

Neural Networks Applied For Image Compression And Denoising

Approach using a basic linear model.....	1
Effect of epochs on the results.....	1
Effect of learning rate on the results.....	3
Effect of the compression / latent space size on the results.....	6
Testing our model with other types of images.....	7
Calculation of the Compression Ratio.....	11
Autoencoder with convolutional layer.....	12
Comparing our two models' performance.....	12
Convolutional approach's results on different images.....	15
Application of our models for noise removal.....	17
Conclusion.....	19

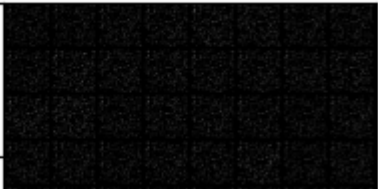

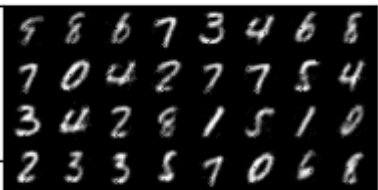
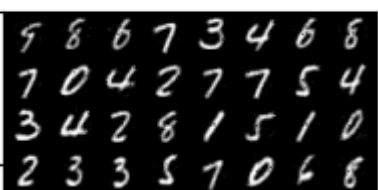
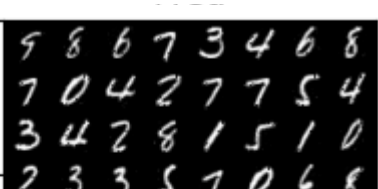
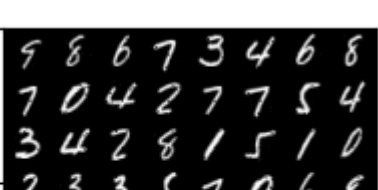
Approach using a basic linear model

Here is our analysis for the first model we wrote which is a simple multi-layer perceptron (MLP). We conducted multiple tests in order to get a deeper understanding of the impact of each factor on the final result.

For training the first model we used the MNIST dataset which contains low resolution (28 x 28) greyscale images that represent handwritten digits from 0 to 9. Each image is labeled so we know what each digit actually is. This allows us to perform supervised training (which means that, since we know the correct labels, we can use them for reinforcement during training).

Effect of the number of epochs on the results

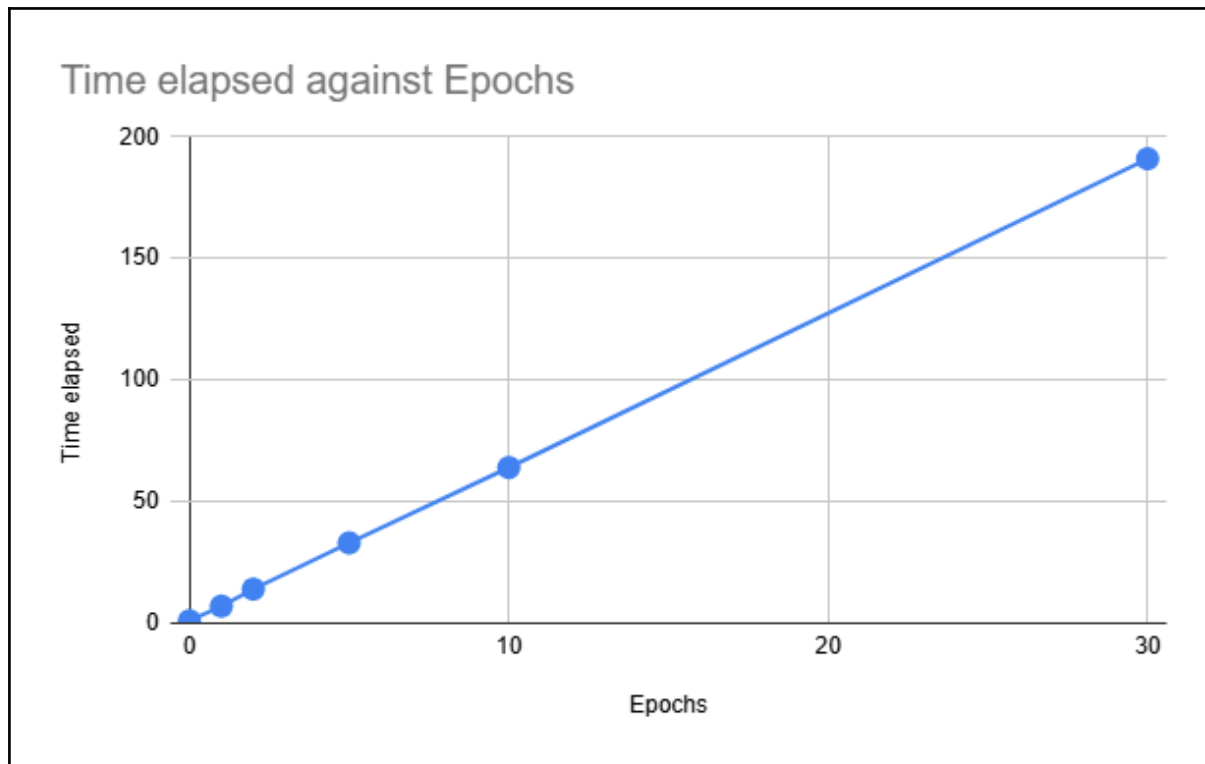
We wanted to see the results of our network to see what level of training time / epochs is necessary to get a result that's visually satisfactory. To do that, we varied the number of epochs while keeping all other parameters static (latent space of size 128, learning rate 0.0001).

Epochs	Decompressed Visualisation	Original vs Decompressed difference	Execution time
0		0.140	1 s
1		0.073	7 s
2		0.053	14 s
5		0.034	33 s
10		0.029	64 s
30		0.020	191 s

We can see that 1 epoch is already good enough for our Autoencoder to learn to compress and decompress the basic shapes of the digits. Some are already understandable but they all are blurry.

More epochs allow the reconstruction to get less and less blurry as we can see with our results for 5 and 10 epochs. But there are still small defects that make the reconstruction differ from the original image.

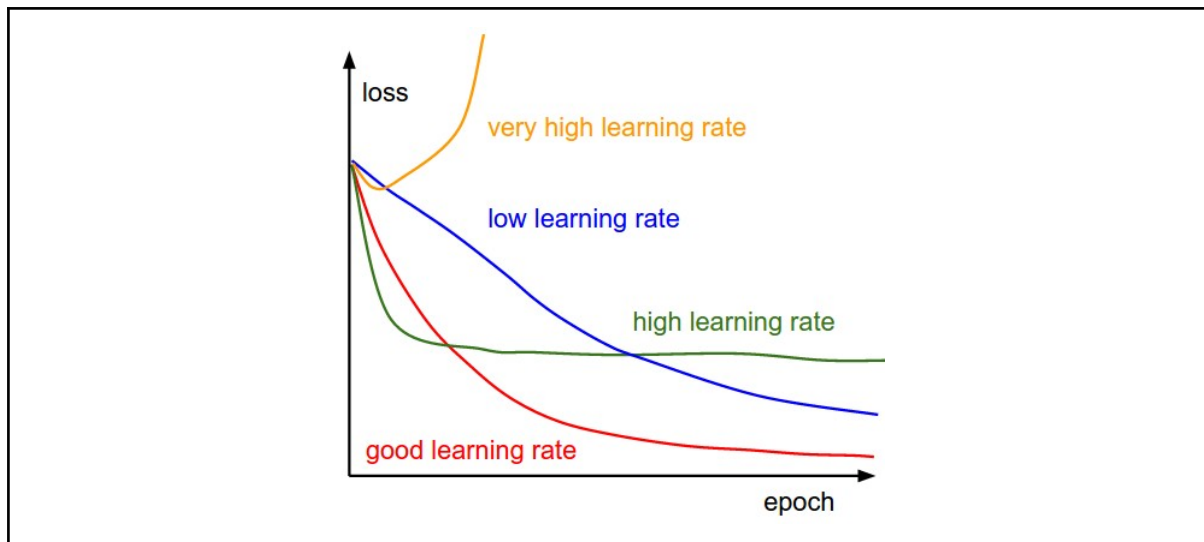
With 30 epochs, all the digits are recognizable and most of the defects are gone: we get a difference score of 0.020. However, training the model for that number of epochs takes a longer duration of time (3 minutes in our example).



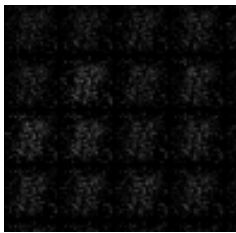
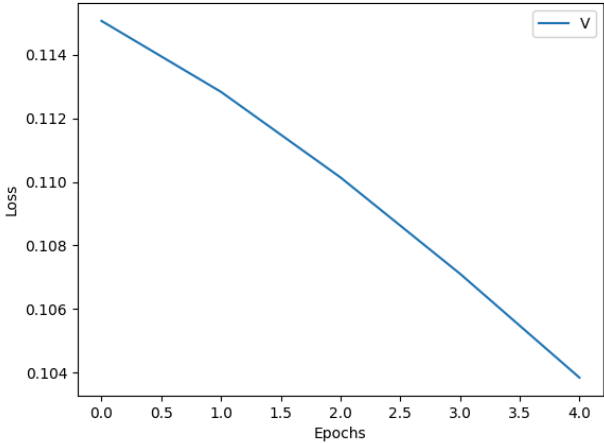
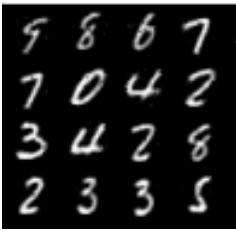
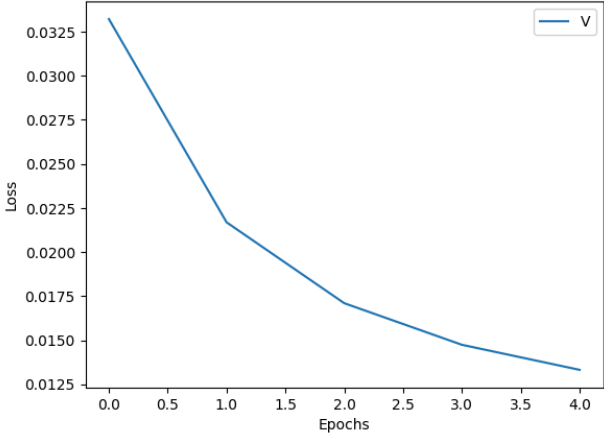
Graph made using our time measurements

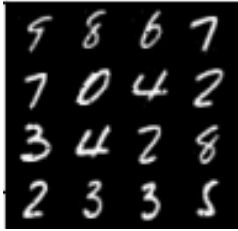
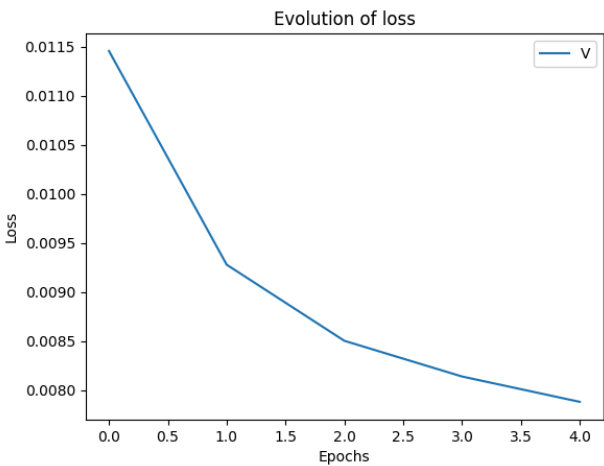

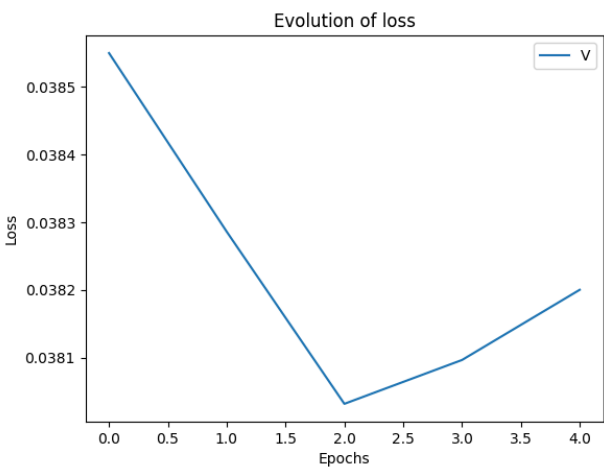
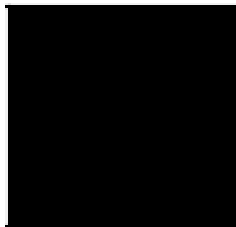
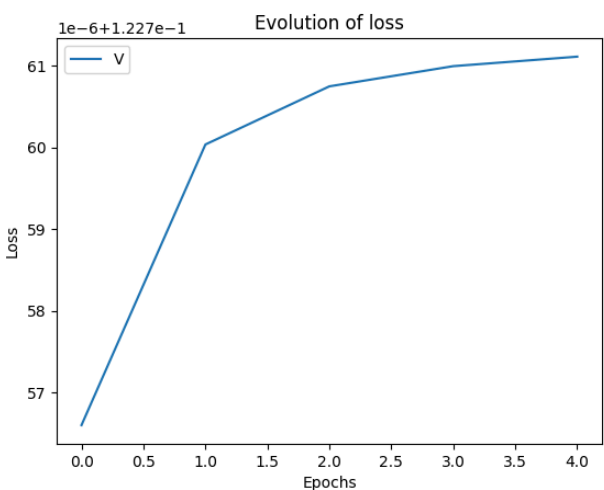
We plotted the duration of each training session and we can observe that, as we expected, it's linear. This reflects the fact that a single epoch always takes the same amount of time. This highlights the fact that choosing a good learning rate and other hyperparameters is of paramount importance to reduce execution time and save computations.

Effect of learning rate on the results



Influence of learning rate on loss during training

Learning Rate	Decompressed Visualisation	Validation Loss during training	Diff. Value
0.000001		<p>Evolution of loss</p> 	0.139
0.0001		<p>Evolution of loss</p> 	0.034

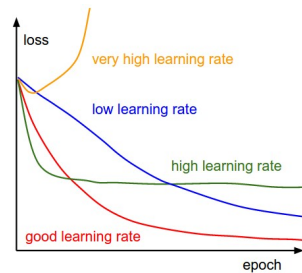
0.001			0.022
0.01			0.076
0.1			0.117

We can observe that, as expected, for a very small learning rate (like 0.000001) the loss decreases but at an extremely slow rate. Therefore the final result is not adapted to the data at all since it would have needed many more epochs to arrive at a minimum.

In the other extreme for very large learning rates such as 0.01 or 0.1, we observed that the Adam gradient descent method does not manage to find a minimum and we even get increasing loss instead of decreasing loss.

In our experiments the learning rates that worked better in our experiments of 5 epochs were in the middle like 0.001 and 0.0001.

This way we got to observe in practice what the following graph presents.

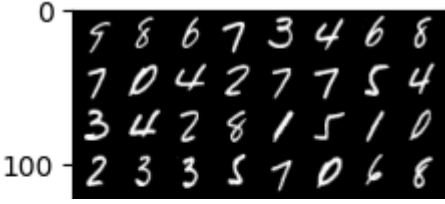

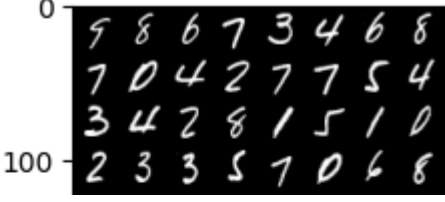
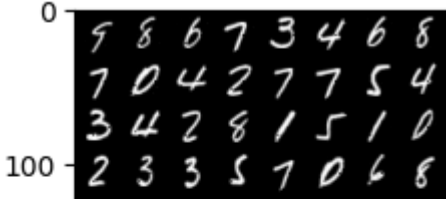
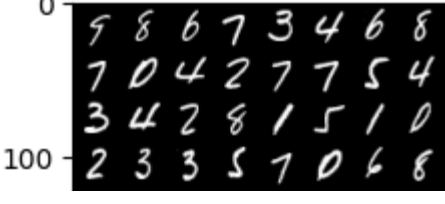
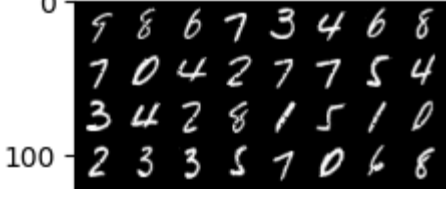


We think superimposing our curves to get an actual visual comparison can be an interesting next step but we didn't have the time to do it for this TP.

Effect of the compression / latent space size on the results

One of the most important tests for the task at hand, compression, is to see what is the minimal latent space possible to get a good enough reconstruction of our images. To assess this we trained and tested our autoencoder for different latent space sizes (1, 16, 128, 512). Though it obviously isn't useful for compression, we also decided to include a test with a massive latent space of 16384 to see what would happen.

Size of the latent space	Original	Decompressed
1		
16		

128		
512		
16384 (not compressed, just a test)		

We noticed that the greater the latent space, the better the result will be.

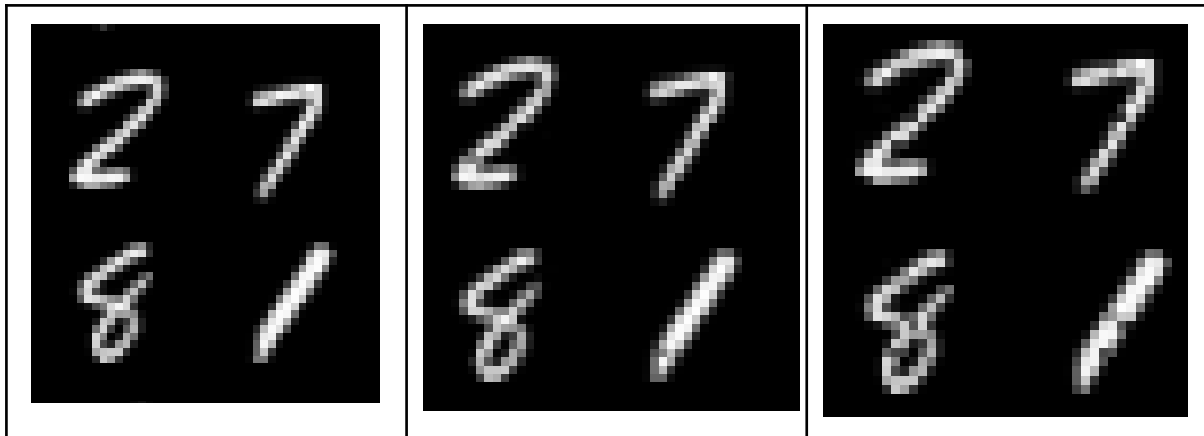
We can see that for a ridiculously reduced latent space of 1, all the uncompressed images look the same. This makes perfect sense because the latent is a single value and cannot represent the information contained in an image because the amount of data stored in an image is much larger than a single value.

For a latent space of 16, we observe that the digits are more recognizable but very blurry and some don't represent the original digits very well. This means that this method, with these hyperparameters, doesn't allow us to compress the images to such an extreme level. It may be the case that 16 is too little and it is impossible to get a good result but that would require more proof.

Larger latent spaces like 128 or 512 perform much better. However the improvement of 512 over 128 is small and we think it is not worth it to lose a significant portion of the compression rate for such a small improvement.

Our experiment of using a very large latent space shows in our opinion the dangers of overfitting to our training data. We can see that it actually performs worse on our test data, there are more pixels that are black when they shouldn't and we interpret that this means our model is not generalizing enough.

Original	Latent 512	Latent 16384
----------	------------	--------------

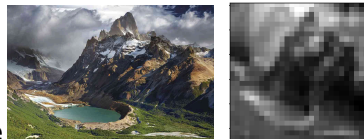


Testing our model with other types of images

For the images we decided to try out different types of images with different characteristics. One of them is this cartoon skeleton character with a black background.



It is meant to be a more complex shape but still with the same background as the training data.



Another one is this one which is a picture of nature probably taken by a camera and is even more complex.



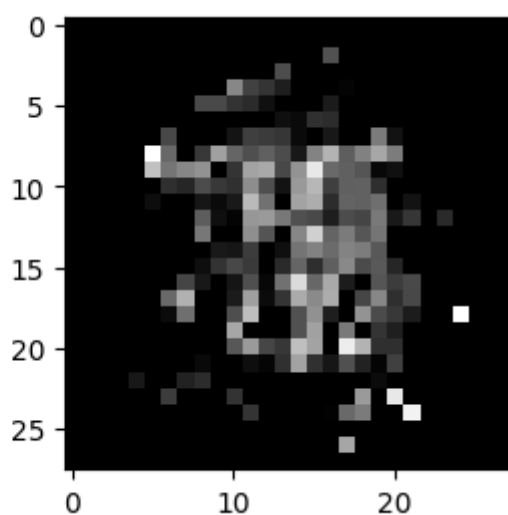
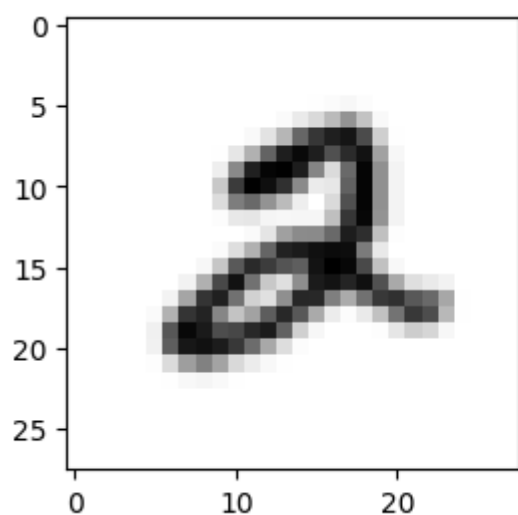
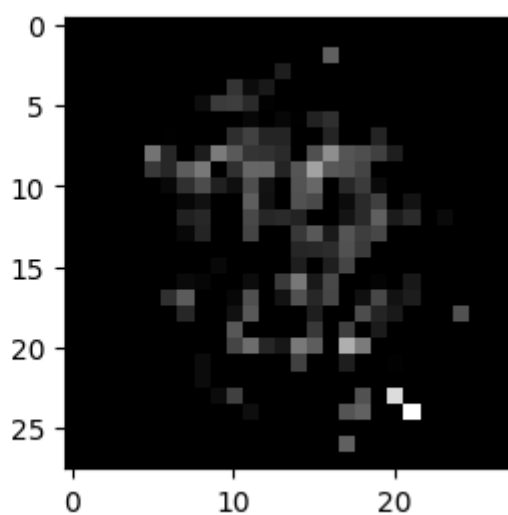
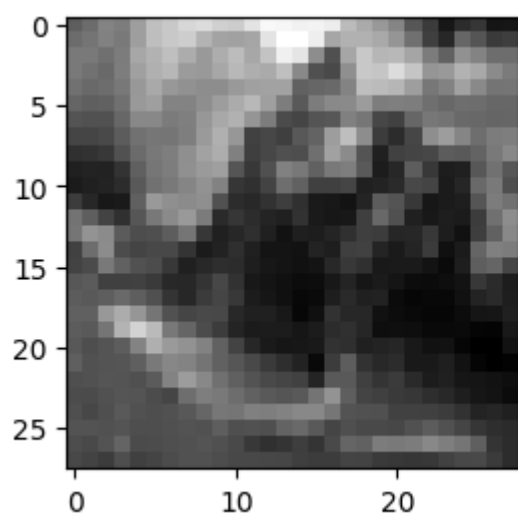
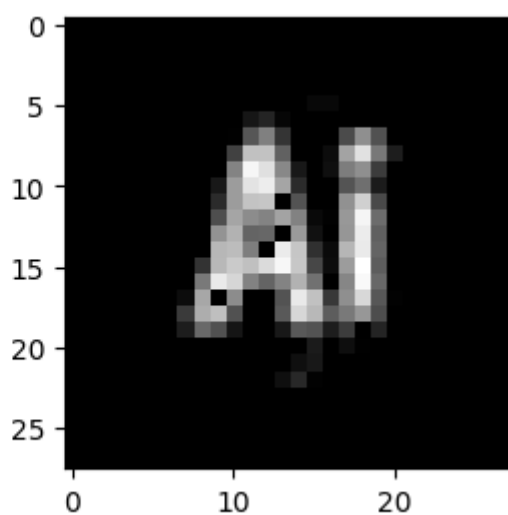
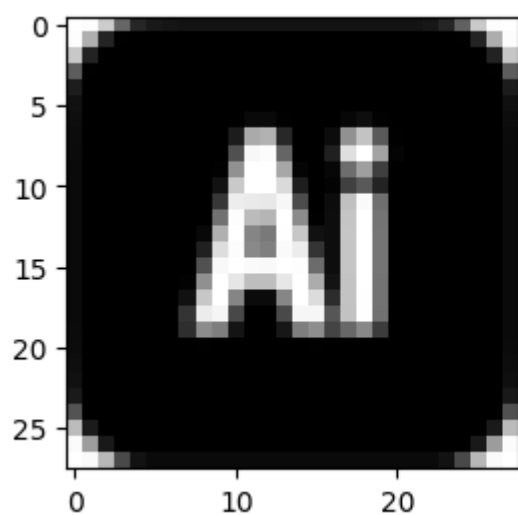
Then, the reverse color “2” digit from the database allows us to test if the same shape can be recognized even when the colors are inverted.

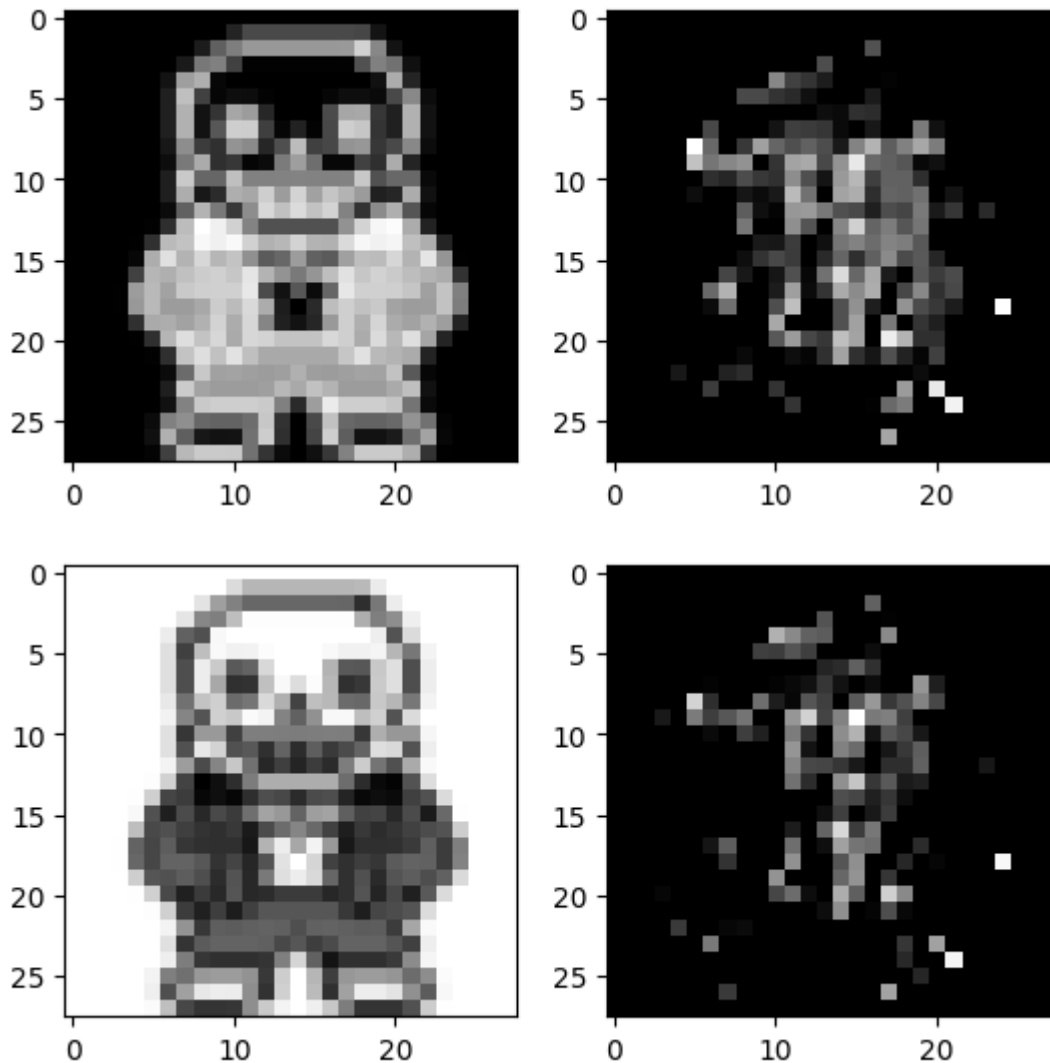
Finally the image that contains the text “Ai” in a white font with a black background



looks very similar to the images in the database but with new characters that are not in the training data and thus have never been seen by the model.

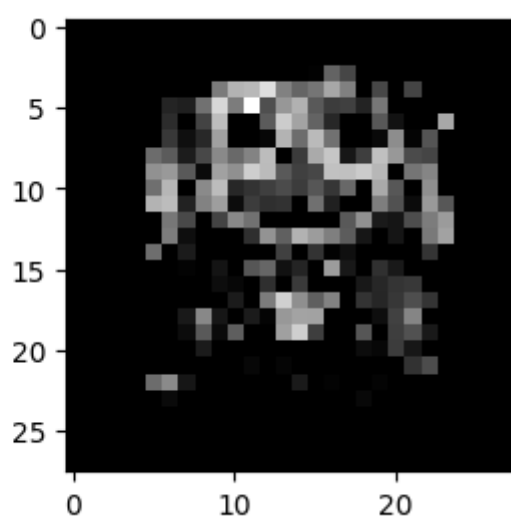
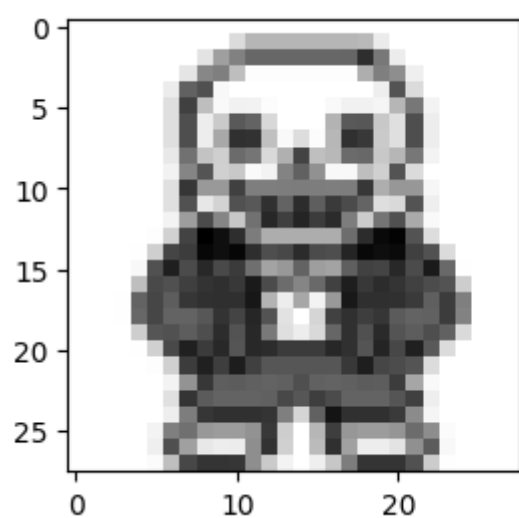
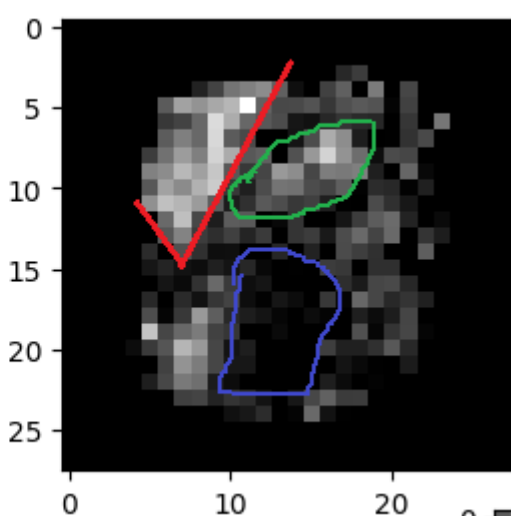
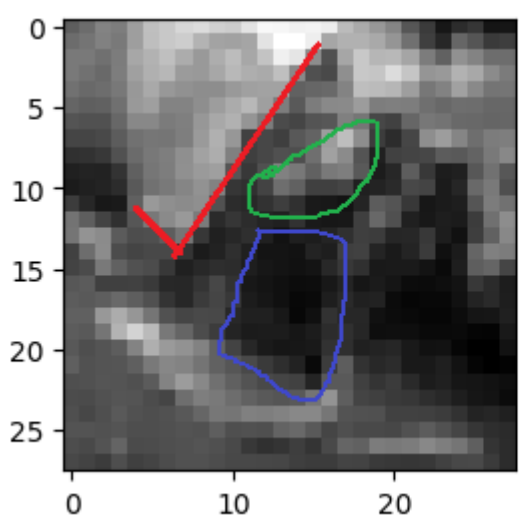
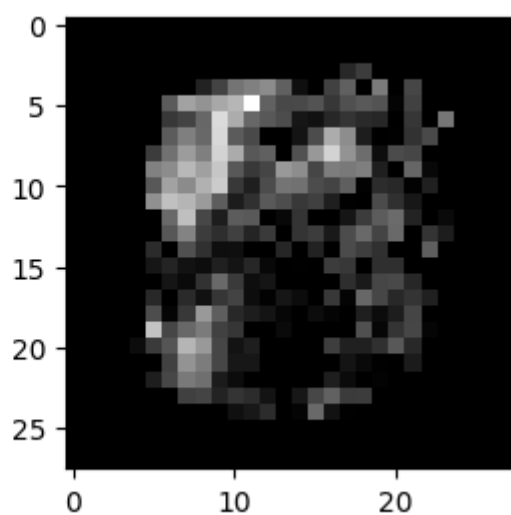
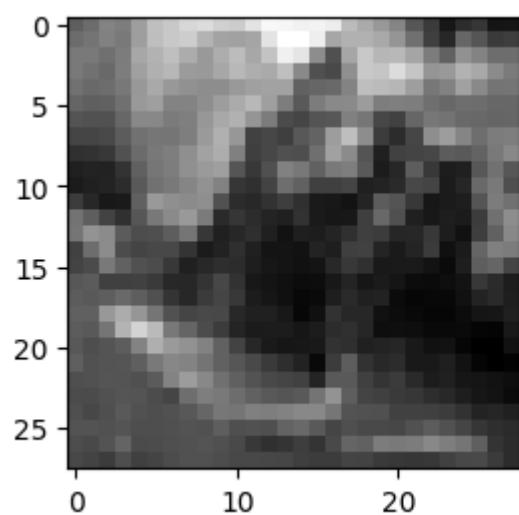
Here are our results for a latent space of 128:

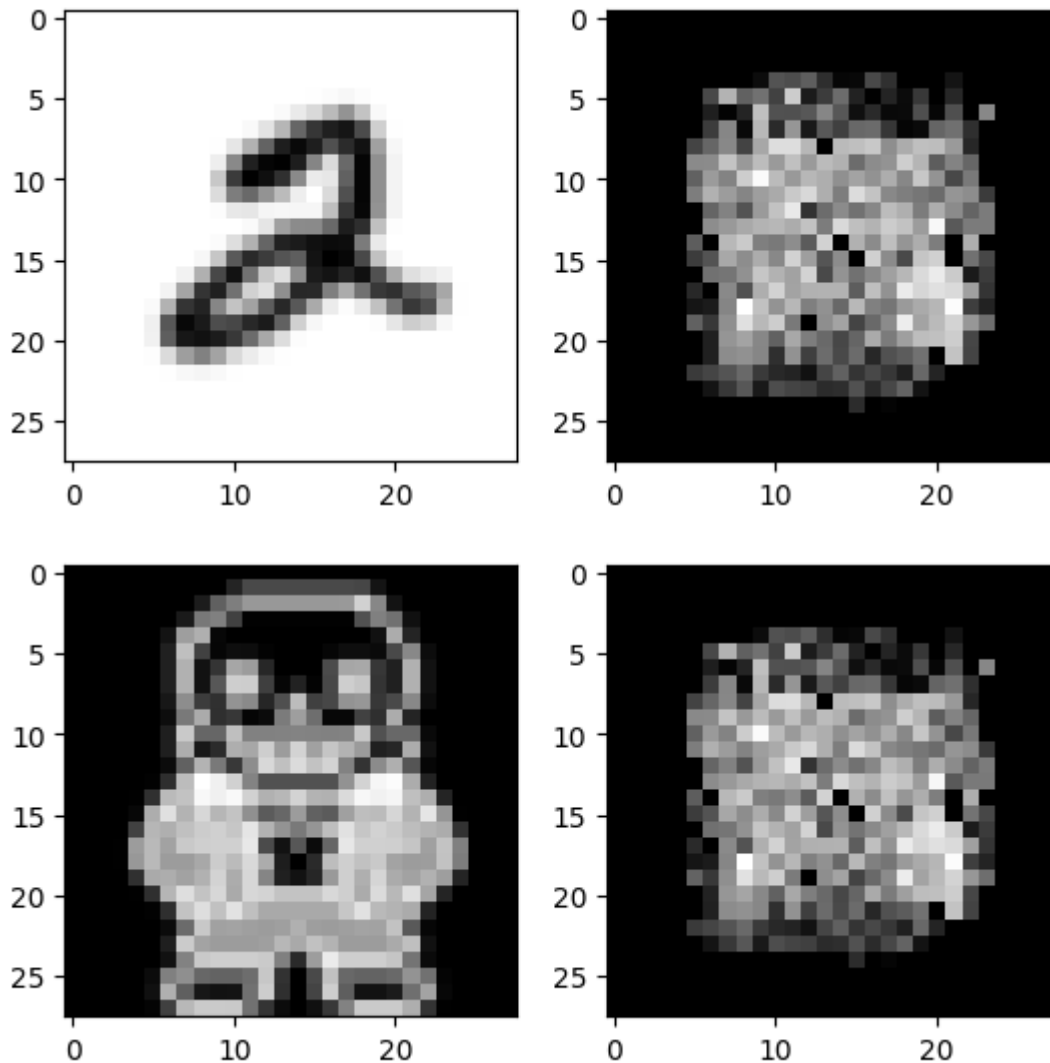




The model fails to recreate any of these images except for the one that spells “AI”, though with some errors. This shows us that the training data is incredibly important in determining how well and to what images the model will generalize well. Even minor changes like color inversion or using a character instead of a number will throw it off except if it’s very similar to the training data like the image that has the word “AI”.

We also noticed that the model with the giant latent space manages to recreate some images in a more recognizable way but fails completely for others as can be seen in the following captures (Latent 16384):





Calculation of the Compression Ratio

We can calculate the data compression ratio by using the formula:

$$\text{compression ratio} = \text{uncompressed size} / \text{compressed size}$$

The uncompressed image is represented by a matrix of floating point values. Its dimensions are 28 x 28. Therefore the total amount of pixels in the image is 784. So the total size of the uncompressed image is 784 floating point values.

The compressed image is represented by an array with a set size so we know its size in memory. We tested with the values 1, 16, 128, and 512 therefore the compression ratios are:




Latent space size	512	128	16	1
Compression rate	1.5	6.1	49	784

As we've just seen, the size of the encoder and decoder are not a factor in the compression ratio. Therefore the model can be quite large to capture the nuances of the data and achieve a good result.

Autoencoder with convolutional layer

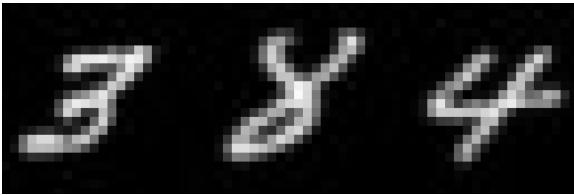
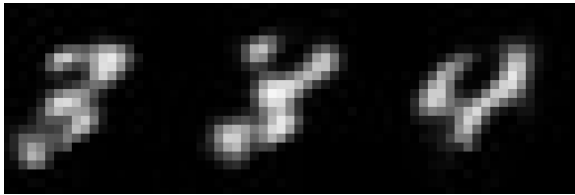
Comparing our two models' performance

Using a very restrained latent space allows us to clearly see the performance differences between the perceptron approach and the approach that includes a convolution step. Here we tried both using a 16 sized latent space.



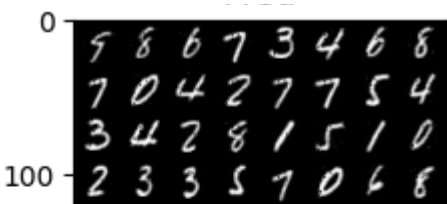
Original Images	
Linear only	
Convolutional → Linear	

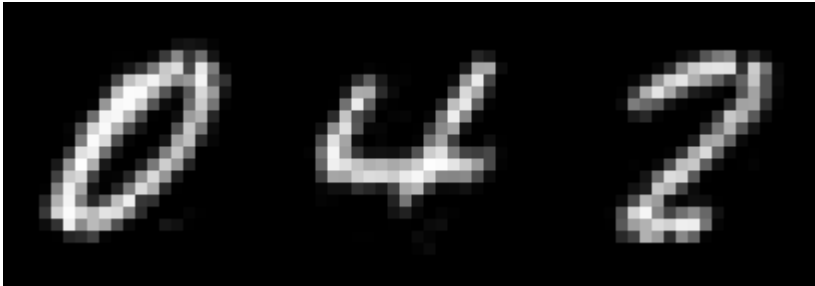
We can see that though they're still somewhat blurry, the images reconstructed by the Convolutional → Linear approach are far clearer and more readable than the other approach.

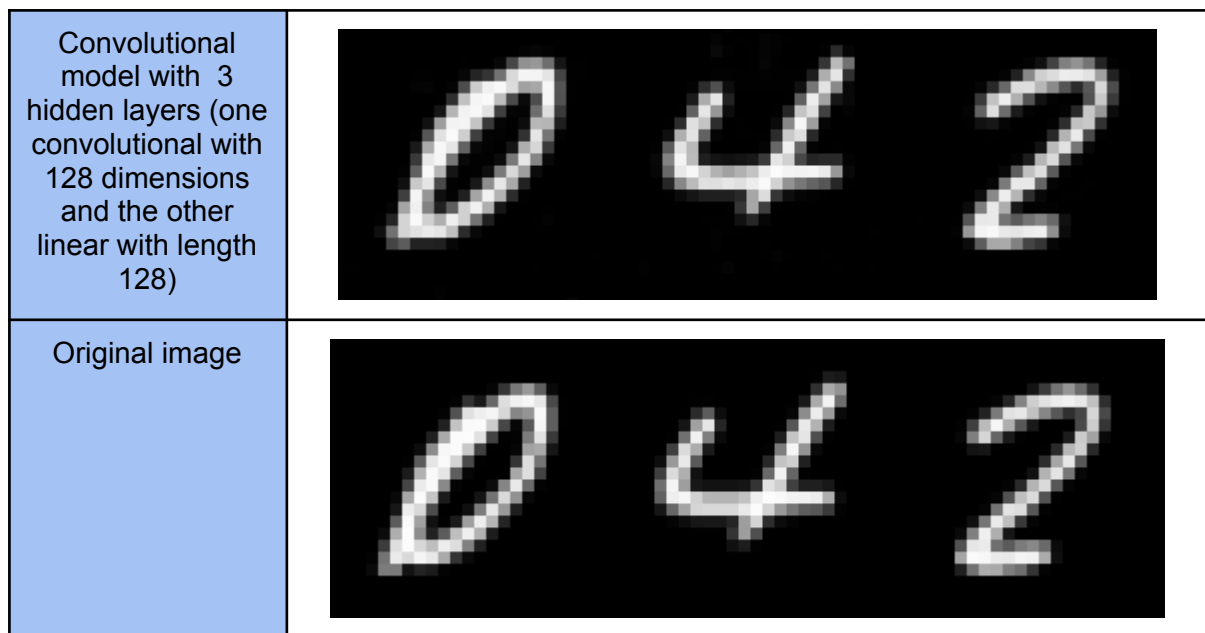
However the latent space could still be said to be too small and lose too much information. As we can see in the following image, the 3 is preserved, the 8 seems to turn into a 3 and the 4 loses its shape.

Original	Compressed to latent space 16
	

Next we wanted to compare the compression performed by our Linear Model and our Autoencoder models for the same size of latent space. We chose a shared latent space of length 128 for this comparison.

Original Test Images	
Convolutional model with 3 hidden layers	
Linear Perceptron with 128-sized Latent Space	

Linear Perceptron with 128-sized Latent Space	
---	--

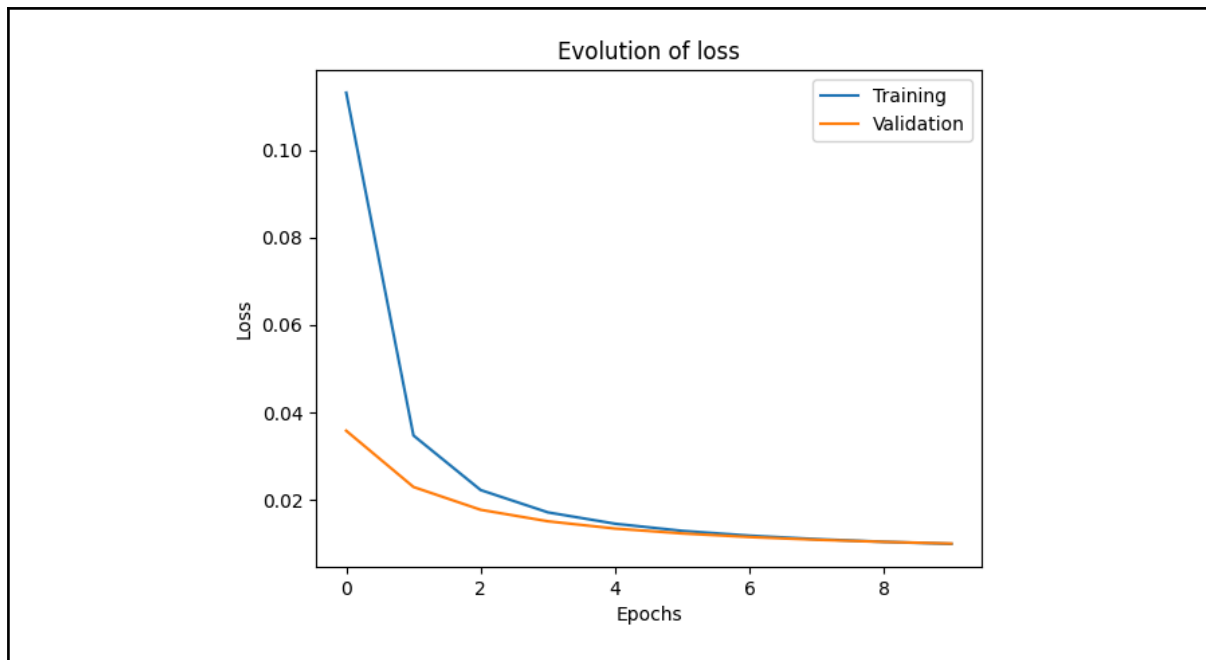


When zooming in on these examples we can see that the convolutional model's reconstruction is more precise. For example, the extra pixels at the bottom left of the 0 are preserved by it but not by the simple model. The simple model also makes some pixels darker and shortens a line from the 4, while the convolutional one doesn't make these mistakes.

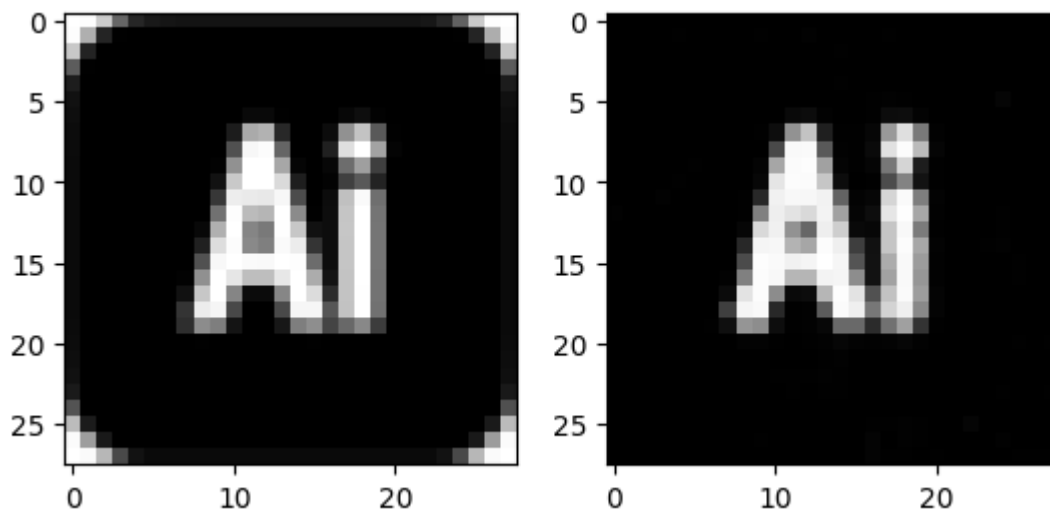
Here is also the comparison of the actual values:

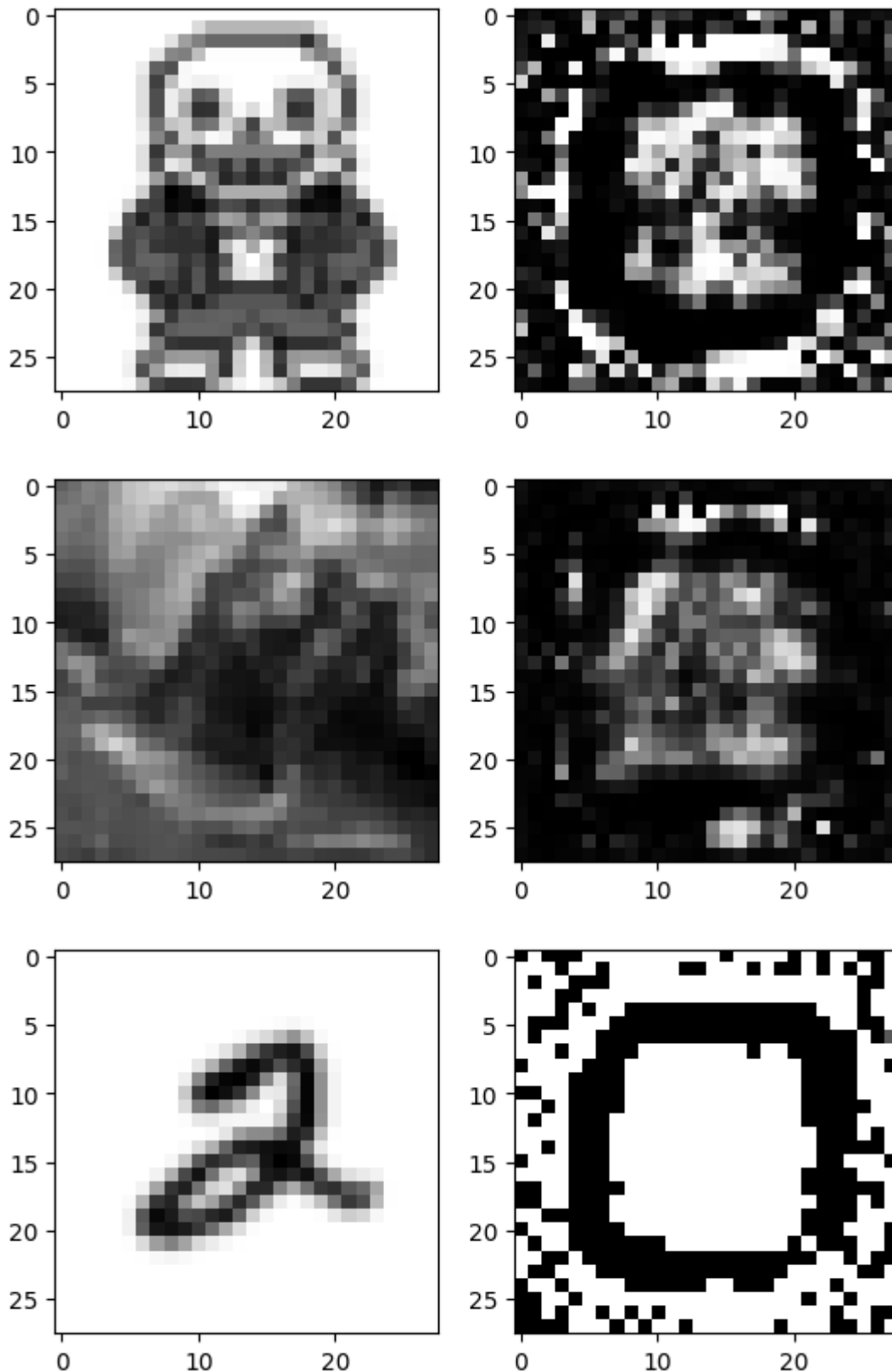
	Execution time	Image difference value
Simple Linear	121 s	0.029
Convolutional	64 s	0.013

Additionally, here is a graph showing the evolution of loss during training for this model. We see that it decreases until reaching a low value and starting to stabilize.



Convolutional approach's results on different images



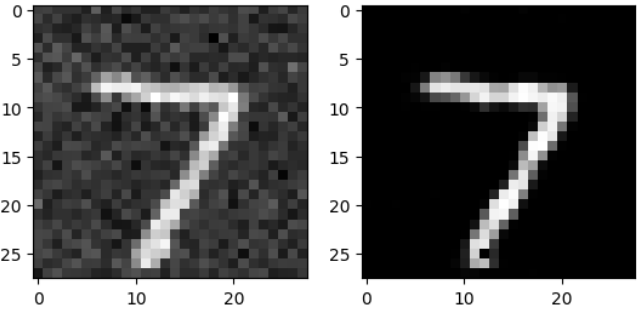
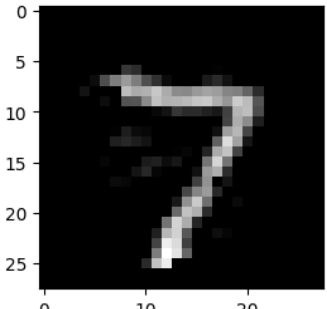
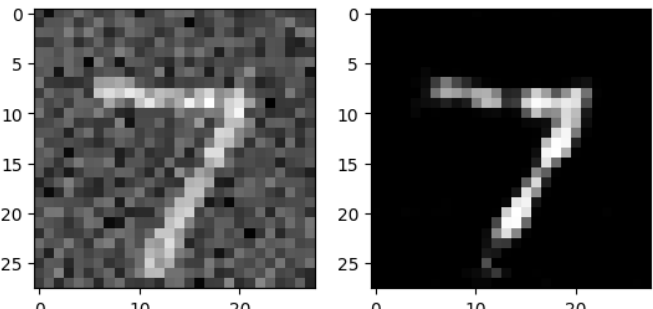
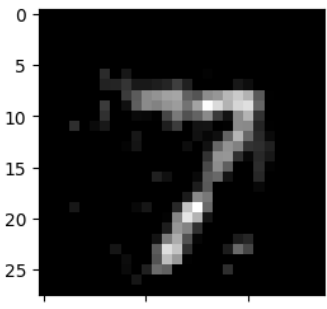


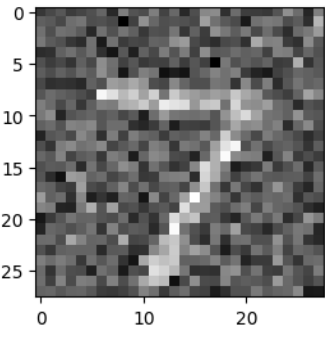
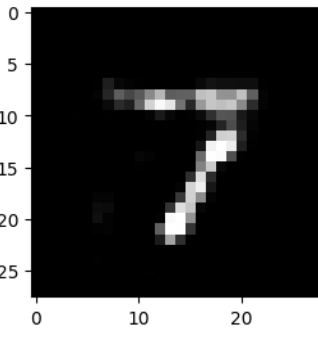
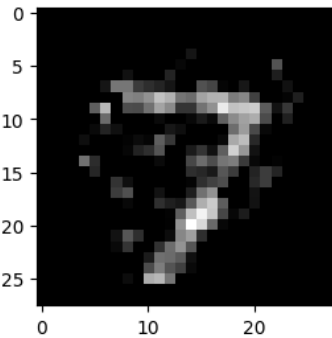
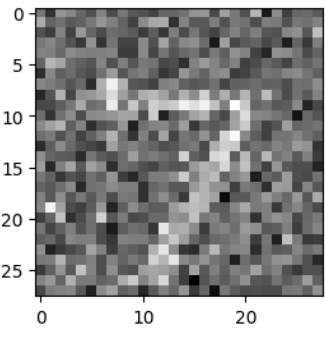
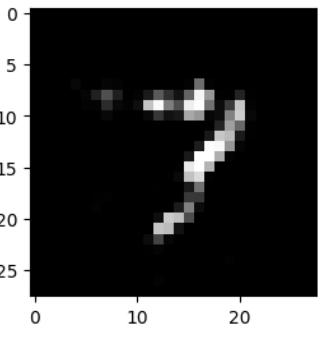
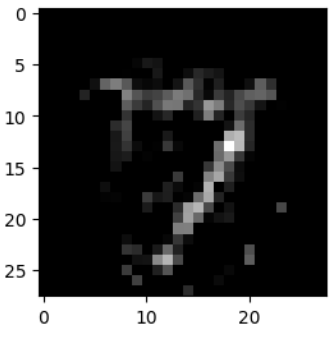
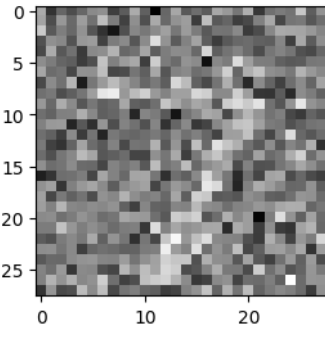
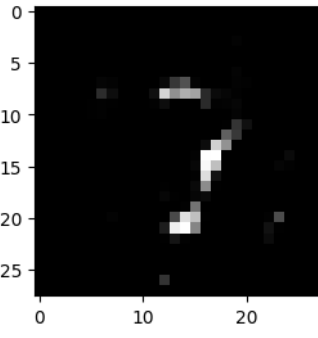
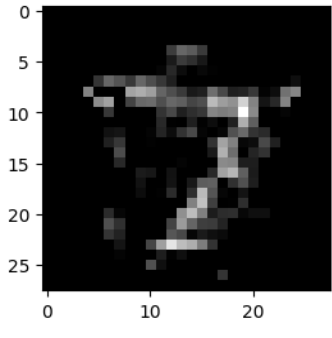
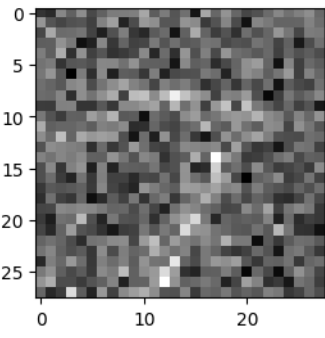
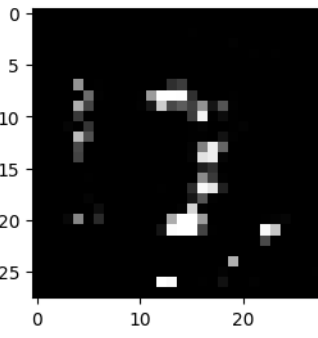
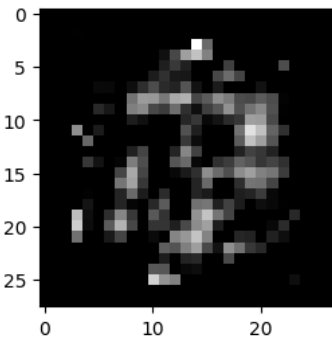
We can see that the only image that gets a good result is the one with the text “AI”. This reinforces the idea that with such a restrained dataset, our model is going to learn the patterns of the model specifically and even minor changes such as inverting the colors of the image stray far enough away from the training data that the model fails.

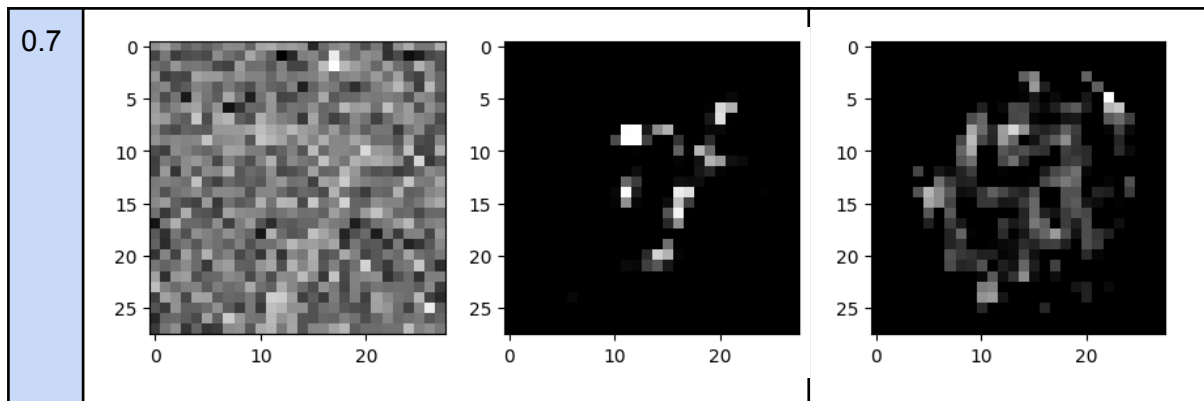
However, with the text “AI” image, the model performs remarkably well. It manages to capture the black dot inside the A for example. This means that the patterns that the model learned fit very well with the white-on-black-background, centered characters of this image. This is evidence of the generalization capability of AI and how it is determined by the training data.

Application of our models for noise removal

Now we will show and describe the results of our experiments on noise removal. We tested two different models on this, our “Deep” (3 hidden layers) convolutional -> linear model with a latent space of length 64 and our basic linear model from the beginning of the LAB with a latent space of 128.

Noise factor	Convolutional -> linear model and with a latent space of length 64	linear model latent space of 128
0.1		
0.2		

0.3	 	
0.4	 	
0.5	 	
0.6	 	



Here, for the interpretation of why the autoencoder works so well, we will quote what we wrote in our python notebook:

-> We think that since the convolutional layer extracts features, the encoder will focus on finding the details that are specific to the learned features and will be able to score those characteristics high even if there is noise. Then, since the original images from the training data weren't noisy, the encoder only knows to use the latent space to create clean images, so it does that and we get a result that contains the learned features that are in the input image but with none of the noise.

-> When the noise gets strong enough it drowns out the signal and even our autoencoder can't detect those features anymore. Some parts of the digit then possess "holes". However, new features that were not in the test image don't start to appear until the noise is very high which attests for the resilience of this method.

On the other hand the simple model does not perform feature detection so it is only handling the raw pixel values and basing its decision on them. Therefore since it learned to recreate the input image somewhat directly it will recreate some of the noise too. This is why we observe that even though most of the noise has disappeared for noise factors 0.1 and 0.2, there still is some visible noise.

Then noise preservation clubs rapidly and though some of the images can still be kind of recognizable to a human, the task of reducing noise is a complete failure with this model.

For every single test, the model with convolution, even if it doesn't manage to recreate the original image, does away with most of the noise, while the simpler model starts recreating the noise in the center and is noisier for every test.

Conclusion

This lab allowed us to develop an understanding of how AI image processing, and in this case, compression, works. It also allowed us to learn and practice our Python Torch skills.

By experimenting with various architectures (linear vs convolutional), modifying hyperparameters like learning rate and latent space size, and testing on different kinds of images, we saw firsthand how each choice impacts both reconstruction quality and generalization.

Finally, exploring compression ratios and applying our models for tasks like noise removal gave us a broader perspective on the practical strengths and limitations of these techniques.