Lab 5 - Autoencoders for digit reconstruction

Vidaich Francesco

1 Introduction

The goal of this assignment is to build an autoencoder to reconstruct the digits of the MNIST dataset, which was imported from the `torchvision.datasets` library. The model is trained and tested using the PyTorch framework, which allows to use GPUs to speed up the computations. BAH The autoencoder structure will be analyzed to understand how the size of the hidden layer impacts the performance of the model. Moreover, corrupted data will be used to test the model under different conditions, and a new one will be developed in order to denoise input data efficiently. In the end we will briefly explore the generative capabilities of autoencoders.

2 Code Structure

The provided script *Lab_05.py* was used as a starting point for this exercise. In particular, the code was split in different files: the main program is replaced by a notebook, and many portions of the file have been used to build functions in *testing_tools.py* or where used to build the `Autoencoder` class, contained in *autoencoder_structure.py*. This class has been expanded with the following new methods:

- `self.train_epoch()`: Performs one epoch of the training using all data given by the dataloader in input;
- `self.test_epoch()`: Computes the loss value of the network over the data given by the dataloader in input;
- `self.get_enc_representation()`: Encode all samples contained in the input dataset;
- `self.generate_from_encoded_sample()`: Plot or store the reconstruction of the encoded sample given as argument.

3 Training the Autoencoder

3.1 Random Search

Before training any model, a Random search was performed to find the best values of the training parameters `lr` (the learning rate), `L2_decay` (the weight decay of the optimizer for *L2 regularization*) and of the `dropout` probability of the neural network. The size of the hidden layer, most important parameter of the Autoencoder, was kept constant to 4 during this procedure since changing it has a big impact onthe performance and the effects of the other parameters would have been neglected (see next subsection).
The `Param_generator` class was built to produce combinations of values of the three parameters, which were sampled using

```
drop_vals = uniform(drop_min, drop_max, n_combins)
lr_vals = 10**(uniform(lr_min, lr_max, n_combins))
L2_vals = 10**(uniform(L2_min, L2_max, n_combins))
```

where we chose `drop_vals` $\in [0, 0.5]$, `lr_vals` $\in [10^{-4}, 10^{-1}]$ and `L2_vals` $\in [10^{-6}, 10^{-2}]$ as intervals. Using 30 combinations of parameters we performed a fairly simple Random Search (Cross-Validation was not implemented since the needed computation time would have increased a lot). The goal of this process was not to find the best combination of training parameters, but to find a good one to train the different models for the rest of the exercise. After training and evaluating all the networks, the combinations of parameters were sorted based on their results and the best one was saved in the first cell of the notebook as the variable `best_params`. These were `dropout=0.1508`, `lr=3.47e-4`, `L2_decay=4.65e-06`.
After finding `best_params`, a full training (150 epochs) of the network with 4 hidden neurons was performed. It reached a final loss value of 0.02995 and its weights were stored as *best_net_params_4.pth*.

3.2 Sizes of the hidden layer

As anticipated in the previous section, the size of the hidden layer of the autoencoder is its most important parameter. As one could imagine, increasing it should bring large improvements to the model since the input would be encoded using more information, making the decoding (reconstruction of the input) much more accurate. To verify this assumption, we trained networks with different number of neurons in the hidden layer (from 1 to 10) keeping fixed all the other parameters. The final loss values of every network are showed in Figure 1.

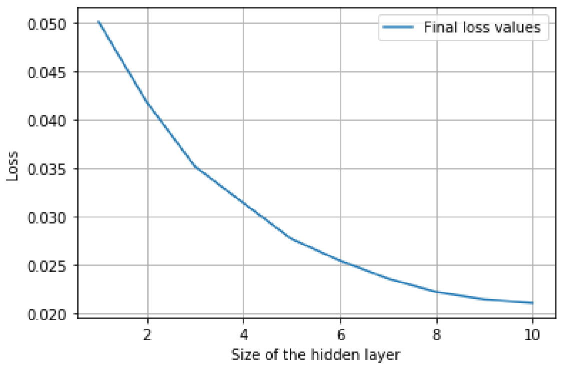

Figure 1: *Final loss values of Autoencoders with different size of the hidden layer (after 80 epochs)*

These values scales exponetially with the size of the hidden layer and they agree with our expectations.

4 Corrupted Data

4.1 Testing the Autoencoder

Autoencoders are models with some reconstruction capabilities that can be used to retrieve information from corrupted data. Here, we will test those skills with two different types of corrupted data: one corrupted with Gaussian noise and the other with partial occlusion. The script *corrupt_data.py* contains the functions that corrupt the dataset in the two ways, which will be both analyzed by the model trained after the Random Search and stored in *best_net_params_4.pth*.

4.1.1 Gaussian noise

The corrupted dataset is obtained just by adding a Gaussian noise to the original dataset and clipping the result to be contained in the interval $[0, 1]$. The `Gauss_noised_dataset()` function takes in input also the `std_noise` to modify the noise intensity (by default it is 0.2). In Figure 2 and 3 are shown a correct reconstruction and an inaccurate one:

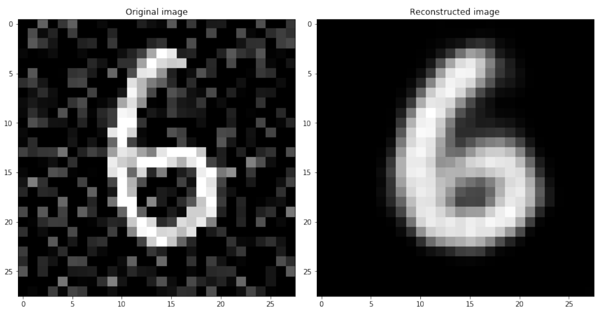

Figure 2: *Correct reconstruction of a number 6*

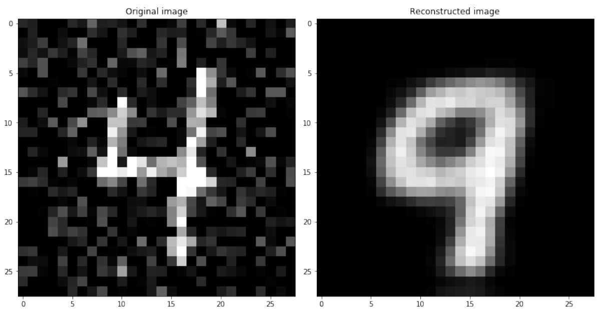

Figure 3: *Inaccurate reconstruction of a number 4 (model decoded it as a 9)*

After some tests, it was observed that, increasing the noise intensity, the quality of the reconstructed images don't change, but they become more inaccurate.

4.1.2 Partial occlusion

The `Obscured_dataset()` generate squares of size given as argument and random position inside the images. Then it hides the portion of image covered by the square by setting all the contained pixels to zero. In Figure 4 and 5 are shown a correct reconstruction and an inaccurate one:

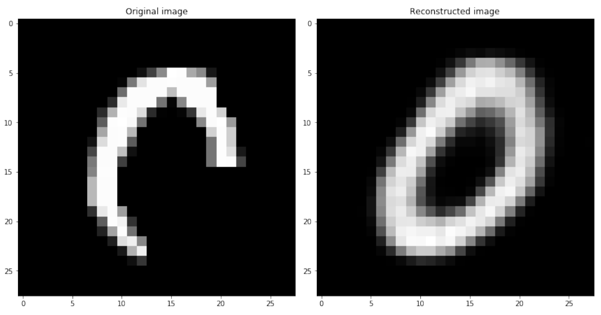

Figure 4: *Correct reconstruction of a number 0*

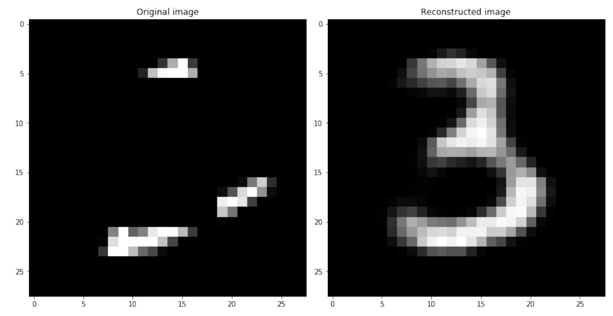

Figure 5: *Inaccurate reconstruction of a number 2 (model decoded it as a 3)*

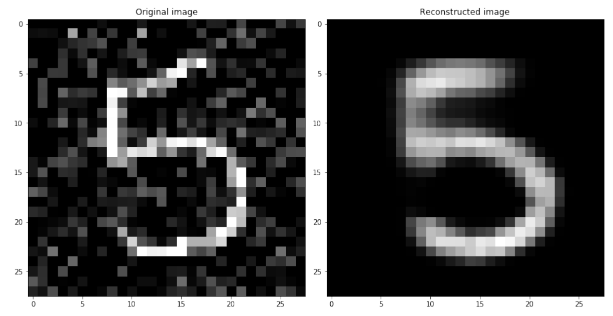

Figure 6: *Final loss values of Autoencoders with different size of the hidden layer*

By default, the squares have size 10×10 pixels. By increasing it, it is possible to hide whole numbers (almost like the case shown in Figure 5). In these cases the model takes some "wild guesses": it tries to interpolate from the obscured sample, so the reconstructed image often represent a number, but sometimes it can be almost a random one.

4.2 Denoising model

The already trained model provides some denoising capabilities, but it is possible to improve this results by developing a *Denoising model*, obtained by training an autoencoder to reconstruct clean images starting from the corrupted versions. Both versions of each image had to be contained in the same dataset: the `combined_dataset` class takes the original dataset as an argument during its initialization and builds it corrupted version. After building train and test dataset, the new network was trained and evaluated with its `self.train_batch()` and `self.test_batch()` methods using the `denoise_mode = True` flag, which means that the loss function has to be computed using the two different version of each image. The final loss value (obtained after 100 epochs) was equal to 0.030541, which is impressive because it is really close to the one obtained with the standard autoencoder described at the end of section 3.1.
In Figure 6 it is shown an example of its reconstruction from a noisy image.

5 Generative Capabilities

In this last section, we will explore the generative capabilities of autoencoders by sampling points in the encoded space and decoding them. In particular we want to check if the autoencoder described at the end of section 3.1 is regular enough to allow for a smooth sampling. This was done by taking two points in the encoded space and connecting them with a straight line. Starting from one of them, we can travel along this line and iteratively decode each new "position" in the encoded space, which will be described by sets of values of the encoded components. After building an animation of the output with two random points, we asserted that the encoded space was regular and allowed smooth sampling.
From this positive result, we wanted to try morphing digits between them. To do so, we first had to use the `net.get_enc_representation()` method to retrieve the encoded representation of the digits contained in the MNIST test dataset. Then, the `testing_tools.morphing_digits_gif()` function takes in input the trained autoencoder, the encoded representation of the dataset, the starting and the ending digits of the morphing process. The function uses the encoded dataset to compute the two centroids of the samples corresponding to the two digits in the encoded space and then use them as starting and endong point of the procedure described above. Here are linked some examples of this morphing process:

- From digit 2 to digit 0
- From digit 3 to digit 9
- From digit 1 to digit 4

6 Conclusions

The results obtained in each section are satisfying. Autoencoder are a very powerful tool: with only 4 hidden neurons their reconstruction capabilities are impressive, even with corrupted data. When training a model with the specific goal to reconstruct data corrupted with the same kind of disorder, the results further improve. In the future we could try training a model on obscured data to check if it performs as good as the denoising model of section 4.2. Moreover, the encoded space has been proved to be regular enough to allow for a smooth sampling.