Image denoising with deep learning

Date: 04/11/23

Authors: *SAGDULLIN Damir* and *LAIOLO Léo*

Pedagogical referent: *Mai Khuong NGUYEN VERGER*

Lesson: Artificial Intelligence Image Processing

School:

Report

Introduction

Image denoising has emerged as a critical application, enabling the restoration of degraded images to their original states. The significance of image denoising extends far beyond mere visual aesthetics. In fields like medical imaging, autonomous vehicles, and surveillance, it plays a pivotal role in enhancing the reliability and accuracy of image analysis.

The aim of this project is to showcase the efficiency of AI-based denoising techniques, specifically employing autoencoders. In our investigation, we will introduce random noise to images of hand-written digits, sourced from the widely recognized MNIST dataset, and subsequently task a neural network with the intricate task of restoring the original, noise-free images.

By the end of our investigation, we hope to offer a comprehensive understanding of the potential of AI in image denoising

I) Generating noise

To find effective image denoising techniques, our project involves the application of various noise types to images from the MNIST dataset. These noise types include *Gaussian*, *salt-and-pepper*, and *periodic*. In this section, we will explain each noise type and how they are introduced to the images during data preprocessing.

They are all illustrated in the *Data preprocessing* when executing the *Source code* in `demonstration` mode.

1) Gaussian

The first noise type we employ is *Gaussian noise* makes a picture blurry and add shades by slightly changing the color of each pixel. The changes follow a *Gaussian* distribution. Here is the function φ applied on each image's pixel z :

$$\varphi(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-\mu)^2/(2\sigma^2)}$$

In the code below, we used: $\mu = 0$ and $\sigma^2 = 0.01$.

2) Salt-and-pepper

Moving on to the second noise type, *salt-and-pepper* noise (or *impulse noise*) randomly adds white and black dots on a picture. hese dots are created by extreme alterations in pixel values. Example: a white dot is a pixel value turned into 255 and a black dot is a pixel value turned into 0.

3) Periodic

The final noise type we employ is periodic noise, which introduces periodic waves onto the images. Unlike Gaussian and salt-and-pepper noise, periodic noise does not rely on randomness. Instead, it is characterized by a function φ is applied on each pixel z at the coordinates x and y on the image:

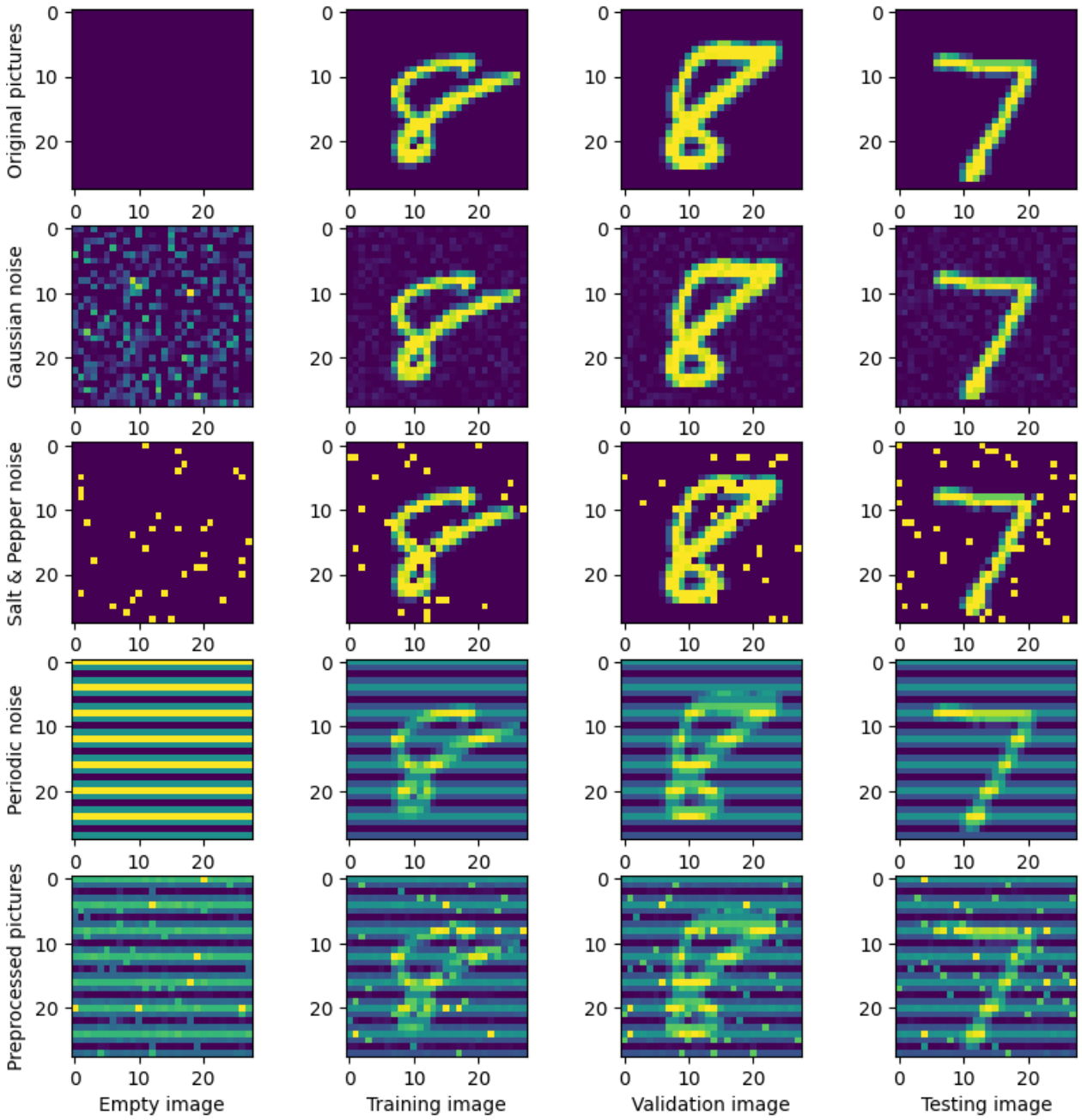
$$\varphi(z_{x,y}) = z_{x,y} + f(\alpha \times x)$$

Where f is a periodic function and α is a constant corresponding to the wave length of the periodic noise. We can replace x by y inside f to change the orientation of the noise. In this project, we used the following function to generate periodic noise:

$$\varphi(z_{x,y}) = z_{x,y} + cos(1.5 \times \pi \times x)$$

The constant $\alpha = 1.5 \times \pi$ was arbitrary defined.

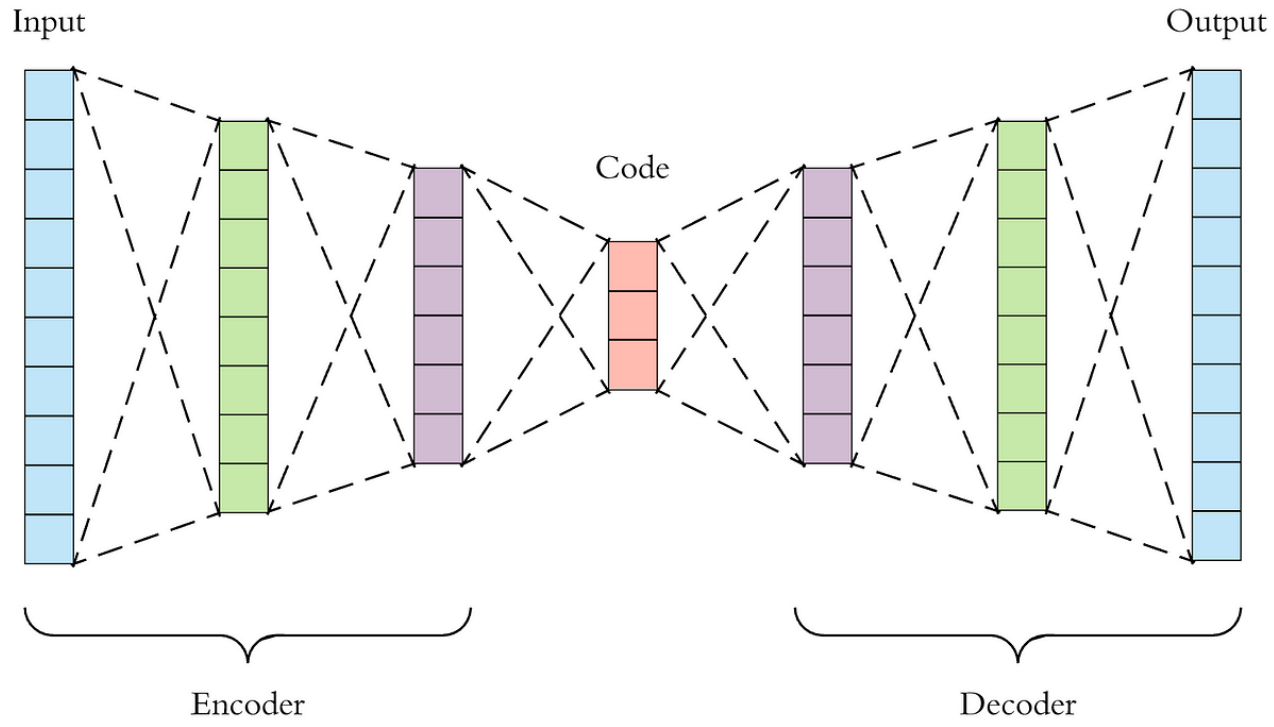
With this understanding of the noise types and their characteristics, we can proceed to explore the the source code involved in applying these noises to the MNIST dataset for subsequent denoising with autoencoders.



II) Autoencoder architecture

An *autoencoder* is a type of neural network designed for unsupervised learning, which means it doesn't require labeled data for training. It is primarily used for dimensionality reduction, feature learning, and reconstruction tasks.

The architecture of the autoencoder consists of multiple layers of neurons. In the illustration provided below, there are seven layers. These layers are organized into two parts, the encoder and the decoder.



The encoder is the first part of the autoencoder. It takes the input data, in this case, the noisy images, and encodes or compresses the input into a lower-dimensional representation. This compact representation captures the most important features of the data while eliminating noise.

The decoder is the second part of the autoencoder. It takes the lower-dimensional representation and reconstructs an output that should ideally be very similar to the original input. In our project, this output would be the denoised image.

In this project, we are using a linear autoencoder composed of fully connected layers. Fully connected layers, also known as dense layers, are those in which each neuron is connected to every neuron in the previous and subsequent layers. This architecture is suitable for basic autoencoder tasks.

Autoencoders are trained to minimize the difference between the input data (here, noisy images) and the output data (here, denoised images). This is typically done by using a loss function that measures the reconstruction error, such as mean squared error (MSE).

It's important to note that autoencoders come in various forms including convolutional autoencoders, variational autoencoders (VAEs) and others each designed for specific tasks and with variations in their architectures and training objectives.

III) Loss function and optimizer

To train our neural network, we need to define a loss and an optimizer. The loss function is needed to evaluate the performance of the model. The optimizer is the algorithm that will make the model learn by modifying its weights and parameters.

Here, we used the loss function *Mean Squared Error* (MSE). MSE is well known for generative AI evaluation. Indeed, reproducing a given image is equivalent to a regression task on a vector. The loss is define by calculating the mean of the squared difference between the prediction and the expectation:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Where Y_i is the expected value, $\hat{Y}_i$ is the predicted value and n is the number of datas. In our project of image denoising, Y_i corresponds to the value of the pixel i of the target image, $\hat{Y}_i$ corresponds to the value of the pixel i of the predicted image and n is the number of pixels in the image. Our aim is to generate the closest image to the target, so we have to reduce the differences between the pixel values. This means that our model will try to make the MSE tend to 0.

To reduce the model's loss, we use the optimizer named *Stochastic Gradient Descent* (SGD). This optimizer is often used in machine learning to train neural networks. It estimates the best weights of each neurone by using the following formula:

$$w_{i,t+1} = w_{i,t} + \eta(r - o)x_i$$

Where $w_{i,t}$ is the weight i of a neurone at the instant t, η is the learning rate, r is the value that the weight should have had so the network predicted the exact correct value, o is the neurone's predicted value and x_i is the input value of the neurone.

IV) Performance analisis

In this section, we analise the accuracy of our models and try to built the best one to denoise image. To do so, we used some training methods.

1) Classic Autoencoder training

First, we built our models with the classic method:

1. Training to predict denoised images from noisy images
2. Evaluation
3. Tuning the hyperparameters
4. Repeating the steps above

Based on multiple trainings of different models, we found efficient hyperparameters. Those values are repertoried in the table below.

Hyperparameter	Best Value
Learning rate	0.8
Epochs	50
Size of the code layer	500
Batch size	-
Number of layers	5

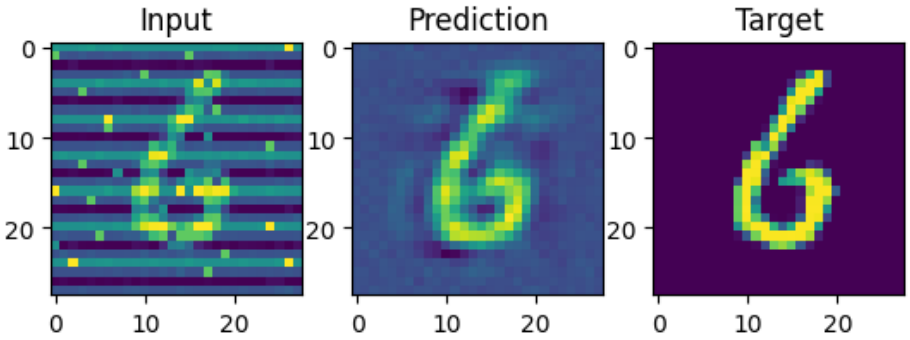
Those hyperparameter's values were found by training linear models on images preprocessed with Gaussian and salt-and-pepper (amount of 0.1) noises. The loss score of our tested models can be found in the section Results of this notebook.

We observed that the loss of the model decrease slower as the number of layers increase. So the model must train longer. While a smaller model could reach the same performance with less time. Looking at the loss curves, we can see that the small models reached a minimum with approximately 30 epochs. The big models seems to keep learning, even after 50 epochs. We didn't try to push the trainings further because it would have been too long. For example, the biggest model we built had 9 layers and a code layer with 500 neurones it took 20 minutes to train it (without GPU). The batch size doesn't seems to have a huge impact on the model's accuracy. As we expected, the size of the code layer is

really important. With a small layer, the encoder will have to leave some features out and the decoder won't have enough datas to build an accurate picture.

The linear autoencoders that we built were able to highly reduce the noise of pictures but sometime couldn't draw a readable digits. Indead, the digits were blurry (illustrated in the section Training the model). This shows that the loss score isn't enough to evaluate our models.

It seems that there isn't any "best model". There is only a comprise between the accuracy and the number of epochs. We can train a model as long as we can, he will reach its limit after a certain number of epochs.



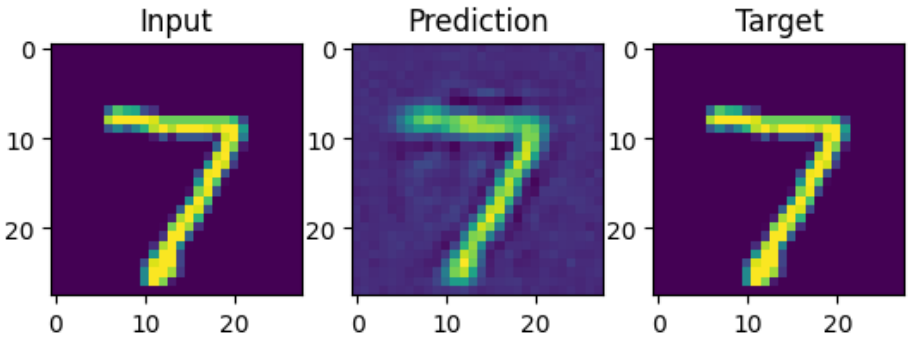
	Noise			**Hyperparameters**					**MSE Loss**		
Model	*Gaussian*	*Periodic*	*Salt-and-pepper* amount	Number of layers	Size of code layer	Learning rate	Epochs	Batch size	Train	Val	Test
Linear	X		0.1	5	10	0.1	10	256	0.00152	0.00152	0.00077
Linear	X		0.1	5	100	0.5	10	256	0.00069	0.00070	0.00035
Linear	X		0.1	5	100	0.8	10	256	0.00037	0.00037	0.00037
Linear	X		0.1	5	100	1	10	256	0.00039	0.00039	0.00039
Linear	X		0.1	5	100	1.5	10	256	0.00038	0.00038	0.00038
Linear	X		0.1	7	100	0.8	10	256	0.00038	0.00038	0.00038
Linear	X		0.1	7	100	0.8	20	256	0.00050	0.00050	0.00050
Linear	X		0.1	5	100	0.8	20	256	0.00029	0.00030	0.00030
Linear	X		0.1	5	100	0.8	50	256	0.00021	0.00022	0.00022
Linear	X		0.1	9	100	0.8	50	256	0.00032	0.00032	0.00032
Linear	X		0.1	5	100	0.8	10	128	0.00037	0.00037	0.00037
Linear	X		0.1	5	100	0.8	10	32	0.00038	0.00038	0.00038
Linear	X		0.1	5	500	0.8	10	256	0.00031	0.00031	0.00031
Linear	X		0.1	5	500	0.8	50	256	0.00018	0.00018	0.00018
Linear	**X**		**0.1**	**5**	**500**	**1**	**50**	**256**	**0.00016**	**0.00017**	**0.00017**
Linear	X		0.1	9	500	0.8	10	256	0.00031	0.00031	0.00031
Linear			0	5	100	0.5	10	256	0.00056	0.00056	0.00028

Model	Noise			Hyperparameters					MSE Loss		
	Gaussian	*Periodic*	*Salt-and-pepper* amount	Number of layers	Size of code layer	Learning rate	Epochs	Batch size	Train	Val	Test
Linear	X		0.5	5	100	0.5	10	256	0.00091	0.00093	0.00047
Linear	X	X	0.2	5	100	0.5	10	256	0.00050	0.00050	0.00050
Linear	X	X	0.1	5	100	0.5	10	256	0.00044	0.00044	0.00045
Linear	X	X	0.5	5	500	1	50	256	0.00036	0.00044	0.00044

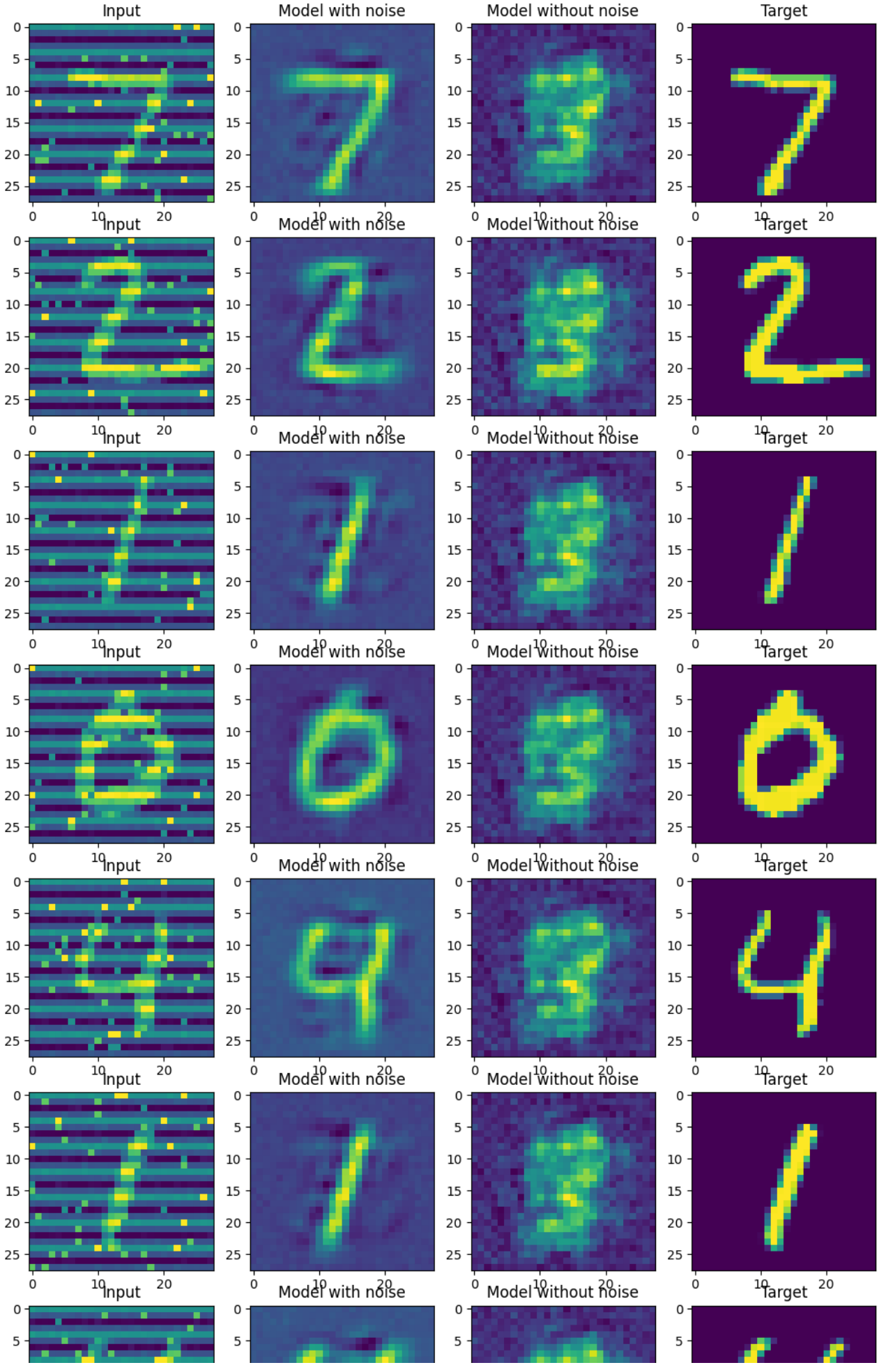
2) Training without noise

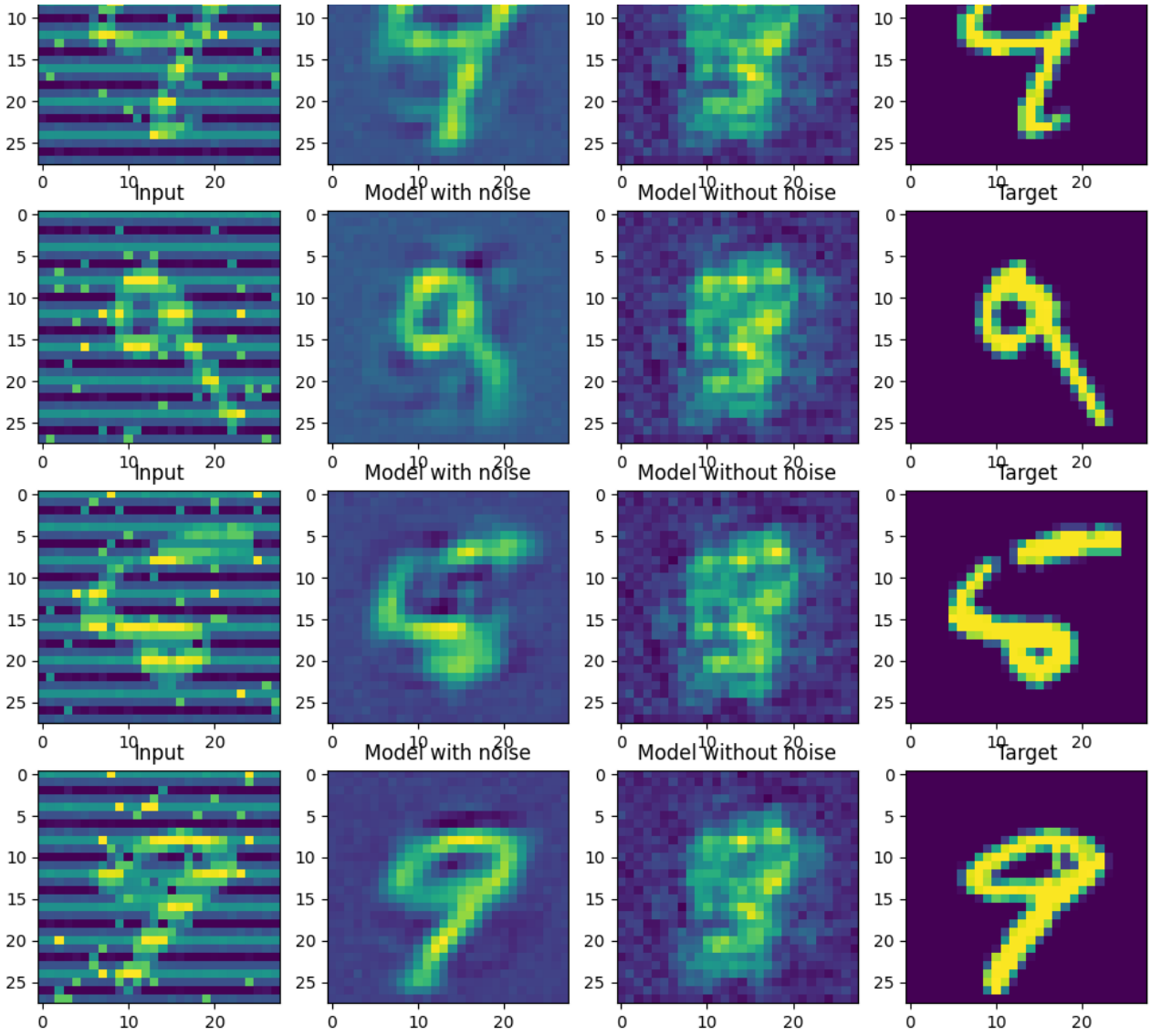
Our project explored as well the impact of training autoencoder models on noisy images versus clean (denoised) images and tested their ability to handle noisy input data. Initially, we observed that models trained exclusively on noisy images struggled to reproduce clean images. This was expected since their primary task was noise removal. **However, these models encountered challenges when presented with noisy input images, as they were not explicitly trained to generate noise-free results.**

To address this limitation, we decided to train a separate set of models on clean, denoised images to see if they could better reproduce noise-free input images (in the section Models comparison). Surprisingly, we found that models trained on clean images to reproduce clean images had worse performance when dealing with noisy input data (as illustrated in the section Comparison). This suggested that they were ill-equipped to handle noise, as they were optimized for noise-free reconstructions.

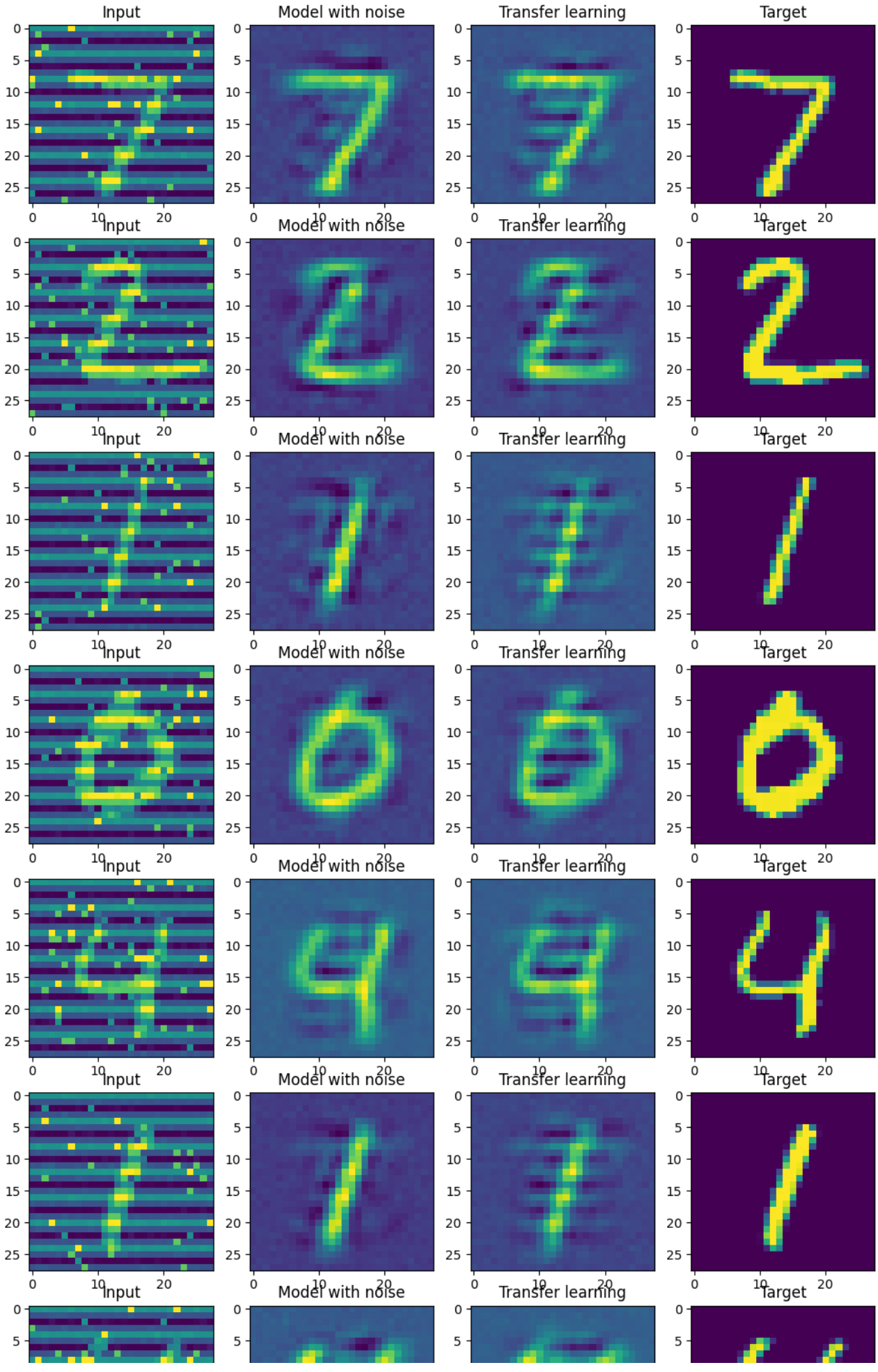


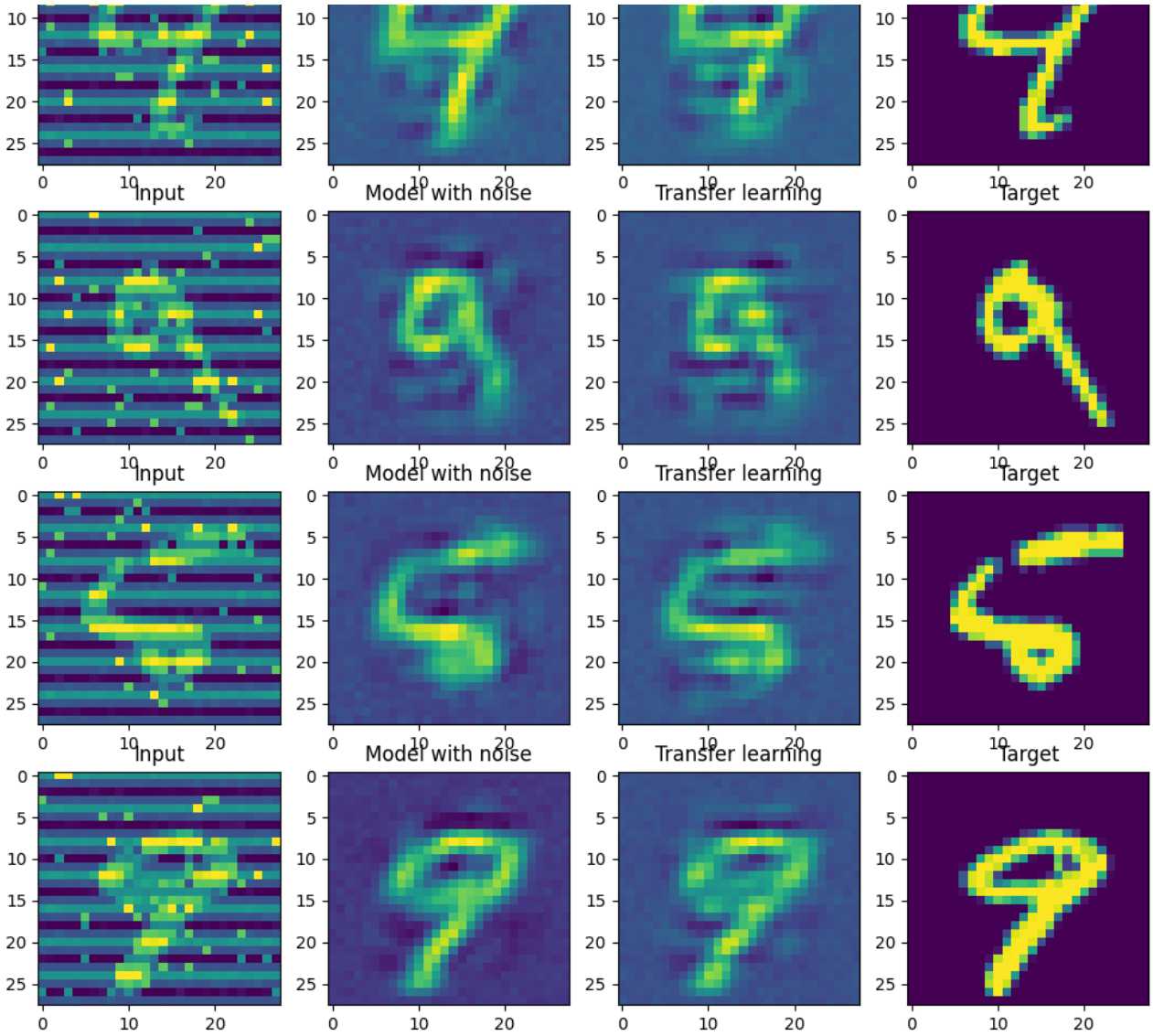
To overcome this issue, we fine-tuned the models trained on clean images with noisy data. This fine-tuning process significantly improved their ability to handle noisy images. We achieved a test loss of 0.00021, which was only slightly higher than the test loss of 0.00017 achieved by the models initially trained on noisy images. This result demonstrates the adaptability of autoencoder models through fine-tuning, allowing them to efficiently denoise and reconstruct images even when not explicitly trained for noise removal.


Input
Model with noise
Model without noise
Target



We tried another way of training our models, using the transfer learning (in the section Transfer learning). This method consist to train a model on a slightly different task (most of the time simpler) then train it on the wanted task. This technique is commonly used in computer vision with deep neural networks. In our experiment, we applied transfer learning on the model trained on free-noise. With only 10 epochs the model reach an almost better accuracy than our first model trained directly to denoise images.

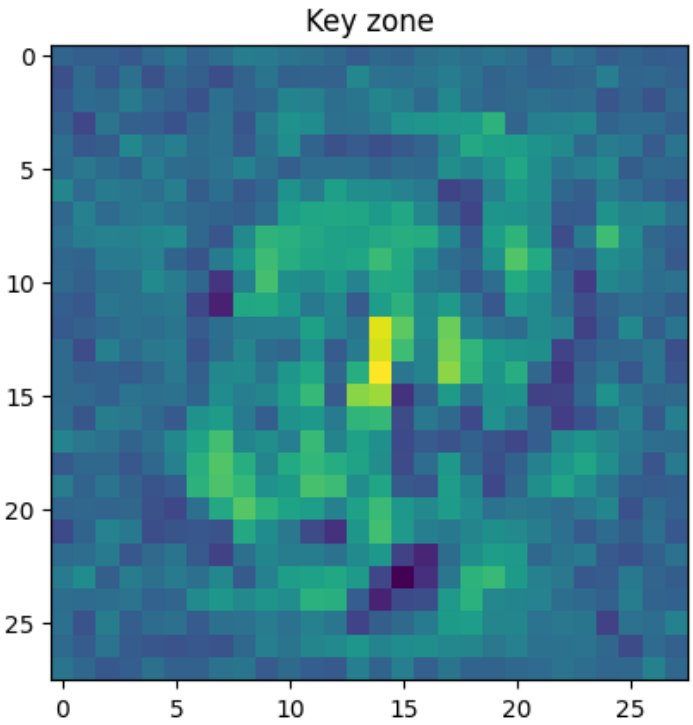

Input
Model with noise
Transfer learning
Target



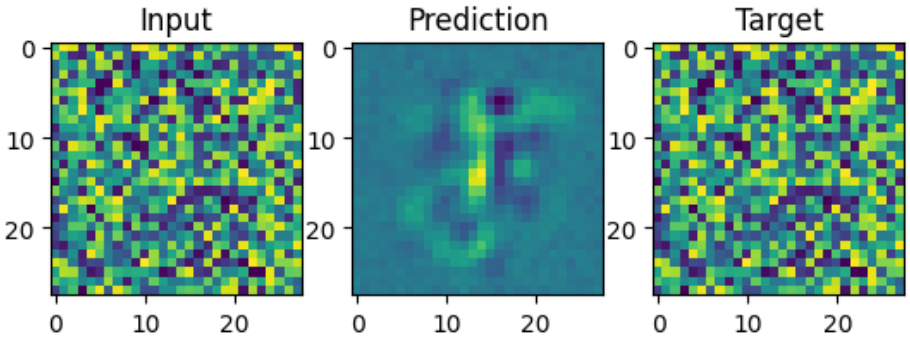
3) Generating images

Autoencoders are commonly known for being able to generate images. We didn't try to generate digits at first but realized that it could be possible with a simple experiment (illustrated in the section Generating digits).

Once our models trained, we wanted to know how they differenciate noise from interesting features. We tried to illustrate it by generating an image from a blank input. The result given by our model was a blue dark square with a light spot in the middle. Somehow the spot looks like a 3. Our interpretation is that the model learned that the digits are generally written in the middle of the picture and the rest is noise, so he darkens the background and he enlights the middle of the image. We also realized that the digit 3 contains two curves that are common to many of the other digits. For example, 2 and 9 have an upper right curve, like 3. This experiment showed that our model recognize digits, amoung noise, based on their position in the image and their curves.



Observing that given a blank image the model's output looks like a 3, we tried with random inputs. The results weren't similar to digits. Moreover, even if our inputs were all differents, it seemed like the output stayed the same.



Generating images from random inputs might be too difficult for our linear autoencoder. Studies showed that linear autoencoders aren't efficient for image generation.

Conclusion

This investigation has demonstrated that linear autoencoders excel in identifying key areas and reducing low noise in and around those areas. However, they struggle to produce complete images. While this project may seem straightforward, it put light on the limitations of linear autoencoders.

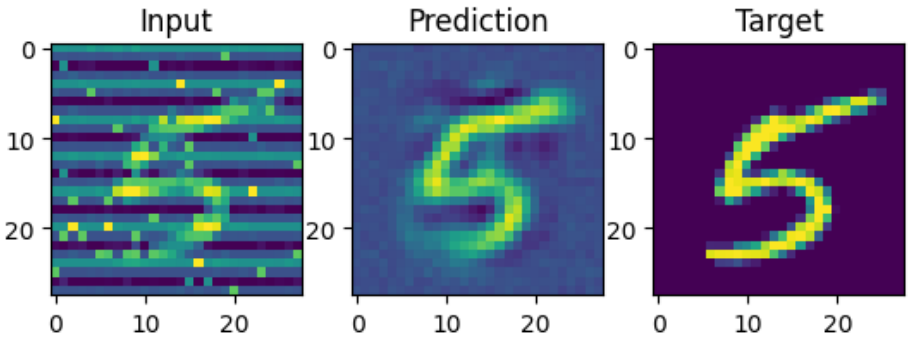
Moving forward, there is room to explore the capabilities of this type of autoencoder by integrating another neural network to recognize the digit within the image. By comparing the predictions of this additional network for the denoised image and the original one, we can aim for more accurate and complete image reconstructions.

Another intriguing idea for future research involves applying hysteresis to the autoencoder's output image. This could enhance the consistency of the denoised image, ensuring it more faithfully reproduces the borders of key objects.

Furthermore, for more robust image denoising, it might be worthwhile to investigate other types of autoencoders, such as Variational Autoencoders (VAEs) and Convolutional Autoencoders. VAEs introduce probabilistic elements into the

encoding process, allowing for richer and more diverse image generation. Convolutional Autoencoders, on the other hand, are specifically designed for image-related tasks, utilizing convolutional layers to capture spatial features more effectively.

Incorporating VAEs and Convolutional Autoencoders into future work could potentially give more promising results in terms of denoising efficiency and image completion. These models could address the limitations observed with linear autoencoders in this project by providing more comprehensive and accurate image reconstruction in image denoising tasks.



Bibliography

- *Self-Supervised Dynamic CT Perfusion Image Denoising With Deep Neural Networks*, written by Dufan Wu, Hui Ren, and Quanzheng Li
- *Gaussian noise*, Wikipedia
- *Impulse noise (salt-and-pepper)*, authored by KEA (Image & Data Processing, Analysis and Computational Geometry Research Group)
- *Periodic noise in images*, authored by Fernando Chamizo
- *Various Optimization Algorithms For Training Neural Network*, written by Sanket Doshi on Medium