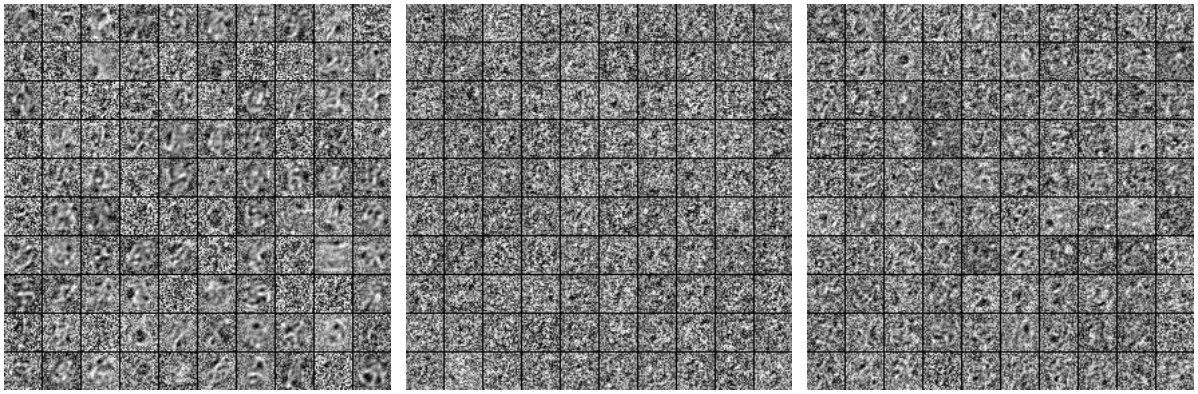

Figure 5.3.1: 100 random out of 900 first layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

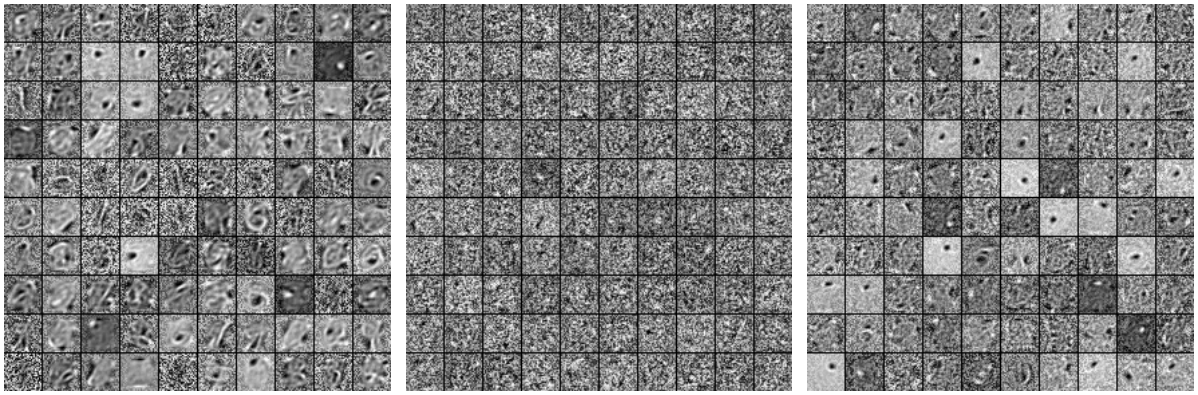

Figure 5.3.2: 100 random out of 900 first layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

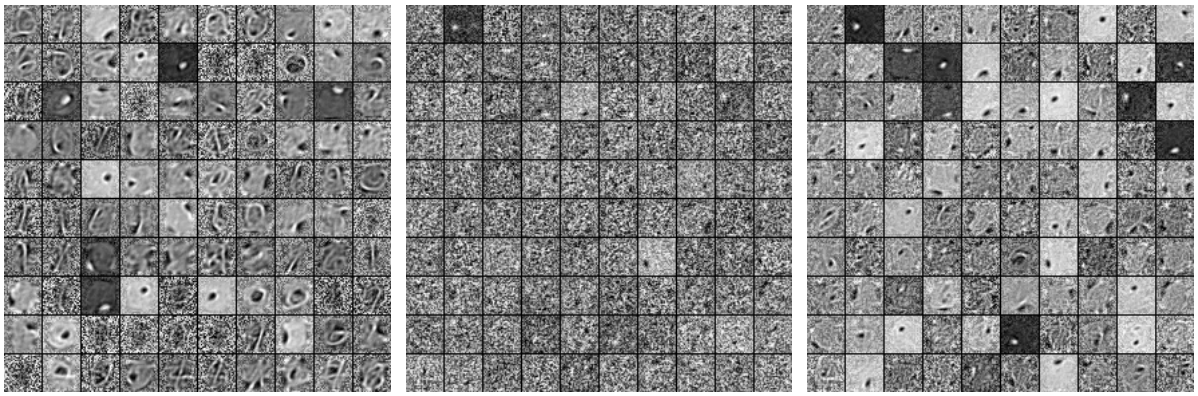

Figure 5.3.3: 100 random out of 900 first layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

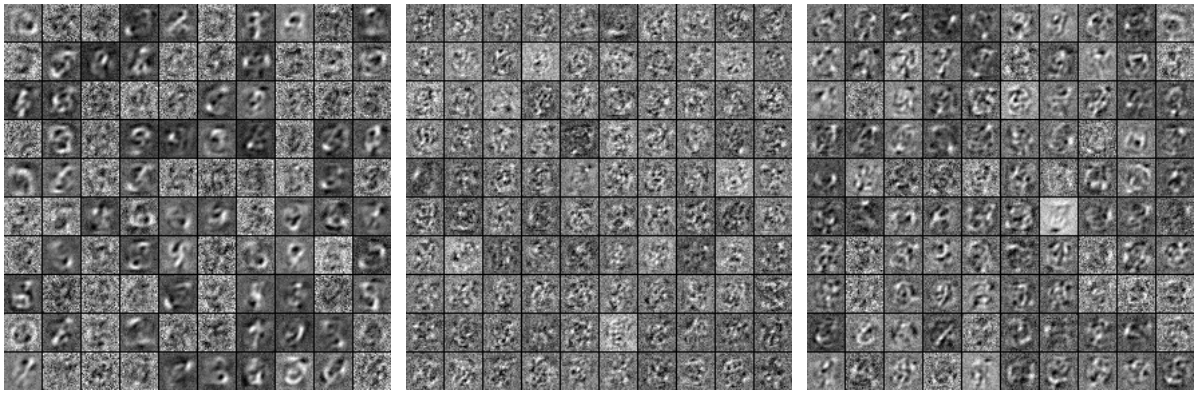

Figure 5.3.4: 100 random out of 400 second layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

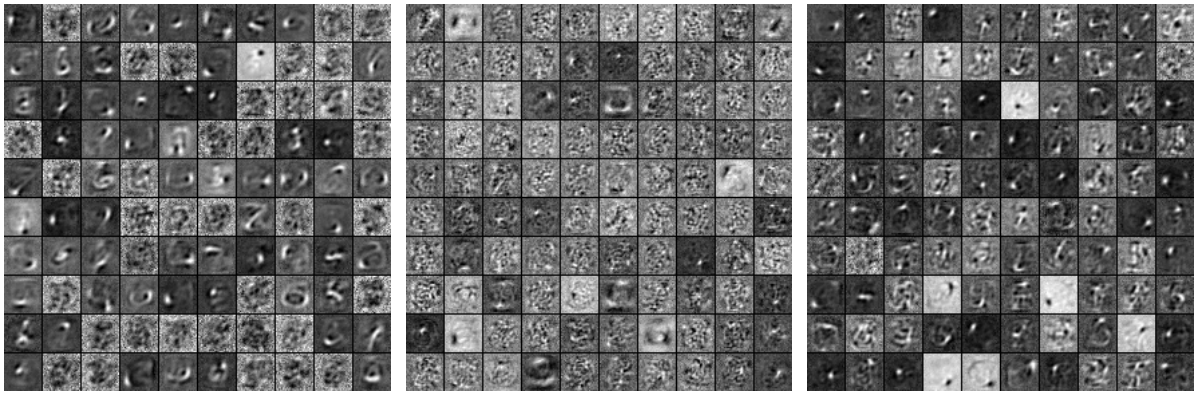

Figure 5.3.5: 100 random out of 400 second layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

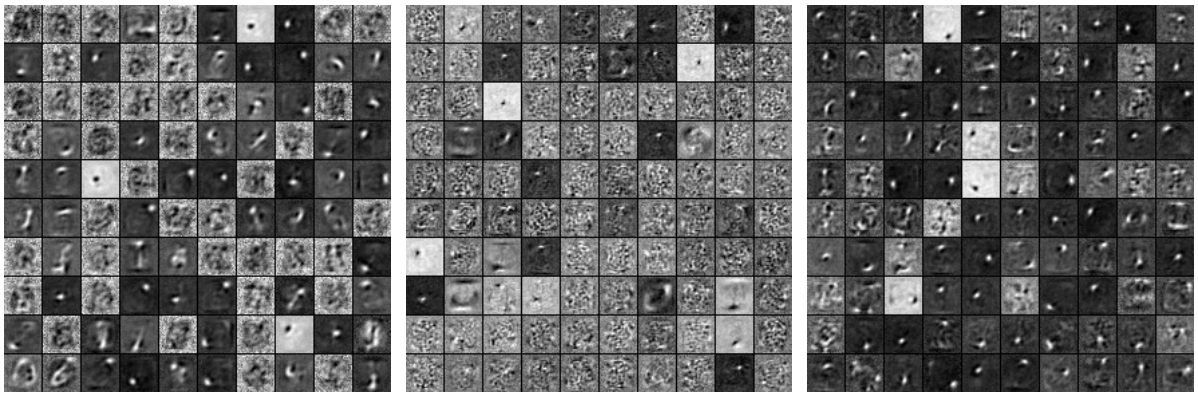

Figure 5.3.6: 100 random out of 400 second layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

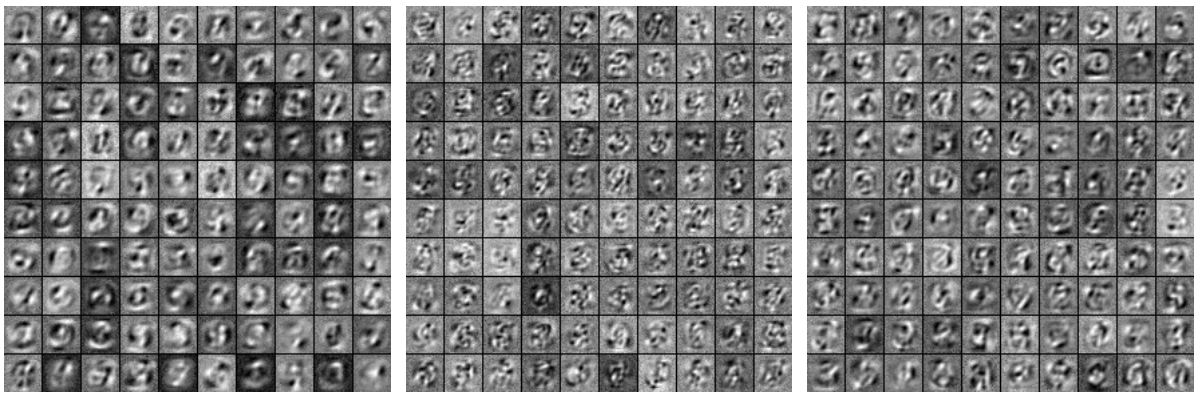

Figure 5.3.7: Third layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

Figure 5.3.8: Third layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

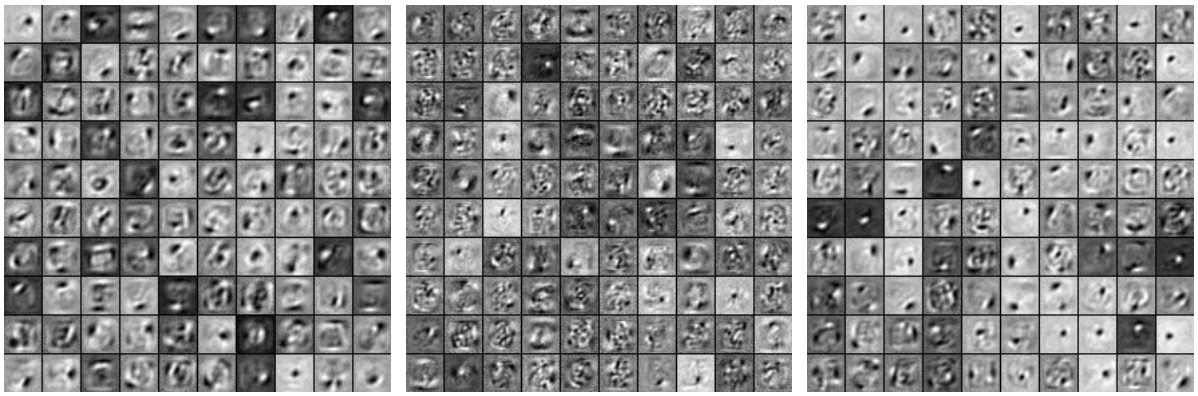

Figure 5.3.9: Third layer filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

5.4 Internal representations

After having looked at the filters, it can be expected to achieve better filtered internal representations when the available training samples are many. It is also expected, but difficult to attest, that the internal representations in the third layer are of greater interest for a classification task, i.e. easier to split in different and meaningful classes.

When the different architectures were compared using the filters, the stacked denoising autoencoder showed a better understanding of the input data with respect to its basic counterpart, the stacked autoencoder, while the DBN had still a quite similar encoding to the SdA.

On the other hand, an internal representation, being a sequence of activations, can not be shaped as interesting images and this is well understandable if one takes a look at the ones obtained for the first 10 digits and presented in figure 5.4.1. So it is difficult to attest the quality of the internal representations by eye inspection of the results. Histograms and Gini coefficients may help to assess various properties of the retrieved internal representations.

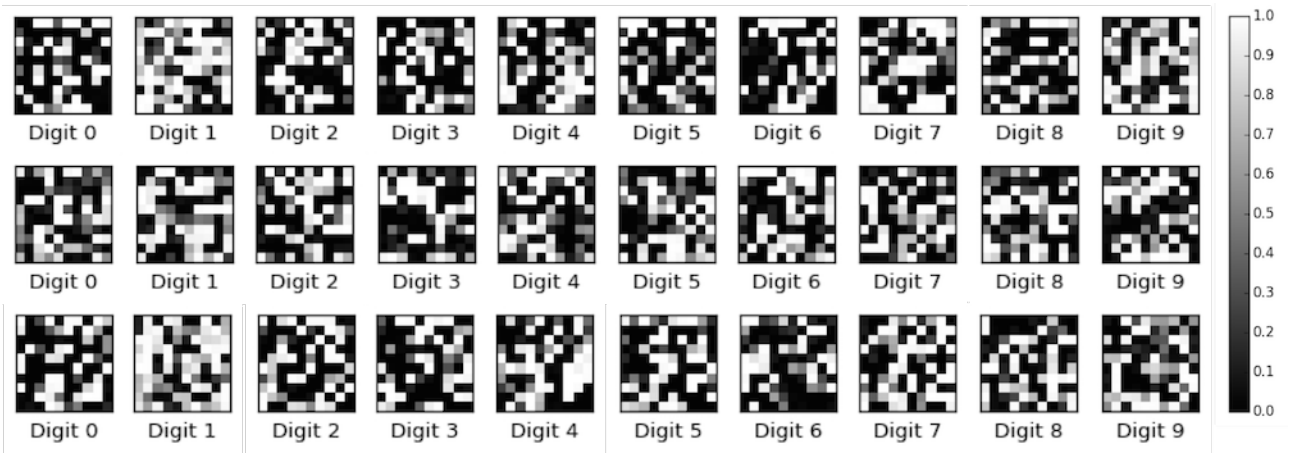

Figure 5.4.1: Internal representations in the third layer of the first 10 digits of the MNIST-100 test set, respectively for a DBN (top), a SA (middle) and a SdA (bottom). These internal representations are obtained by feeding the original digits to the network.

5.4.1 Histograms

The three architectures are able to classify a test set composed of 10 different digits (MNIST) with just a few errors. If only the well classified data is selected, using the labels provided with each input, what remains are 10 sets of correctly classified digits.

For each of those digits a third layer internal representation is created by the architecture in analysis. So, a first approach to look numerically at the internal representations is to create a histogram of all the activations per digit of the third layer, with real values in range $[0, 1]$ split in 20 bins, as shown in figures 5.4.2, 5.4.3 and 5.4.4.

The histograms clearly show that most of the values are high (1) or low (0) activations, independently from the training data available: MNIST-10, MNIST-50 and MNIST-100 have almost the same (scaled) number of activations per corresponding values. In order to extract further information from the previous histogram, it is possible to compute the mean of all the internal representations for each classified digit, from now on referred as **mean-digits**. This allows to visualize the sparseness of the high (1) and low (0) activations (see figures 5.4.5, 5.4.6 and 5.4.7).

The distribution tends to have an equal number of activations per value, this means that the high (1) and low (0) activations do not take place in the same hidden units of the third layer with different digit samples.

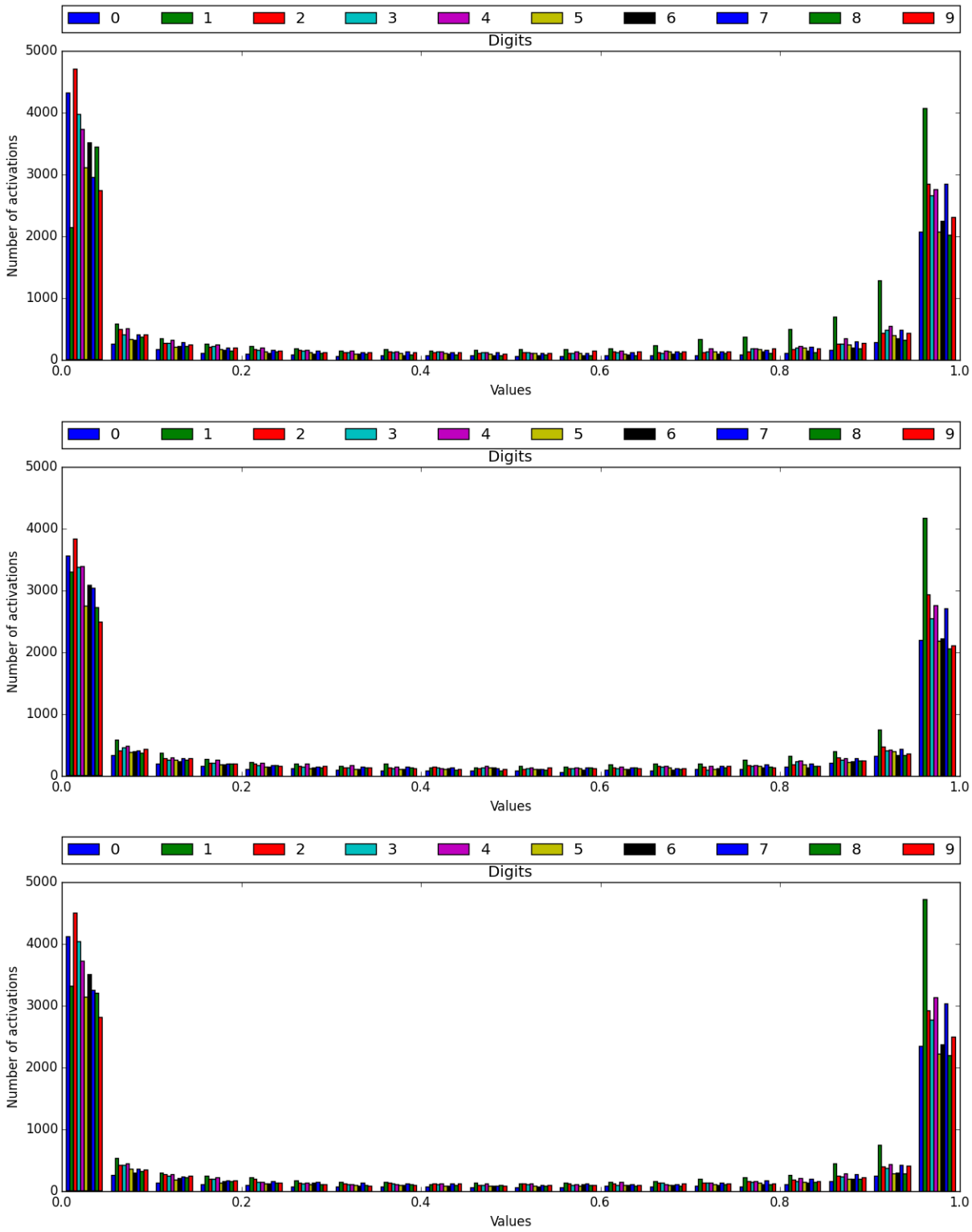

Figure 5.4.2: Histograms of third layer internal representations respectively for a DBN (top), SA (middle) and SdA (bottom), trained on MNIST-10. The y-axis shows the number of activations, the x-axis has 20 bins for values in range $[0, 1]$. Each color represents a different digit.

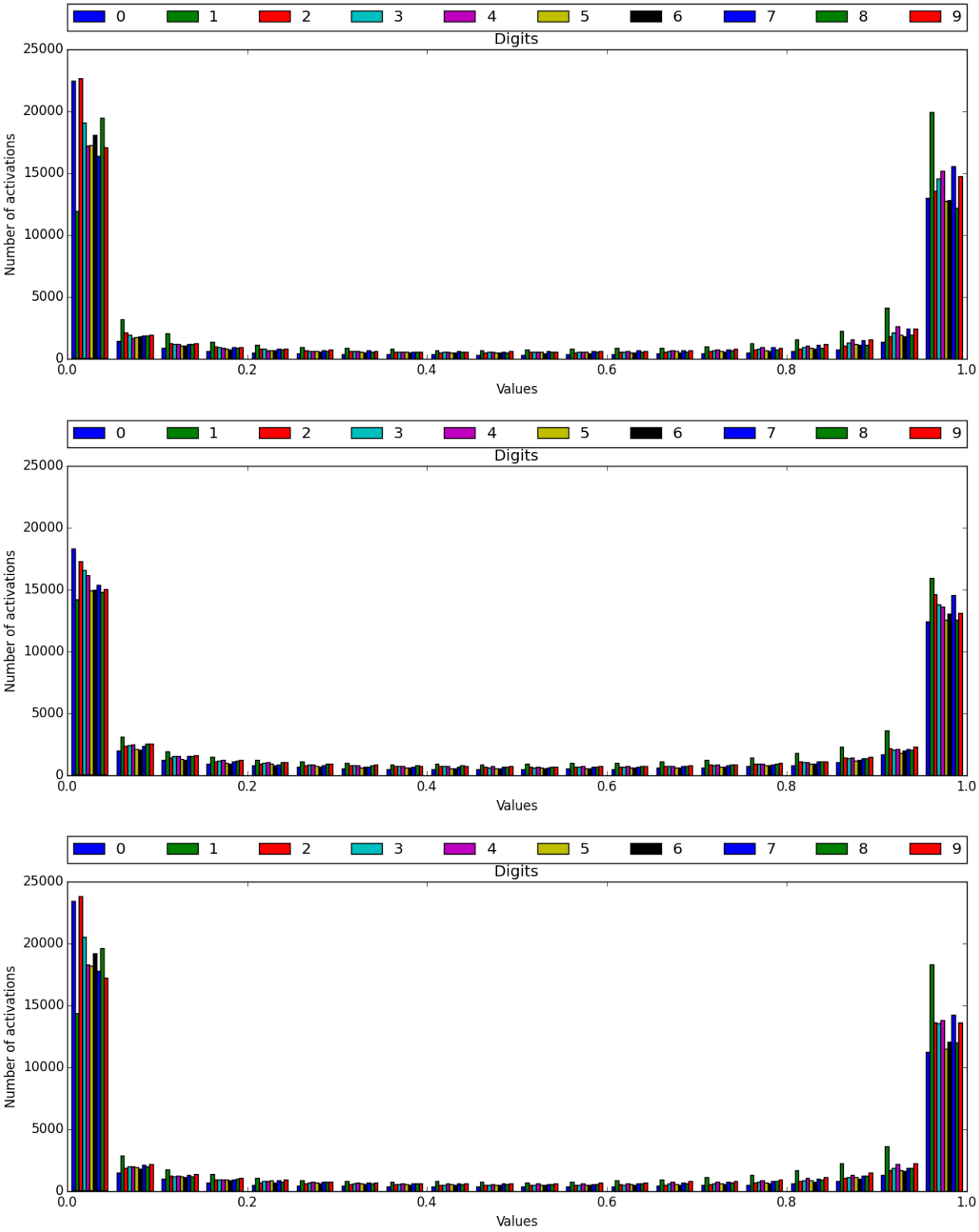

Figure 5.4.3: Histograms of third layer internal representations respectively for a DBN (top), SA (middle) and SdA (bottom), trained on MNIST-50. The y-axis shows the number of activations, the x-axis has 20 bins for values in range $[0, 1]$. Each color represents a different digit.

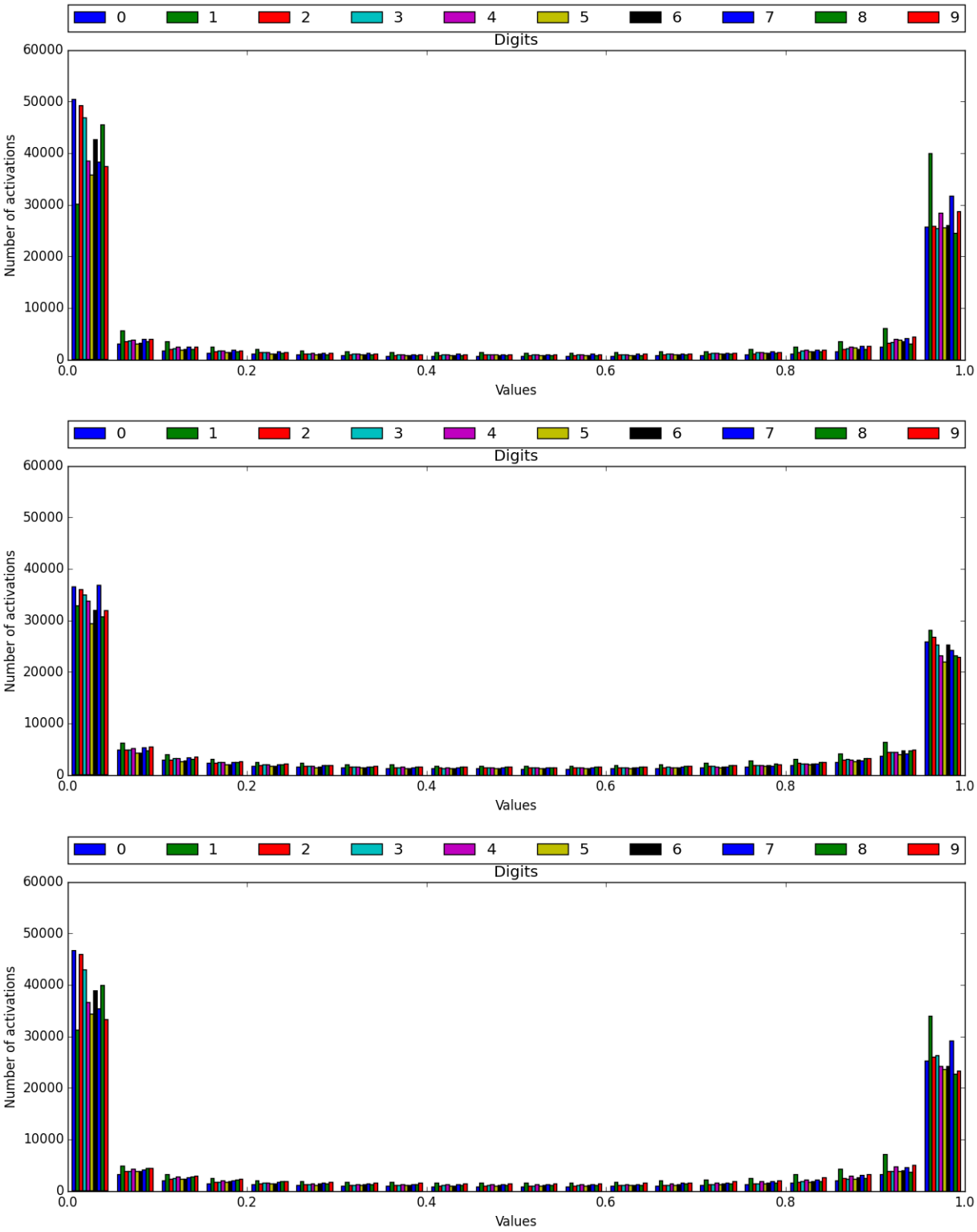

Figure 5.4.4: Histograms of third layer internal representations respectively for a DBN (top), SA (middle) and SdA (bottom), trained on MNIST-100. The y-axis shows the number of activations, the x-axis has 20 bins for values in range $[0, 1]$. Each color represents a different digit.

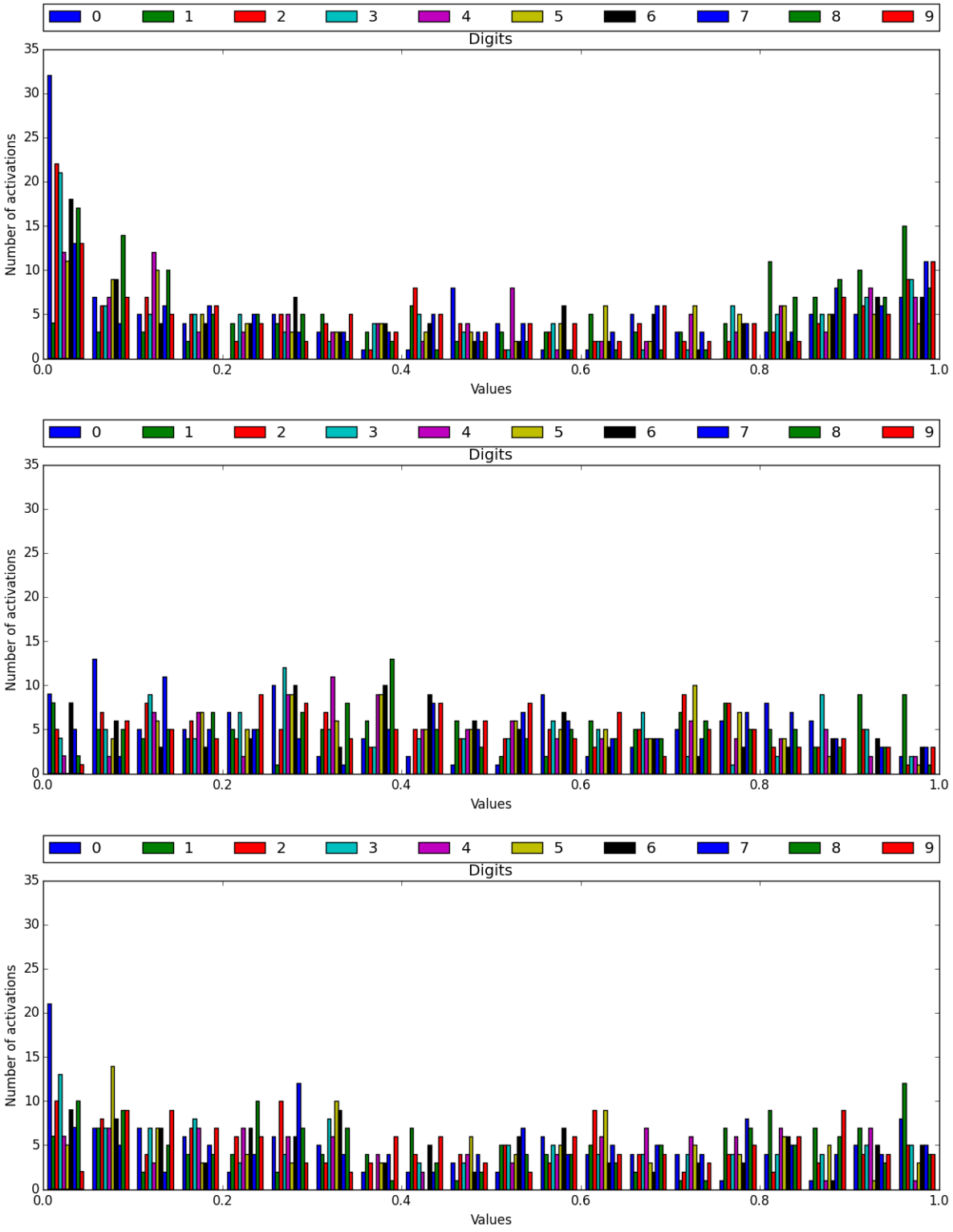

Figure 5.4.5: Histograms of third layer internal representations of the Mean-Digits respectively for a DBN (top), SA (middle) and SdA (bottom), trained on MNIST-10. The y-axis shows the number of activations, the x-axis has 20 bins for values in range $[0, 1]$. Each color represents a different digit.

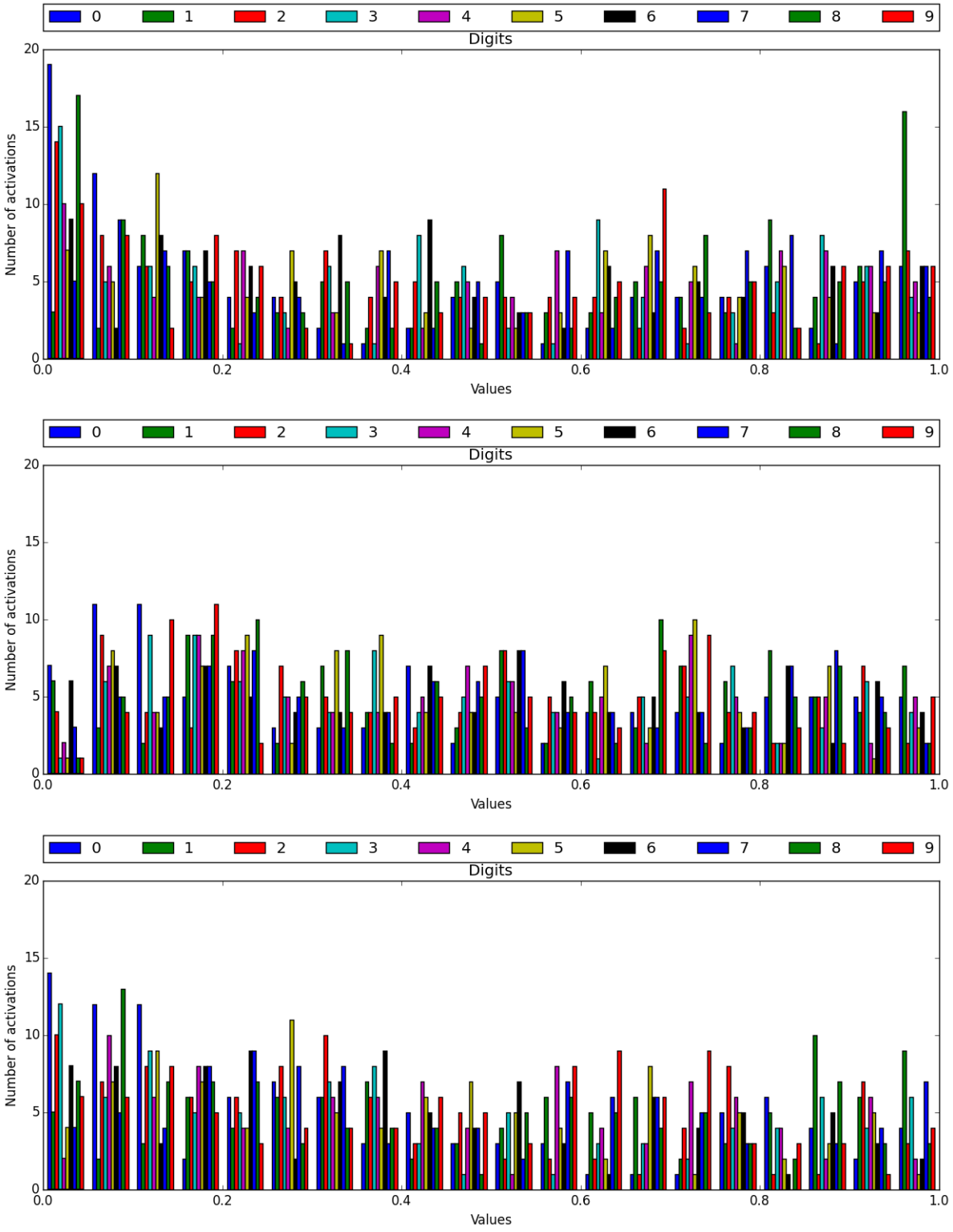

Figure 5.4.6: Histograms of third layer internal representations of the Mean-Digits respectively for a DBN (top), SA (middle) and SdA (bottom), trained on MNIST-50. The y-axis shows the number of activations, the x-axis has 20 bins for values in range $[0, 1]$. Each color represents a different digit.

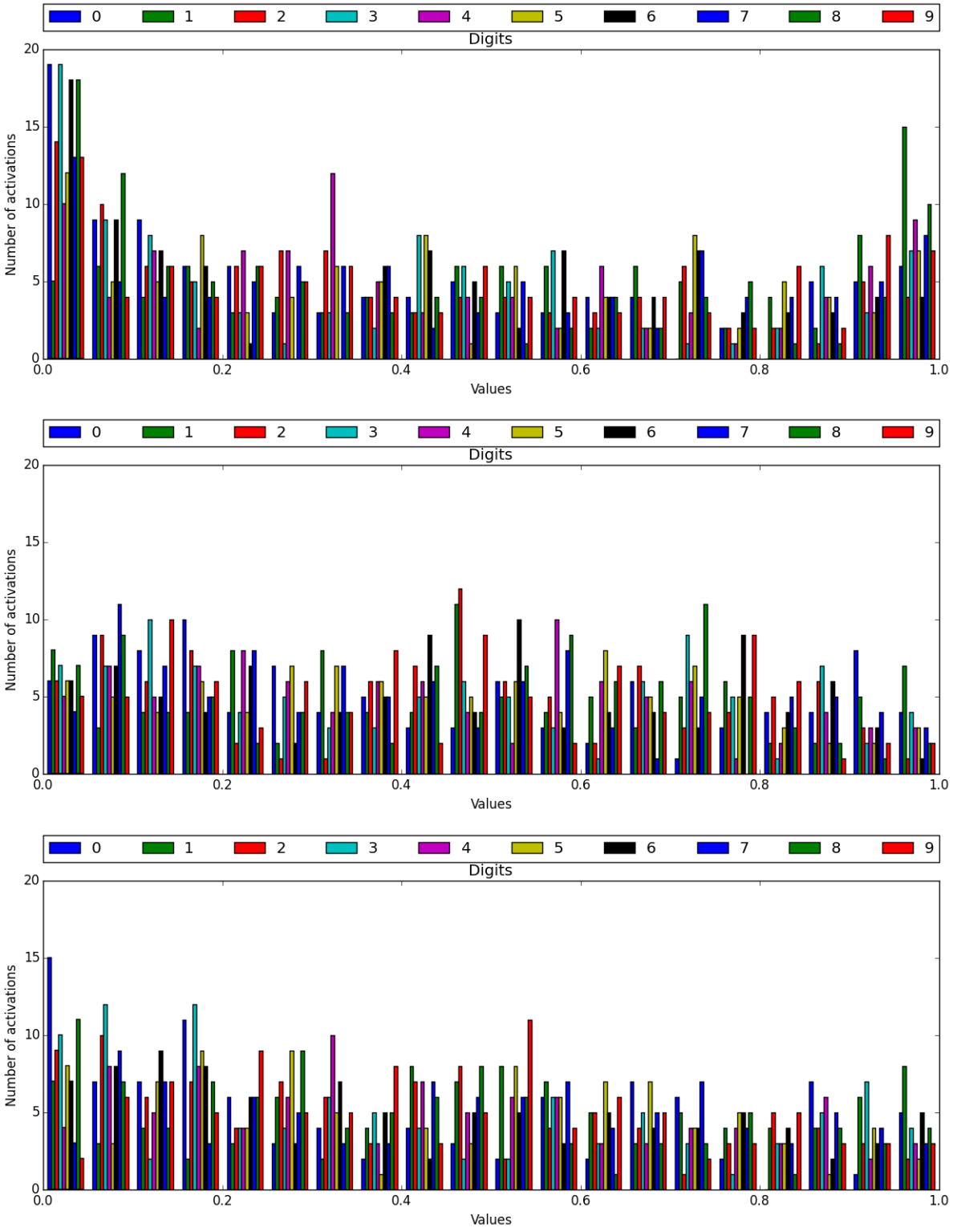

Figure 5.4.7: Histograms of third layer internal representations of the Mean-Digits respectively for a DBN (top), SA (middle) and SdA (bottom), trained on MNIST-100. The y-axis shows the number of activations, the x-axis has 20 bins for values in range $[0, 1]$. Each color represents a different digit.

5.4.2 Sparseness

The Gini coefficients are calculated for each digit's third layer internal representation, as a mean to quantify the sparseness of the high (1) and low (0) activations. For a more compact reading the mean and the variance of the Gini coefficients is provided per each digit (see table 5.2). A Gini value of 1 means that a vector is sparse, a Gini value of 0 means that the vector is non-sparse. Vectors with random values in the range $[0, 1]$ have a Gini coefficient that tends to $0.\bar{3}$.

All the digits appear to have a representation that is sparser than a random sequence, but not extremely sparse. This corresponds to the expected trade-off between density and sparseness of the internal representations that the three architectures should provide. It is worth to point out that all the architectures encode the information almost with the same sparsity.

Table 5.2: Gini coefficients. A hidden representation is sparse if its Gini coefficient is close to 1. Since a random sequence would have a Gini coefficient close to $0.\bar{3}$, the internal representations can be qualified as sparser than a random sequence.

Arch.	**Dataset**		**Digit**										**Mean**
			0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
DBN	MNIST-10	**Mean**	**0.6252**	**0.3424**	**0.5736**	**0.5432**	**0.5205**	**0.5318**	**0.5602**	**0.4806**	**0.5769**	**0.4971**	**0.5251**
		Var	0.0017	0.0020	0.0020	0.0029	0.0043	0.0035	0.0028	0.0031	0.0024	0.0025	0.0027
	MNIST-50	**Mean**	**0.5988**	**0.3941**	**0.5791**	**0.5263**	**0.4889**	**0.5295**	**0.5441**	**0.4795**	**0.5606**	**0.4959**	**0.5197**
		Var	0.0028	0.0028	0.0029	0.0026	0.0041	0.0038	0.0039	0.0032	0.0026	0.0028	0.0031
	MNIST-100	**Mean**	**0.6239**	**0.4354**	**0.6061**	**0.5941**	**0.5362**	**0.5374**	**0.5739**	**0.5163**	**0.5991**	**0.5251**	**0.5547**
		Var	0.0022	0.0026	0.0028	0.0029	0.0032	0.0031	0.0030	0.0030	0.0021	0.0025	0.0027
SA	MNIST-10	**Mean**	**0.5689**	**0.4330**	**0.5217**	**0.5241**	**0.5102**	**0.5102**	**0.5365**	**0.4929**	**0.5210**	**0.5035**	**0.5122**
		Var	0.0009	0.0006	0.0009	0.0009	0.0015	0.0009	0.0009	0.0010	0.0009	0.0011	0.0010
	MNIST-50	**Mean**	**0.5529**	**0.4411**	**0.5055**	**0.5100**	**0.5057**	**0.5096**	**0.5018**	**0.4889**	**0.5025**	**0.4949**	**0.5013**
		Var	0.0015	0.0004	0.0007	0.0007	0.0017	0.0010	0.0008	0.0008	0.0008	0.0009	0.0009
	MNIST-100	**Mean**	**0.5419**	**0.4837**	**0.5226**	**0.5250**	**0.5309**	**0.5159**	**0.5058**	**0.5428**	**0.5059**	**0.5157**	**0.5190**
		Var	0.0007	0.0003	0.0005	0.0005	0.0006	0.0005	0.0007	0.0006	0.0006	0.0006	0.0006
SdA	MNIST-10	**Mean**	**0.6004**	**0.4145**	**0.5601**	**0.5528**	**0.5112**	**0.5446**	**0.5561**	**0.4897**	**0.5507**	**0.4987**	**0.5279**
		Var	0.0021	0.0009	0.0017	0.0026	0.0025	0.0021	0.0027	0.0019	0.0016	0.0015	0.0021
	MNIST-50	**Mean**	**0.6290**	**0.4265**	**0.5909**	**0.5584**	**0.5226**	**0.5610**	**0.5701**	**0.5208**	**0.5623**	**0.5124**	**0.5454**
		Var	0.0027	0.0018	0.0017	0.0027	0.0050	0.0033	0.0033	0.0026	0.0025	0.0037	0.0030
	MNIST-100	**Mean**	**0.5979**	**0.4456**	**0.5836**	**0.5671**	**0.5357**	**0.5406**	**0.5555**	**0.5040**	**0.5728**	**0.5152**	**0.5418**
		Var	0.0019	0.0030	0.0014	0.0018	0.0024	0.0026	0.0021	0.0023	0.0018	0.0025	0.0022

5.5 Mapping of the internal representations (F-Mapping)

As described in section 4.3.4, sorting the filters $\mathbf{F^i}$ and $\mathbf{F^j}$ of the same layer of two different architectures i and j, one can obtain a mapping function f_M of their internal representations (F-Mapping). All the F-Mapped filters are analyzed in this section.

5.5.1 Comparison of sorted filters using MSE

Two sets of filters $\mathbf{F^i}$ and $\mathbf{F^j}$, corresponding to the same layer of architectures i and j, are first normalized in range $[-2, 2]$. For each row, namely one filter, the mean of the first three values and last three values is used as background value. This allows to center the background at value zero (see figure 5.5.1).

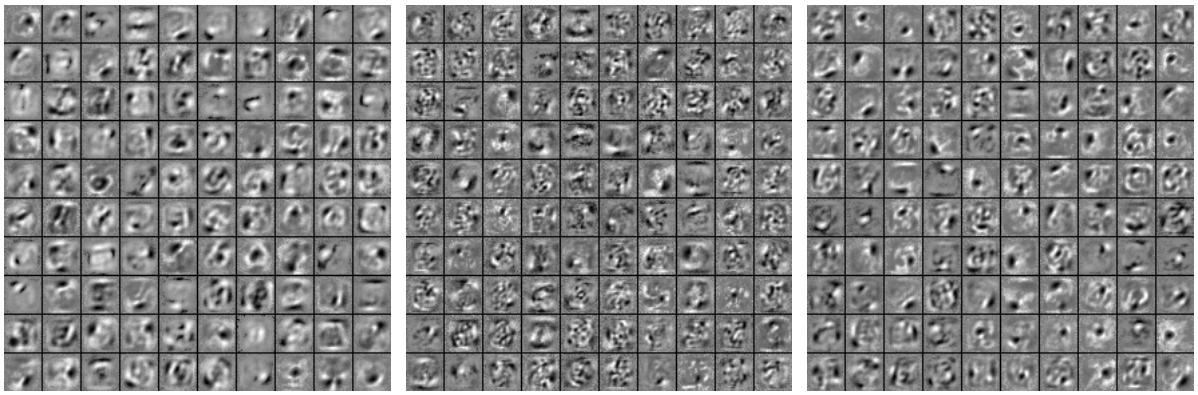

Figure 5.5.1: Third layer normalized filters respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

Successively, the two filters are F-Mapped, i.e. sorted with the algorithm described in section 4.3.4, in order to minimize the Mean Squared Error (see figure 5.5.2). The MSE gives more importance to the values in ranges $[-2, -1]$ and $[1, 2]$, that are the filters' white and black marks, leaving what is closer to the background, range $[-1, 1]$, less relevant. Also, positive and negative values are treated equally. At the end of this process an F-Mapping function f_M for the two sets of filters is obtained, the original filter sets can be shown coupled together using their F-Mappings (see figures 5.5.3, 5.5.4 and 5.5.5).

As expected, the numerical results show that the total sum of Mean Squared Errors for two sets of filters in the third layer is reduced by the sorting algorithm (see tab 5.3). In particular, the filters produced with more training samples and produced by the SdA w.r.t. the SA are more accurately compared one another.

Table 5.3: Sum of Mean Squared Errors applied to couples of architectures' third layer filters, before and after the F-Mapping.

Compared Architectures	**Dataset**	**Sum of Mean Squared Errors**	
		Before F-Mapping	**After F-Mapping**
DBN and SA	MNIST-10	68.2926	50.6575
	MNIST-50	64.5213	46.7702
	MNIST-100	60.9511	43.0220
DBN and SdA	MNIST-10	70.9706	43.9734
	MNIST-50	61.5818	38.7536
	MNIST-100	56.5720	33.6455

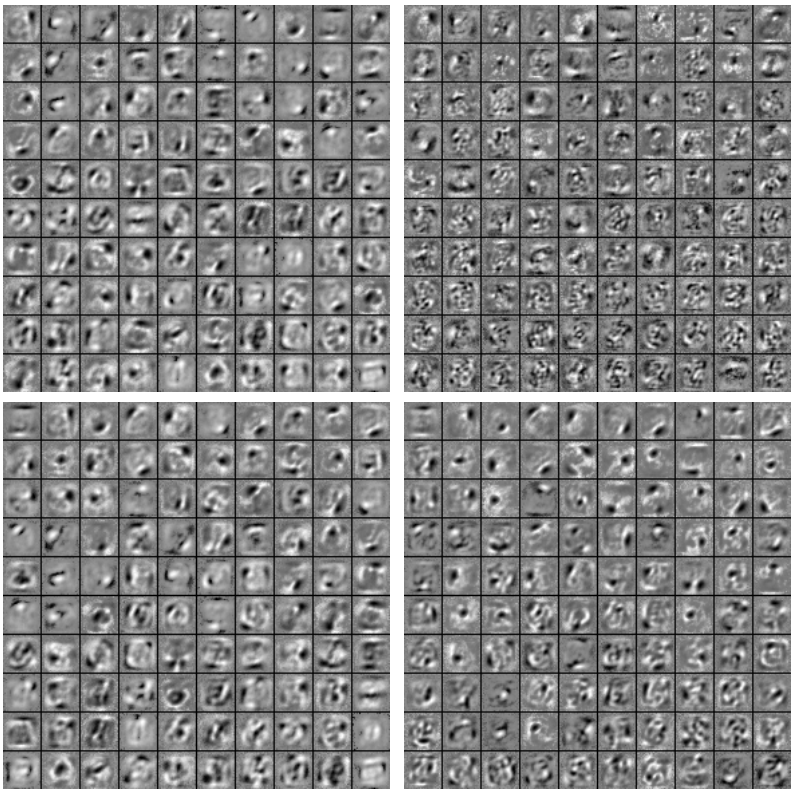

Figure 5.5.2: Third layer normalized and sorted filters. The top row contains respectively a DBN filter set (left) sorted to match a SA filter set (right). The bottom row contains respectively the same DBN filter set (left) sorted to match a SdA filter set (right). All architectures are trained on MNIST-100.

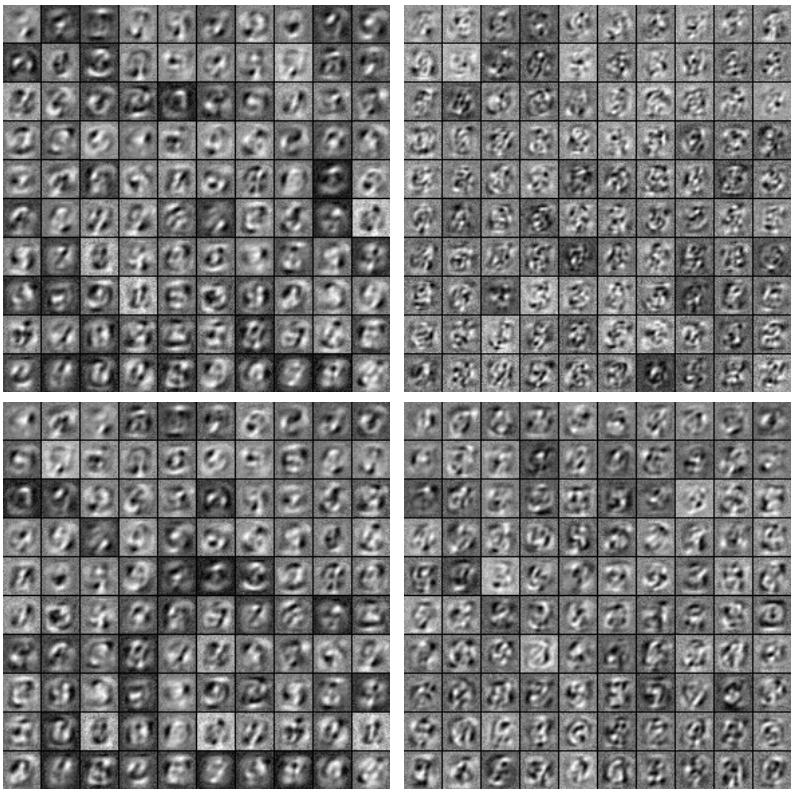

Figure 5.5.3: Third layer original sorted filters. The top row contains respectively a DBN filter set (left) sorted to match a SA filter set (right). The bottom row contains respectively the same DBN filter set (left) sorted to match a SdA filter set (right). All architectures are trained on MNIST-10.

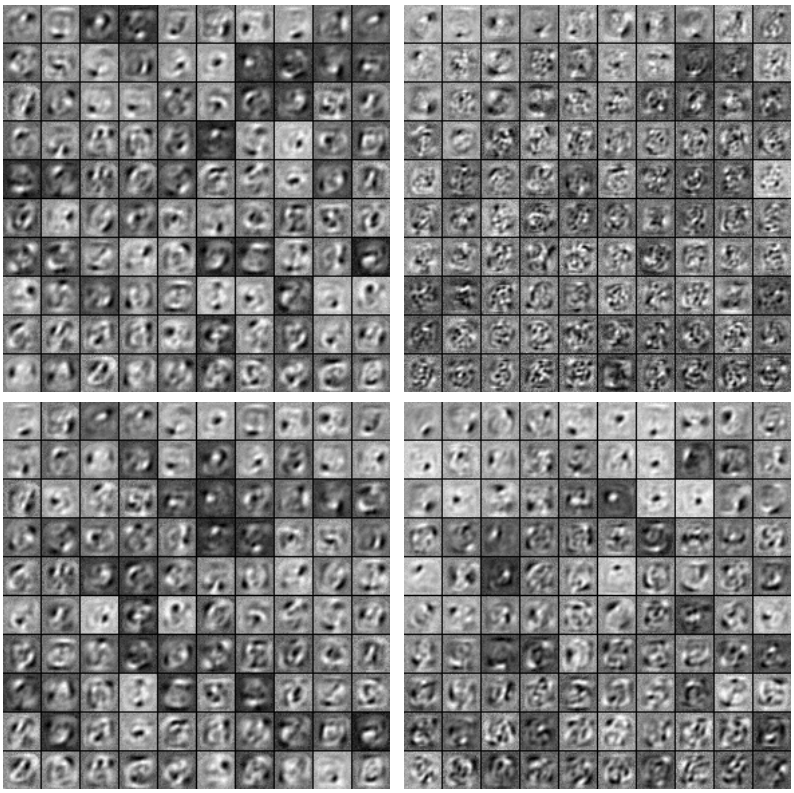

Figure 5.5.4: Third layer original sorted filters. The top row contains respectively a DBN filter set (left) sorted to match a SA filter set (right). The bottom row contains respectively the same DBN filter set (left) sorted to match a SdA filter set (right). All architectures are trained on MNIST-50.

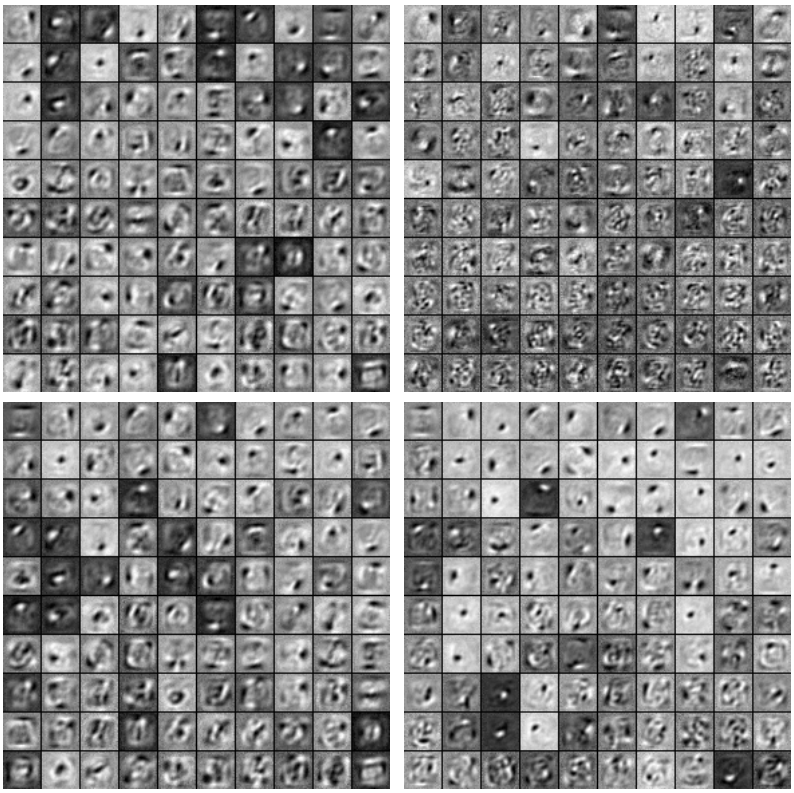

Figure 5.5.5: Third layer original sorted filters. The top row contains respectively a DBN filter set (left) sorted to match a SA filter set (right). The bottom row contains respectively the same DBN filter set (left) sorted to match a SdA filter set (right). All architectures are trained on MNIST-100.

5.5.2 Comparison of sorted internal representations using MSE

Once an F-Mapping f_M is defined for each architecture, the internal representations of different architectures are compared to each other with MSE (see table 5.4). The reduction of MSE after the F-Mapping suggests that the procedure can make a better comparison of two different internal representations more significant, by aligning the information corresponding to same features. A visual comparison is provided for MNIST-100 but it is still not very helpful (see figure 5.5.6).

Table 5.4: Sum of Mean Squared Errors applied to couples of third layer first-10-digits samples (one for each digit), before and after the F-Mapping.

Compared Architectures	**Dataset**	**Sum of Mean Squared Errors**	
		Before F-Mapping	**After F-Mapping**
DBN and SA	MNIST-10	3.4568	2.4632
	MNIST-50	3.8427	2.5062
	MNIST-100	3.3708	2.6395
DBN and SdA	MNIST-10	3.5851	2.4489
	MNIST-50	3.6783	2.4457
	MNIST-100	3.6795	2.4540

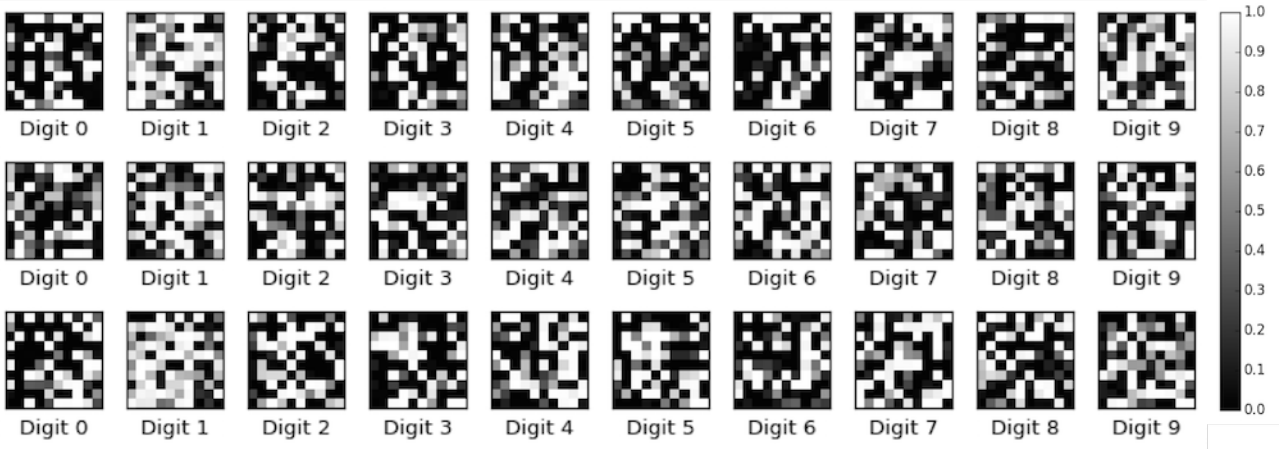

Figure 5.5.6: F-Mapped internal representations in the third layer of the first 10 digits of the MNIST-100 test set, respectively for a DBN (top), a SA (middle) and a SdA (bottom).

5.6 Input reconstruction

In this section, all the input reconstructions are presented and numerically compared using MSE.

5.6.1 Reconstruction of single activations

All the internal representations with the single activations (see Methods, subsection 4.3.6.1) are reconstructed as inputs (see figures from 5.6.1 to 5.6.9).

This reconstruction of the single activations didn't prove to be very useful to understand neither the information encoded in the internal representations nor the similarity of the architectures, since the images produced are difficult to read and quite confusing, both in the first layers and the deepest ones. In spite of this, the single activations are related to the filters and may eventually be used, in conjunction with the filters, to achieve a better understanding of the internal representations. One algorithm proposed in literature is the activation maximization (Erhan et al., 2010).

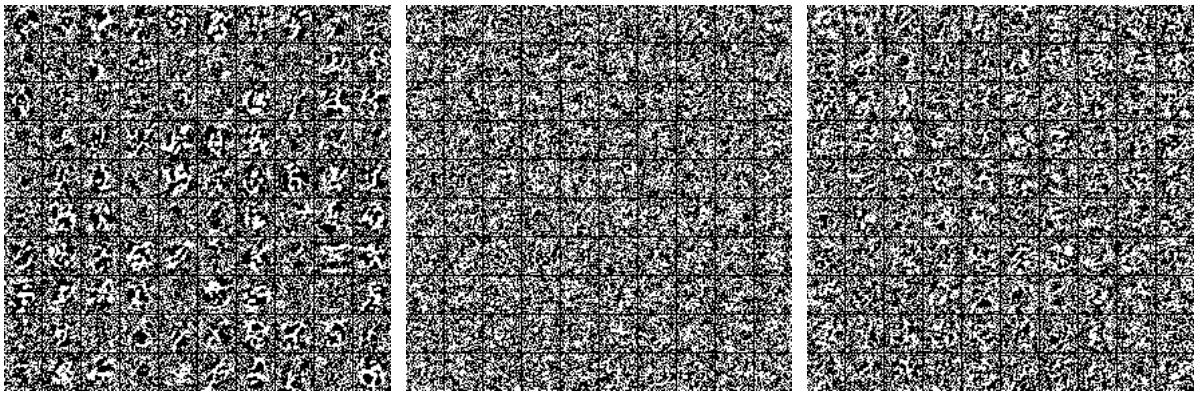

Figure 5.6.1: 100 random out of 900 input reconstructions of the single activations in a first layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

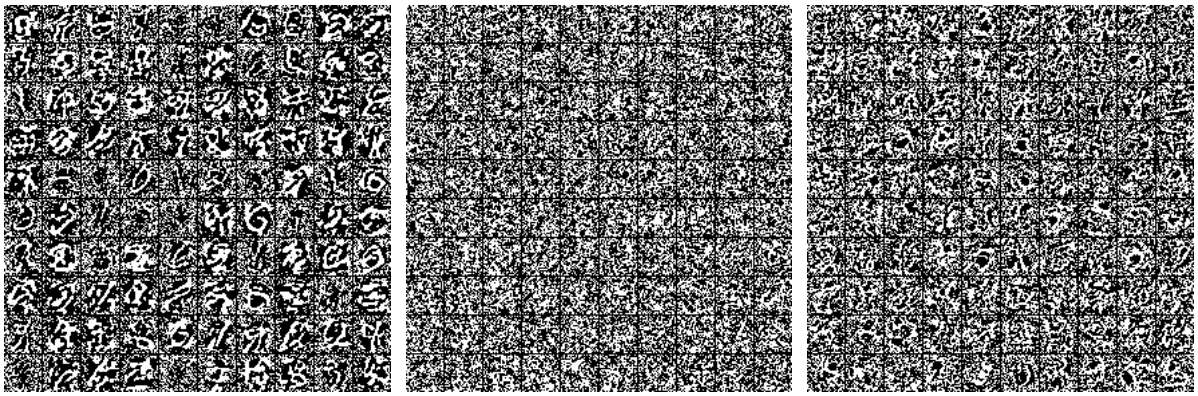

Figure 5.6.2: 100 random out of 900 input reconstructions of the single activations in a first layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

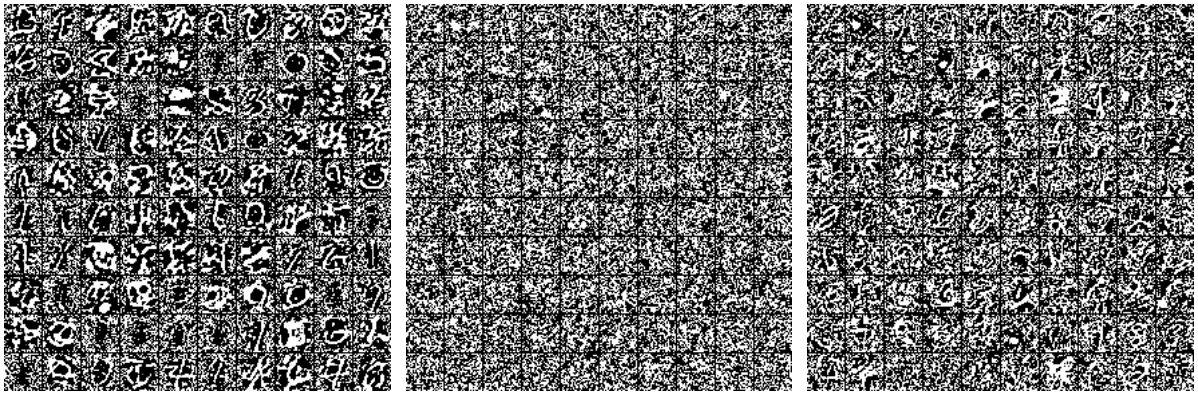

Figure 5.6.3: 100 random out of 900 input reconstructions of the single activations in a first layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

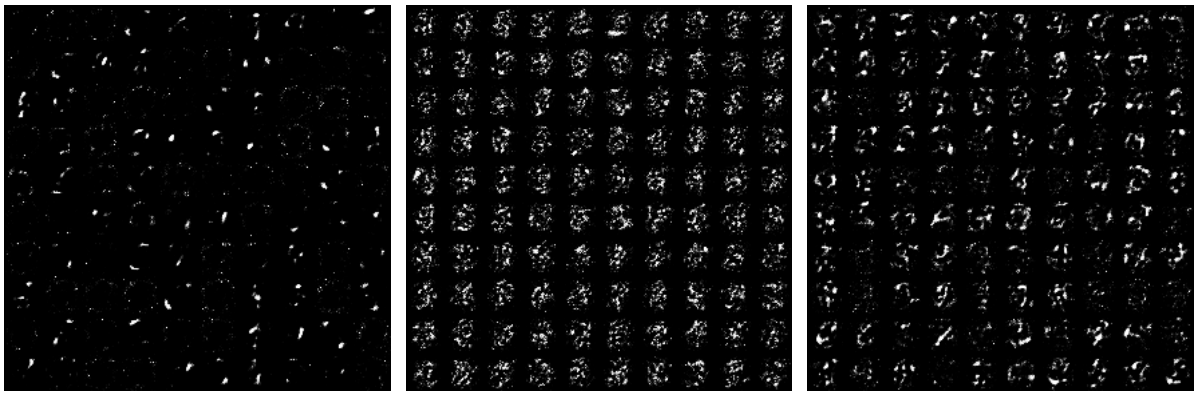

Figure 5.6.4: 100 random out of 400 input reconstructions of the single activations in a second layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

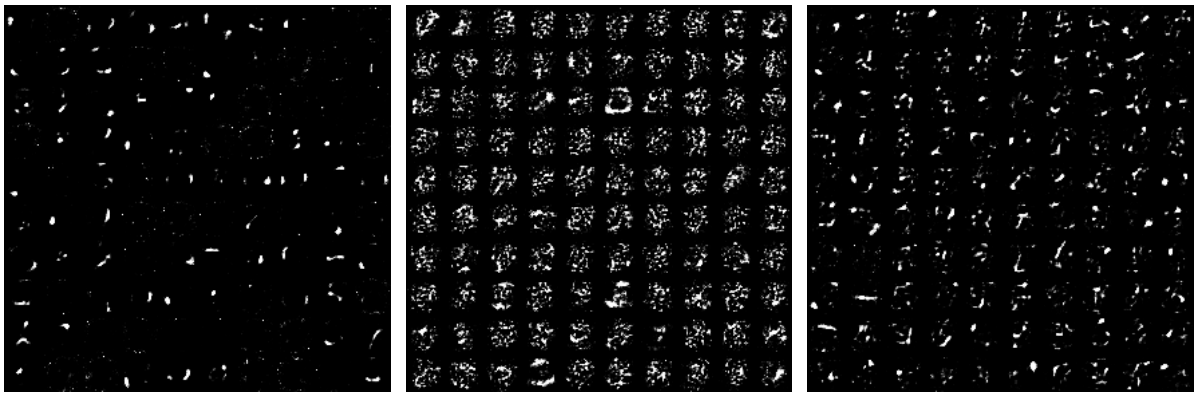

Figure 5.6.5: 100 random out of 400 input reconstructions of the single activations in a second layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

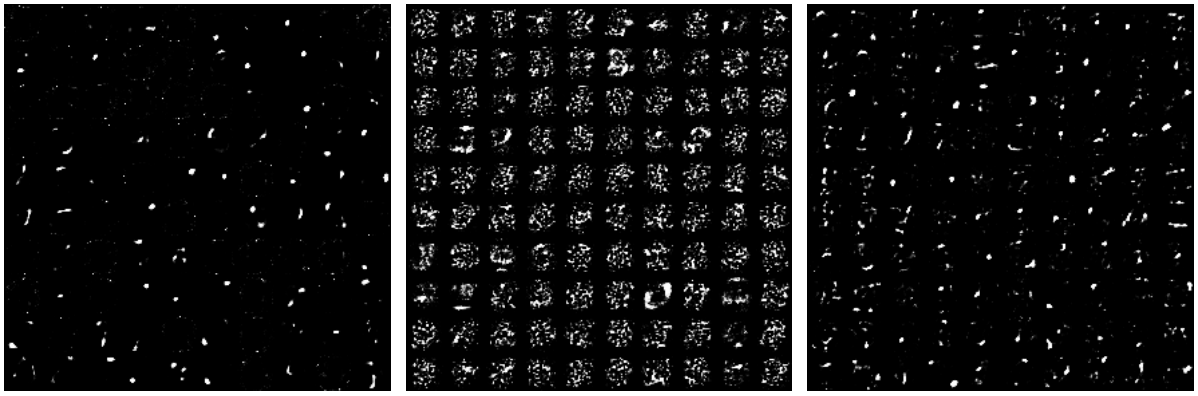

Figure 5.6.6: 100 random out of 400 input reconstructions of the single activations in a second layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

Figure 5.6.7: Input reconstructions of the single activations in a third layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-10.

Figure 5.6.8: Input reconstructions of the single activations in a third layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-50.

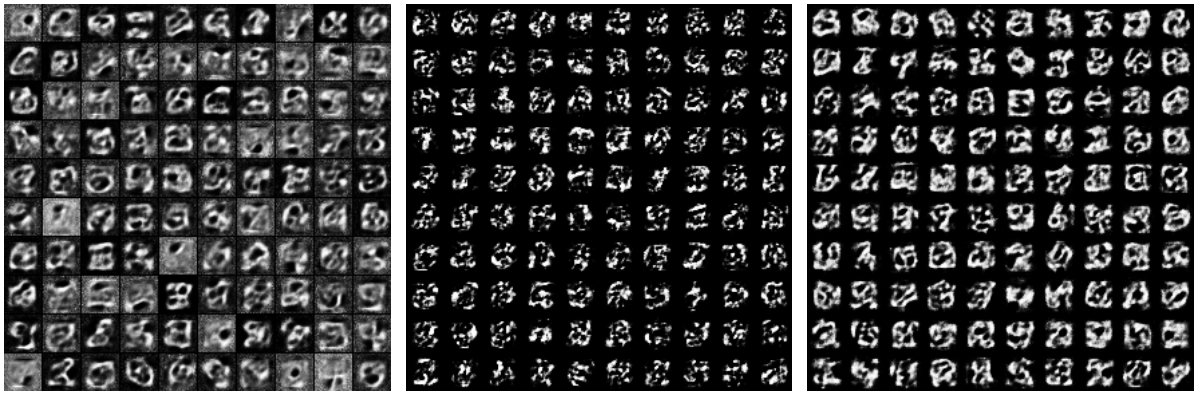

Figure 5.6.9: Input reconstructions of the single activations in a third layer respectively for a DBN (left), SA (center) and SdA (right), trained on MNIST-100.

5.6.2 Reconstruction of digits

A reconstruction of the first-10-digits samples used before is shown in figure 5.6.10. This confirms that all the structures are able to reconstruct their internal representations, but the SA obtains a worse reconstruction.

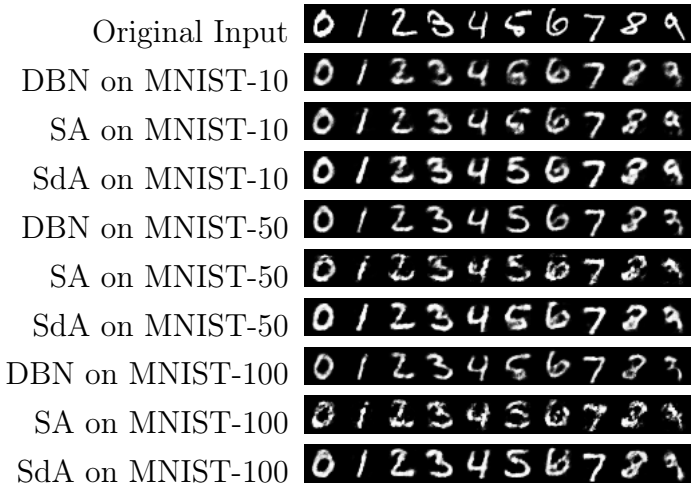

Figure 5.6.10: Reconstruction of the input, by each of the three architectures, starting from the third layer internal representations of the first-10-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

5.6.2.1 Reconstruction of the mean-digits

The mean-digits are internal representations artificially made in order to test the capacity of the architectures to generalize over the internal representations learnt. The reconstructions of the mean-digits are provided in figure 5.6.11.

Since the test is successful, it is possible to take into consideration hand-made modified versions of the internal representations originally produced by the architectures. The DBN appears less able than the SdA to reconstruct this kind of internal representations. The SA fails to reconstruct the input in a good shape.

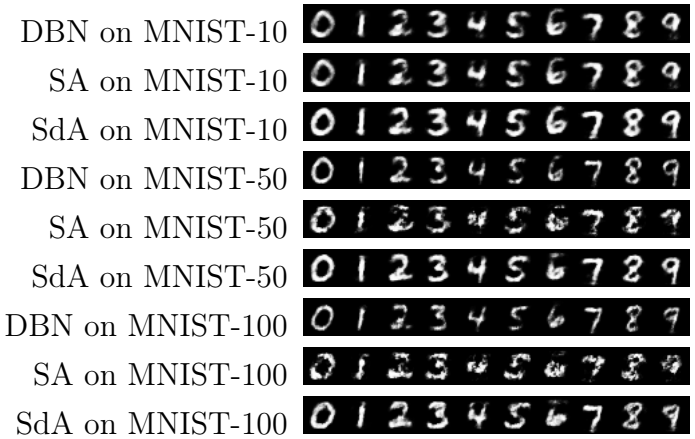

Figure 5.6.11: Reconstruction of the input, by each of the three architectures, starting from the third layer internal representations of the mean-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

5.6.2.2 Filtering the mean-digits

Since the histograms on the mean-digits show that the activations are less sparse when the mean is performed on the internal representations, it is interesting to filter the mean-digit internal representations in order to restore proper values for the high (1) and low (0) activations.

The reconstructions of the filtered mean-digits are provided in figure 5.6.12 and the images appear better than the reconstructions of the mean-digits. Only the SA fails to reconstruct the input in a good shape.

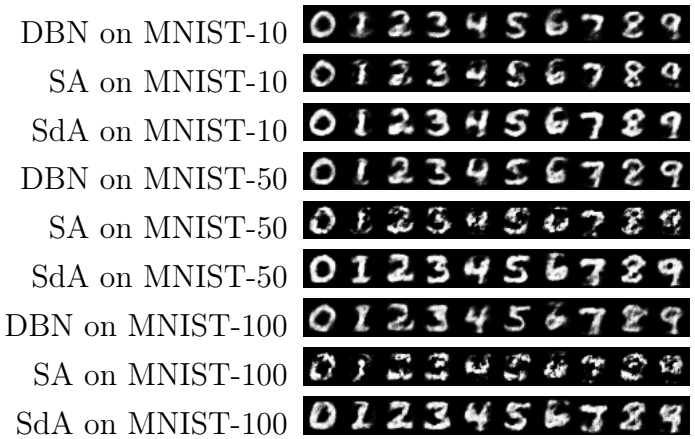

Figure 5.6.12: Reconstruction of the input, by each of the three architectures, starting from the third layer internal representations of the filtered mean-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

5.6.3 Feeding internal representations to other architectures

A first look at the reconstructions of the first-10-digits (see figure 5.6.13), the mean-digits (see figure 5.6.15) and the filtered mean-digits (see figure 5.6.15) gives an idea of the plasticity of the internal representations.

The images obtained are clearly not the expected input. This means that the raw internal representations are not good to be used by other architectures.

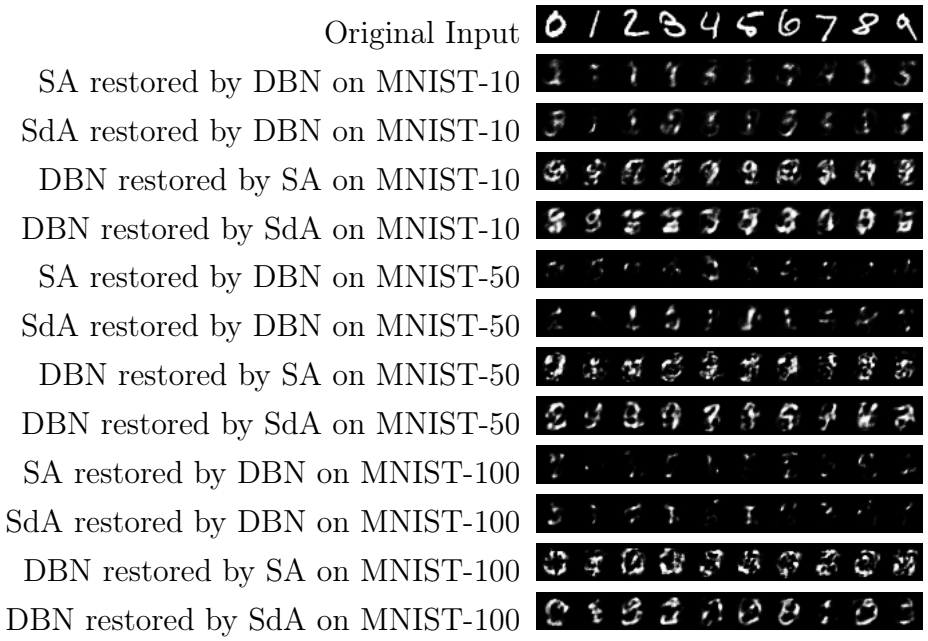

Figure 5.6.13: Reconstruction of the input of another architecture, by each of the three architectures, starting from the third layer internal representations of the first-10-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

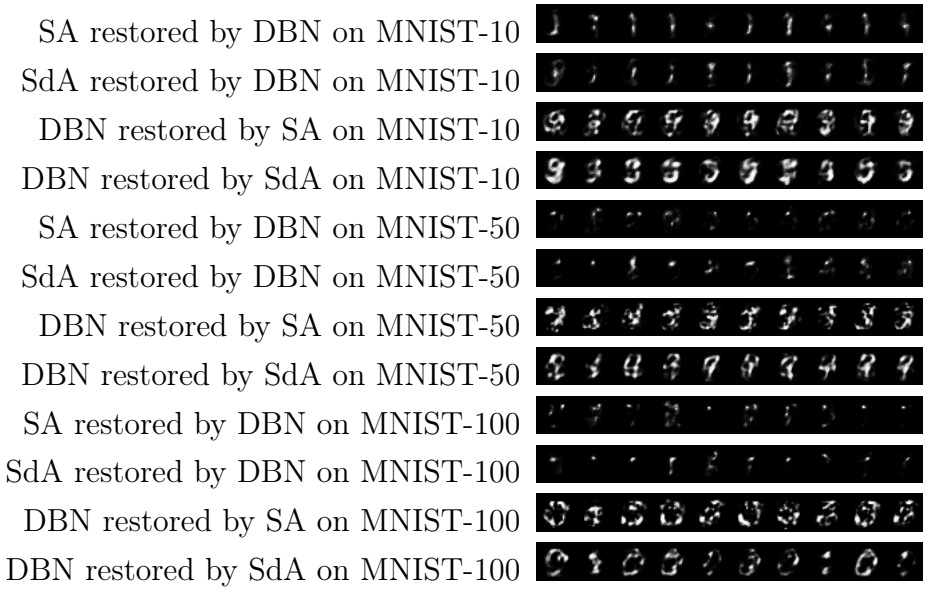

Figure 5.6.14: Reconstruction of the input of another architecture, by each of the three architectures, starting from the third layer internal representations of the mean-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

Figure 5.6.15: Reconstruction of the input of another architecture, by each of the three architectures, starting from the third layer internal representations of the filtered mean-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

5.6.3.1 Inverse F-Mapped representations

Using the inverse F-Mappings (see methods, subsection 4.3.7.1), it is now possible to reconstruct the F-Mapped internal representations of the first-10-digits (see figure 5.6.16), the mean-digits

(see figure 5.6.17) and the filtered mean-digits (see figure 5.6.18) in order to evaluate if there is a gain in the reconstruction.

The results are not always correct, but some input images are quite good reconstructions of the respective internal representations. In particular, while the first-10-digits are not reconstructed at all, the mean-digits work quite well for the SdA, and partially for the SA, but the DBN fails. Using the filtered mean-digits instead, the DBN is finally able to restore some of the hybrid internal representations.

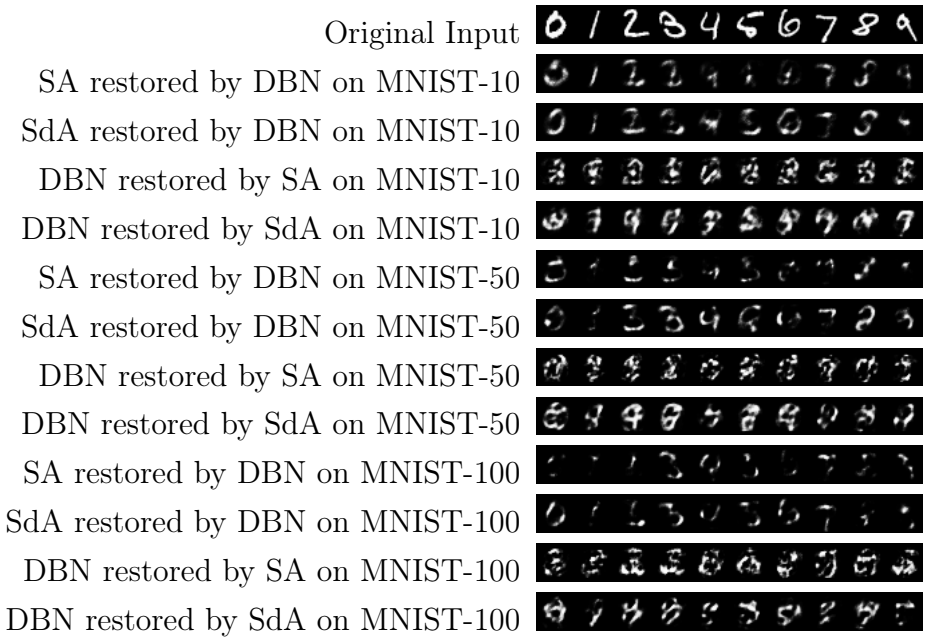

Figure 5.6.16: Reconstruction of the input of another architecture, by each of the three architectures, starting from the F-Mapped third layer internal representations of the first-10-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

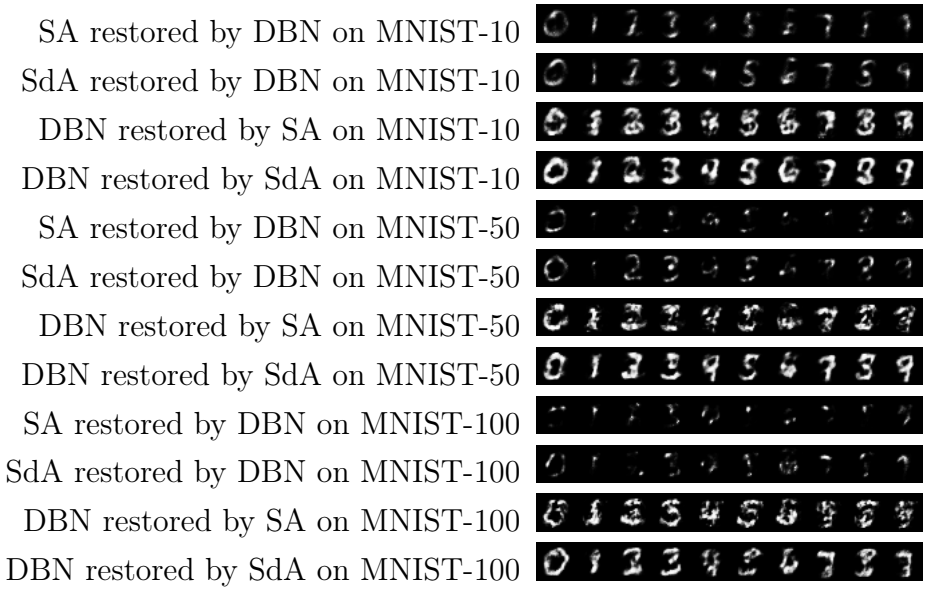

Figure 5.6.17: Reconstruction of the input of another architecture, by each of the three architectures, starting from the F-Mapped third layer internal representations of the mean-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

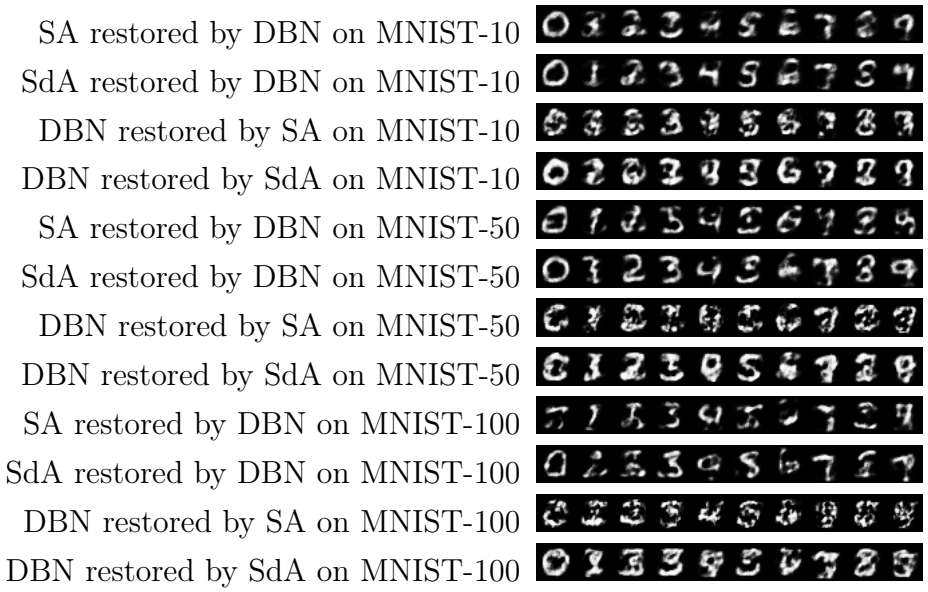

Figure 5.6.18: Reconstruction of the input of another architecture, by each of the three architectures, starting from the F-Mapped third layer internal representations of the filtered mean-digits in the test sets for MNIST-10, MNIST-50 and MNIST-100.

A table of the sum of Mean Squared Errors (MSEs) between the first-10-digits original input and all the reconstructions described in this section is provided (see table 5.5). The error always decreases when the F-Mapping is applied to an internal representation.

Table 5.5: Sum of Mean Squared Errors on the first-10-digits original input and the reconstructions, before and after the F-Mapping.

Architecture	**Dataset**	**Sum of Mean Squared Errors**					
		First-10-digits		Mean-digits		Filtered Mean-digits	
DBN	MNIST-10	0.3863		0.7483		0.7935	
	MNIST-50	0.3082		0.7283		0.8590	
	MNIST-100	0.3164		0.7662		0.8602	
SA	MNIST-10	0.2454		0.7355		0.8042	
	MNIST-50	0.3765		0.7846		0.8555	
	MNIST-100	0.4308		0.8768		0.9830	
SdA	MNIST-10	0.4256		0.8664		0.9884	
	MNIST-50	0.3958		0.8795		1.1516	
	MNIST-100	0.3691		0.8368		1.0857	
		No F-M	**F-M**	**No F-M**	**F-M**	**No F-M**	**F-M**
SA restored by DBN	MNIST-10	0.9632	0.7339	0.9994	0.9111	0.8659	0.7923
	MNIST-50	1.0280	0.8120	1.0404	0.9890	0.9881	0.7758
	MNIST-100	0.9929	0.8780	1.0373	1.0031	0.9303	0.9192
SdA restored by DBN	MNIST-10	0.9463	0.6623	0.9953	0.8268	0.8730	0.7848
	MNIST-50	0.9762	0.6796	1.0434	0.9172	0.9987	0.9394
	MNIST-100	1.0051	0.7611	1.0086	0.9299	0.9523	0.7906
DBN restored by SA	MNIST-10	0.9965	1.0146	0.9863	0.8443	1.0515	0.9128
	MNIST-50	1.0315	1.0079	1.0306	0.7947	1.1065	0.9396
	MNIST-100	1.1264	1.1638	1.0852	0.9030	1.2019	1.0354
DBN restored by SdA	MNIST-10	1.0155	1.0567	0.9339	0.8505	1.1258	0.9759
	MNIST-50	0.9078	1.1368	0.9032	0.8805	1.1194	1.1039
	MNIST-100	1.0781	0.9935	0.9779	0.8927	1.2478	1.1426

Chapter 6

Discussion and conclusions

The goal of this work is to understand the extent of the similarities and the overall pros and cons of using either Restricted Boltzmann Machines (RBMs), Autoencoders (AE) or denoising Autoencoders (dAs) in deep networks.

The three architectures have been finely designed and implemented to perform the state-of-the-art classification of MNIST digits and evaluated mainly on that task.

6.1 Overview of the results

For what concerns the robustness to corrupted input, none of the architectures seems to be outstanding in the classification of noisy inputs. For highly corrupted inputs the Stacked denoising Autoencoder (SdA) and the Stacked Autoencoder (SA) both perform better than the Deep Belief Network (DBN). For highly corrupted inputs with more training data available, the stacked autoencoder happened in this one case to perform better than the stacked denoising autoencoder. Otherwise the DBN has a slightly better performance.

So, quite surprisingly, the SA could be less prone to errors on corrupted inputs than the SdA, when enough data is available, but further work is required to confirm such a result. In general, SdAs and RBMs suffer of the same sensitivity to noisy input data.

When some filters are visualized, there is no doubt that the SdA can have more defined features than the SA. The details appearing in the filters always increase w.r.t. the training data available, and DBNs filters are practically comparable with SdA filters.

On the deep internal representations there are two aspects to discuss, the way they look if shaped as images and their numerical composition.

The first does not say much about how good or bad an internal representation is w.r.t. another one, but this layout makes more sense for the purpose of comparing different architectures. In order to achieve this, the F-Mapping - presented as a novel approach - can be applied

to the different internal representations to obtain aligned (feature-wise) representations (see figure 5.5.6). This method led to an experimental proof (hybrid input reconstructions) that the internal representations produced by different architectures are very similar to each other, and surely comparable.

The second aspect, on the other hand, can reveal useful characteristics, such as the sparsity and the number of activations. All internal representations, for the three architectures, show the same amount of sparseness. The histograms reveal fewer high (1) activations than low (0) activations in all the internal representations.

6.2 Conclusions

The results reported in this thesis confirm a fact that seems to be implicitly known in literature: if no improvement is given to the basic algorithms of restricted Boltzmann machines and autoencoders, it is just a matter of preference which one to use because the performance is comparable.

An important point is that the various internal representations produced by the three architectures on the same dataset are also quite similar in terms of features encoded, because the filters converge to the same details.

The input reconstruction data is encouraging, but it is also affected by a lot of different factors. One that needs to be mentioned is the fine-tuning, which changes the weights of the deep network, thus it alters the reconstruction functions of the single-units. This is allowed because the classification unilaterally goes from the input to the classification output and there is no need, after the layer-wise training, of an output from each single layer when only hidden layers are used to propagate the encoding to a next hidden layer or the final classification layer. The reconstruction is not even specifically optimized during the fine-tuning, since the training error is evaluated on the final classification.

Despite this, the results allow to make some conclusions: since hybrid input reconstructions, i.e. feeding internal representations to other architectures, were not possible on raw internal representations but they have been successfully obtained with the F-Mapping, the method proposed in this thesis appears to be valid and the internal representations are proved to be indeed very similar and comparable. In case one needs to use an hybrid architecture, interchangeability is feasible.

6.3 Future work

Since all the filters in this thesis have been produced by linear combination of the previous units, a good idea for future work is to apply Activation Maximization(Erhan et al., 2010) hoping for

better filters. It can directly influence the quality of the F-Mapping, which is executed above the filters and can be absolutely improved with further research. Then a new study on the internal representations could give better results.

The same analysis can also be extended to other types of architectures, especially the new ones mentioned in section 1.2, that could eventually diverge from the behavior hereby described.

All the architectures surely need to be tested on other datasets in order to collect more data and add more details to the conclusions proposed in this thesis.

Finally, it is just a matter of curiosity but the fine-tuning stage of a deep network seems to disrupt the inverse ability of the architectures to reconstruct the deep internal representations as good input images. It could be useful to balance this in order to achieve bidirectional deep networks. It would empower the idea that biological neural networks are able to project their signals back, in order to produce synthetic low level inputs that are fed to other neural pathways (Damasio, 2008).

Bibliography

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., & Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Bengio, Y. (2007). *Learning deep architectures for AI.* Technical Report 1312, Dept. IRO, Universite de Montreal. Preliminary version of journal article with the same title appearing in Foundations and Trends in Machine Learning (2009).

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and trends in Machine Learning*, 2(1), 1–127.

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8), 1798–1828.

Bengio, Y. & Delalleau, O. (2008). Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6), 1601–1621.

Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., et al. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 153.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., & Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy).*

Bourlard, H. & Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4-5), 291–294.

Chen, M., Weinberger, K. Q., Sha, F., & Bengio, Y. (2014). Marginalized denoising auto-encoders for nonlinear representations. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* (pp. 1476–1484).

Coates, A., Ng, A. Y., & Lee, H. (2011). An analysis of single-layer networks in unsupervised feature learning. In *International conference on artificial intelligence and statistics* (pp. 215–223).

Cohen, W. W., McCallum, A., & Roweis, S. T., Eds. (2008). *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*. ACM.

Damasio, A. (2008). *Descartes' error: Emotion, reason and the human brain*. Random House.

Deng, L. & Yu, D. (2014). Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4), 197–387.

Erhan, D., Courville, A., & Bengio, Y. (2010). Understanding representations learned in deep architectures. *Universit e de Montr eal/DIRO, Montr eal, QC, Canada, Tech. Rep*, 1355.

Goodfellow, I., Lee, H., Le, Q. V., Saxe, A., & Ng, A. Y. (2009). Measuring invariances in deep networks. In *Advances in neural information processing systems* (pp. 646–654).

Hinton, G. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8), 1771–1800.

Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7), 1527–1554.

Hinton, G. E. & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786), 504–507.

Hinton, G. E. & Zemel, R. S. (1994). Autoencoders, minimum description length, and helmholtz free energy. *Advances in neural information processing systems*, (pp. 3–3).

Hubel, D. H. & Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1), 215–243.

Hurley, N. & Rickard, S. (2009). Comparing measures of sparsity. *Information Theory, IEEE Transactions on*, 55(10), 4723–4741.

Krizhevsky, A. & Hinton, G. (2009). Learning multiple layers of features from tiny images.

Krizhevsky, A., Hinton, G. E., et al. (2010). Factored 3-way restricted boltzmann machines for modeling natural images. In *International Conference on Artificial Intelligence and Statistics* (pp. 621–628).

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.

Lee, H., Battle, A., Raina, R., & Ng, A. Y. (2006). Efficient sparse coding algorithms. In *Advances in neural information processing systems* (pp. 801–808).

Lee, H., Ekanadham, C., & Ng, A. Y. (2008). Sparse deep belief net model for visual area v2. In *Advances in neural information processing systems* (pp. 873–880).

Lorenz, M. O. (1905). Methods of measuring the concentration of wealth. *Publications of the American statistical association*, 9(70), 209–219.

Olshausen, B. A. et al. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583), 607–609.

Olshausen, B. A. & Field, D. J. (2004). Sparse coding of sensory inputs. *Current opinion in neurobiology*, 14(4), 481–487.

Poultney, C., Chopra, S., Cun, Y. L., et al. (2006). Efficient learning of sparse representations with an energy-based model. In *Advances in neural information processing systems* (pp. 1137–1144).

Ranzato, M. & Hinton, G. E. (2010). Modeling pixel means and covariances using factorized third-order boltzmann machines. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on* (pp. 2551–2558).: IEEE.

Rasmus, A., Raiko, T., & Valpola, H. (2014). Denoising autoencoder with modulated lateral connections learns invariant representations of natural images. *arXiv preprint arXiv:1412.7210*.

Rumelhart, D., Hinton, G., & Williams, J. (1986). Learning internal representations by error propagation. parallel distributed processing, vol. i, rumelhart, d. e. and mcclelland, jl.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61, 85–117.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory.

Sohn, K. & Lee, H. (2012). Learning invariant representations with local transformations. *arXiv preprint arXiv:1206.6418*.

Tan, C. C. & Eswaran, C. (2008). Performance comparison of three types of autoencoder neural networks. In *Modeling & Simulation, 2008. AICMS 08. Second Asia International Conference on* (pp. 213–218).: IEEE.

Tieleman, T. (2008). Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning* (pp. 1064–1071).: ACM.

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In Cohen et al. (2008), (pp. 1096–1103).

Yang, J., Yu, K., Gong, Y., & Huang, T. (2009). Linear spatial pyramid matching using sparse coding for image classification. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (pp. 1794–1801).: IEEE.

Index