Neural Network and deep learning: homework 4
of Tommaso Tabarelli

Abstract

In this homework we are asked to modify a given python script in order to train an neural network implementing an autoencoder. The goal is to learn main features of the autoencoder, understand the main parameters to tune and use the autoencoder itself to try to learn MNIST digits images and to reconstrcut them after having corrupted them.

Code understanding

First of all we had to undestand how the code works and how the Autoencoder model is implemented using pytorch. Moreover, we had to understand how the data is loaded, structured (in the given script a *thorcvision dataset* is used) and how to handle it. We were also supposed to understand how the train process is implemented and how to use it to train other networks and try to reconstruct images. During training, the use of tecniques to prevent overfitting is suggested.

Code development

My choice was to work with jupyter notebook since I find it more familiar and cleaner; anyway a python script was finally developed as homework requests.

Data is loaded in two ways depending on where it is used: in the training file (jupyter notebook) the data is loaded (dowloaded and then loaded from local file) as in the given example, while in the python script *trained_model.py* the data is loaded from a *MNIST.mat* file containing the corrupted images.

There are two kind of data loaded: training data and test data. Training data is then split into training set and validation set; the lenght of the validation set was chosen to be 20% of the lenght of the training set (arbitrary choice); the rest of the training data was used as actual training set.

After having understood the given example, the choice was to modify some parameters and see what effects they have on training procedure and its time duration. To try to avoid overfitting, in the training procedure an early stopping test was implemented; it monitors the validation error at every epoch and stop the training if the last validation error is larger than the average of the previous *patience* errors, where patience is a parameter chosen by the user (my choice was to start with *patience*=5 and see if it gives a good result according to the validation error plot; they seemed not *good*, so I decided to change the patinence value to 10 and the validation error plot seemed better, so for the actual training the choice was to use this last value). Since the number of batches per epoch is large (they are 48000/500 = 96) this time the use of a *K-fold cross validation* tecnique was avoided due to the huge amount of time to train when looking for the best parameters set: a simple train-validation split is used.

The parameter changes were focusing on the number of hidden units, the encoded dimension, the learning rate, the maximum number of epochs, the batch size:

- hidden units: one expects that the larger the number of hidden units the network has, the larger the number of representations it can make of what it "sees" through the convolutional layers. Starting from this assumption, increasing the number of hidden units should increase the complexity of the patterns it can recognize, being able to achieve better performances on the digit reconstructions and/or denoising. Thus, in general, increasing the hidden units should result in

a better performance, but of course it implies that there is a larger number of parameters to be tuned, increasing the training time and risking an easier overfit (with increasing parameters also the number of elements in the training set should be increased).

- encoded dimension: the intuition tells that the larger it is the more different *patterns* the network can learn, in the sense that the network should be able to better recognize the images since it can represent them on a larger and more complex pool of possibilities; as before, this increasing performance leads to a larger training time (and should be supported by a larger train set).

- learning rate: the learning rate affects the weights update speed; larger learning rates make the update faster but more unstable, in the sense that the procedure can explore a larger portion of the loss function domain, but at the same time it can prevent it to stop in a local minimum, thus "ruining" the training.

- maximum number of epochs: increasing the maximum number of epochs generally leaves the train procedure last longer. Anyway, since an *early stopping* procedure has been implemented, this parameter was not considered "important" and was fixed to be quite large (400) so that the train procedure should stop because of the early stopping test.

- batch size: the larger the batch size, the smaller the number of "partial updates" the training procedure acts during a single epoch. With this in mind, this parameter was modified to be 500 (the given value was 512). This choice was made in order to have batches of the same size (the actual train test has size equal to 48000 = 60000*(1-0.8), which is a multiple of 500) and to try to balance the decrement in the number of updates in a single epoch due to the validation split, which decreased the elements in the train set and thus the number of batches.

The parameters were tuned using a grid search on the following parameters choices:

- hidden units: [50, 100, 500]

- encoded space dimension: [2, 4, 6, 8]

- learning rate: [0.1, 0.01, 0.001]

All data, parameters, validation loss plot and other analysis statistics were saved in properly created directories to be eventually retrieved.

To test the *goodness* of the autoencoder some test images (the first ten in my case) were corrupted with different noises or applying different cuts. Considering that the images are represented by *grey-scale* values from 0 to 1, they were corrupted in different ways:

- at beginning a gaussian noise having mean $\mu = 0.5$ and different values of standard deviation σ was applied to the images simply summing it and fixing all eventually saturated values to be 0 if lower than 0 and 1 if larger than 1. This try was very unsuccesful because using $\mu = 0.5$ caused almost half of the images pixels to be *grey* (having indeed grey-scale values around 0.5) and this made the reconstruction very hard for the network, which trained on "black-dominated" images, leading very often to unsatisfactory results.

- the gaussian noise was then modified to have $\mu = 0.0$ so that the "base color" would remain black and the noise emerged as "intuitively expected". Of course, this is not properly a gaussian noise since all the negative contributes are neglected (saturated values are fixed as before). The noise was tested to have σ going from 0.05 to 0.45 with steps of 0.05.

- another kind of corruption was to "cut" the images, thus setting to 0 (black) some portions of them. The cuts were performed in horizontal and vertical directions. The "saved" portions of the images were chosen to be those in the upper-left corner, while the cut portions were those in the right and in the bottom parts of the images. Some combinations of the two cuts were tried having fractional values of the saved portion of 0.5, 0.7, 1.0 (the combination of horizontal cut equal to 1.0 and vertical cut equal to 1.0 was skipped).

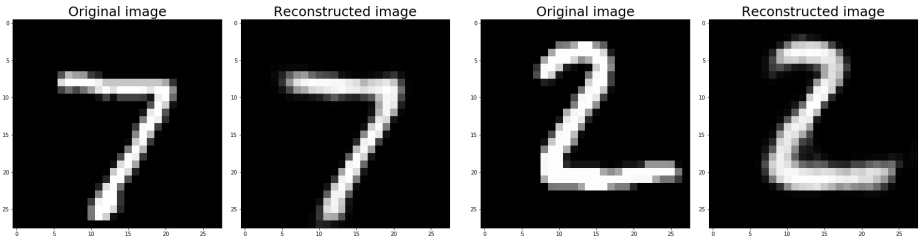
Results

The best newtwork turned out to be the one with:

- hidden units: 500
- encoded space dimension: 8
- learning rate: 0.001

It was then re-trained using the whole train set (thus using all 60000 images in the loaded train set) for a number of epochs corresponding to the trained epochs in grid search +10. This choice was made to try to avoid overfitting in absence of the *early stopping* test (indeed in this case there is no validation set and so an ealy stopping procedure is not possible). Furthermore, the number of epochs were increased because also the number of the elements on which the train procedure was performed was increased; the value 10 is an arbitrary choice.

The results of the network reconstructions are generally quite good, as one can see in the following images.



(a) (b)

Figure 1: Figures (a) and (b) show the first test images reconstructed by the "final" network.

The results on the noisy images as those for the cut images depend on the level of corruption of the images. A thing to notice about reconstruction is that the network works in an almost intuitive way: when there are some cuts that leave the visible part with some features similar to those of other digits, sometimes the network confuses them and during reconstruction the network guesses another digit or a sort of mix of the two (the original one and the one that has the "common feature"): a good example is Fig. (2d) in which the upper part of the digit *0* is confused with the upper section of a *7*, so the network guess for the reconstruction is a sort of mix of the two.

In the appendix the created corruption and cut images are reported (not all of them): they are the first ten images of the *test set* having applied the aforementioned noises and cuts.

Comments

This exercise was useful to undestand how to deal with autoencoders, how to train them and how to use them to try to reconstruct images (simple digit images in our case).

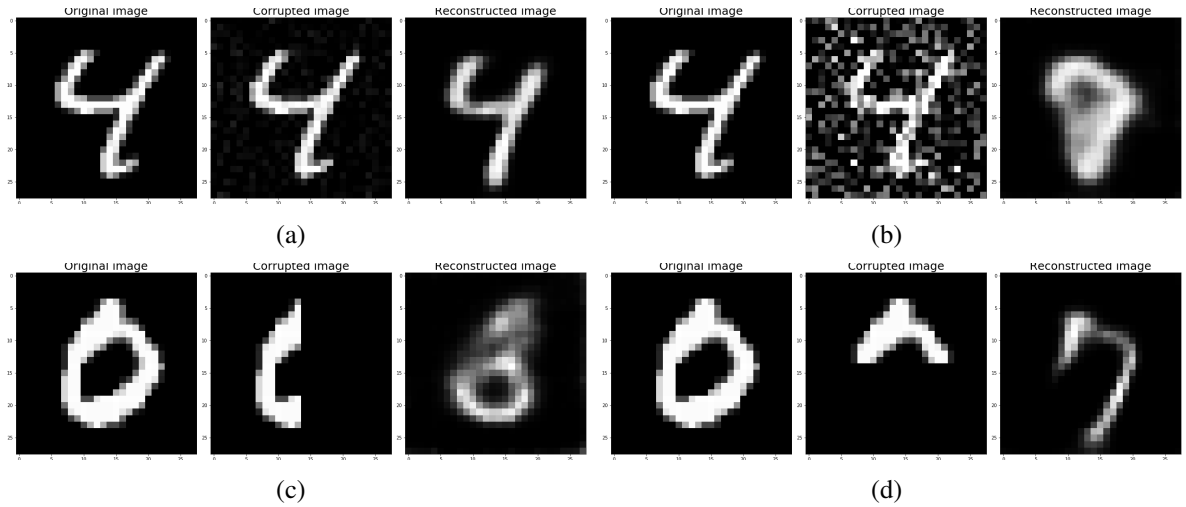
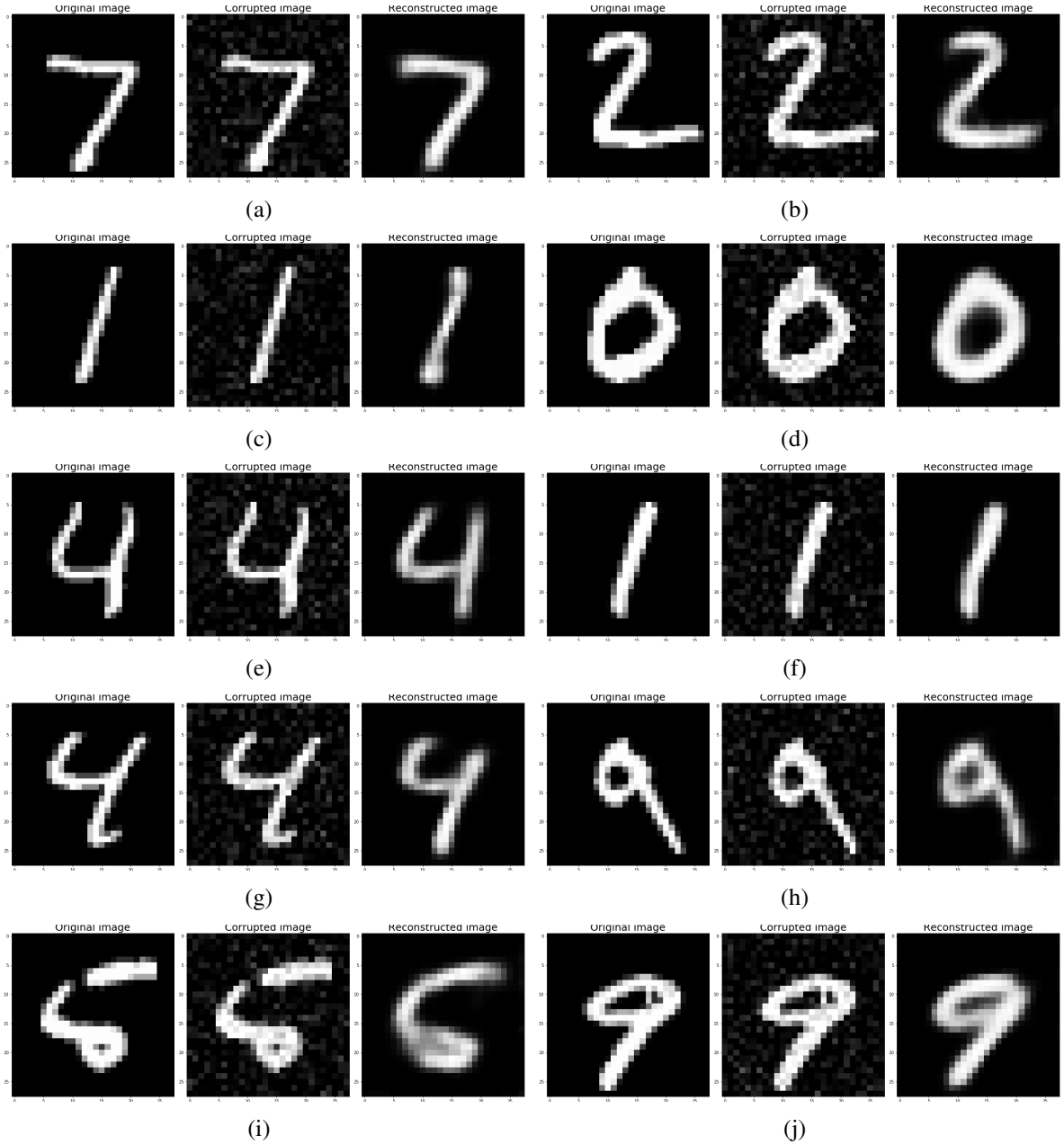


Figure 2: Figures (a) and (b) show a noisy test image with noise having $\sigma = 0.05$ (a) and $\sigma = 0.35$ (b). Figures (c) and (d) show another image corrupted with two different cuts (50% horizontal cut in (c) and 50% vertical cut in (d)) and reconstructed.

Appedix

Here some results are reported.

Noise: $\sigma = 0.10$



(a) (b)

(c) (d)

(e) (f)

(g) (h)

(i) (j)

Figure 3

Noise: $\sigma = 0.20$

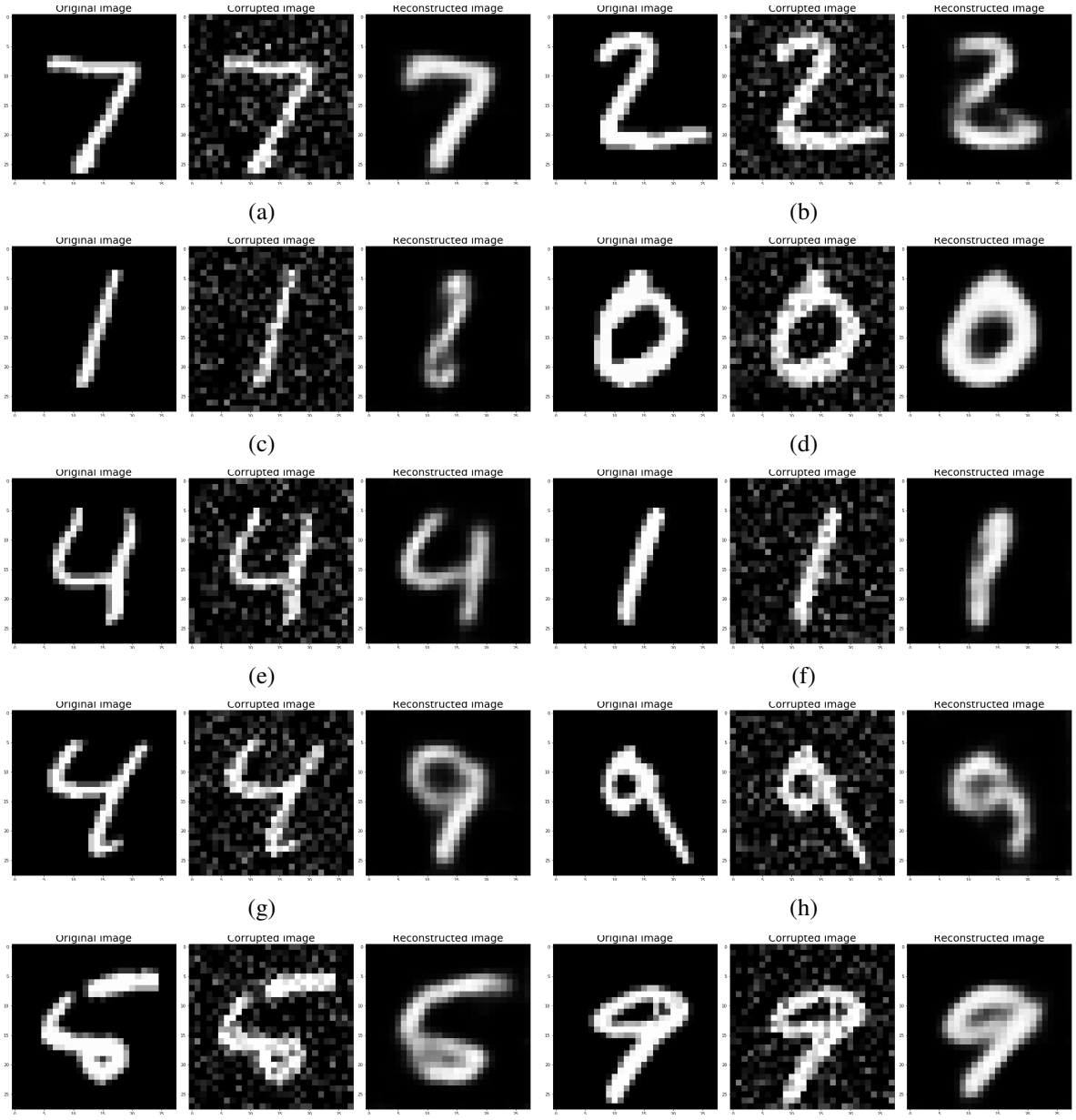
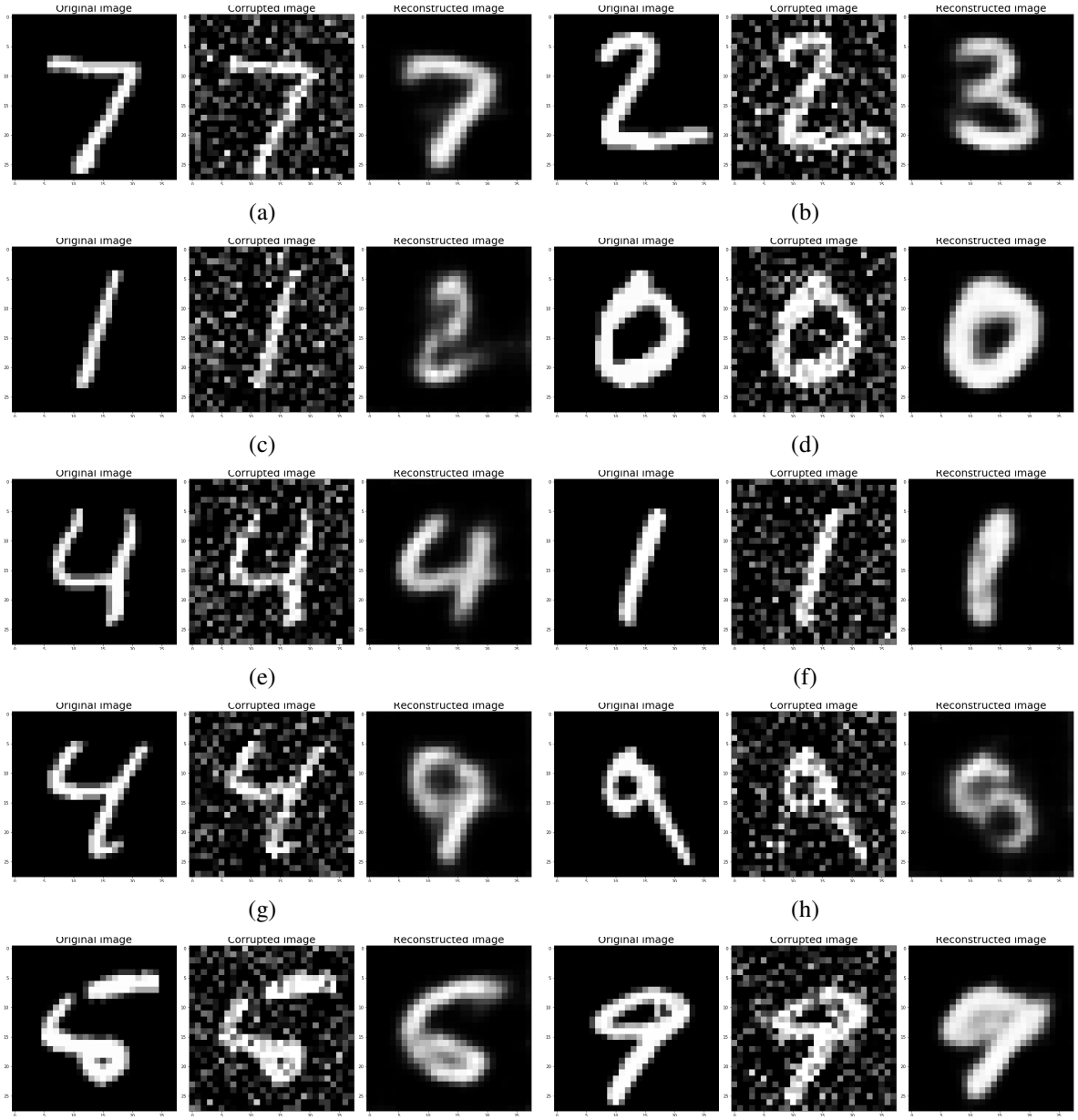


Figure 4

Noise: $\sigma = 0.30$



(a) (b)

(c) (d)

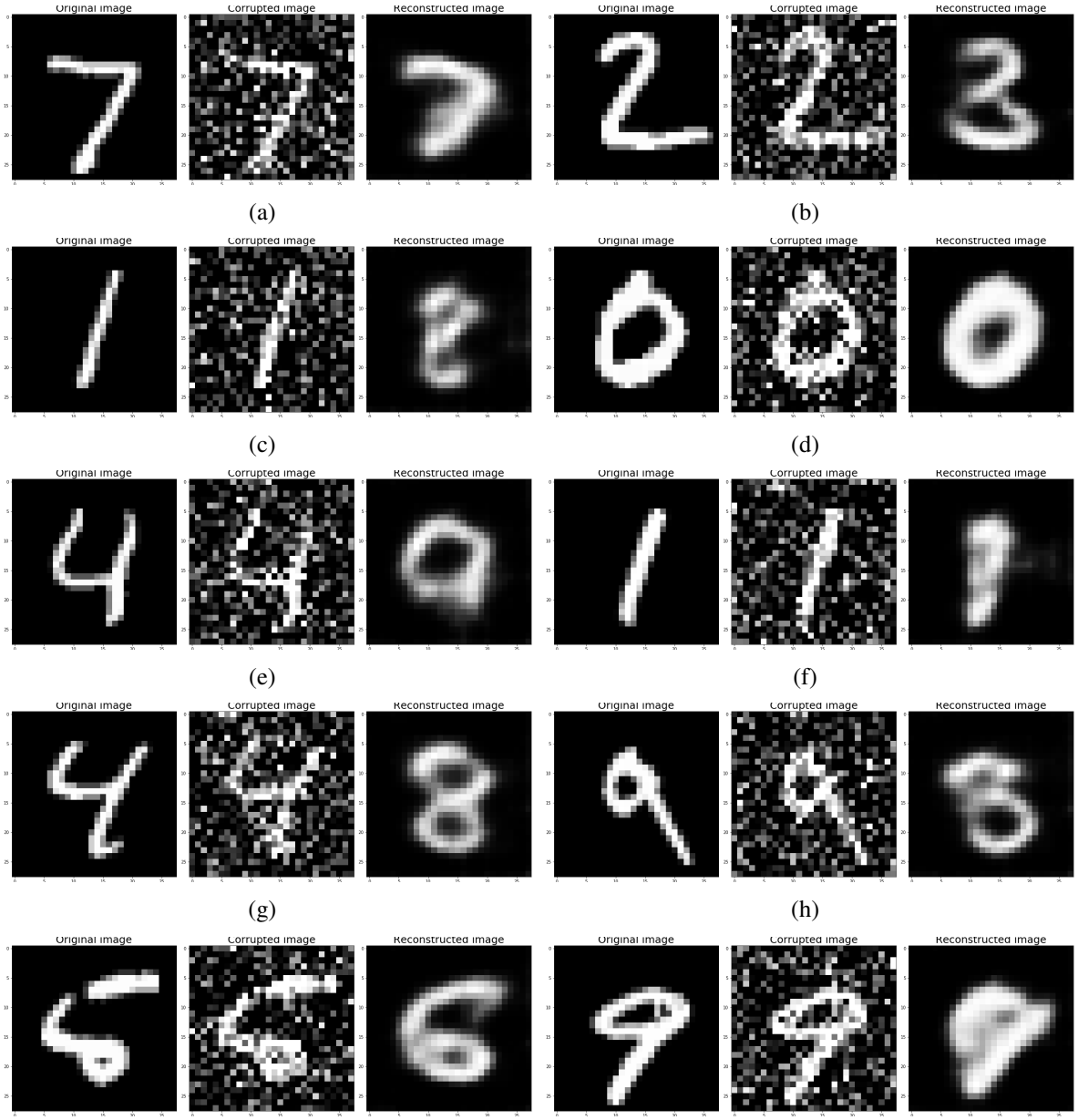
(e) (f)

(g) (h)

(i) (j)

Figure 5

Noise: $\sigma = 0.40$



(a) (b)

(c) (d)

(e) (f)

(g) (h)

(i) (j)

Figure 6

Cut: horizzontal = 1.0, vertical = 0.7

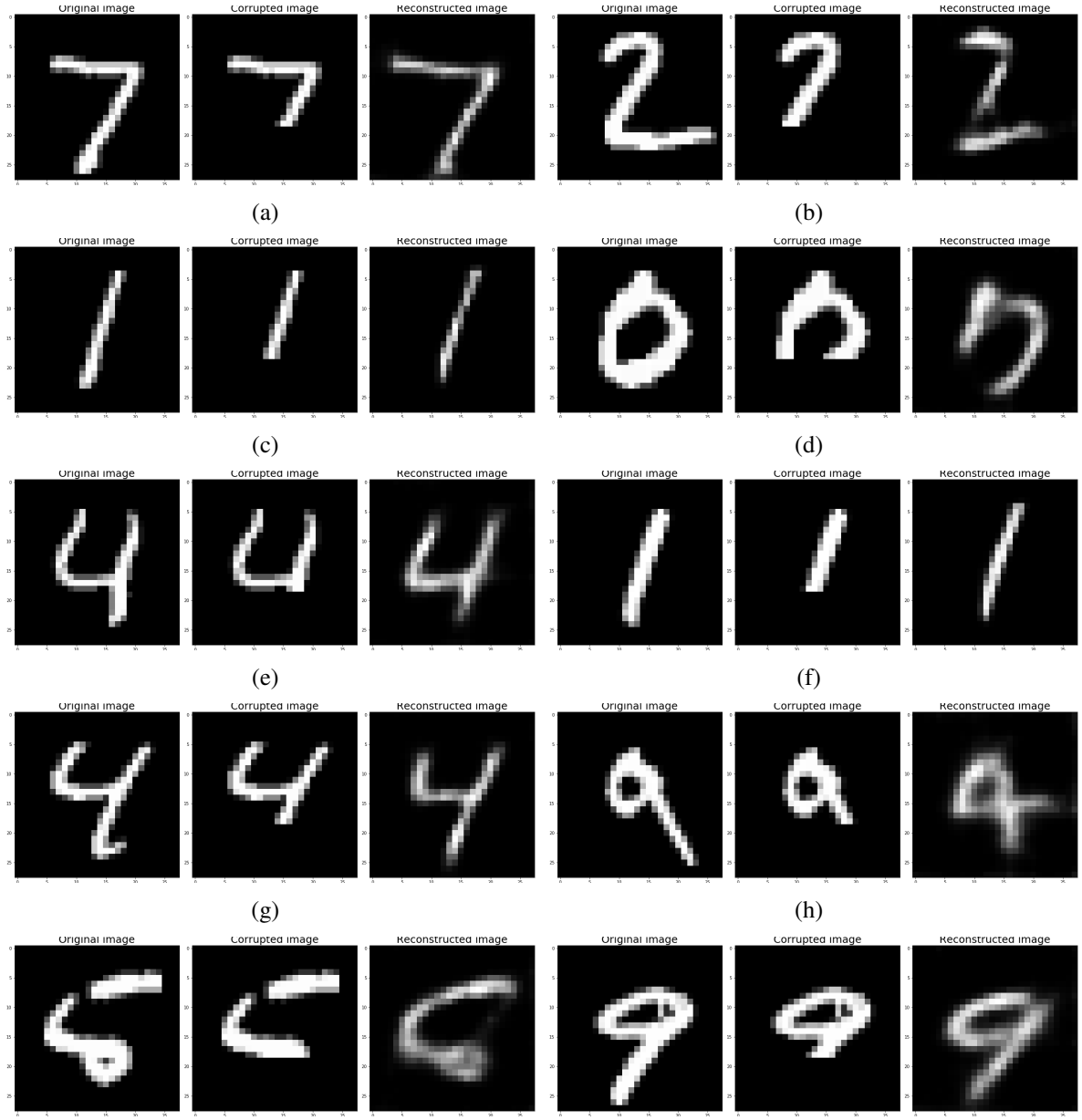


Figure 7

Cut: horizzontal = 0.7, vertical = 1.0

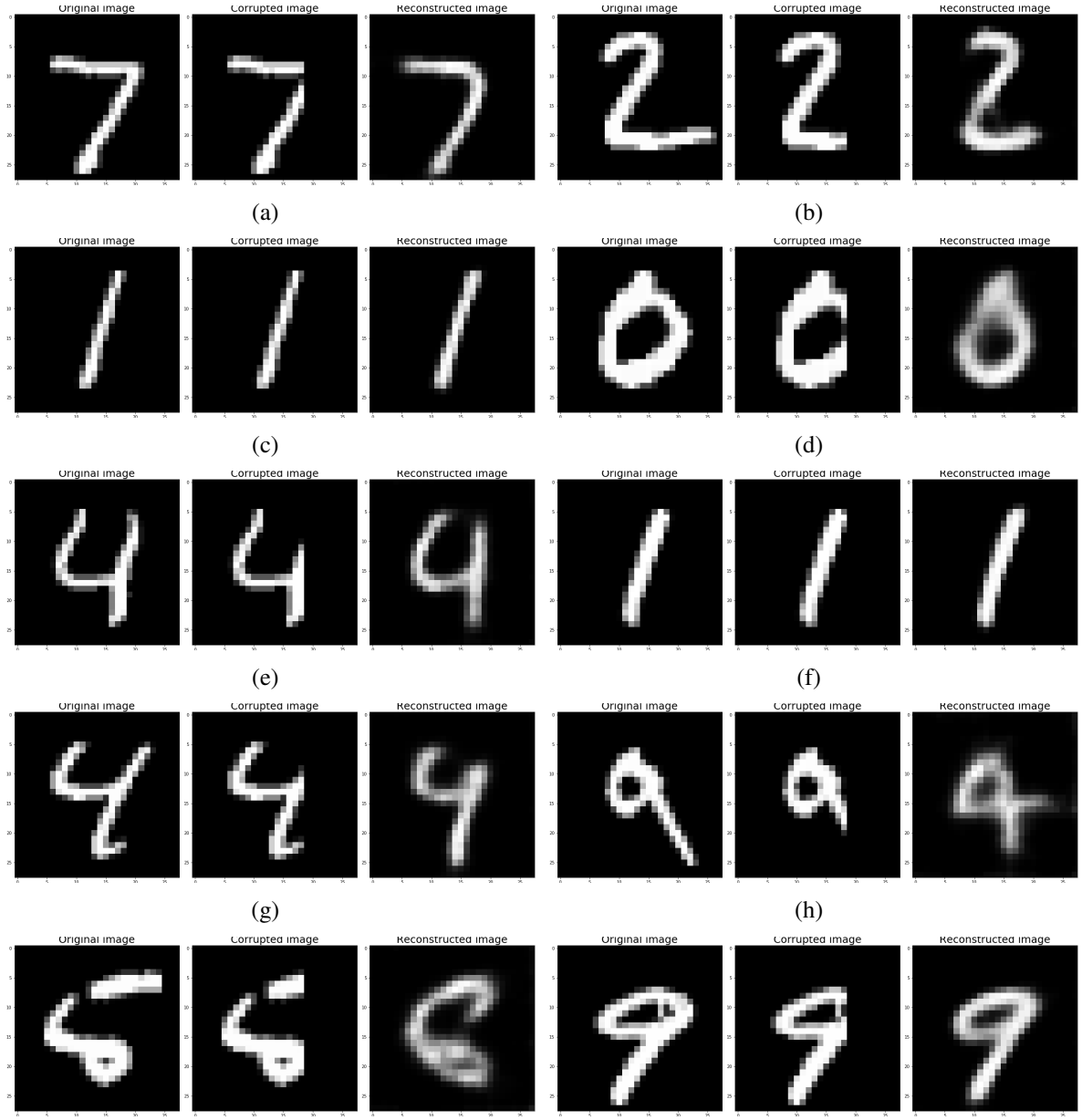


Figure 8

Cut: horizzontal = 0.7, vertical = 0.7

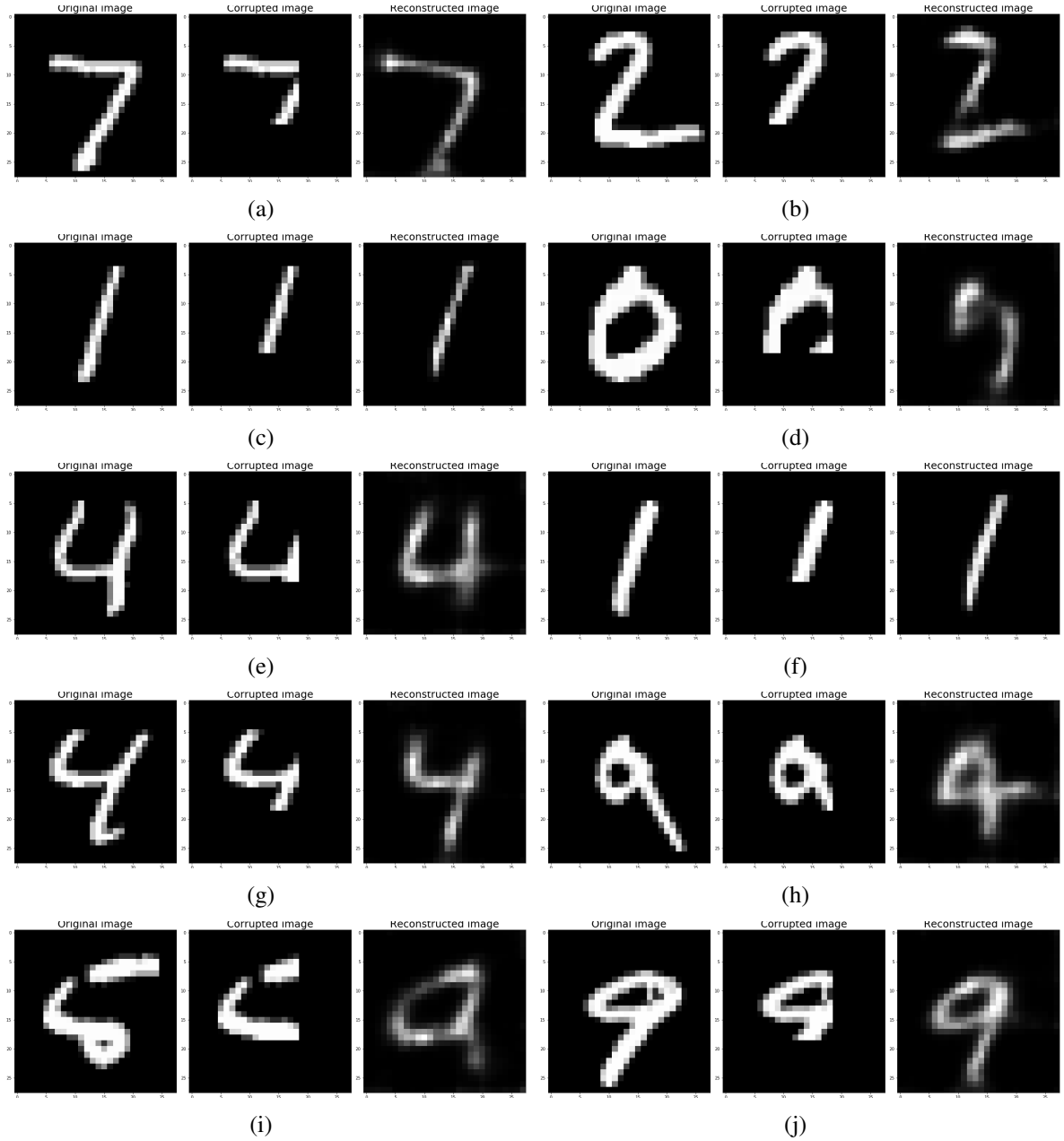


Figure 9

Cut: horizzontal = 0.5, vertical = 0.5

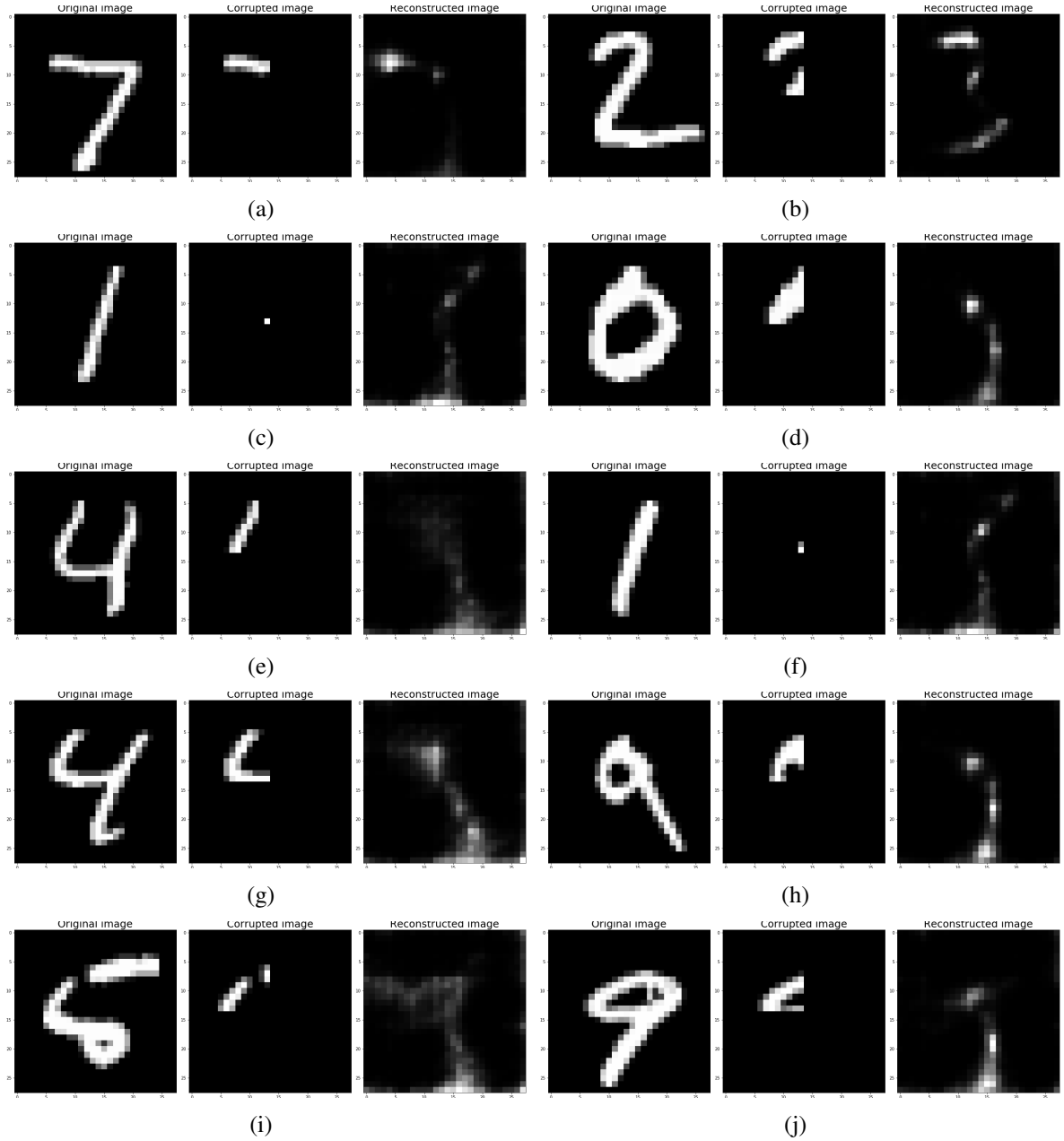


Figure 10