University of Burgundy

Assignment I

Learning to Sum and to Generate Images of Numbers

Kimberly Grace Sevillano Colina

---

1 Introduction

The realm of machine learning and image analysis has seen significant advancements with the advent of neural network architectures like Multi-Layer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), and autoencoders. This assignment aims to provide hands-on experience with these architectures, focusing on understanding and manipulating them in a structured and incremental manner.

The primary goal is to design a model capable of interpreting images of numbers, performing addition on these images, and subsequently visualizing the result. This involves the integration of encoders, decoders, and addition models, denoted by E, D, and P respectively.

2 Part I - Environment Setup & Inputs

2.1 Environment Setup

To set up the conda environment for working on the local machine, follow the steps below:

1. Create a new conda environment: `conda create -n ML_Learning2Add_SeviColi`
2. Activate the environment: `conda activate ML_Learning2Add_SeviColi`
3. Install Jupyter Notebook: `conda install jupyter`
4. Navigate to the workspace directory of the course and run Jupyter: `jupyter-notebook`

2.2 Input Images

This assignment utilizes the MNIST digits dataset, renowned for its collection of handwritten digits. Each image in the MNIST dataset is a grayscale snapshot with a resolution of 28x28 pixels, where every pixel value ranges from 0 (indicating white) to 255 (indicating black)

3 Part II - Reconstruction Autoencoder

3.1 Architecture

3.1.1 Architecture 1: Convolutional Transpose Autoencoder

Encoder : This part utilizes three convolutional layers with 32, 64, and 128 channels, respectively. Each of these convolutional layers is followed by a max-pooling layer. The extracted features are then transformed through two fully connected layers, first from a 128x3x3 dimensional space to a 256-dimensional space, and then to a 2-dimensional space.
Decoder1 : In this section, it starts with two fully connected layers that expand the 2-dimensional latent representation to a 256-dimensional space and further into a 128x7x7 dimensional space. This is followed by three transposed convolution layers, aimed at gradually reconstructing the original image. The first transposes from 128 to 64 channels, the second from 64 to 32, and the last from 32 to 1 channel. ReLU activation is used in the transposed convolution layers, and the sigmoid activation in the final layer to complete the image reconstruction.

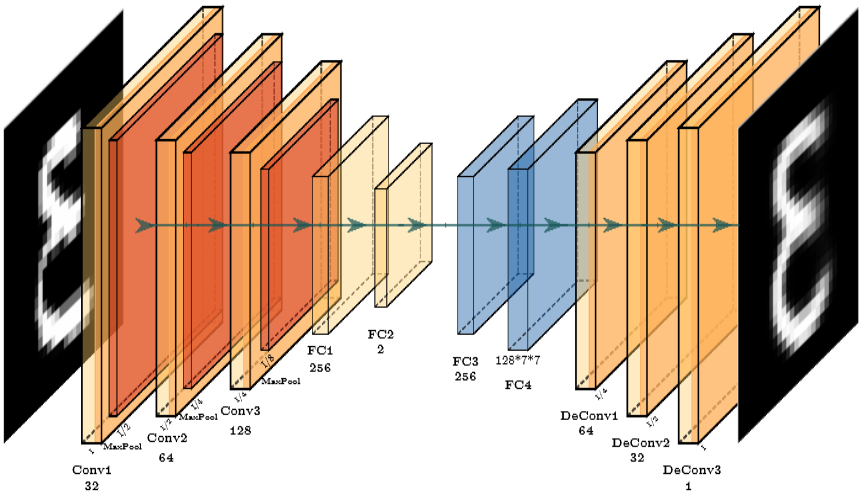


Figure 1: Architecture of the Reconstruction Autoencoder 1.

Overall, this Autoencoder architecture employs a combination of convolutions and transposed convolutions for effective encoding and decoding of images, facilitating the reduction and subsequent reconstruction of essential image features.

3.1.2 Architecture 2: Autoencoder architecture combining Conv2d layers and Upsampling

Encoder : The same as Architecture 1.
Decoder2 : The Decoder starts with two fully connected layers that expand the 2-dimensional latent representation to 256 and then to a 128x7x7 dimensional space. Next, upsampling is employed to increase the image size, followed by a convolutional layer that reduces the channels from 128 to 64. This process is repeated, reducing from 64 to 32 channels in the next convolutional layer. Finally, a third convolutional layer reduces from 32 channels to 1, with a sigmoid activation to generate the reconstructed image.

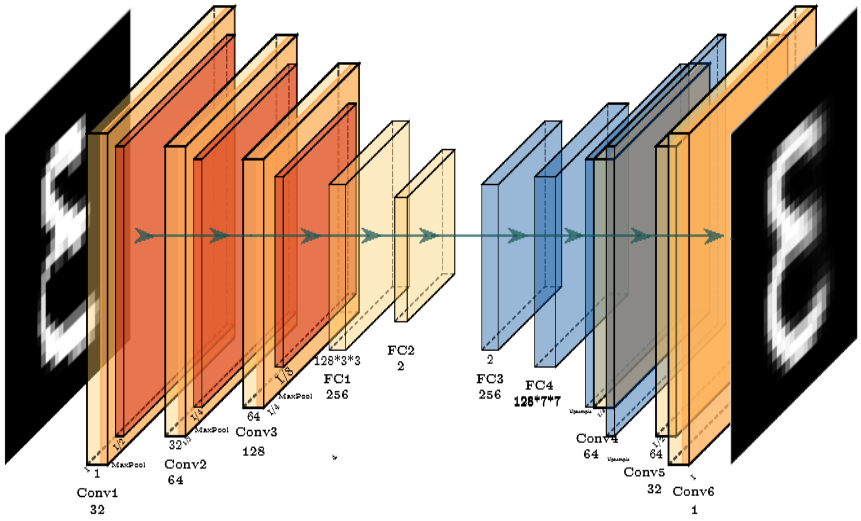


Figure 2: Architecture of the Reconstruction Autoencoder 2.

In summary, this Autoencoder architecture with Conv2d and Upsampling utilizes convolutions to encode the image into a reduced latent space and then employs upsampling and additional convolutions for decoding or reconstructing the image, maintaining and recovering its essential features.

3.1.3 Architecture 3: Autoencoder architecture combining Conv2d layers with Interpolation

Encoder : The same as Architecture 1.
Decoder3 : The Decoder begins its process by expanding the 2-dimensional latent representation back to a 256-dimensional space and further into a 128x7x7 dimensional space using two fully connected layers. It then employs interpolation to upscale the image size (14x14 after the first interpolation, and 28x28 after the second). Each interpolation step is followed by a convolutional layer that reduces the channels successively (from 128 to 64, and then from 64 to 32). Unlike in previous architectures, no interpolation is applied after the last convolutional layer, which is a 32 to 1 channel reduction with a sigmoid activation for final image reconstruction.

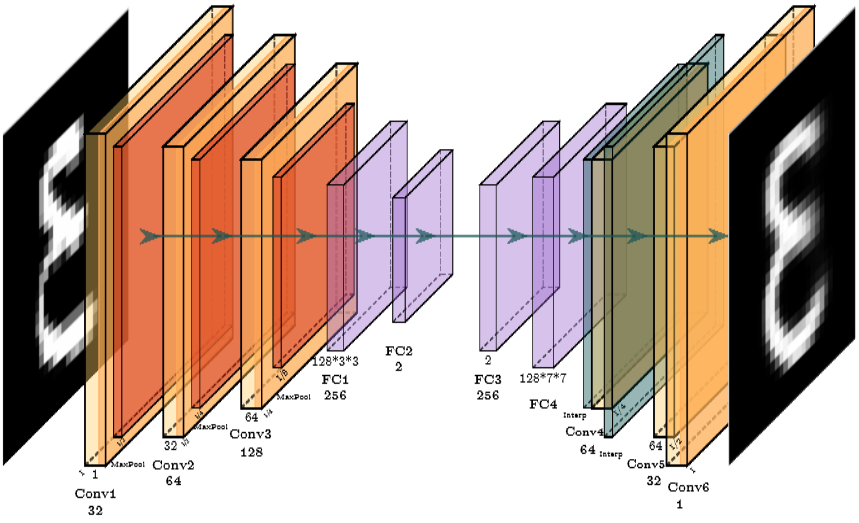


Figure 3: Architecture of the Reconstruction Autoencoder 3.

Overall, this Autoencoder with Conv2d and Interpolation effectively encodes images into a compressed latent space using convolutional layers and max-pooling, and then decodes or reconstructs the images using a series of fully connected layers, interpolations, and convolutional layers. This architecture is particularly notable for its use of interpolation as a means to upscale the image size during the decoding phase, which is a key distinction from the previous architecture that used upsampling.

3.2 Questions

3.2.1 Which activation should be used in the output layer of your decoder model? Why?

The sigmoid activation function must be used in the output layer of the decoder because we want the reconstructions to be in the same range as the input images, which is [0, 1] for normalized images. Also, if we use a binary cross-entropy loss later, we need the outputs to be in this range.

3.2.2 How could you ensure the same conditions to train and evaluate your models, i.e., you would like that the same model trained twice on the same data to have similar performance (and similar weights)? This is useful for comparing versions of models. How could you do this?

To ensure that the model behaves deterministically and that two runs with the same data produce similar results, we must set the seeds and configure certain parameters in PyTorch.

```
def set_seed(seed_value=42):
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)
    torch.backends.cudnn.deterministic = True
set_seed(42)
```

3.2.3 Train your models using an L2 reconstruction loss for 100 epochs and Adam with a learning rate 0.001. Please provide the reconstruction error curves (using RMSE metric average for all pixels) for the training and validation data sets for your three models using TensorBoard or wandb. Which one is the best architecture and which one you would stick with? Why?

DecoderConvTranspose (Blue Graph): This model achieves the lowest RMSE in both training and validation, suggesting it is the most effective at reconstructing the MNIST dataset images. The use of transposed convolutions seems to be beneficial, as they can directly learn the upsampling filters during the training process, potentially resulting in more detailed and accurate reconstructions.

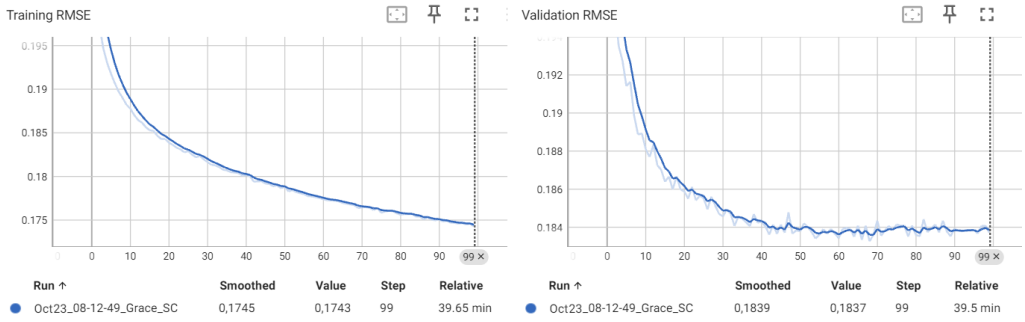


Figure 4: Training and Validation Error Curve for Autoencoder 1

Figure 5: Input and reconstruction image for Autoencoder 1

DecoderConvUpsample (Fuchsia Graph): The performance of this decoder is slightly inferior to the DecoderConvTranspose, but it still maintains a competitive RMSE, especially in the validation phase. This decoder uses nearest neighbor upsampling followed by convolutions, which is a simpler approach that can sometimes lead to less refined outputs. Despite this, the model is still effective and exhibits good generalization from the training to the validation dataset.

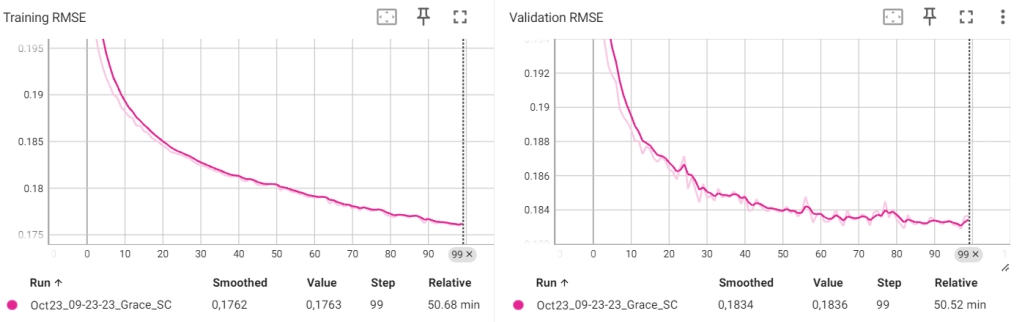


Figure 6: Training and Validation Error Curve for Autoencoder 2

Figure 7: Input and reconstruction image for Autoencoder 2

DecoderInterpolation (Orange Graph): The decoder with interpolation has the highest RMSE among the three. It uses a fixed method for upsampling (nearest interpolation) followed by convolutions, which might be less flexible than the learned parameters of transposed convolutions in the DecoderConvTranspose. The interpolation technique may result in a loss of some high-frequency details, which could explain the slightly higher RMSE.

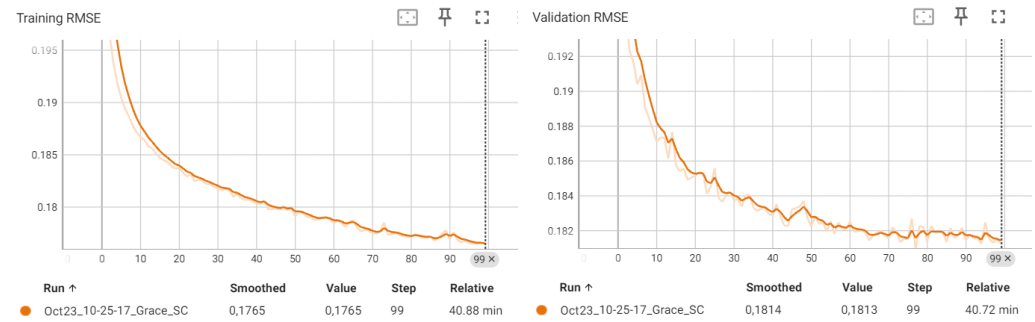


Figure 8: Training and Validation Error Curve for Autoencoder 3

Figure 9: Input and reconstruction image for Autoencoder 3

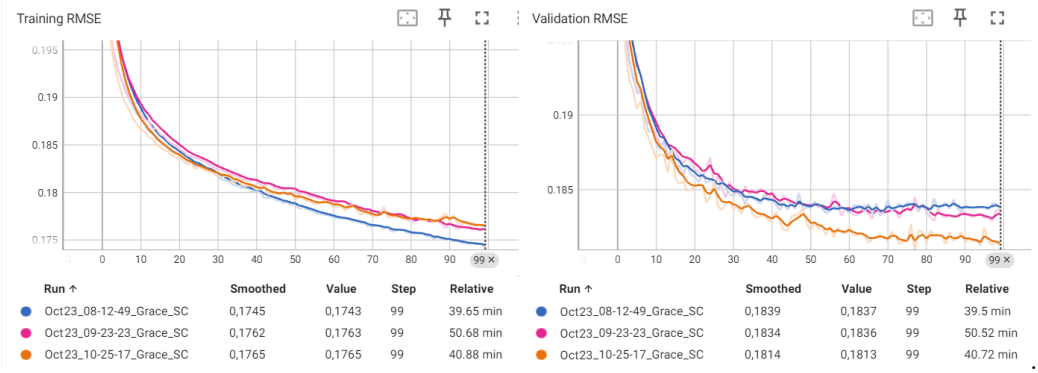


Figure 10: Comparison of Training and Validation Error Curves for Three Autoencoders with L2 (MSE) loss.

In summary, the DecoderConvTranspose is the most successful at minimizing the RMSE, likely due to its ability to learn the upsampling process. The DecoderConvUpsample offers a simple yet effective approach, but it doesn't quite match the performance of the learned transposed convolutions. The DecoderInterpolation, while still a valid approach, seems to be less suited for this task compared to the other two, potentially due to its less nuanced way of increasing the image resolution. Each decoder has its merits, but the DecoderConvTranspose appears to provide the best balance between complexity and performance for this specific application

3.2.4 Do you observe any overfitting in the obtained training curves? How would you select the best weights over the epochs for a specific architecture? How would you select the best weights over the epochs for a specific architecture?

No, I don't observe any clear signs of overfitting. The RMSE for both the training and validation sets is decreasing and stabilizing, which indicates that the models are generalizing well to unseen data. There is no significant uptick in the validation RMSE, which would suggest that the model is beginning to memorize the training data rather than learning generalizable patterns.

To select the best weights over the epochs for a specific architecture, I would typically use a combination of checkpointing and early stopping. I'd save the model's weights whenever there is an improvement in validation RMSE. This way, even if the model begins to overfit later during training, I can always revert to the best-performing checkpoint. Early stopping would be employed as an additional safeguard to halt training if the validation loss starts to increase consistently, which is a sign of overfitting.

Given that the validation RMSE in my current models does not show an upward trend, I would choose the weights from the last epoch for each model. However, I would still implement checkpointing during training as a best practice, ensuring that I can always select the best weights based on validation performance.

3.2.5 Please retrain your models now using binary cross-entropy as the reconstruction loss. What do you observe compared when using L2?

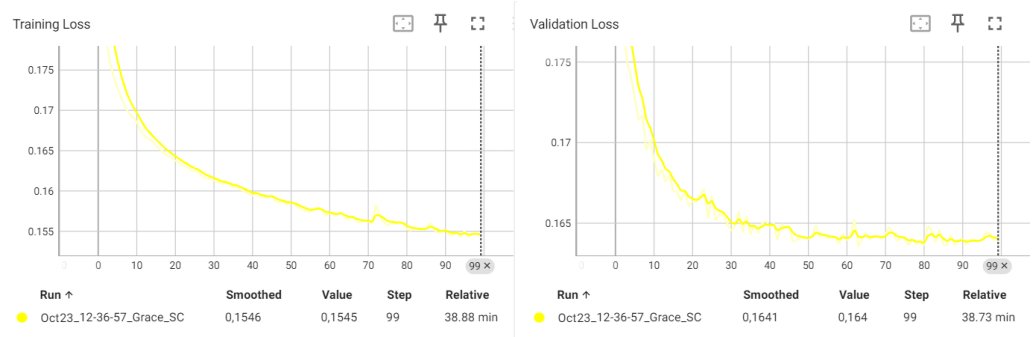


Figure 11: Training and Validation Error Curve for Autoencoder 1 with BCE loss

Figure 12: Input and reconstruction image for Autoencoder 1 with BCE loss

Figure 13: Training and Validation Error Curve for Autoencoder 2 with BCE loss

Figure 14: Input and reconstruction image for Autoencoder 2 with BCE loss

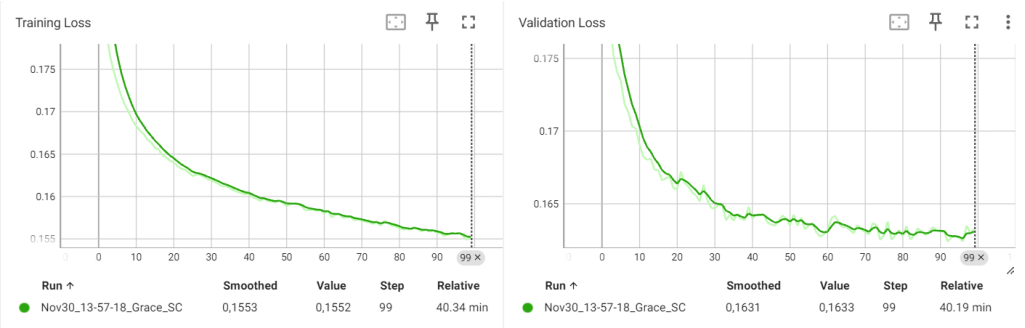


Figure 15: Training and Validation Error Curve for Autoencoder 3 with BCE loss

Figure 16: Input and reconstruction image for Autoencoder 3 with BCE loss

Observations:

• With L2 (MSE): The blue, fuchsia, and orange autoencoders display a swift decrease in both training and validation RMSE in the initial iterations, after which the metrics stabilize. Stabilization occurs around the 100-step mark, taking about 40 minutes for all cases. The final training RMSE settles at approximately 0.1745, 0.1762, and 0.1765, respectively, and for validation around 0.1839, 0.1834, and 0.1814. Convergence is quick at the start but plateaus after roughly 20 steps.

• With BCE Loss: The yellow, sky blue, and green autoencoders show a smoother and more continuous decrease in both training and validation loss. The loss stabilizes at lower values than with L2, around 0.1546, 0.1556, and 0.1553 for training, and 0.1641, 0.1643, and 0.1631 for validation. These also stabilize near the 100-step mark with similar timings to L2, but they achieve lower loss values. The loss reduction is more uniform and lacks the dramatic initial drops seen with L2.

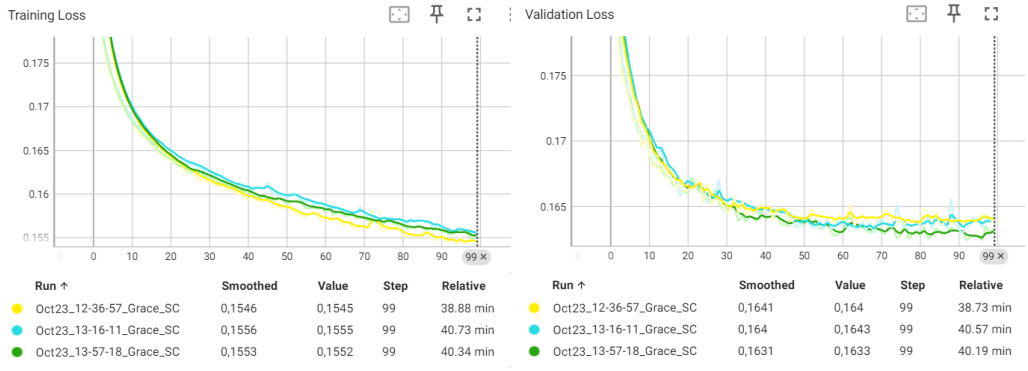


Figure 17: Comparison of Training and Validation Error Curves for Three Autoencoders with BCE loss.

Comparison:
Loss behavior with BCE is smoother and more stable compared to L2, stabilizing at lower values. This might indicate that BCE is a more effective loss metric for this problem, at least with the current hyperparameters and architecture in use. BCE loss seems to facilitate finer tuning during training, which might result in a model that generalizes better on unseen data, as suggested by the lower validation loss compared to L2. The choice between L2 and BCE may depend on the data type and specific task; BCE tends to perform well with classification problems or when the output is probabilistic, while L2 is commonly used in regression tasks.

In summary, while both methods achieve convergence, BCE loss demonstrates a more stable behavior and reaches convergence at a lower value than L2, suggesting a potential preference for BCE in this particular scenario.

3.2.6 Why it is possible to use binary crossentropy (BCE) with the data from MNIST?

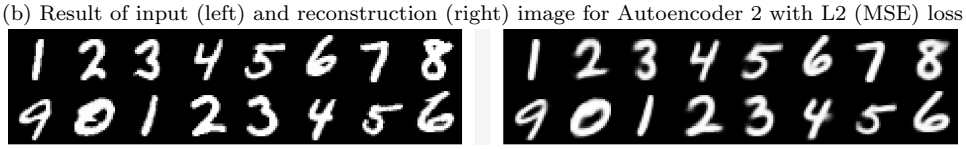
Binary Cross-Entropy (BCE) is well-suited for the MNIST dataset because after normalization, the grayscale pixel values lie within the [0,1] range. BCE is designed for such binary or near-binary values, especially when the model's output is bounded between [0,1], typically achieved with a sigmoid activation function in the final layer. In the context of autoencoders, the task of reconstructing an image can be seen as a set of binary classifications for each pixel, thus making BCE an appropriate choice for the loss function. The reconstruction considers whether each pixel should be activated (close to 1) or not (close to 0), aligning with the binary nature of BCE.

Moreover, the charts comparing training and validation loss suggest that BCE provides better results for the MNIST dataset compared to L2 loss. This may be due to BCE offering more informative gradients for optimization or being more inherently aligned with the binary nature of the normalized MNIST pixel values.

3.2.7 Change the dimension of the code from 2 to 7 in your encoder (and operations in the decoder if needed). Please show visualizations for five reconstructed numbers for each loss.

(a) Result of input (left) and reconstruction (right) image for Autoencoder 1 with L2 (MSE) loss

(b) Result of input (left) and reconstruction (right) image for Autoencoder 2 with L2 (MSE) loss



(c) Result of input (left) and reconstruction (right) image for Autoencoder 3 with L2 (MSE) loss

Figure 18: Comparison of input and reconstruction image results from the training of three L2 lossy autoencoders (MSE) each with a 7-dimensional vector.

(a) Result of input (left) and reconstruction (right) image for Autoencoder 1 with BCE loss

(b) Result of input (left) and reconstruction (right) image for Autoencoder 2 with BCE loss

(c) Result of input (left) and reconstruction (right) image for Autoencoder 3 with BCE loss

Figure 19: Comparison of input and reconstruction image results from the training of three BCE lossy autoencoders each with a 7-dimensional vector.

The Tensorboard tool was used for the visualizations

Since changing the code size in the autoencoders can influence their capability. A larger code size means that the encoder can potentially capture more information from the input, which the decoder can use to produce better reconstructions. However, it is also essential to be cautious, as increasing the code size too much can lead to overfitting.

4 Part III - Learning to Perform Addition from Images

4.1 Samples visualization of the train loader

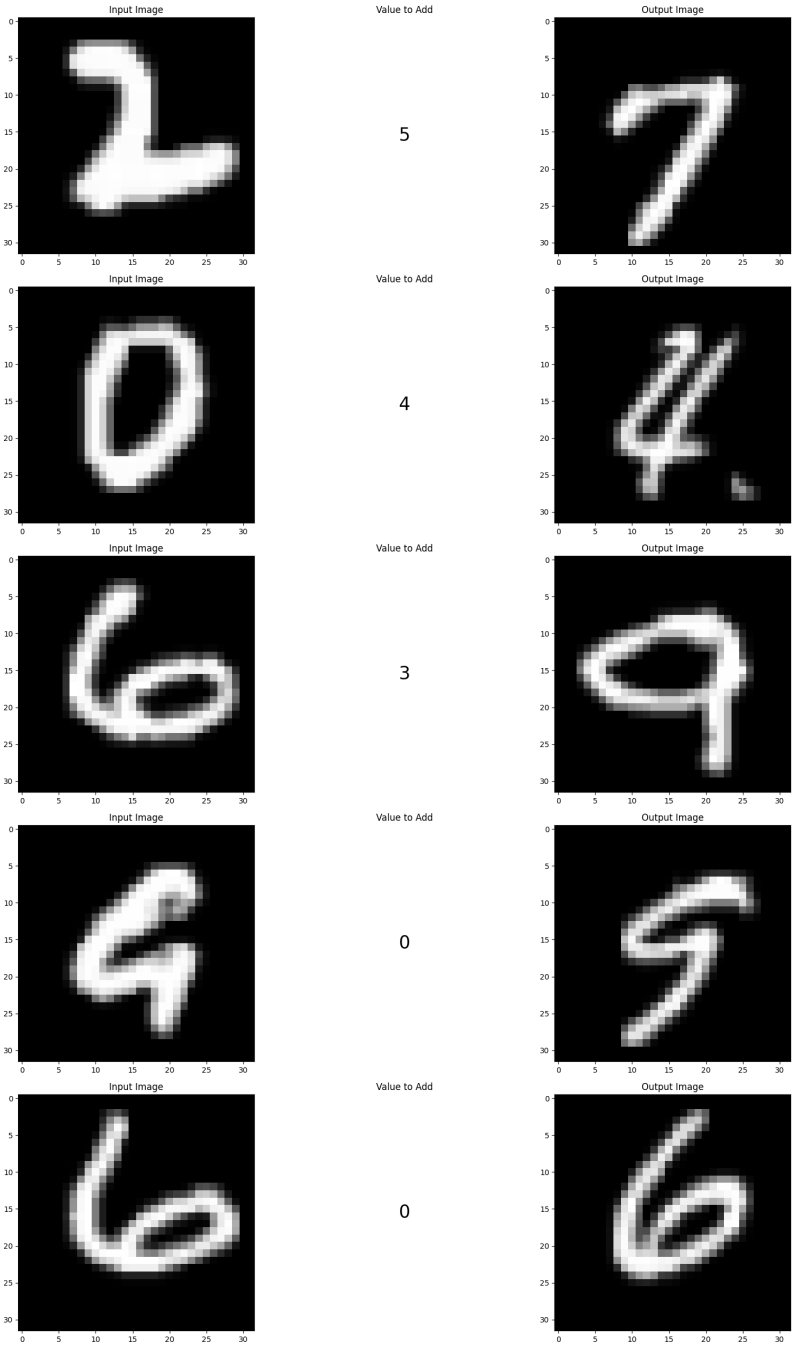


Figure 20: Training and Validation Error Curve for Autoencoder 3 with BCE loss

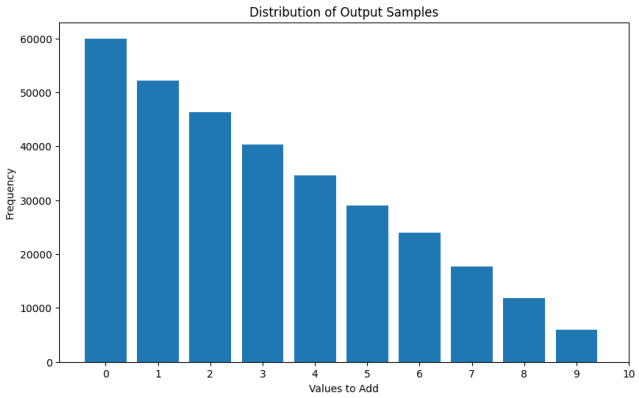


Figure 21: Display of the distribution of output samples

4.2 Please visualize some input and output samples to understand what is being done. Please display the distribution of the output samples. Is this created dataset balanced?

When we look at the output samples after using the transform_data_add function, we see that they are not evenly spread out. This happens because the function creates pairs of numbers to add together, and it's more common to get smaller sums than larger ones. For example, there are more chances to get a sum of zero than a sum of nine because we only consider sums that are single digits.

Despite this, the actual images used for training the model are evenly distributed. This means that for every type of image we want to recognize, we have the same amount.

In simple terms, while the sums are not evenly spread out, the images themselves are. This is good because it helps the model learn equally from all different types of images.

Note! ;

After the transformation, our data loses the characteristics of data and targets. This is important to note when creating inferences with our trained model. For example, when we will see below and we want to extract the label of the image of a number in the function: "get_image_of_number"

4.3 Train your complete training model for 100 epochs using Adam with a learning rate of 0.001. Use as an evaluation metric the L2 reconstruction error between the expected added image and the one provided by the model. Was our model affected by overfitting?

In the TensorBoard-generated graphs, there is a decreasing loss for both the test and training sets. This suggests no evident signs of overfitting, as typically indicated by the test loss diverging and increasing while the training loss continues to decrease, which is not observed here.

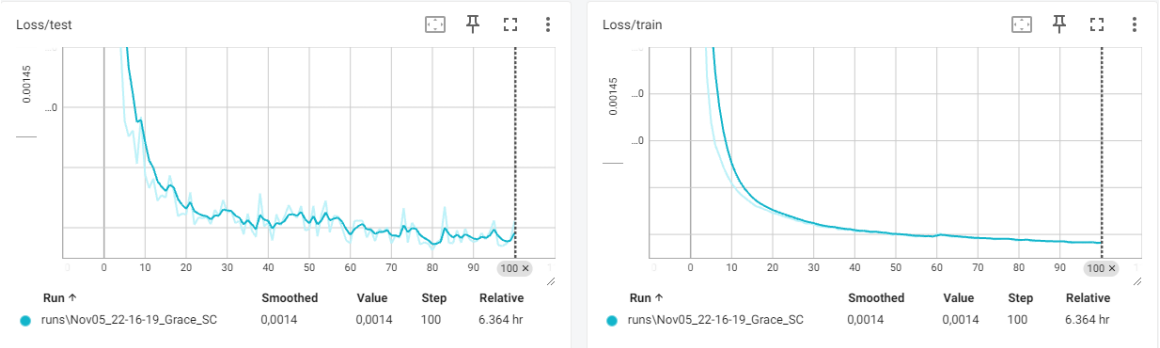


Figure 22: Training and test Error Curve for Complete model (with adder) loss.

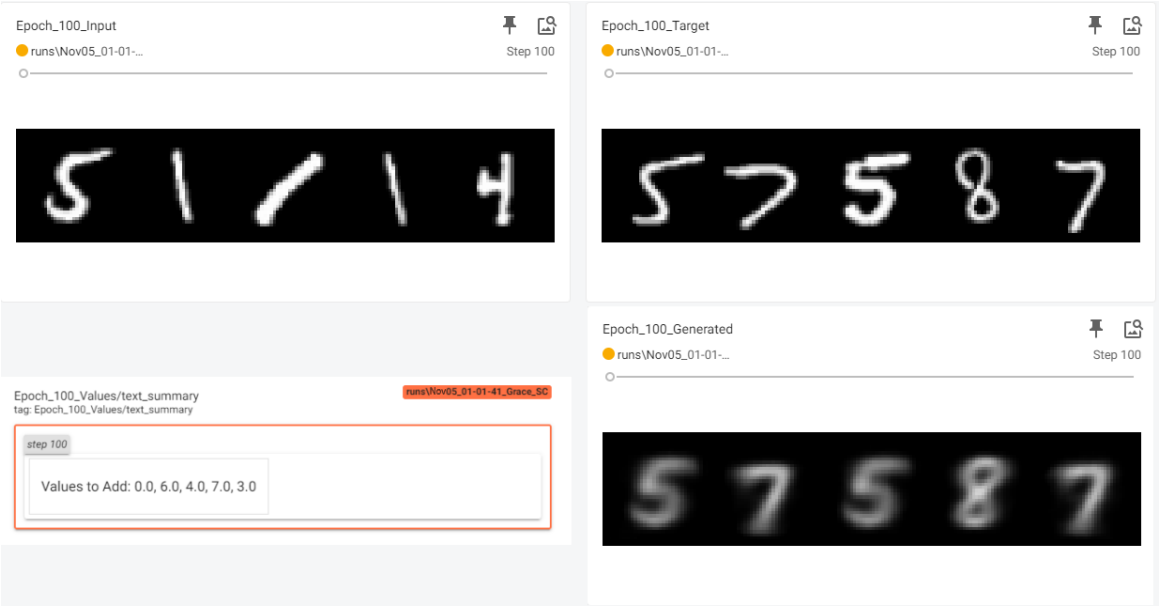


Figure 23: Display of the input, target and generated images as well as the numerical value added.

4.4 Create an inference/test model using the learned encoder, decoder, and model addition submodels (that is only solid line modules should remain in the test as shown in the overview image). Your inference model should output only the expected number images of the addition.

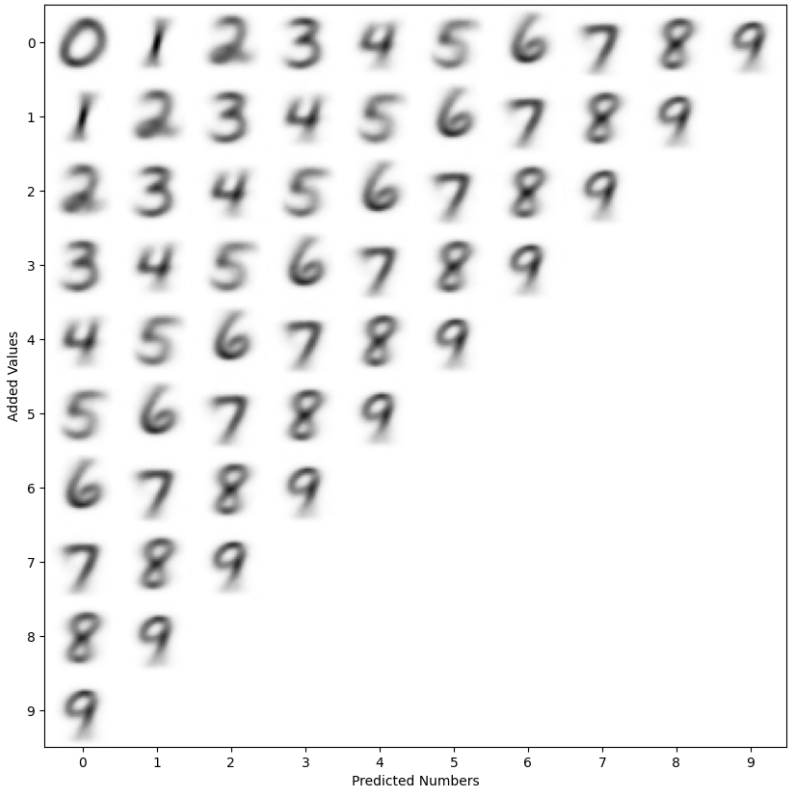
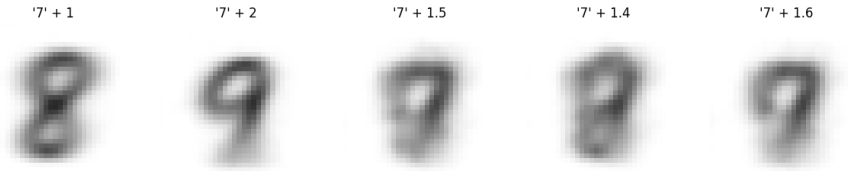


Figure 24: Display of the inference model using the learned encoder, decoder, and model addition submodels (expected number images of the addition)

4.5 Finally, provide and visualize the generated output of your model to the following inputs: ('image of 7', 1), ('image of 7', 2) and ('image of 7', 1.5), ('image of 7', 1.4), ('image of 7', 1.6). What can you observe?



The inference tends to be wrong when adding numbers with decimals, and the model tries to show only the sum between the base number and the decimal integer. PS: When the decimal is equal to or greater than 0.5 the image is blurred but tends to resemble the image of the base number + decimal integer.

5 Part IV - Addition with Generative Model

Let's turn this model into a generative one, such as that we now handle distributions and generate new samples from it.

5.1 Please re-train your variational model and perform the inference again for the following inputs : ('image of 7', 1), ('image of 7', 2) and ('image of 7', 1.5), ('image of 7', 1.4), ('image of 7', 1.6). Compare the generated results with the previous model in Part III. What can you observe?

5.1.1 Unbalanced data

Since training was costing my model, as it was generating pure fuzzy nines (9), one of the strategies I did was to balance my training and validation data. I know at the beginning of my report I said it was not a problem, but because I was only getting fuzzy nines in each training I included this data treatment and generated the train_loader_tra_VAE.

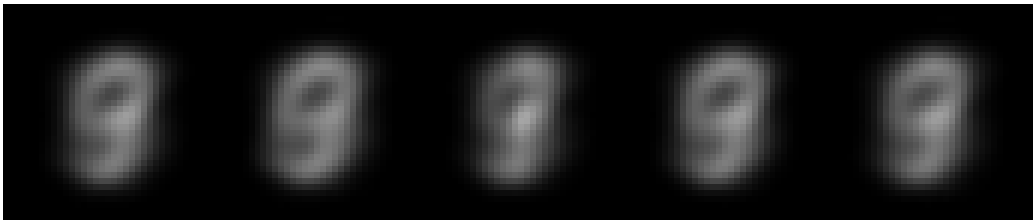


Figure 25: Display of the bad images generated.

I want to make a small comment here , although I was training with transformed and balanced data (train_loader_tra_VAE), unfortunately my model was still generating only fuzzy nines (9) , so I created the ValueToLatent function , which converts the aggregate value to a latent space that can be handled in the CompleteModelVAE function but this time with the VAE components. And with that essential change my model was able to train correctly.

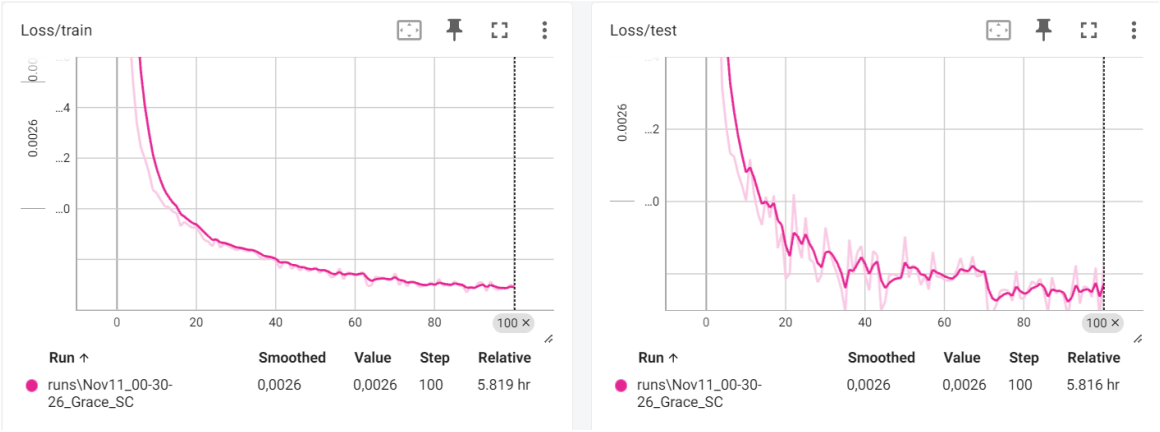


Figure 26: Training and test Error Curve for Complete model VAE loss.

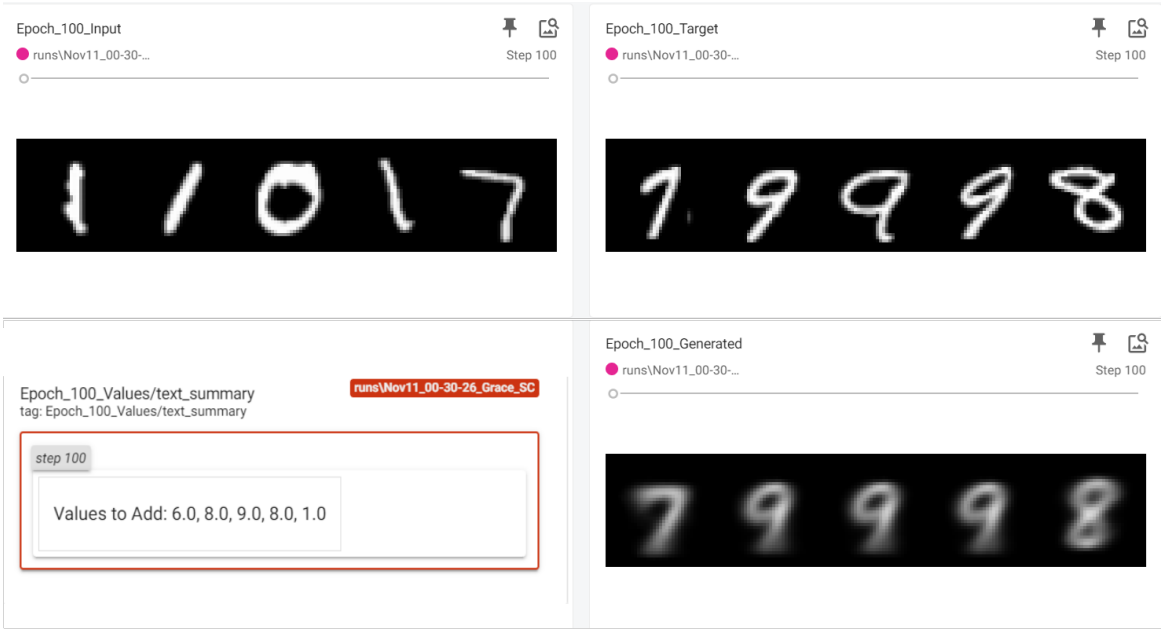


Figure 27: Display of the input, target and generated images (VAE) as well as the numerical value added.

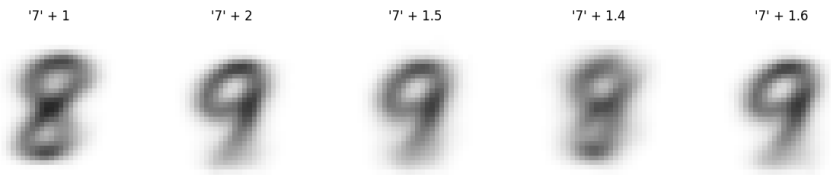


Figure 28: Display of the inference VAE model.

Unlike the autoencoder the VAE when the decimal is greater or less than 0.5 the inferred image is averaged correctly, but when it is 0.5 the inferred image is averaged to the previous base number but in a blurred way.

Continued Generative changes ...

5.2 Finally, go a step further and also make your MLP addition model (P) a generative one, i.e., it now also outputs the mean and standard deviation from the distribution. Compare the results against the two previous models.

This last task has taken me a long time to perform, especially to improve the quality of the reconstruction and the generation of the correct numbers in my VAE generative model, since I had to experiment with different weights in the loss function.

L2_orig	L2_added	kld_weight_encoder	kld_weight_adder	Image generated
0.1	0.5	0.00001	0.00001	only fuzzy nines
0.1	0.5	0.0001	0.0001	only fuzzy nines
0.3	0.7	0.0001	0.0002	only fuzzy nines
0.5	0.5	0.0005	0.0005	only fuzzy nines
0.7	0.7	0.0002	0.0002	only fuzzy nines
1	1	0.0005	0.0005	only fuzzy nines
0.5	0.5	0.0001	0.0002	just a number with fuzzy nines
0.6	0.8	0.00015	0.00015	just a number with fuzzy nines
1	1	0.0001	0.0001	just a number with fuzzy nines
1	1	0.0001	0.0001	just a number with fuzzy nines
1.2	0.8	0.00005	0.00005	just a number with fuzzy nines
1	0.2	0.0001	0.0001	generates numbers different from the target
1	0.4	0.0001	0.0005	generates numbers different from the target
1	0.4	0.0001	0.0005	generates numbers different from the target
0.9	0.1	0.0002	0.0002	generates numbers different from the target
0.1	1	0.00001	0.00001	correct

Table 1: All the weights I tested until I got to the right one.

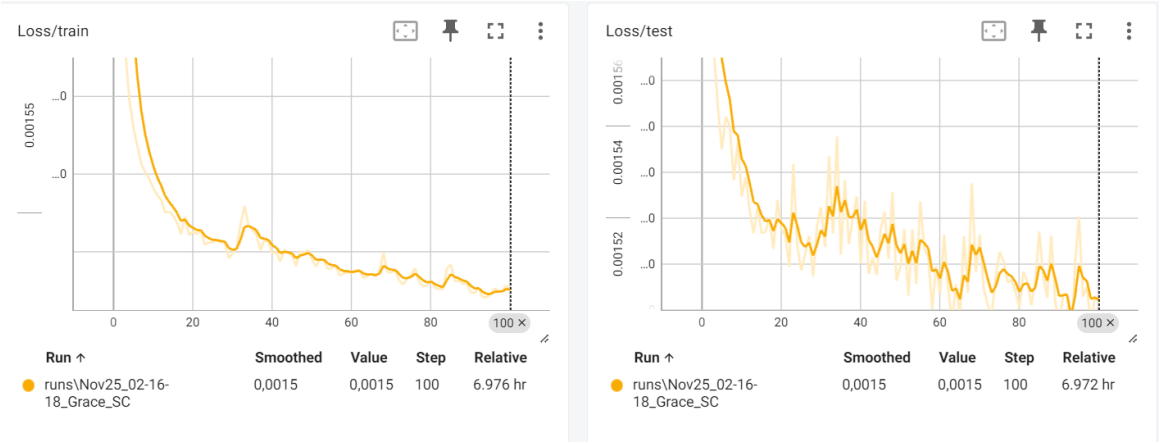


Figure 29: Training and test Error Curve for Complete model VAE loss.

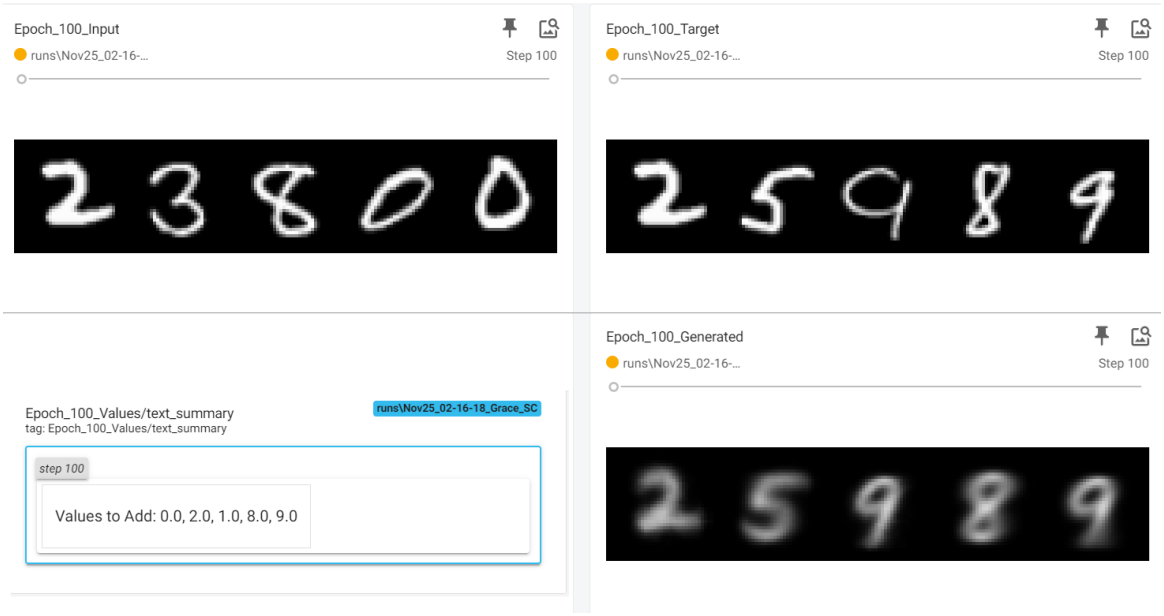


Figure 30: Display of the input, target and generated images (VAE) as well as the numerical value added.

Figure 31: Display of the inference GEN model.

Unlike the VAE the generative model when the decimal is equal to or greater than 0.5 the inferred image is correctly averaged to the next base number, and when it is less than 0.5 the inferred image is averaged to the previous base number correctly.