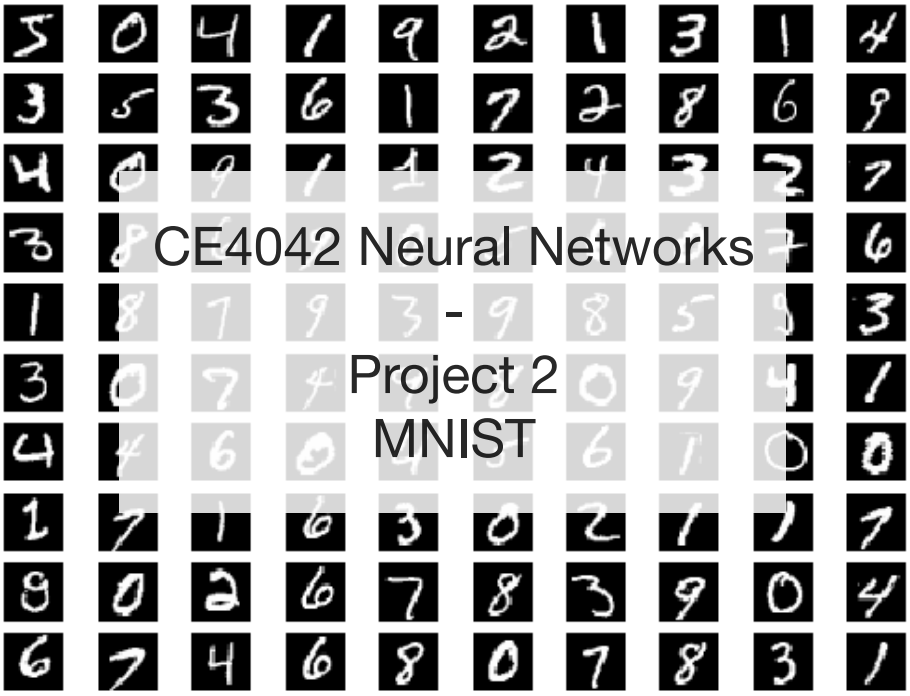
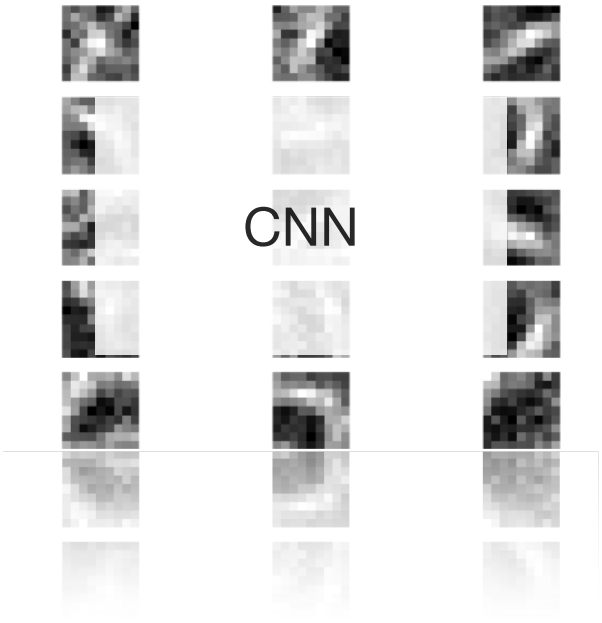
Durand Tom
n1701729g

13.11.2017



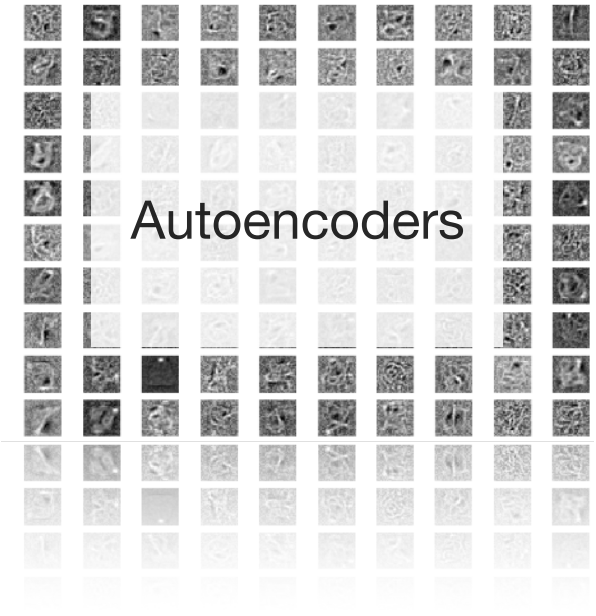
CE4042 Neural Networks - Project 2 MNIST



CNN

&

Autoencoders



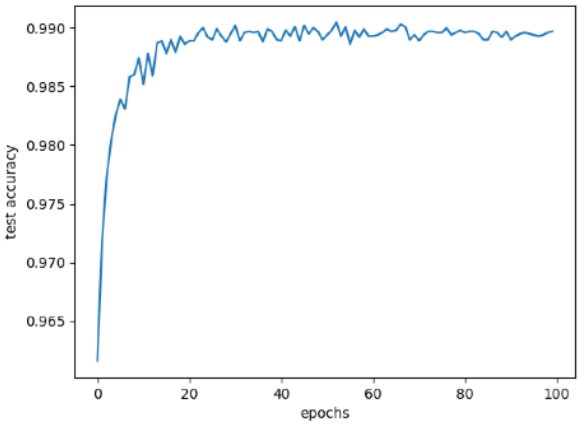
Part A : Deep convolutional neural networks

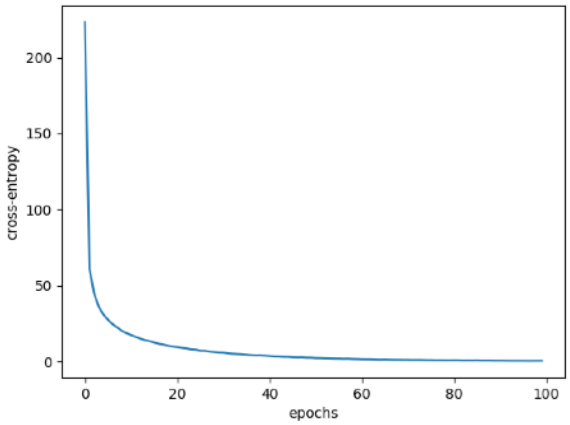
Recognize an object is quite an easy task for humans. In fact, in a glance we can interpret a scene with a good accuracy thanks to our brain and especially our visual cortex. However, we struggled during a long time to give the computer the ability to analyse and understand a scene. We can use some descriptors to identify certains objects like for instance SIFT but it still has limited performances. But now, with the development of neural networks and deep learning, we manage to have really high accuracy, even better than human sometimes, in object recognition. The principe is to reproduce the behaviour of the visual cortex by extracting some features of different order at each layer. This is done by producing a set of filters for each layer with which we will make a convolution on the image (or the previous layer). Then, with the learning, the filters will automatically extract characteristic features of the data. By adding convolutional layers followed by pooling layers (to reduce dimensions, make the model more robust to translations) and then some fully connected layers we can obtain really good recognition rates. On this part we will evaluate the performances of a model composed of 2 convolutional layers with a pooling layer after each of them followed by a fully connected layer of 100 neurons and finally a softmax layer of 10 neurons as output. We will also evaluate different algorithms designed to fasten and improve the training that is relatively heavy in deep learning architecture as CNN.

1- Mini batch gradient descent

For this first training we use the basic mini batch gradient descent technique. The size of the batch is 128, the learning rate 0.05, the decay parameter: 10^{-4} and the activation of hidden layer is ReLU. We can see on the following graphs that it did the job pretty well, the accuracy being pretty high and the cross entropy around 0. We can notice that the decay parameter avoids overfitting because the accuracy doesn't go down with training and that the number of epochs is sufficient because the convergence doesn't evolve any more from 60 epochs to 100. For this part we used the full dataset for training and testing.

Program to run: project2a_1.py





Now let's see the features maps obtained for two digits at the convolution and pooling layers with this network.

Image input 28*28

Features extraction at
the first layer
15 filters (size 9*9)

15 * 20 * 20

Max pooling first layer
(size 2*2)

15 * 10 * 10

Features extraction at
the seconde layer
20 filters (size 5*5)

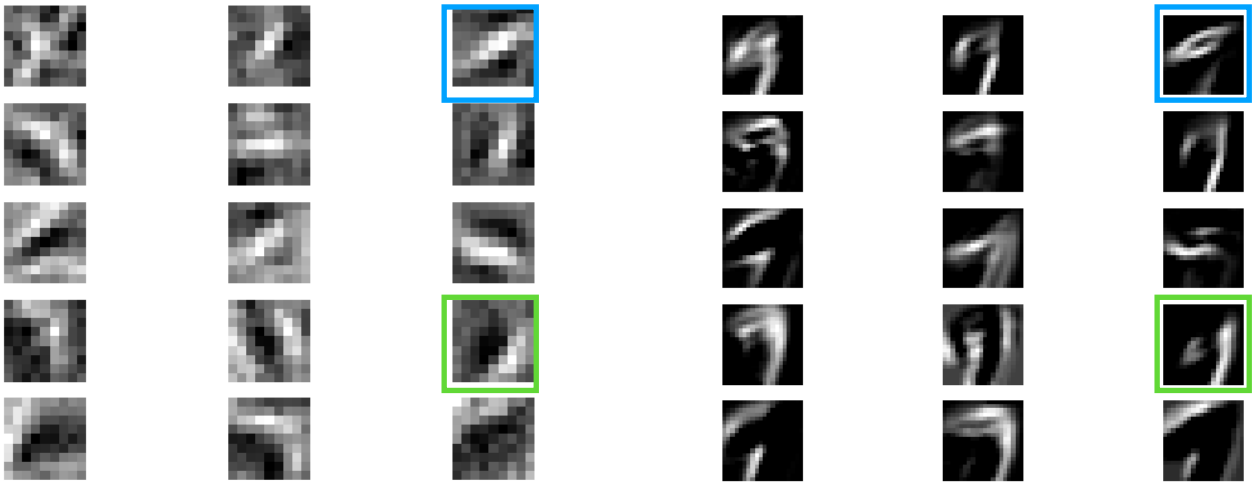
20 * 6 * 6

Max pooling first layer
(size 2*2)

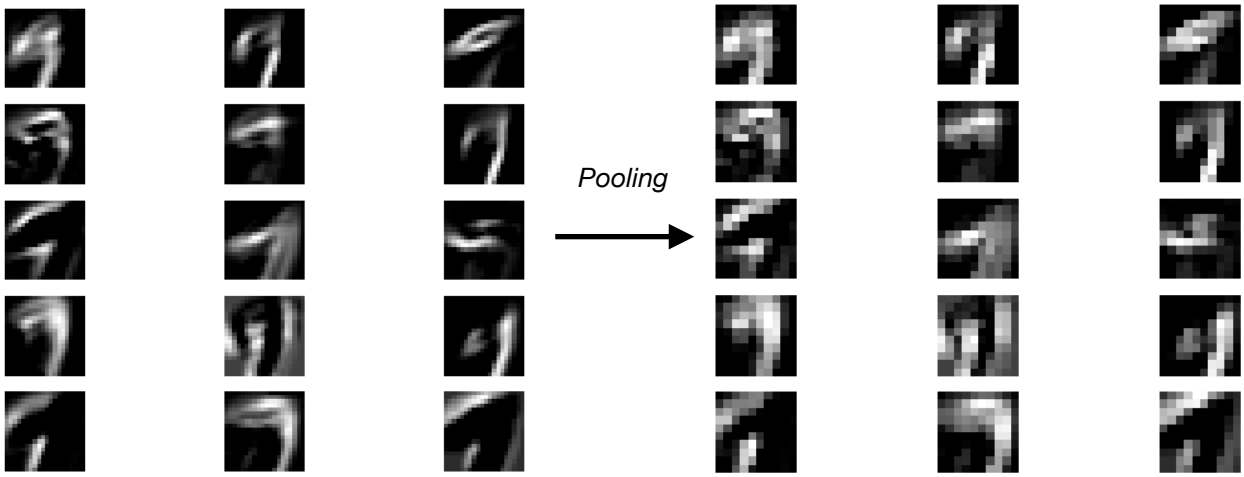
20 * 3 * 3

As we can see the first layer seems to be more sensible to edges (first order feature). Indeed if we look at the filter's weights learned the correlation with the edges is quite easy to find. Then some numbers will automatically responds with a higher activation for the edges they contain.

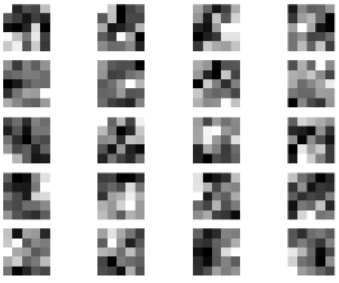
Weights learned and correlations with features for the digit corresponding to number 9.



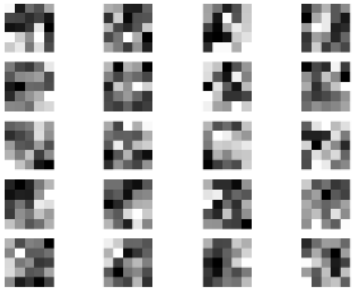
Moreover we can see that doing pooling just lower the definition but brings more robustness to translations. It also keeps the importance of strong features and makes them easier to exploit for the network by lowering the dimensions.



If we look at the second layer we can see that it is more difficult to interpret the features extracted thanks to the filters. We can just say that they are some higher features, usually it should correspond to basic combinations of edges like corners or basic shapes. One of the reason why it is difficult to state is that the weights obtained are quite noisy. It can be a sign of not enough training or overfitting (or maybe just a too low definition for us to state). We can notice that when I used the full dataset to train the weights are a bit smoother than if I use just 12 000 samples.



Weights learned at second convolutional layer. All the training dataset is used for the left image and 12000 samples are used for the right dataset.



To conclude on first question we can see than the network success to extract the important features of the images. This was possible mainly thanks to the use of 2 convolutional and pooling layers that extract different orders features discriminative for the classification. Those layers, connected to an other fully connected layer and a softmax layer do a precise classification (accuracy of about 99%) that proves their efficiency for image classification problems. Now that we saw that this kind of network is very efficient for this task we will se if we can improve its performances.

2- Mini batch gradient descent with momentum

One problem that can be faced by deep learning architectures using gradient descent is very low convergence rates due to high curvature or noisy gradient. Indeed the gradient descent algorithm can oscillate a long time before it finds the minima. To solve this problem we can use the momentum. The principle is to remember the global direction of the past gradients and continue to move in the same way. By doing this the learning should be more efficient and the convergence faster.

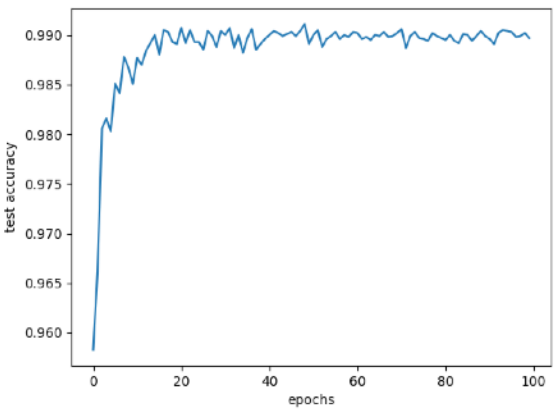
The new term added during learning is called the velocity term and the momentum γ will be fixed at 0.1 for this question. These are the new learning equations for the weights:

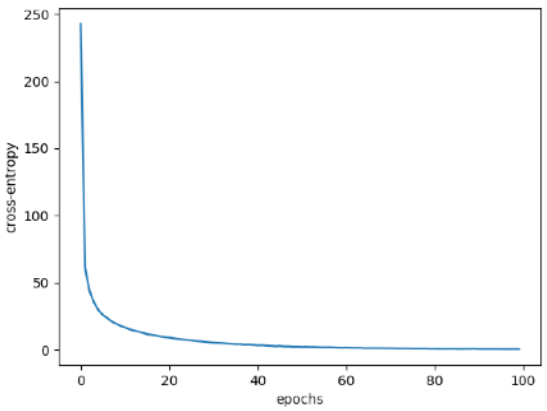
$$V = \gamma V - \alpha \frac{\partial J}{\partial W}$$

$$W = W + V$$

For the others parameters we will keep the same values as before:
batch size = 128
learning rate = 0.05
decay parameter = 10^{-4}
full dataset for training and testing

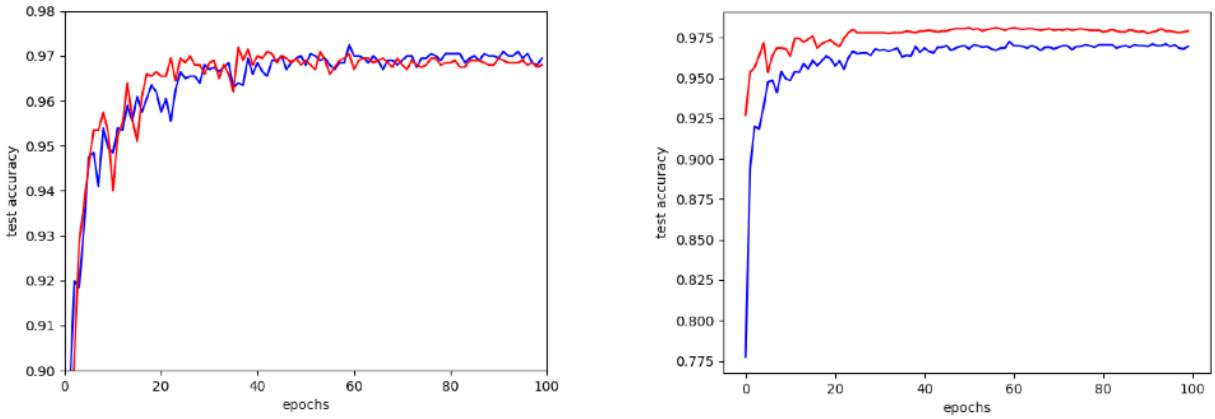
Program to run: project2a_2.py





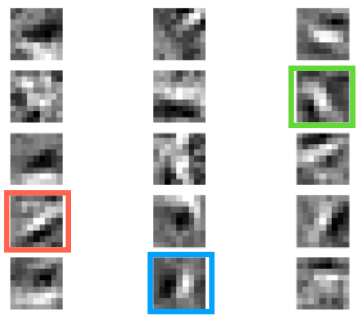
We can see that for this case it also provides really good performances. But with all the dataset for training the convergence is to fast to really judge the improvements brought by the momentum on the gradient descent. This is why we plotted the test accuracy of the gradient descent with and without momentum an other time with just 12000 samples to train.

We can observe a small improvement thanks to the momentum (red curve) with an accuracy of 0.97 obtained at around 22 epochs against 40 epochs for basic gradient descent. An other point is that for this experiment we kept a momentum of 0.1 for all the training but usually we will increase it to around 0.9. This is because the more we advance in the training, the more we can trust past directions of the gradients. In the case of the MNIST dataset the convergence seems quite easy and quick to find. We can then try to increase the momentum parameter. We can see that with a value of 0.9 (high importance of the previous gradient directions) the efficiency is way bigger. This result is just an observation and needs to be nuanced, it is just better for this dataset.

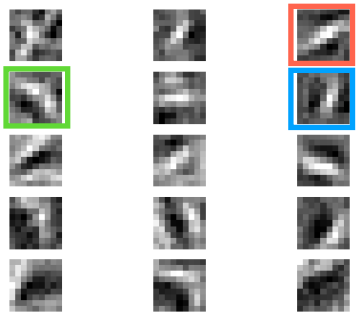


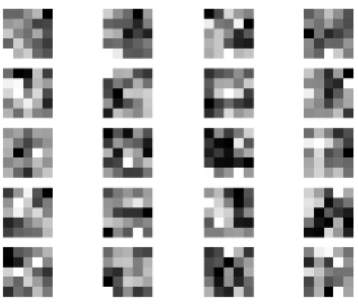
Test accuracies for GD (blue) and GD with momentum (red). 12000 samples are used for all trainings and 2000 for testing. The left figure corresponds to a momentum of 0.1 and 0.9 for the right one.

About the features maps at the convolution and at the pooling they are very similar to what we obtained with the gradient descent. It makes sense because the momentum is mainly use to fasten the convergence, not to have that much better accuracies. Moreover the accuracy being very high for both techniques and the architecture being also the same the differences are really small.

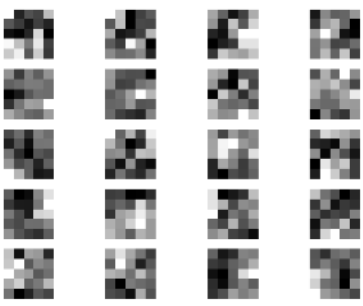


If we compare the weights obtained we can notice that almost the same filters are realised. Quite smoother filters are obtained with the momentum that shows a better efficiency (less noisy).





The features maps are displayed in the annexe in order to make the report more readable.



3- Mini batch gradient descent with RMSProp

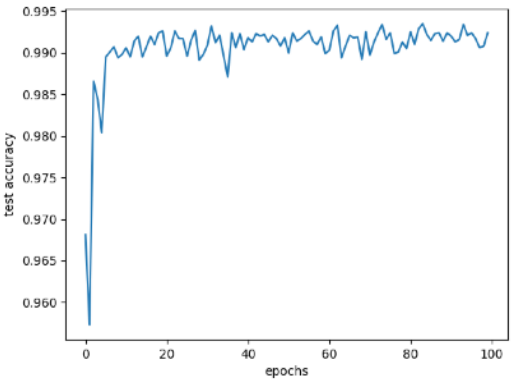
An other algorithm used for learning is RMSProp. These are the equations for learning:

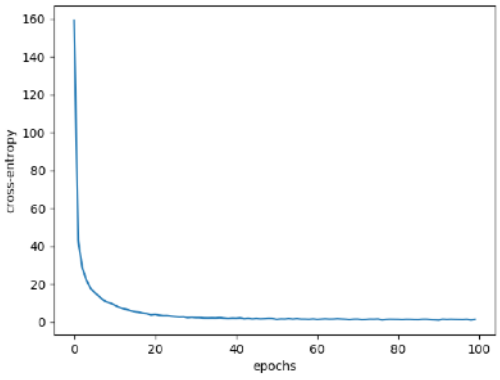
$$r=\rho r+(1-\rho)(\frac{\partial J}{\partial W})^2$$
$$W=W-\frac{\alpha}{\sqrt{\varepsilon+r}}\cdot(\frac{\partial J}{\partial W})$$

We can see that it has the same idea the momentum in the way that we are taking in account the past gradients. This value will be used to modify the learning rate as annealing but according to the square gradient and not only the index of the iteration. This allows to have a low learning rate when the recent gradients are really high and a high one when the average is low to avoid respectively the weights to explode or vanish. This is a recurrent problem in deep networks that limit the efficiency of the network even when we have a decay or ReLU activations that should also limit this phenomenon.

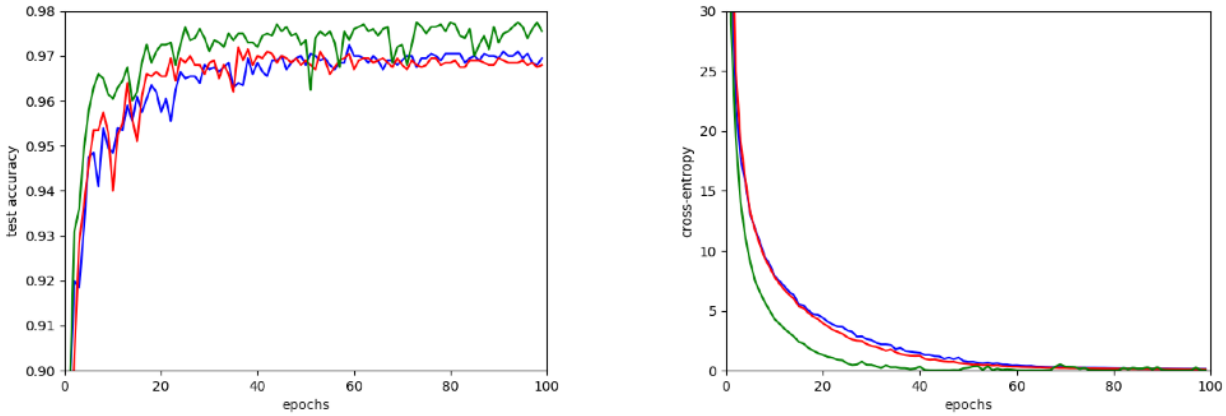
For this algorithm the parameters need to be determined empirically (even if some standards appeared for some architectures) :
batch size = 128
learning rate = 0.001
decay parameter = 10^{-4}
rho=0.9
epsilon = 10^{-6}
full dataset for training and testing

Program to run: project2a_3.py





The performances are really good with a very high accuracy (around 0.993 for the best epochs). The convergence is also quite quick even if we can observe big jumps in the accuracy at the beginning. That may be due to the penalisation of big variations in the training for the weights according to the training equations. Now let's compare it with the previous experiments, using a smaller part of the dataset as before.

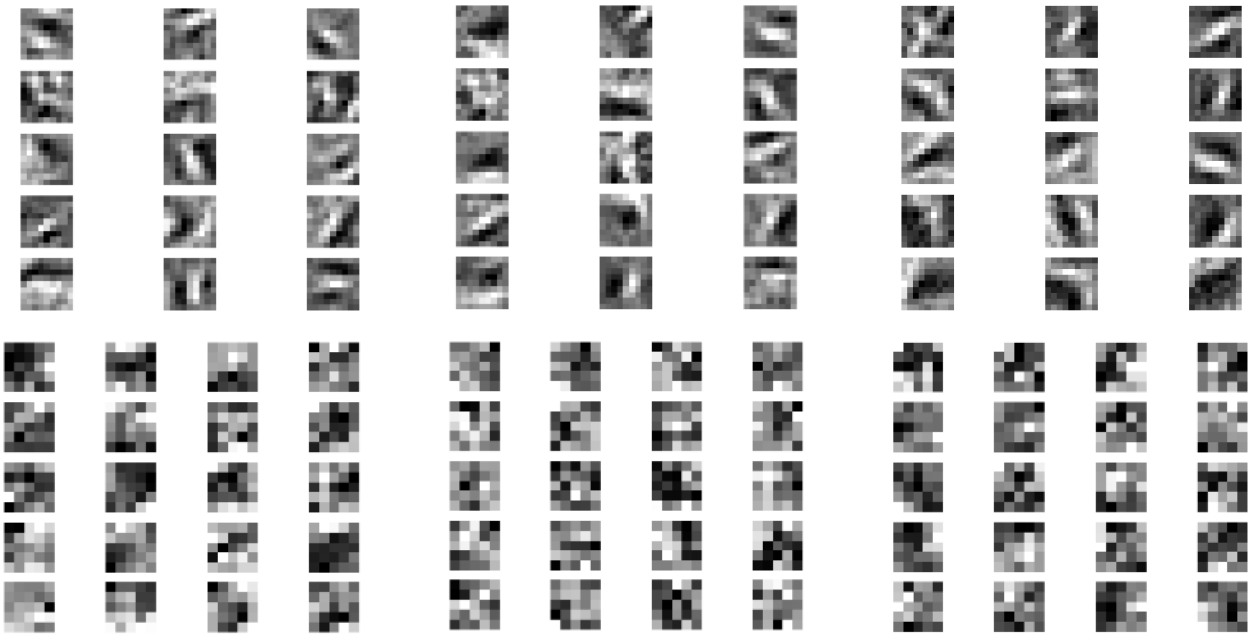


Test accuracies for GD (blue), GD with momentum (red) and RMSProp (green). 12000 samples are used for all trainings and 2000 for testing.

We can see on the figures above that RMSProp performs way better than other algorithms with a faster and higher convergence. Moreover the parameters for the trainings and algorithms were given for all the experiments and maybe with other tuned values it would have perform even better.

About the features maps (available in the annexe) the comments are the same as before, we can't really see differences. We can also see a lot of similarities between the weights obtained but it makes sense for the reasons explained before on the GD with momentum. However we can see that for RMSProp the weights seem smoother with more grey pixels to transit from white ones to blacks ones and that is a sign of a good generalisation. This is surely do to the objective of RMSProp to avoid certains weights to explode and vanish, making the average smoother. This is why we have for instance less totally black pixels with RMSProp and more dark grey ones (noticeable if we compare the weights of RMSProp and GD).

Weights learned for, from left to right, RMSProp, GD with momentum and GD on the entire dataset.



Part B : Autoencoders

An other type of neural network is the autoencoder. The principle is to give the possibility to the network to extract or find characteristic features of the input with a basic fully connected architecture. To train an autoencoder (one hidden layer) we gave him the input with some random noise and we fix as output the input without noise. The error is then the difference between the reconstructed input and the clean input. By doing this we want the network to extract the features of the input necessary to rebuild it without loosing informations. It can then be used to remove the noise from an image or find an other way to represent the input in an other dimension. It can be useful for data compression for example. We can also designed stacked autoencoder by adding some layers trained separately to reconstruct the initial input with the output of the precedent layer. Then we can use those layers as hidden layers for a classification network by adding an output layer like softmax. The convergence will be fast with the pre-trained layers of the autoencoder because they already extract characteristic features of the input. An other point is that the training of autoencoder is unsupervised and doesn't require labelled data (that is sometimes hard to obtain). It is then very attractive to build a classification model if we have a lot of data but just few labelled. Some constraints can also be added to make models not only designed to reconstruct the output but also to learn more about the data distribution like regularized or sparse autoencoders.

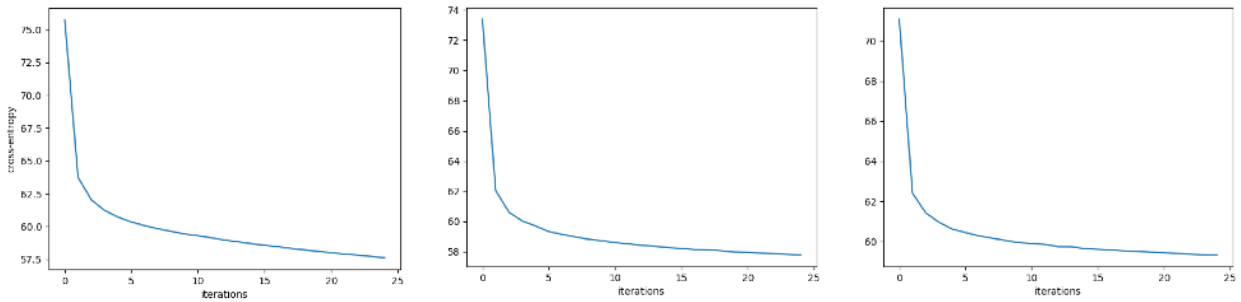
1- Stacked denoising autoencoder

The denoising autoencoder we will design in this part is composed of 3 layers of respectively 900, 625 and 400 neurons. We train each layer separately, looking for their ability to reconstruct the images given the outputs of the past layer. For each of them the batch size will be 128, the learning rate 0.1 and the activation function a sigmoid. We will do 25 epochs using all the dataset to train.

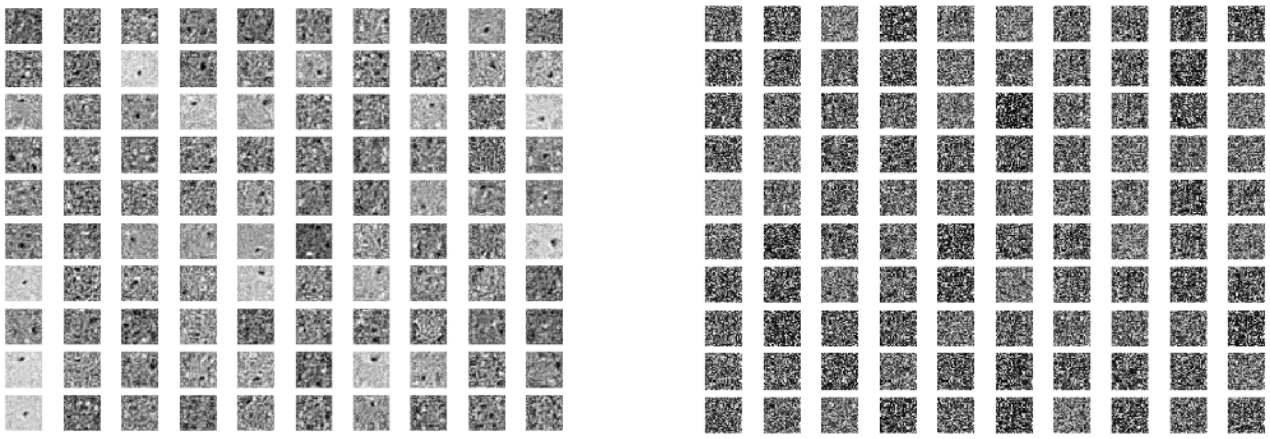
For the first layer we will apply some noise at the inputs using a binomial distribution with a probability of 0.1. This way we will learn the first weights (28*28 to 900 and 900 to 28*28 that is the transpose of the first one) with the associate biases. Then to train the second one we want to learn the weights 900 to 625 (and by the same occasion 625 to 900) and the biases. To do so we use at input the output of the first layer (for the noisy images) and then compute the loss on the reconstructed images by just updating the weights of the second layer. We will use the same principle for the third layer.

You can find just below the results of training for all layers.

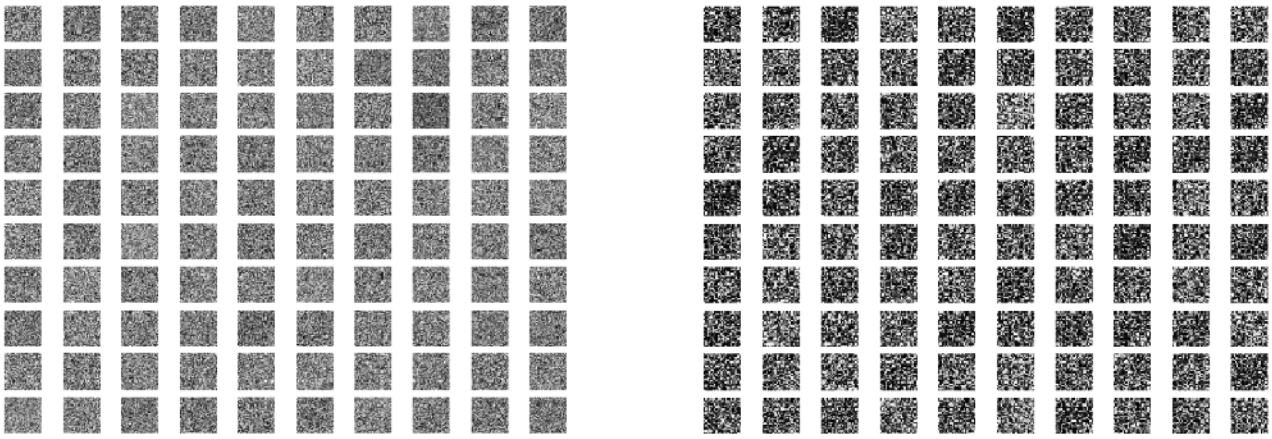
Program to run: project2b_1.py



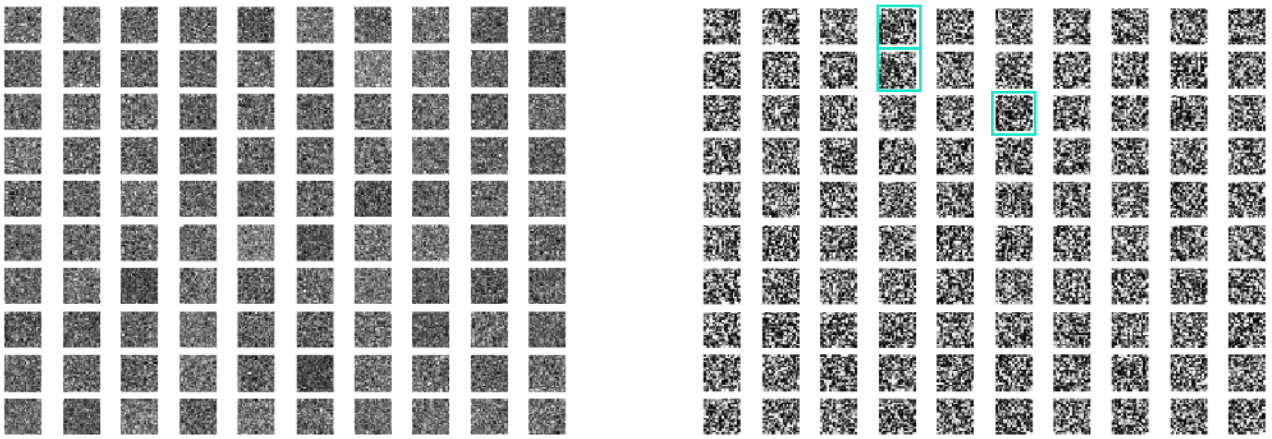
Training cost (cross-entropy) for each layers from 1 to 3 respectively.



First layer: weights learned (for 100 neurons) and activation for 100 images.

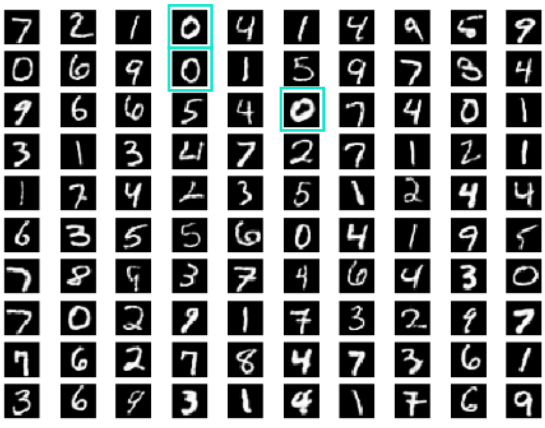


Second layer: weights learned (for 100 neurons) and activation for 100 images.



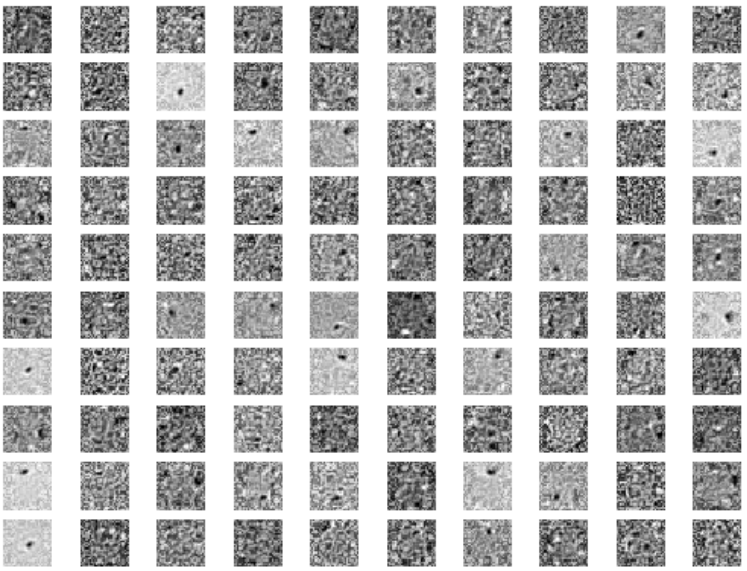
Third layer: weights learned (for 100 neurons) and activation for 100 images.

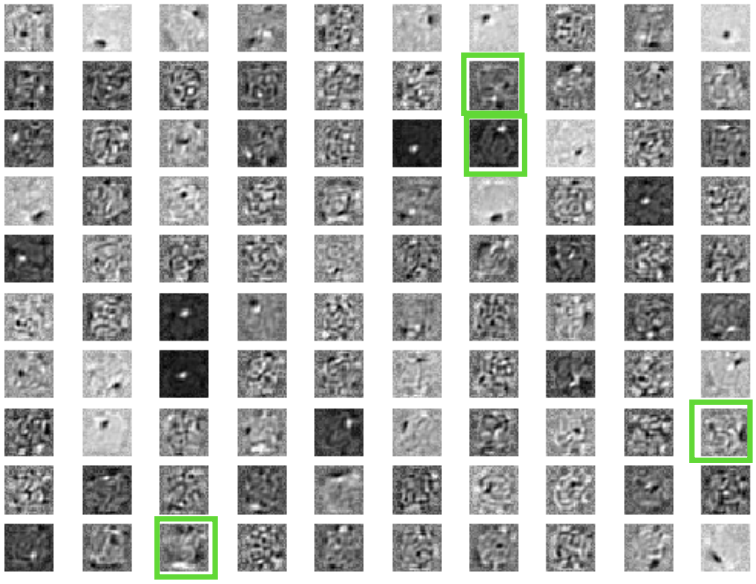
Reconstructed images :



The first thing we can notice all the costs converge and all to almost the same value (around 60). We observe just a tiny augmentation from the first layer to the second and third one. This means that to denoise the input we shouldn't use 3 layers but only the first one (for this dataset). However the fact that we have almost the same convergence also means that from the training of the first layer to the full stacked DAE (Denoising autoencoder) we almost don't loose any information even if we lower the number of neurons. Moreover the reconstructed outputs are very close to the the input samples so it does the job pretty well.

The interpretation of the weights is very hard. Indeed, we just display the weights as images for some of the neurons and those one are quite noisy. This may be due an overcomplete DAE as a first layer that will add some informations that can be interpreted as noise. Still we can see that the neurons seem to respond according to some patterns with some important black and white points (mainly on the first layer). We can see below the differences with a undercomplete DAE (right figure, 169 neurons in the layer) where we can even clearly read some numbers in the weights (green frames) and see some patterns that we can imagine useful to represent the data.



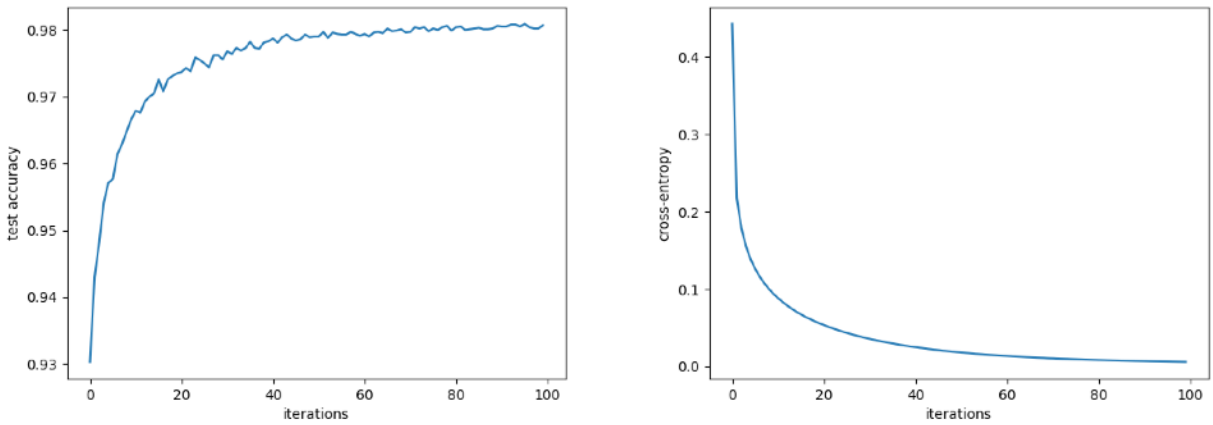


About the activations on all the layer the observations are also difficult to explain. We can just say that for a human it is really hard to see how the information is kept on those representations, specially on the first two layers. The blue frames on the activation of the third layer (see last page) are quite similar and we can see that they all correspond to zeros in the reconstructed images but the correlation between same numbers is still hard to find.

2- Five-layer feedforward neural network pre-trained with stacked DAE

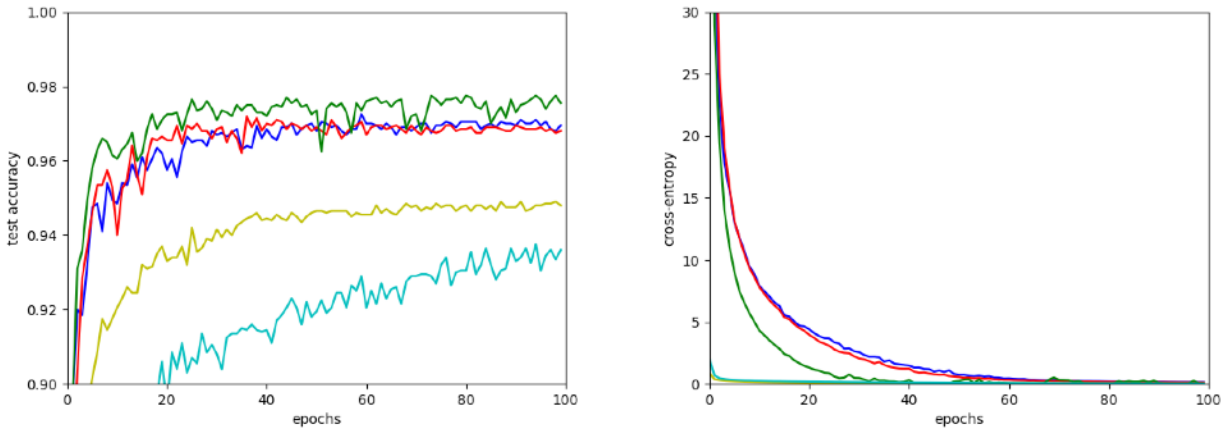
Now that we know that we can keep almost all the information characteristic of the images in the 3 layers of our DAE we can try to use them as hidden layers of a recognition network. Indeed, adding just a softmax layer as output and doing some training to update all the parameters of the network should lead to a very fast training, the weights of the hidden layers being already trained to extract important features of the images.

Program to run: project2b_2.py



Accuracy and error for the five-layer network trained on the whole dataset.

First we can see that the learning is very smooth and begins, as we guessed, at a very high accuracy and a low error. This means that the use of the layers of our DAE is very efficient to help the training.



Test accuracy and error rate agains epochs for CNN with GD (blue), GD with momentum (red), RMSProp (green), five fully connected layer with autoencoder pre-training (yellow) and without (cyan).

We can see above the comparison with the others networks (when we train with only 12000 samples). The first thing we can say is that the CNN perform the best with a really quick and high convergence on the accuracy. About the cross entropy error there

is an issue that is the calculation for it wasn't the same for both parts so we can not really exploit it (for the CNN we didn't divide it by the number of batch in an epoch). Anyway we can still see that all converge. Also to state on the efficiency of our pre-trained model we needed to show the results if we just use the same network but without the layer of the stacked autoencoder. We can see that the accuracy is way better with those layers with a faster and smoother convergence. If we compare the weights obtained we can also see that the network without pretaining doesn't seem to really extract features, the weights being very noisy and uniform. As we said before, the autoencoder based network has a good accuracy even if low supervised training is done and that is not the case for the others so it is a really big advantage.

3- Five-layer feedforward neural network pre-trained with stacked DAE with momentum and sparsity constraint

Because our autoencoder is overcomplete at the first layer it could just have make the identity function as his activation and that is not what we want, we want it to extract specific features. Moreover, using a lot of neurons can allow the network to learn more features but those ones risk to be very correlated if we just increase the number of neurons. The idea of the sparsity constraint is to restrict the activation output of the hidden neurons in order to lower the dependencies between the features. By doing this it should increase the number of relevant features.

To do so we add a penalty term to the cost function of the hidden units that forces the average activation of a layer to be really low.

$$J_l(W,b,b')=J(W,b,b')+\beta D(H)$$

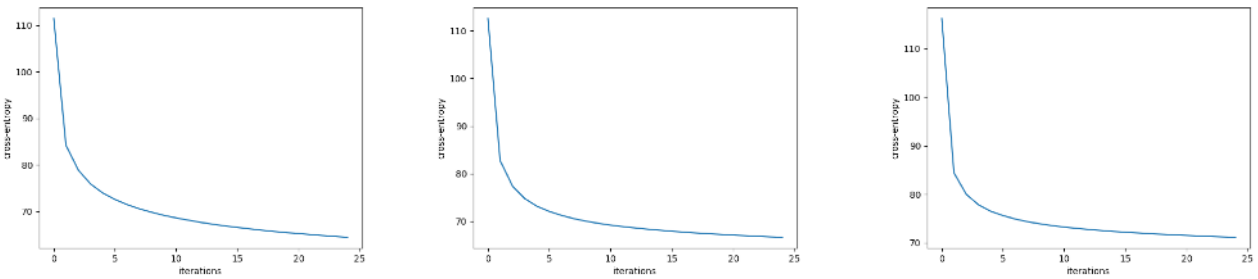
$$D(H)=\sum_{j=1}^{M}\rho\log\frac{\rho}{\rho_j}+(1-\rho)\log\frac{1-\rho}{1-\rho_j}$$

ρ=*target activation*
ρ_j=*activation of the* j^{th} *neuron*
β=*penalty term*

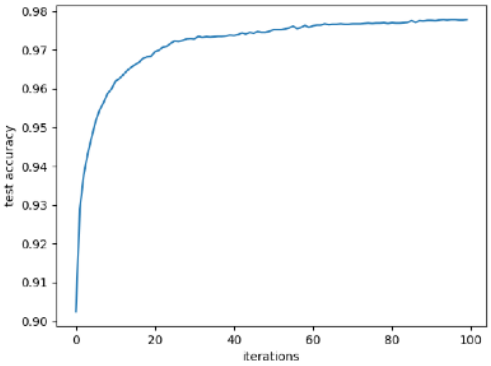
We can see with the formula that it will force all the neurons (the M neurons) of each layer to have an average activation very low. In this question we used 0.05 as the target activation (sparsity parameter) and 0.5 for the penalty term. We also add the momentum algorithms to speed up the learning with a parameter at 0.1. Below are the learning curves obtained for the training on all the dataset.

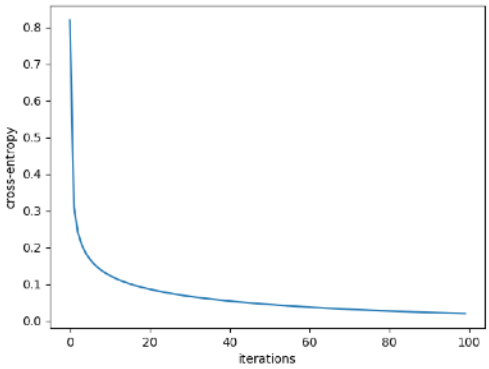
Program to run: project2b_3.py

First let's see the results of the training of the stacked autoencoder.

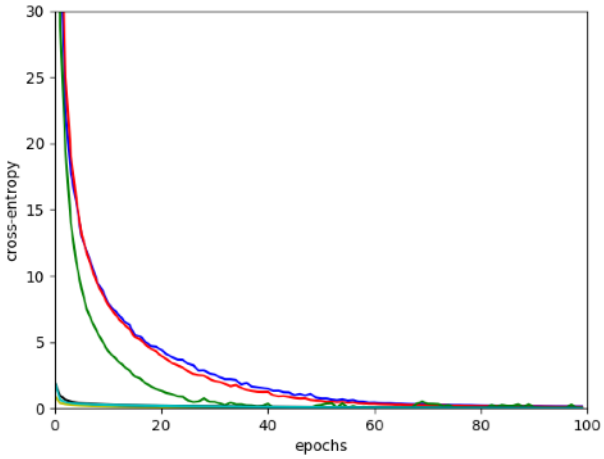


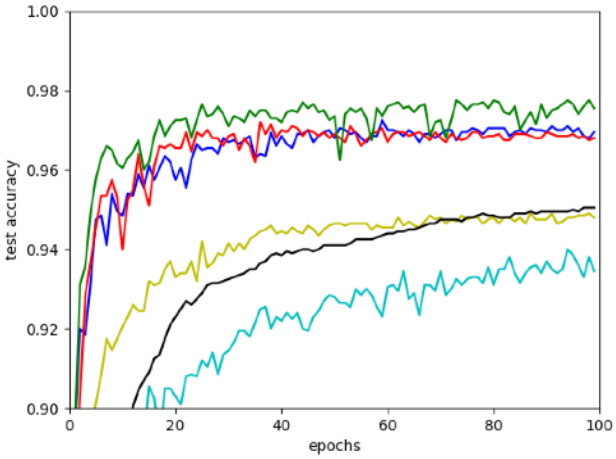
The first thing we observe is that maybe we could have pushed the training a bit more but it took too much time on the whole dataset. However the convergence is almost obtained with a cost of about 66, 68 and 72 for the layers 1, 2 and 3 respectively. We can see that is a bit above the results from first part (around 60). However, the results for the classification network are very good (see below) : high accuracy since the first epoch, smooth convergence and a very high accuracy at the end. We can observe that the accuracy is still increasing and as I said I should have trained a bit more.





We can also compare it to the other method used in this project (with only 12000 samples to train).

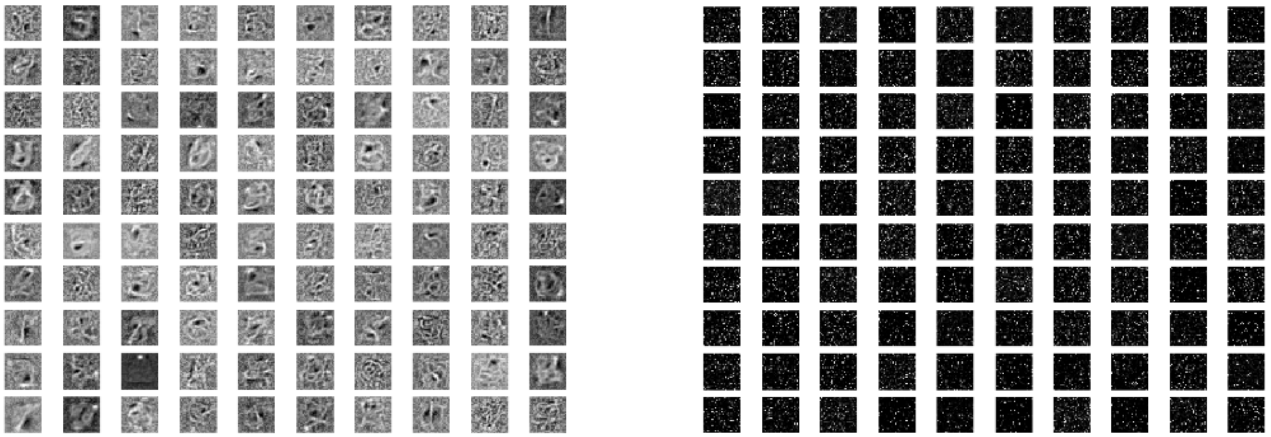




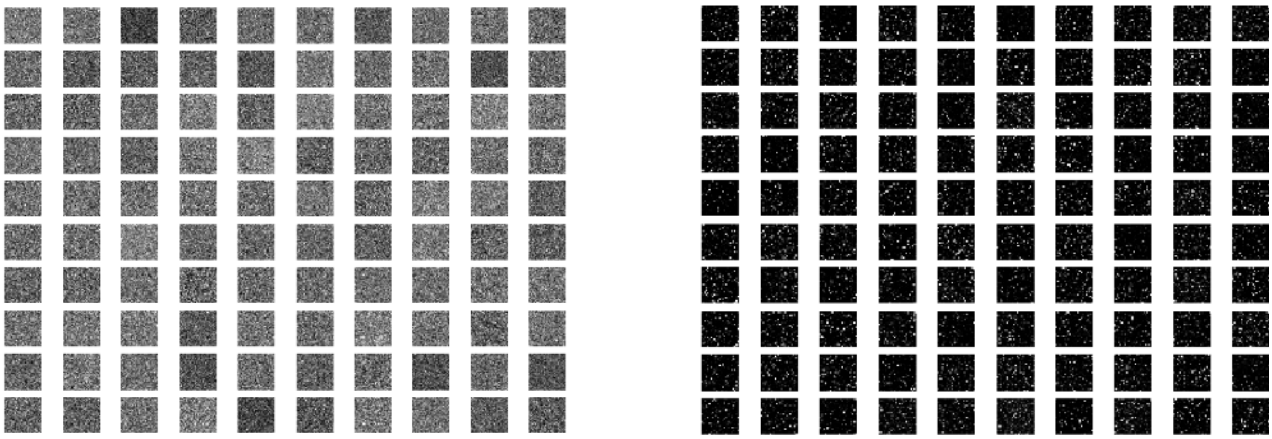
Test accuracy and error rate agains epochs for CNN with GD (blue), GD with momentum (red), RMSProp (green), five fully connected layer with autoencoder pre-training with gradient descent, sparsity constraint and momentum (black) just gradient descent (yellow) and without pretraining (cyan).

We can see that our pre-training with sparsity constraint leads to a better accuracy (that is still increasing) that without. It is also noticeable that the sparsity slows down the convergence a bit because even with the momentum it is slower than without the sparsity. Finally the accuracy curve is very smooth. However, the convolutional networks perform better on this classification task.

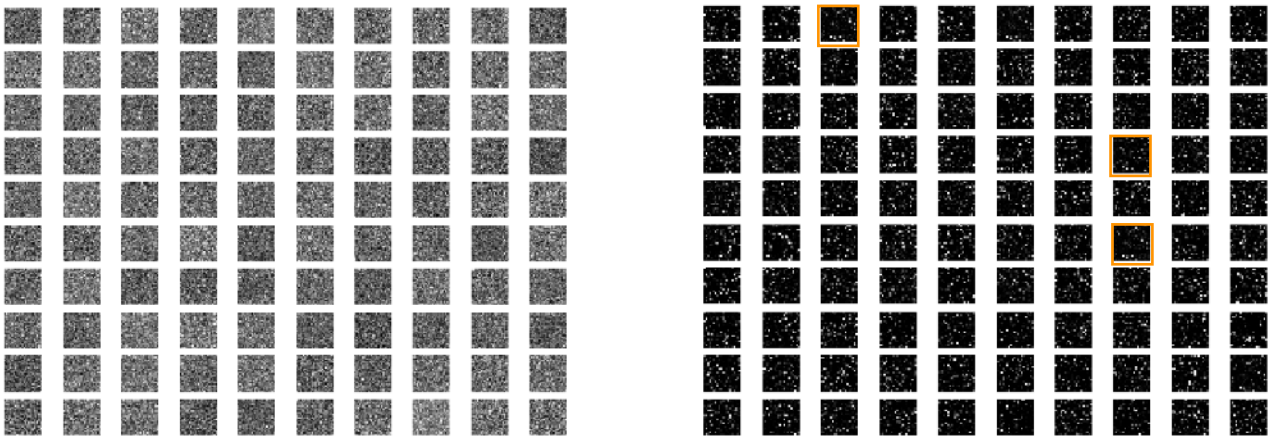
Now that we saw the performances we can move to the representation of the weights, the activations and the reconstructed inputs to see the impact of the sparsity.



First layer: weights learned (for 100 neurons) and activation for 100 images.

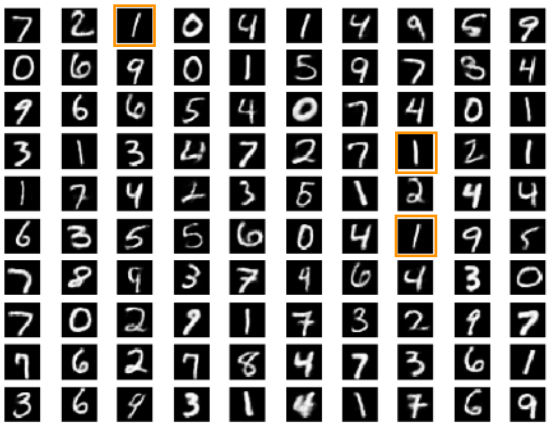


Second layer: weights learned (for 100 neurons) and activation for 100 images.

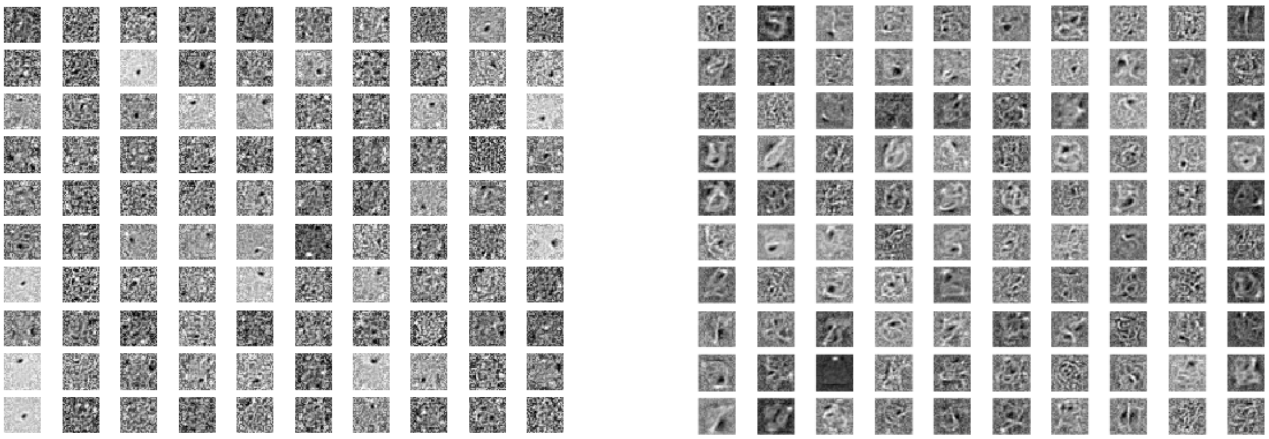


Third layer: weights learned (for 100 neurons) and activation for 100 images.

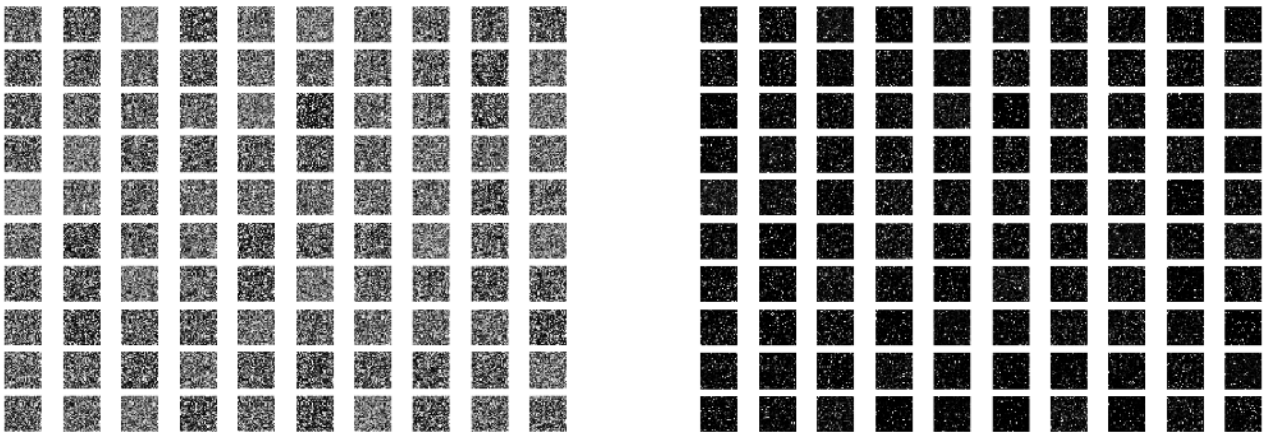
Reconstructed images :



First in both case we obviously manage to reconstruct the inputs images very well. Then if we compare the weights obtained we can clearly see that the sparsity leads to way more precise weights learned for each neuron that we can easily interpret.



Weights without (left) and with (right) sparsity constraint for the first layer.
The corresponding activations are below.



The activations also differ a lot. We can directly see the sparsity of the activation with only few neurons active. It is mainly due to the fact that with the sparsity the neurons will be more specialized to respond only to some particular inputs in order to keep the mean activation low. It is then also easier to recognize some characteristic activations for some digits, especially on the third layer (see last page, orange squares). To conclude on this part we can say that the sparsity brings more specialization for the neurons that may slower the learning but also increase the efficiency of the network with better features extraction.

Conclusion

During this project we went through the convolutional neural networks and the autoencoders with the MNIST dataset.

We saw the efficiency of the first one to solve some image classification problems with its ability to extract features and combine them to differentiate classes. We also explored some algorithms designed to improve the training. It is interesting to see how the topic of object recognition evolves and improves so quickly with new neural networks algorithms and architectures. Recently a new paper was published by the well-know Geoffrey Hinton about a new type of network, the capsule network, that outperforms all the past models on the MNIST dataset. It fixes one of the problem of the convolutional neural networks that is that they are really sensible to variations in the orientation of the scene. In fact with a new orientation the filters will not respond in the same way and after the pooling layers the treatment will be completely different. We will not talk more about this new model but the basic idea is that the research in computer vision is far from its end and there is no doubt that more improvements will be done in the future thanks to neural networks.

The autoencoders are also very interesting with their ability to extract relevant features of the data to express them in different dimensions. As we said it is a very powerful unsupervised type of neural network that can also be use to pre-train a model like we did. However we also saw that some constraints like sparsity need to be add in order to improve its efficiency.

This project was also the occasion to train the networks on larger datasets and imagine the power needed for very big datasets and deep architectures.

Annexe

1- Features maps for gradient descent with momentum

2- Features maps for RMSProp

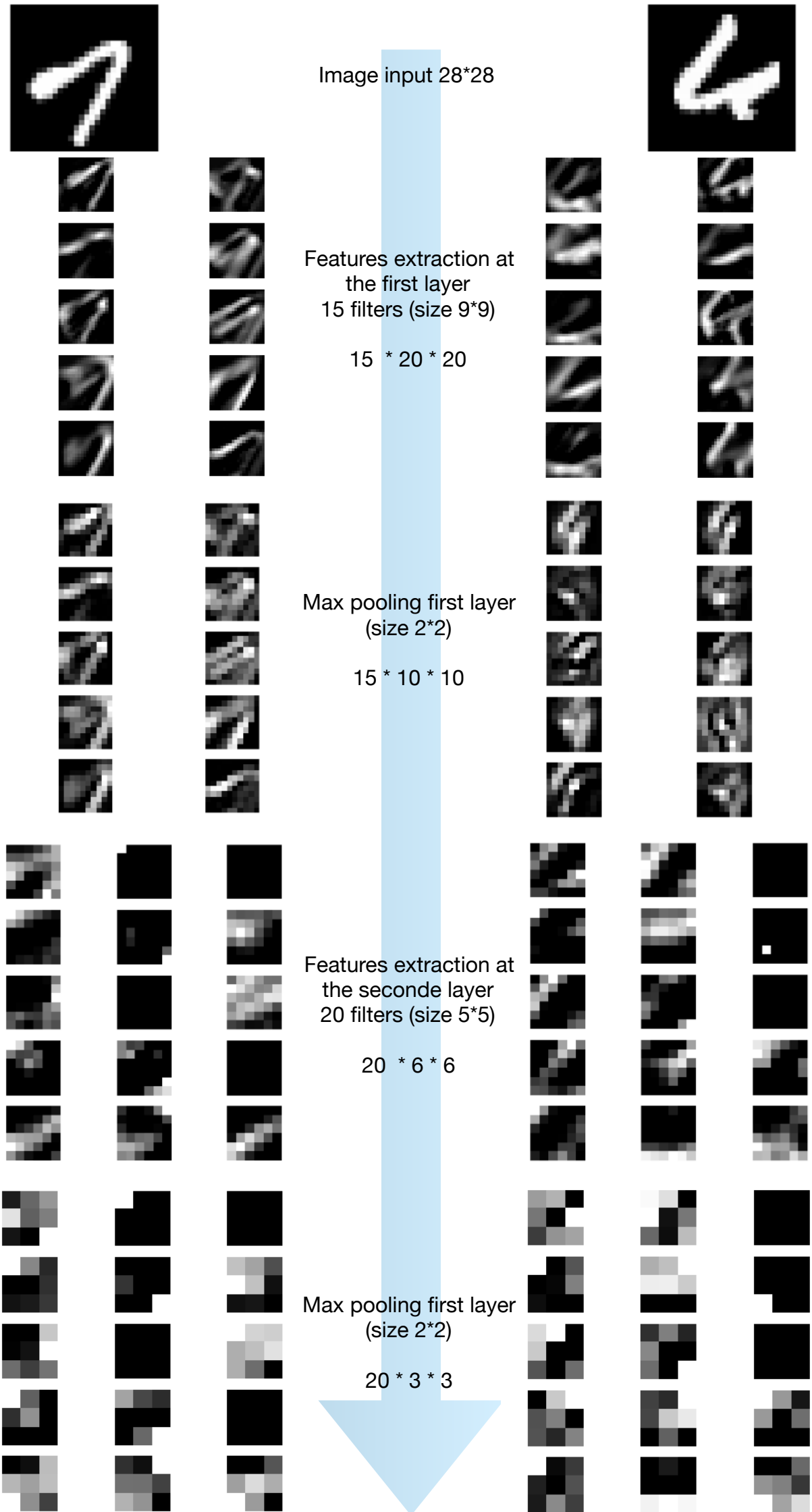


Image input 28*28
Features extraction at
the first layer
15 filters (size 9*9)
15 * 20 * 20
Max pooling first layer
(size 2*2)
15 * 10 * 10
Features extraction at
the seconde layer
20 filters (size 5*5)
20 * 6 * 6
Max pooling first layer
(size 2*2)
20 * 3 * 3

Image input 28*28

Features extraction at
the first layer
15 filters (size 9*9)

15 * 20 * 20

Max pooling first layer
(size 2*2)

15 * 10 * 10

Features extraction at
the seconde layer
20 filters (size 5*5)

20 * 6 * 6

Max pooling first layer
(size 2*2)

20 * 3 * 3