Exploration of deep Auto-Encoders architectures for dimensionality reduction

Anthony T.Garant

École Polytechnique de Montréal

Presented to

Prof. Christopher Pal

1 INTRODUCTION

Dimensionality reduction can be useful for many applications such as visualization, classification, storage of data and information retrieval with technique such as semantic hashing [1]. Works on dimensionality reduction using neural networks [2] have shown that using a deep auto encoder structure allowed for better reconstruction of images and two dimensions visualizations compared to a classical technique: principal component analysis (PCA). A key element to the success of training a deep model architecture was the use of unsupervised pre-training. Hinton et al. [2] used greedy layers wise pre-training with restricted boltzmann machines (RBM). Vincent et al. [3] proposed to use denoising auto encoder to extract robust features which yield similar results to RBM. Our first model, explores the impact of depth and pre-training with a deep auto-encoder architecture pre-trained with denoising auto-encoders.

Recent works in image recognition exploited deep convolutional architecture which lead to breakthroughs in the field [4]. These results give the intuition that convolution might be useful for image reconstruction with auto encoder. A convolutional auto-encoder requires the implementation of deconvolution layers. This procedure has previously been explored to understand the visual aspect retrieved by the convolutional network. The deconvNet was introduced by Zeiler et al. [5] and used by the stacked what where auto-encoders [6] to build a convolutional auto-encoder by pairing it to a convNet. Furthermore, Simonyan et al. [7] demonstrated that de deconvnet can be generalized by a gradient-based method. We present a similar architecture to stacked what-where auto-encoders with the difference that a gradient-based method is used for the deconvolutional layers of the decoder.

2 AUTO-ENCODERS

We begin by recalling the basic auto-encoder [8]. The goal of an auto-encoder is to reconstruct its own input. Id does so by encoding its input in a usually lower dimensionality code (or compressed) layer. It then attempts to decode this compressed layer to reconstruct the input. Its analogy with the human brain makes it particularly interesting. Humans memory are not built from complete information; they reconstruct the images from few compressed information that they actually remember. [9] Auto-encoders are traditionally used to reduce the dimensionality of the data or to extract useful features. The intuition behind feature extraction is that if it is able to reconstruct an input from a compressed version of it, it must have learned the important characteristics (features) of the input. Its most basic implementation is a neural network with a single hidden-layer (that represents the compressed layer). The network takes an input $x \epsilon [0,1]^d$ and encodes it to the hidden layer h of dimension d', $h(x) = g(b + Wx)$. Where $\boldsymbol{g}$ is an activation function, $\boldsymbol{W}$ is a $d \times d'$ weights matrix and $\boldsymbol{b}$ is a bias vector. The hidden-layer is then decoded to form the reconstruction $\hat{x} \epsilon [0,1]^d$,

$\hat{x} = o(b' + W' * h(x))$. Where $\boldsymbol{o}$ is an activation function (usually sigmoid to get back values between 0 and 1) and $\boldsymbol{W'}$ is a $d' \times d$ weights matrix. It is common to set $\boldsymbol{W'} = \boldsymbol{W^T}$, in which case the auto-encoder is said to have tied weights.

2.1 Denoising auto-encoders [3]

A common problem with auto-encoders, especially if the representation is over complete (the size of the code-layer is greater than the size of the input), is that the network can learn the identity mapping instead of extracting important features. A way around this is to train the network to reconstruct the original input by giving it a noisy version of the input. The noise process is usually to randomly assign a subset of inputs to 0, with a probability v. The result is that the network is forced to learn the most salient features to be able to effectively remove the noise and produce a good reconstruction.

2.2 Deep auto-encoders

Deep auto-encoders [2] are similar to the basic auto-encoders. The only difference being that instead of having a single hidden-layer that corresponds to the code-layer, more hidden-layers are stacked before reaching the code layers in the encoding part. Those layers have corresponding symmetrical layers in the decoding part.

Greedy layer wise pre-training can be performed [8]. In our case, denoising auto-encoders between each layers are used for pre-training. The dimension of the hidden-layer corresponds to the size of the layer below the one we are currently pre-training. The weights of the deep-autoencoder layer is initialized to the weight of the denoising autoencoder. For the first dA, the input is the network input. Otherwise, the previous layer uncorrupted dA output is used.

2.3 Convolutional auto-encoders

The goal of convolutional auto-encoders is to exploit the 2 image structure. Convolutional network (CNN) [10] essentially leverage three ideas. The first idea is the **local connectivity**. Instead of having hidden layers fully connected to its previous layer, its going to be connected to only a small spatially localized portion of the image called a receptive field. The second idea is **parameter sharing**. We will use the same matrix of weighted connection and connect this same matric across a whole image (may have multiple channels) to build a feature map. Many feature maps will be build with this strategy, each having its own weights matrix. The third idea is **pooling and subsampling.** Taking max-pooling for exemple, in a windows (e.g. 3x3) only the highest value will be kept. And this operation is applied across the image.

To build a convolutional auto-encoder, the decoder is composed of unpooling and deconvolutionnal layers. The symmetrical layers have tied weights. To perform a good unpooling, the locations of the max values in the max-pooling layer must be remembered. Once the forward pass is done the inverse layers can be calculated using the gradient-based method [7] which generalize the deconvnet [5].

3 EXPERIMENTS

3.1 Deep auto-encoder

Throughout the experiments, batch-size of 128 is used, 20 epochs are used to train pre-training denoising auto-encoders and 250 epochs are used to fine-tune the auto-encoder. The learning rate during pre-training is set to 0.01. The learning rate during fine-tuning is initially set to 0.01 but is divided by 10 after 50, 100 and 150 epochs. The dataset used for the experiments is MNIST [11].

Architecture	**post pre-training**	**Post fine-tuning**
784-30 (baseline)	9161	76.7024
784-250-30	147.907	67.31
784-500-250-30	198.796	65.0671
784-1000-500-250-30	156.502	63.3451
*784-2000-1000-500-250-30**	300.366	62.19
*784-3000-2000-1000-500-250-30**	274.637	64.259

Table 1: Mean reconstruction costs after pre-training and after fine-tuning in network with variable hidden layers configurations. *The convergence was slower so the number of epochs was increased to 400, with learning rate updates after 100, 200 and 300 epochs

The impact of the depth and the width of the hidden unit in the deep auto-encoder is explored and results are presented in table 1. As in Hinton et Al. [2] paper, the code-layer is set to 30. The mean reconstruction cost is reduced as the depth of the auto-encoder is increased. However, the training time is also increased as more epochs are required to reach convergence. Furthermore, the increased capacity pushes the auto-encoder to overfit the training data as demonstrated in Figure 2. This explains the lower test reconstruction cost.

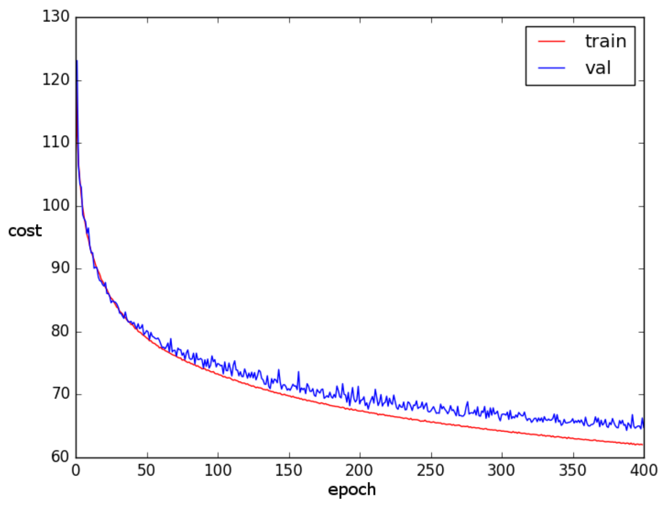

Figure 1: Learning curve of the 3000-2000-1000-500-250-30 network.

A sample reconstruction produced by the 784-1000-500-250-30 network is presented in Figure 2. Interestingly, even if the mean reconstruction cost after the pre-training is fairly large (198.796), most digits are easily discernable. This suggests that the pre-train was able to detect some salient features and reach its goal of initializing the weight close to a good solution. Another, important observation to make is that the produced reconstruction (after fine-tuning) is creating a smoother version of the input. For example, the three zeros at the bottom left corner are somewhat imperfect, yet in the reconstruction these imperfections have been filtered out. This is coherent with the goal of the auto-encoder, by reducing the dimensionality, the network is forced to learn the digits' characteristics to have good generalization performances.

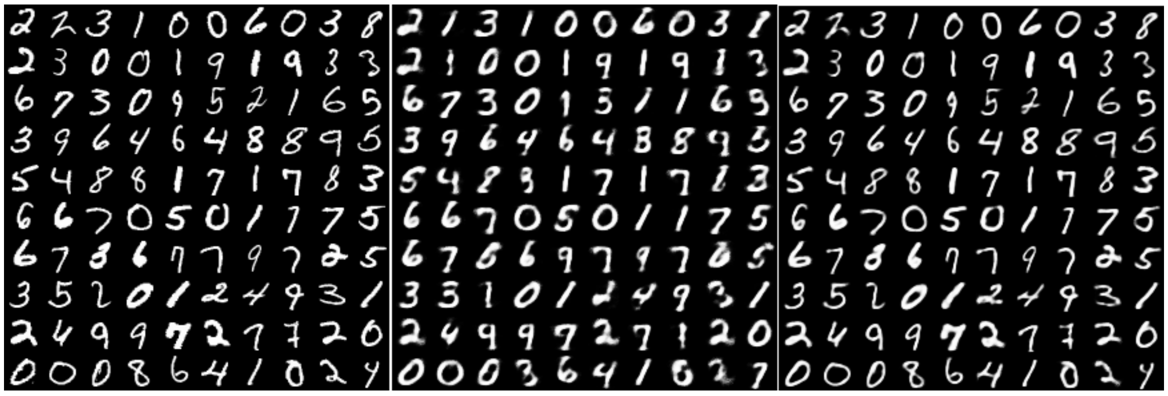
Figure 2: From left to right: Random sample input of 100 images from the test set. Reconstruction of the 100 test inputs after the pre-training with a corruption level of 0.4 and an architecture of 784-1000-500-250-30 (mean cost of 198.796. Reconstruction of the test input after the pre-training and fine-tuning with an architecture of 784-1000-500-250-30 (mean cost of 65.07).

The impact of the corruption level during pre-training on the reconstruction is presented in Table 2. we experiment with the architecture of 784-1000-500-200-30. To our surprise the reconstruction cost, post pre-training, is lower as the corruption level is reduced. Furthermore, after the fine-tuning, regardless of the corruption level, the mean reconstruction cost is similar. Even with no pre-training the cost after fine-tuning is similar, it even converged faster (see learning curves in appendix). This suggests that the network had enough capacity to reproduce the input regardless of the pre-training.

To validate this hypothesis, we repeat those experiments with a reduced code-layer of size 2. With a code-layer of size 2, the reconstructions are blurry (Figure 4) which means that this is not a situation where the network has enough capacity to represent the model. Yet, the same relation between the corruption level and the pre and post fine-tuning cost is exhibited. However, the dimensionality reduction allowed the visualization of the code layer. We observe that although the mean reconstruction cost is similar, the Figure 3-A compared to Figure 3-C suggests that the features that are extracted to reconstruct the input are different. The clusters are less homogeneous and some classes are split in two. For example, three separated clusters of 5 and two separated clusters of 8 can be detected. In contrast, when using pre-training with a corruption level of 0.4, the classes are grouped better. This suggests that without pre-training, the auto-encoder learned smaller segments of the digits (more classes than there actually is). Half of the 8s being with the majority of the 5s and the other half with the 3s shows that it learned both horizontal halves of an 8 instead of learning the complete form of an 8.

Corruption level	**After pre-training**	**After fine-tuning**
No pre-training (baseline)	1010.49	62.5543
0	116.878	63.2004
0.1	122.804	63.0178
0.3	138.967	62.8185
0.4	156.502	63.3451
0.5	192.579	64.0252
0.7	315.839	64.9587
0.9	674.319	70.3085

Table 2: Mean reconstruction costs after pre-training and after fine-tuning on a 784-1000-500-250-30 network.

The 2d visualization helps understand how closely related examples from different classes are. For example, in the Figure 3-D, the 7s are close to the 9s which are really close to the 4. Then, the 8 is just under the 5 and the 3. Those observations make senses but are not particularly useful with digits. However, this property can most likely generalize to other datasets that might be harder to manually group and detect the affinity.

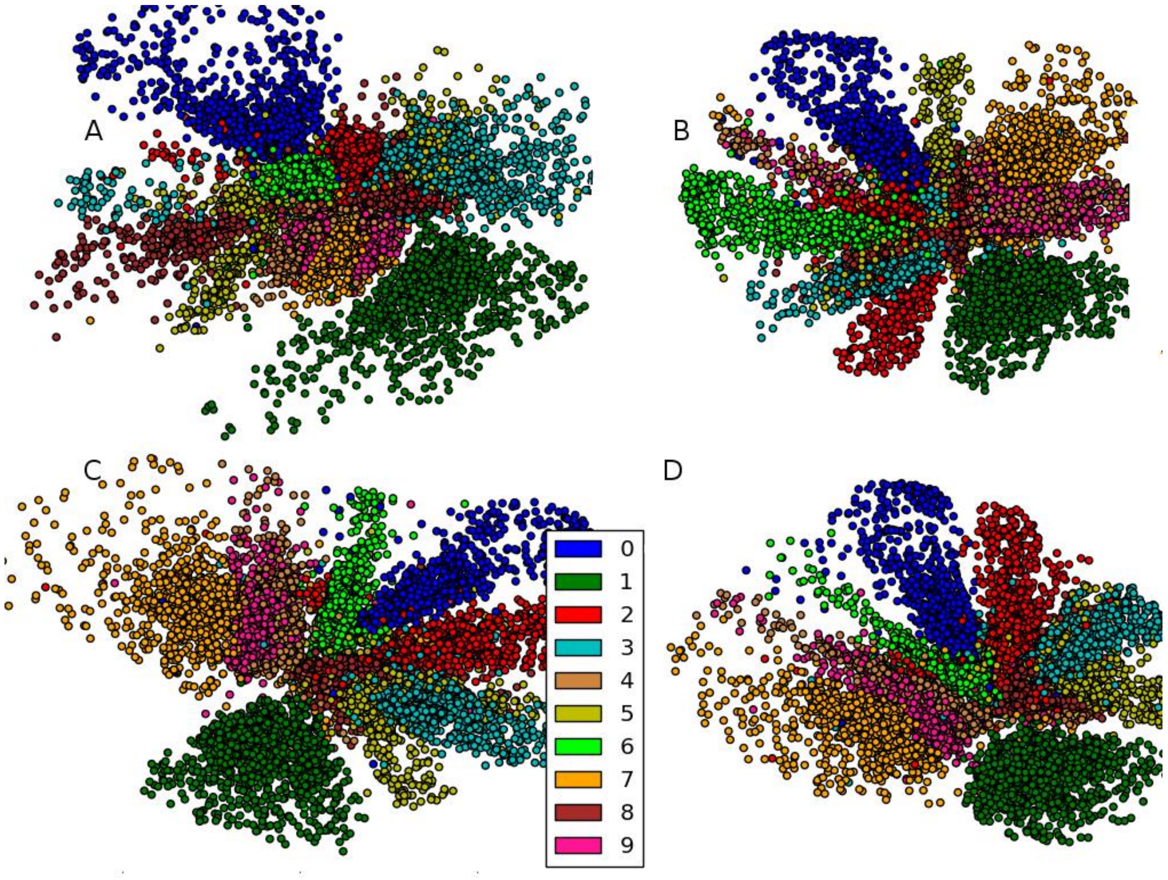

Figure 3: 2d-visualization of the code-layer, encoded with a 784-1000-500-250-2 architecture. **A** No pre-training. **B** No input corruption during pre-training. **C** Corruption level of 0.2. **D** Corruption level of 0.4

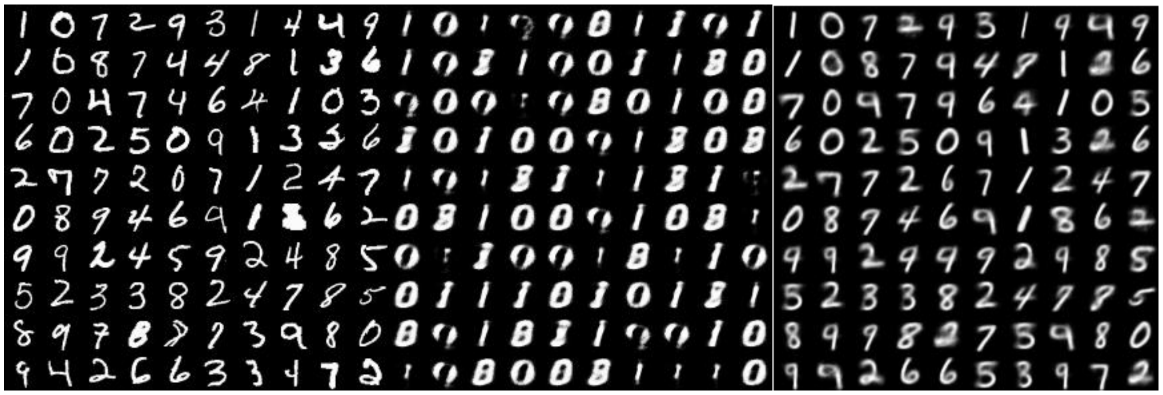

Figure 4: From left to right: Random sample input of 100 images from the test set. Reconstruction of the 100 test inputs after the pre-training with corruption levels of 0.4 and an architecture of 784-1000-500-250-2. Reconstruction of the test input after the fine-tuning and pre-training with an architecture of 784-1000-500-250-2 (cost: 133.282).

3.2 Convolutional Auto-Encoder

We use Zhao et al.[6] notation to describe the tested architectures e.g. (16)5c-2p (32) 3cs-10fc in which '(16) 5c' denotes a convolutional layer with 16 feature maps while kernel size being set to 5. 2p denotes a 2x2 max pooling layer, and 10fc denotes fully connected layer that connects to 10 hidden units. Non linearity (ReLU) is omitted in the notation. Extending from this notation, 'cs' denotes that padding has been added before the convolution to keep the same dimension.

Through out the experiments, batch-size of 128 is used, 30 epochs are used to train the model. The learning rate is initially set to 0.01 but is divided by 10 after 10 and 20 epochs. The dataset used for the experiments is MNIST [11]. Batch-normalization is used after convolutional layers.

We first explore the effect of depth with fully-convolutional auto-encoders. Table 3 presents the results. As expected, as the capacity is increased, the reconstruction cost lowers. It can be observed that the reconstruction cost of the (64)3-(128)3-(256)3-30fc is better than anything obtained with the deep auto-encoder presented in the previous section. With a mean reconstruction cost under 60, it is difficult to tell the images apart (Figure 5).

Architecture	**Post fine-tuning**
(64)3-30fc	71.64
(64)3-(128)3--30fc	68.5001
(64)3-(128)3-(256)3-30fc	59.9759

Table 3: Mean reconstruction costs after fine-tuning in network with different hidden convolutional layers configurations.

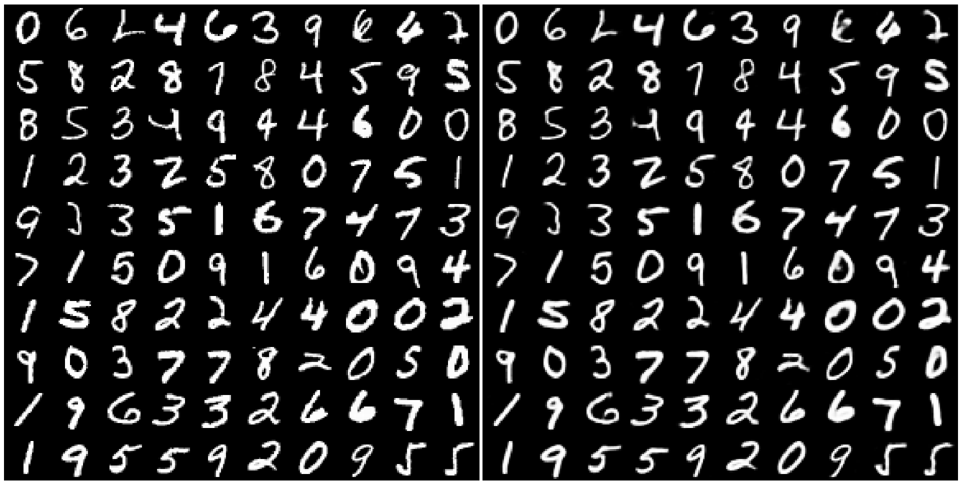
Figure 5: Left: Random sample input of 100 images from the test set. Right: Reconstruction of the 100 test inputs after the fine-training with an architecture of (64)3-(128)3-(256)3-30fc.

Max pooling layers are added to introduce invariance to local translations and to reduce the number of hidden units in hidden layers. Impact on the reconstruction is presented in Table 4. As hoped, the mean reconstruction is reduced in comparison with the fully convolutional layers. To have a better understanding of the features that are retrieved, the code-layer is further reduced to a size of 2. With a single convolutional + max-pooling layer, the impact on the reconstruction cost of the coder-layer size reduction was the biggest. Interestingly, the type or error is different from the errors that the deep-auto encoder committed. With it, the reconstructed characters were blurry when the error was higher. With convolutions, the error represents missing parts from the image (Figure 6). Following theses observations, we attempt to increase the receptive field. Results are presented in Table 6. Using a larger receptive field decreased the mean reconstruction cost even lower. Surprisingly, even if the mean reconstruction cost is lower using convolutional auto-encoders, the 2d visualization shows that the network does not cluster the code-layer in the actual classes (Figure 7). From these observations, we believe that our convolutional auto-encoder might not learn salient features and instead learns the identify function.

Architecture	**Post fine-tuning**
(64)3c-2p-30fc	60.3341
(64)3c-2p-(128)3c-2p-30fc	58.1468
(64)3c-2p-(128)3c-2p-(256)3c-2p-30fc	61.906
(64)3c-2p-(64)3cs-(128)3c-2p-(128)3cs-(256)3c-2p-(256)3cs-30fc	56.0027

Table 4: Mean reconstruction costs after fine-tuning the network with different convolutional + max-pooling configurations.

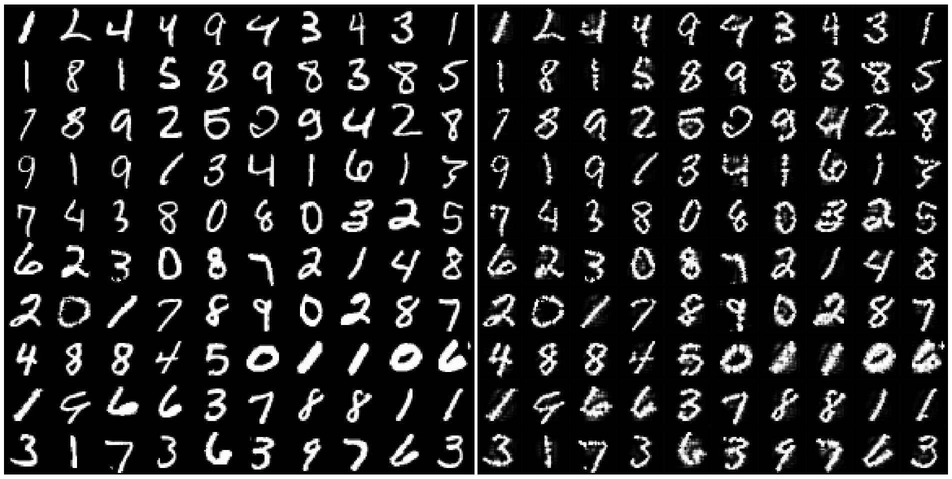

Figure 6: Left: Random sample input of 100 images from the test set. Right: Reconstruction of the 100 test inputs after the fine-training with an architecture of (64)3c-2p-2fc (cost: 86.4014).

Architecture	**Post fine-tuning**
(64)3c-2p-2fc	86.4014
(64)3c-2p-(128)3c-2p-2fc	61.1658
(64)3c-2p-(128)3c-2p-(256)3c-2p-2fc	55.9326
(64)3c-2p-(64)3cs-(128)3c-2p-(128)3cs-(256)3c-2p-(256)3cs-2fc	55.9708

Table 5: Mean reconstruction costs after fine-tuning the network with different convolutional + max-pooling configurations and a code-layer of size 2.

Architecture	**Post fine-tuning**
(64)5c-2p-(128)5c-2p-2fc	52.1375
(64)5c-2p-(128)5cs-(256)5c-2p-2fc	52.4475
(64)8c-2p-(128)3c-2p-2fc	54.6179

Table 6: Mean reconstruction costs after fine-tuning the network with different convolutional + max-pooling configurations, a code-layer of size 2 and varying receptive field.

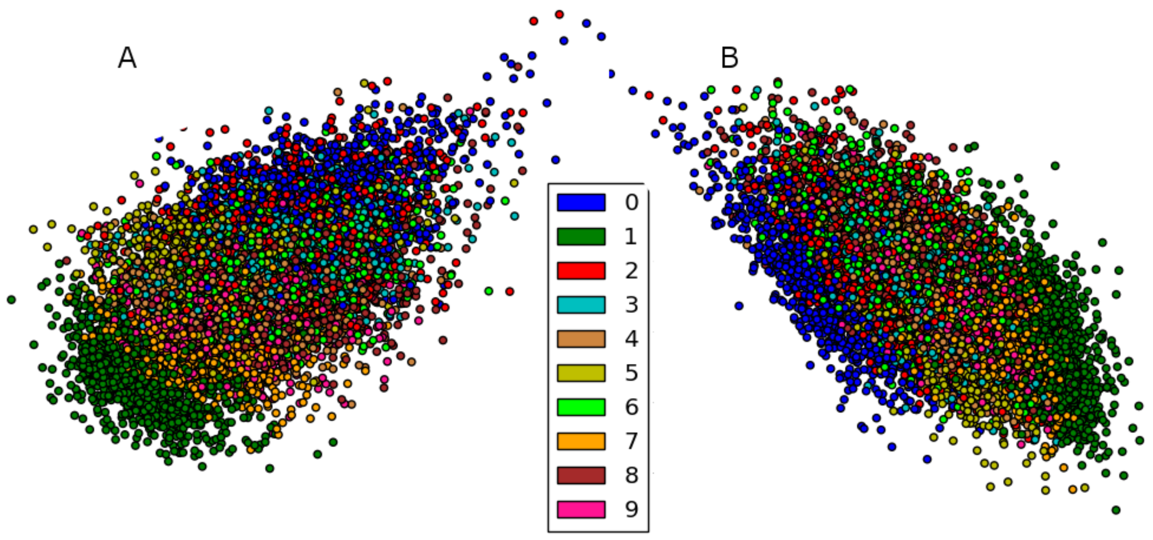

Figure 7: A. The two-dimensional code-layers found by a (64)5c-2p-(128)5c-2p-2fc convolutional auto-encoder (cost: 52.1375). **B** The two-dimensional code-layers found by a (64)3c-2p-(64)3cs-(128)3c-2p-(128)3cs-(256)3c-2p-(256)3cs-2fc convolutional auto-encoder (cost: 55.9708).

4 CRITICAL ANALYSIS

We wanted to learn more about auto-encoders, unsupervised pre-training, convolutional networks and research in general. To first define what we wanted to implement or learn we read many papers. This first step was extremely instructive in itself. Reading articles allowed us to have a clearer understanding of the methods used to evaluate, demonstrate and present a scientific contribution.

Our initial idea was actually to use residual blocks as presented in He et al. paper [12] to form a residual auto-encoder. Our intuition was based on their results suggesting that the skip-connections allow to train effectively a deeper network. Nonetheless, implementing a novel residual auto-encoder is not trivial. Consequently, we decided to use an iterative approach and devised the task in 3 steps: implementing a deep auto-encoder, implementing a deep convolutional auto-encoder and finally adding skip connections to build a deep residual auto-encoder. Unfortunately, we did not have the time to finish the implementation of the residual auto-encoder. No existing implementation on Theano was found for any of the three proposed steps. The deeplearning tutorial implements an auto-encoder, a denoising auto-encoder and stacked auto-encoders. This greatly helped in implementing the deep auto-encoder.

For the convolutional network, the use of Lasagne library definitely helped as it took care of the hardest part, the deconvolution and unpooling layers. We would have learned more about the underlying mathematics by implementing it ourselves, for example in matlab. However, that was not our ultimate goal. We wanted to learn about how the convolutional work and we found that experimenting with it and visualization the results allowed us to gain a good insight on the matter.

Another thing we could have done is to visualize the weight matrices. The 2d visualizations suggest whether the features retrieved were linked to the class or not. However, visualizing the actual features could have been interesting and help us validate whether our convolutional network is learning the mapping or not. Also, the reconstructions were fairly good with the Mnist dataset, it would be interesting to test the architecture's with more complex datasets.

Ultimately, we thought the best way to learn a concept in depth was to learn enough of it to be able to build something new on top of previous works to contribute to the field. The requested structure is a bit different than a classical research paper format and we were not able to finish our residual auto-encoder. Nonetheless, our implementation and in depth analysis of two models. Namely the deep auto-encoder pre-trained with denoising auto-encoders and the convolutional allowed us to learn effectively and in depth the concepts.

REFERENCES

1. Salakhutdinov, R. and G. Hinton, *Semantic hashing.* International Journal of Approximate Reasoning, 2009. **50**(7): p. 969-978.
2. Hinton, G.E. and R.R. Salakhutdinov, *Reducing the dimensionality of data with neural networks.* Science, 2006. **313**(5786): p. 504-7.
3. Vincent, P., et al., *Extracting and composing robust features with denoising autoencoders*, in *Proceedings of the 25th international conference on Machine learning*. 2008, ACM: Helsinki, Finland. p. 1096-1103.
4. Krizhevsky, A., I. Sutskever, and G.E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, in *NIPS*, P.L. Bartlett, et al., Editors. 2012. p. 1106-1114.
5. Zeiler, M.D. and R. Fergus *Visualizing and Understanding Convolutional Networks*. ArXiv e-prints, 2013. **1311**.
6. Zhao, J., et al. *Stacked What-Where Auto-encoders.* ArXiv e-prints, 2015. **1506**.
7. Simonyan, K., A. Vedaldi, and A. Zisserman *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*. ArXiv e-prints, 2013. **1312**.
8. Bengio, Y., et al., *Greedy layer-wise training of deep networks.* Advances in neural information processing systems, 2007. **19**: p. 153.
9. Loftus, E.F. and J.C. Palmer, *Reconstruction of automobile destruction: An example of the interaction between language and memory.* Journal of verbal learning and verbal behavior, 1974. **13**(5): p. 585-589.
10. Jarrett, K., et al. *What is the best multi-stage architecture for object recognition?* in *Computer Vision, 2009 IEEE 12th International Conference on*. 2009. IEEE.
11. Lecun, Y. and C. Cortes, *The MNIST database of handwritten digits.*
12. He, K., et al. *Deep Residual Learning for Image Recognition*. ArXiv e-prints, 2015. **1512**.