Université catholique de Louvain

LINMA2472 : Algorithms in data science

Homework 3 : GAN and CNN to conquer MNIST

Students :	Vincent Cammarano - 5391 21 00
	Louis Parys - 7256 17 00
	Mattias Van Eetvelt - 1660 18 00
Group :	4
Professors :	Jean-Charles Delvenne
	Gautier Krings
	Estelle Massart
	Rémi Delogne
	Bastien Massion
	Brieuc Pinon

November 2022

1 Train a GAN on MNIST for image generation

1.1 Your GAN model

Before describing the architectures of the two models, a few words on the parameters. We kept the dimension of the latent space as defined in the notebook, it is 100 which is a regular dimension for latent space of GAN application, the latent space dimension will be described in detail later. We also kept the batch_size of 32 which gives 32 images at once to our discriminator or 32 vectors at once for the generator. We have kept the architecture given with the project. Here is the detail of the generator :

— nn.Linear(dim_latent_space, 64*7*7) the dimension of the latent space is kept too so it is equal to 100

— nn.ConvTranspose2D(64, 64, kernel_size=3, stride=2, padding=1, output_padding=1)

— nn.BatchNorm2d(64)

— nn.ConvTranspose2d(64, 32,kernel_size=3, stride=1, padding=1)

— nn.BatchNorm2d(32)

— nn.ConvTranspose2d(32, 16, kernel_size=3, stride=1, padding=1)

— nn.BatchNorm2d(16)

— nn.ConvTranspose2d(16, 1, kernel_size=3, stride=2, padding=1, output_padding=1)

All of these layers are stored in variables and initialized on the creation of a generator object. Afterward, they are manipulated in the forward function which is called when the generator makes a prediction (or fake data). In order, we give as input a batch size of latent vector to the Linear layer, his job is to make a linear transformation on the data. The rest of the function is to follow the architecture described above, so to switch between Conv2D layer to reduce the dimension of the data and BatchNorm2d to normalize the data, the output of this layer is activated by a Relu layer which let out only positive input. The output of the generator is a hyperbolic tangent activation, so the range of the values is between -1 and 1.
The architecture of the generator gives a total number of 377121 parameters.

Like the generator, we kept the architecture of the discriminator as given in the notebook :

— nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1, bias=False)

— nn.Dropout2d(0.25)

— nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)

— nn.Dropout2d(0.25)

— nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)

— nn.Dropout2d(0.25)

— nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1)

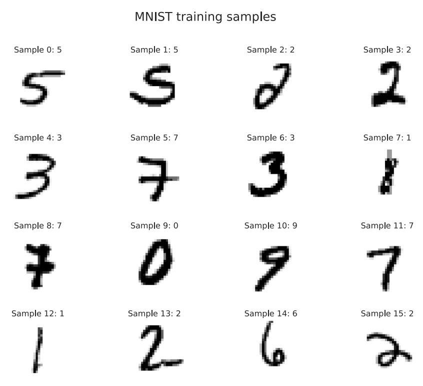
— nn.Dropout2d(0.25)

— nn.Linear(128*7*7, 1)

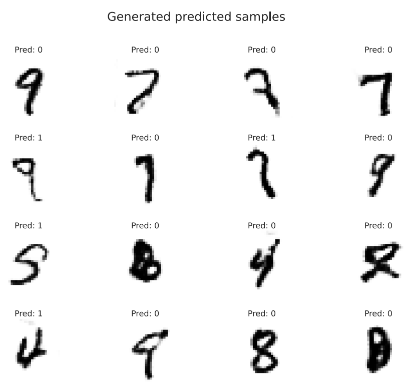
All of these layers are stored in variables and initialized on the creation of a discriminator object. Afterwards, they are manipulated in the forward function, which is called when the discriminator makes a prediction (or classifies true or fake images). We give as input a batch size of images (fake or real). As you can see the structure switch between Conv2d and Dropout2D layers, and at the end we pass it through a linear activation to get only an output of (batch_size * 1) which is the classification of the 32 images given as input. The architecture of the discriminator gives a total of 246497 parameters.

1.2 Training your GAN

We trained our GAN model on the MNIST dataset with 15 epochs because the initial 5 epochs were insufficient to produce a meaningful result. Our generated images, which were supposed to be the number 4, looked like the number 9, but with more training it was able to produce a better number 4. We trained the model with a GPU, a batch_size of 32 and the Adam parameters of the optimize as they were set, so a learning rate of 0.0002 and betas of (0.5, 0.999). For the code of the training part we inspired ourselves from this article [1] We can see the generated images are quite convincing ; the discriminator output has been set as



(a) Images sample from the MNIST dataset

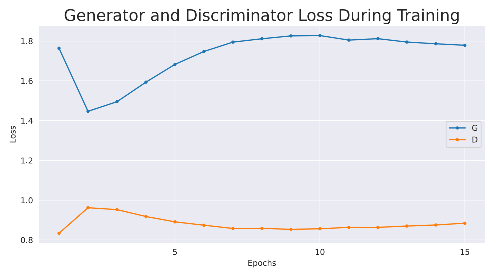


(b) Fake generated images by our generator

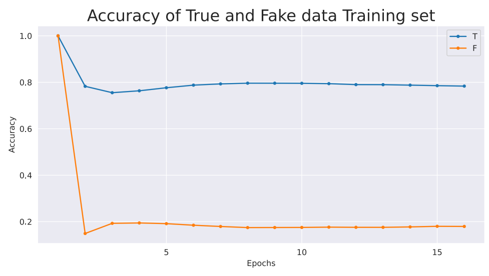
Figure 1

the prediction above each generated image ; if it's a 0, the discriminator classified it as a fake and a one as a true label. Even if the images appear to be well-made, the discriminator can tell that 12 of the 16 images in this batch are fake.

1.3 Error metrics



(a) Evolution of the loss during training of the GAN on MNIST



(b) Evolution of the accuracy during the training of the GAN on MNIST

Figure 2 – Metrics evolution for the GAN

The figure 2a above represents the evolution of the loss during the training phase. We can see at some point the adversary game ; precisely, if you look at the second point, we see the loss of the generator increase because the loss of the discriminator decreases, and this is what we are looking for with GAN models : this game between the two networks, but rapidly the two losses converge. In Figure 2b, the first point is the accuracy before training, and at that step, the discriminator cannot distinguish between fake and true data.

1. `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`

After some training, it can distinguish between both of them since it has high accuracy for true data and low accuracy for fake data. We can see a pattern in both plots as the accuracy and loss converge. We can interpret the accuracy by saying it is because our discriminator is too strong compared to the generator, and we can effectively assess that with the loss, so that's why the discriminator distinguishes well between the fake data and the true data. As said before, the training has been done on a GPU, and for 15 epochs it lasted 12 minutes and 30 seconds, which is reasonable.

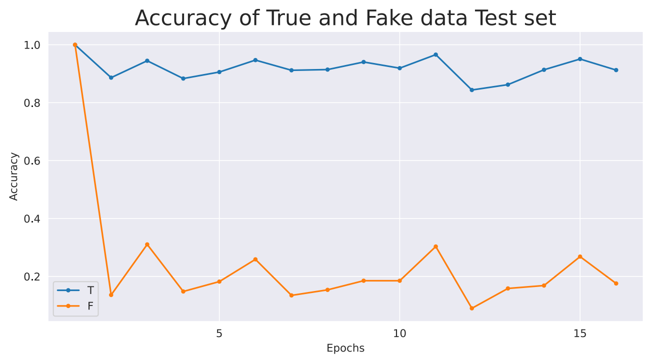


FIGURE 3 – True vs fake images

The results are the same for the accuracy on the test set for fake data, but the accuracy on the true images of the test set is a bit better.

1.4 Improve your GAN model

Three ways of improving the model :

1. Increase the number of epochs for the training ; it will take more time to train the model, but as we experienced, the generator produces better fake images as output.
2. We can add noise to the desired label in order to increase performance. The true label is originally thought of as being 1.0, but if we make the noise 0.9, the output is penalized, forcing the discriminator to learn more in order to avoid the issue of a too quick convergence.
3. One other way to make the GAN stronger would be to give him the label of the image we want to produce ; this would help guide him during the training to produce better results. The last two points have been inspired by this article. [2]

1.5 BONUS : GAN loss function

1.6 BONUS : Discriminator output and loss function

We can use the discriminator output without the last sigmoid layer because we used the BCEWithLogitsLoss instead of the classical BCELoss. As said in the PyTorch documentation [3] *This loss combines a sigmoid layer and the BCELoss in one single class." This version is more numerically stable than using a plain sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.* So by combining the sigmoid layer with the loss function, we don't need to have a sigmoid layer anymore at the end of the discriminator, and since there is no more sigmoid activation, we can see our classification problem as $> 0.0 =$ label 1, because we set it to $1/2$ if it was with sigmoid, and the origin of the sigmoid at $1/2$ is 0.

2. `https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b`
3. `https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html`

2 Train a CNN on MNIST for image classification

2.1 Your CNN model

We have used a 2D CNN which means that the kernel moves in two directions and that the input and output data are three-dimensional. 2D-Cnn is usually used for image data problems, which makes sense here. There are three types of layers in a CNN model : convolution, pooling, and fully connected. The model starts with a convolution layer followed by a pooling one then it repeats itself (with different parameters) to end with a final fully connected layer that makes six layers. Each convolution layer has five parameters : 1 : the number of channels in the input image (input), 2 : the number of channels produced by the convolution (output), 3 : the size of the convolving kernel, 4 : Stride of the convolution, 5 : the padding. The stride being the number of pixels to pass at a time when sliding the convolutional kernel, and padding is used to fill the void created by striding.

2.2 Training your CNN

We used the MNIST dataset for the entirety of the project as it was set by default in the code. Using Google Colab, we were able to use their GPU, which made the training much faster. The optimization parameters were all set by default in the code, and we therefore did not change them. The number of epochs for the CNN model was equal to 15, but we obviously could have used less as the model had an accuracy of 98.74% from the first epoch and the loss didn't diminish during the training, staying at around 1.4. The batch size was 32. In the Adam algorithm, the learning rate (lr_cnn) was set to 0.01, beta 1 to 0.9, and beta 2 to 0.999 (betas_cnn).

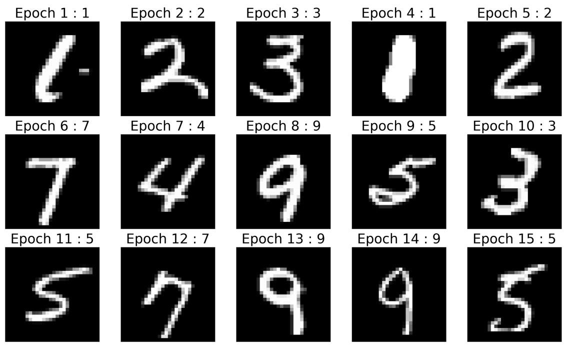


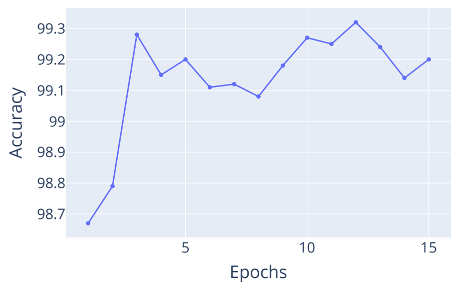
FIGURE 4 – Predictions made by classifier

The predictions are all correct. It was to be expected, given that our CNN model has an accuracy rate of 99.2% after the 15^{th} epoch.

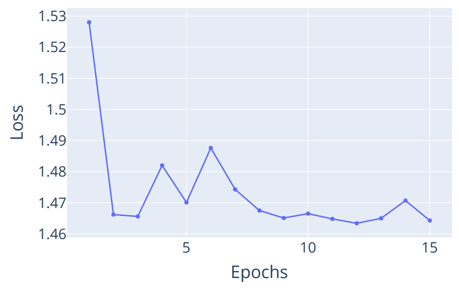
2.3 Error metrics during training

Figure 5 shows the evolution of both the accuracy and the loss over the number of epochs. A nice evolution is observed throughout the epochs for both metrics, with a big improvement around the third epoch.

Even though the accuracy starts with a high value of almost 99%, it is still improved through the 15 epochs. The final accuracy has a value of 99.2%. The same can be said for the loss, when considering a decreasing tendency. The initial loss for the first epoch has a value of 1.528. Unlike the accuracy, the loss is more constant since the eighth epoch. The final loss value is 1.464. This is a pretty high value. It could be explained by the fact that the numbers are handwritten, thus introducing a lot a variability for a given digit. This high variability yields more error when classifying input data, hence the high loss value.



(a) Accuracy



(b) Loss

FIGURE 5 – Evolution of the accuracy and loss over the number of epochs for the CNN

2.4 Convolutional layers

The number of parameters in a neural network grows rapidly with the increase in the number of layers. This can make training for a model computationally heavy. CNNs are very effective at reducing the number of parameters without sacrificing the quality of the models.

At its core, 2D convolution is a relatively simple operation : you begin with a kernel, which is simply a small matrix of weights.This kernel "slides" over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.

A dense layer is linear operation in which every input is connected to every output by a weight (so there are n_inputs * m_outputs weights - which can be a lot !). On the other hand, the convolution layer is a linear operation using a subset of the weights of a dense layer. Nearby inputs are connected to nearby outputs. The weights for the convolutions at each location are shared. Due to the weight sharing, and the use of a subset of the weights of a dense layer, there's far less weights than in a dense layer.

Convolutional layers can also be used in GANs instead of simple two-layer feed-forward networks, both the Generator and Discriminator are implemented as convolutional neural networks (CNNs, or ConvNets). The resulting GAN architecture is known as Deep Convolutional GAN, or DCGAN for short.

2.5 Improve your CNN model

Three ways of improving the model :

1. One way to improve our outputs could have been to increase the size of our dataset. The larger it is, the more our model could train itself on data and, thus, the more accurate the outputs would be. The downside of this method is that it increases the training duration of the model proportionally to the increasing factor of the dataset.
2. Another way to improve the accuracy could have been to increase the number of epochs. Yet it wouldn't have been very useful in our case since our accuracy is already very high.
3. At last, we could have shuffled our dataset (if it hasn't already been done with the MNIST dataset). During training, the training data is presented in batches to the network. This often happens in a fixed order over all iterations, which may lead to certain biases in the network. We wouldn't want to present the classes one at a time, for example, all the ones first, then all the twos, and so on. This could lead to the network unlearning much of the first class by the end of each epoch. Also, reshuffle the training data between epochs could again avoid potential minor biases because of training data order.

2.6 BONUS : CNN loss function

Also called logarithmic loss, log loss or logistic loss. Each predicted class probability is compared to the actual class desired output 0 or 1 and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. The penalty is logarithmic in nature yielding a large score for large differences close to 1 and small score for small differences tending to 0.

Cross-entropy loss is used when adjusting model weights during training. The aim is to minimize the loss, the smaller the loss the better the model. A perfect model has a cross-entropy loss of 0.

Cross-entropy is defined as :

$$L_{CE} = -\sum_{i=1}^{n} t_i log(p_i)$$

for n classes, where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

3 Explore the latent space

3.1 Combine your generator and your classifier

In the first place, for this section, we needed to generate a lot of latent vectors from a uniform distribution, so with a for loop, we generated 1000 latent vectors that we stored in a list. In the second part of this section, we used the stored latent vector to generate fake images. We kept the images only if the predicted label of the generated image was above a threshold of certainty of 0.9. We stored the latent vectors that produced high-quality images in lists, one list for each label, and so we now have our labeled latent vectors. Here is the result of some top-quality images :

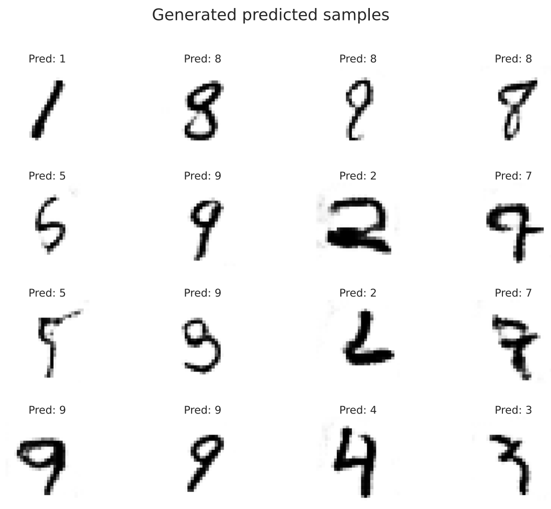
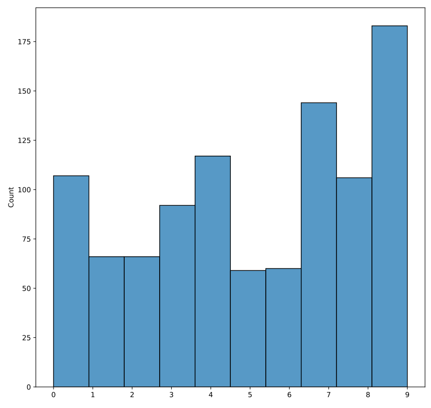
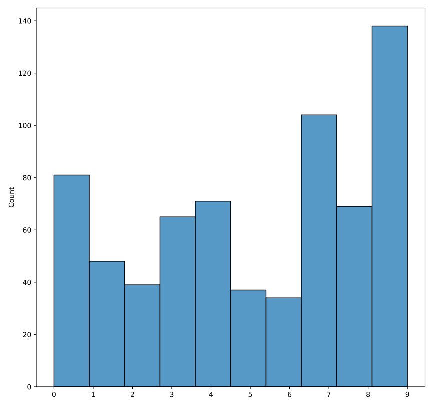


FIGURE 6 – High quality generated images

3.2 Distribution of classes



(a) Distribution with certainty of 0.1



(b) Distribution with certainty of 0.9

FIGURE 7 – Distribution of classes

Obviously, all the labels in the MNIST dataset are represented on the distribution, yet not equally. Nine's seems to be classified with a higher certainty. It seems that the threshold doesn't discriminate against a particular class ; it only discriminates more if the threshold is high, as can be seen on figure 7.

3.3 Mean of your latent vectors

Taking the mean of the latent vectors yields far better results when compared to figure 6, as shown in figure 8. However, the results are not perfect. Indeed, there are some imperfections, as it can be observed for the digit 2.

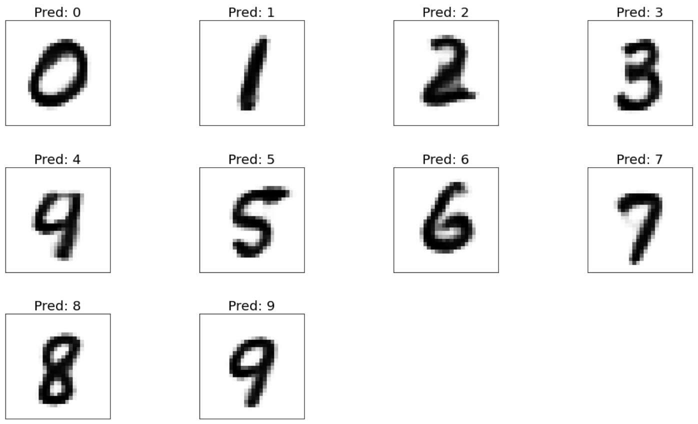


FIGURE 8 – Respective digit image when taking the mean for each latent vector

3.4 Norms and distances

3.4.1 Norms

Table 1 shows that the mean latent vector are closer to the origin than the labeled latent vector. This explains why their norm is lower compared to labeled latent vectors.

TABLE 1 – Norm of label latent vector compare to the norm of the mean latent vectors

	norm random vector	norm mean latent vector
label 0	5.7640	1.3098
label 1	5.3768	1.1777
label 2	5.7668	1.1411
label 3	5.7511	1.2163
label 4	5.6696	1.0644
label 5	5.5562	1.4356
label 6	5.7978	1.4627
label 7	5.5233	0.9839
label 8	5.7553	1.1145
label 9	5.4892	0.7307

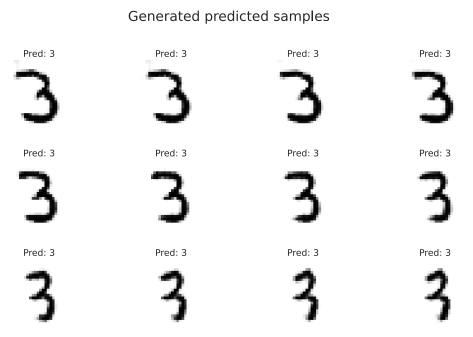
3.4.2 Distances

When we compare the distance between the example of a latent vector of label 0 and the mean latent vector of all labels, the one with the smallest distance is the one with the mean latent vector of label 0. And if we repeat the operation on multiple labeled latent vectors, the smallest distance between them will be the mean latent vector of their corresponding label. That's why we were asked to compare our CNN with the minimum distance criterion (MDM). We implemented it with a function that takes a latent vector and the mean latent vector as input, computes the distance between the latent vector and each of the mean vectors, and then returns the index of the minimum distance, which is the label of the latent vector we passed as input.

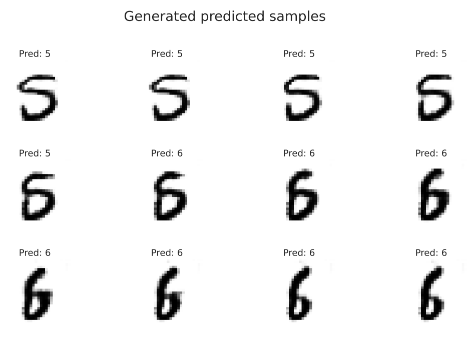
We were asked to compare this new classifier with the classification of our CNN. If we keep the certainty threshold high, like 0.9, the output label of the CNN and the MDM are the same, but if we change the certainty threshold to like 0.5, the MDM criterion gives a better result than our CNN.

3.5 Interpolation for exploration

Our first step in latent vector interpolation is to use two latent vectors with the same label. First, we'll look at the results for two images with the label 3. This is illustrated on figure 9a. With the images above, we can see that we start from the first image and slowly arrive at the last image.



(a) Interpolation between the same class (from 3 to 3)



(b) Interpolation between two different classes (from 5 to 6)

Figure 9 – Interpolations

Figure 9b is a second example of interpolation, but from label 5 to label 6.

We can clearly understand how the images are and how the generator construct an image to another. We drew inspiration from this article[4] for the code of the interpolation and its understanding.

3.6 Noise for exploration

If we look at the first images in figure 11, we can see that the transformation depends a lot on the noise we chose at the start. For example, if we take the images from figure 11a we can see that the little noise added to the mean latent vectors changes nothing. But if we look at a really noisy sample added to the mean latent vectors like figure 11e, we can see that we arrive in the latent space of the image labeled 4 or 1, except for the label 1, 4, and 7, which all stay in their latent spaces.

3.7 BONUS : Mode collapse

As this article says[5] *Each iteration of the generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the trap. As a result, the generators rotate through a*

4. `https://machinelearningmastery.com/how-to-interpolate-and-perform-vector-arithmetic-with-faces-using-a-generative-adversa`
5. `https://developers.google.com/machine-learning/gan/problems`

small set of output types. This form of GAN failure is called mode collapse. What we can derive from this is that the generator is trying to give reasonable output to the discriminator, but then the discriminator cannot learn more and they get stuck. In the article, they advise two types of attempts to remedy this problem by computing other losses. But we think adding a bit of noise to the label can also be a way to avoid this problem.

4 Create what you want

As mentioned before, we only used the MNIST dataset for this homework. We decided to use a binary message to create a message, which will be translated in the final code cell.

The graphical output of the binary message is represented in the following figure 10. We invite the reader to have a look at the last cell of the notebook to be able to translate our message ! In that way, our message is kind of encrypted.

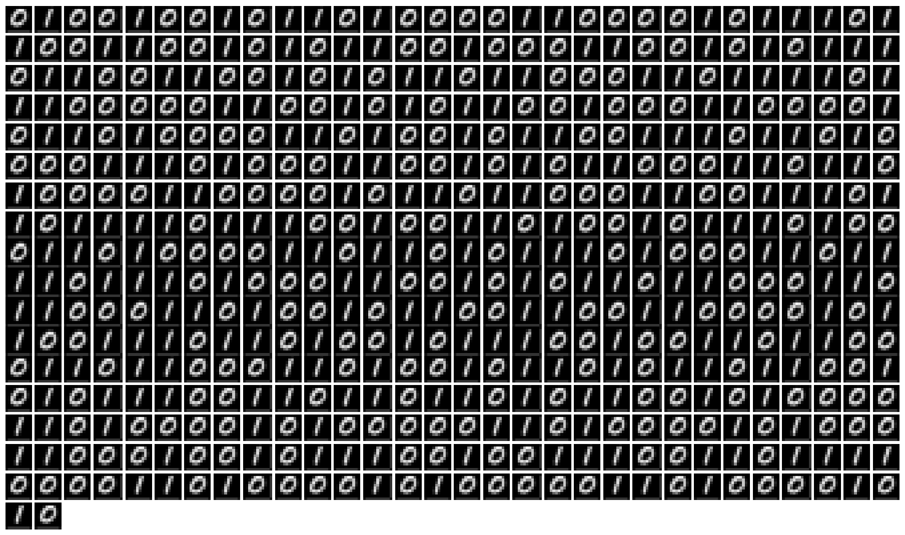
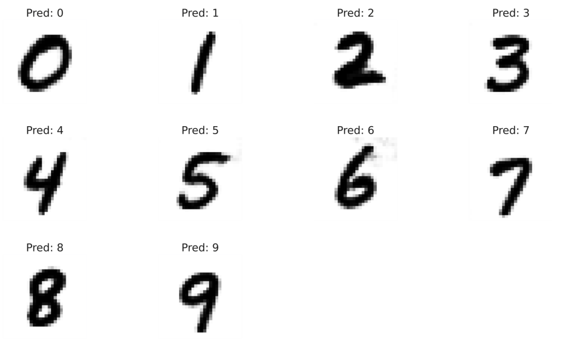
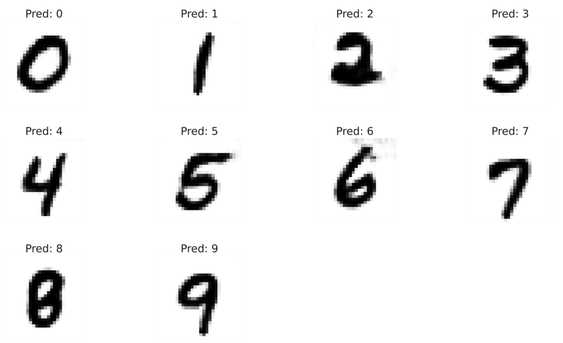


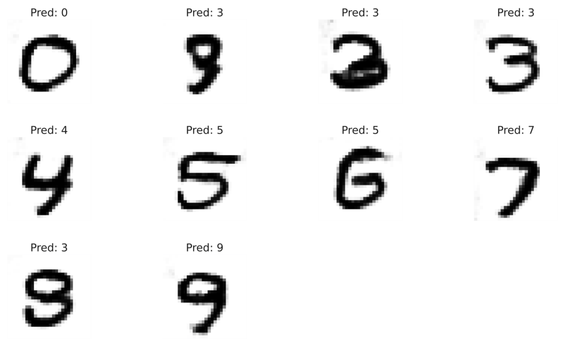
FIGURE 10 – Binary message using GAN and CNN



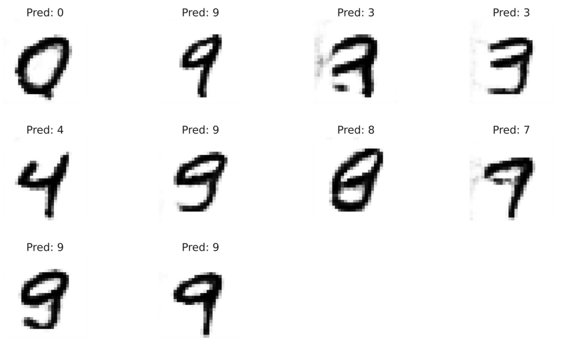
(a) Noisy uniform distribution sampled from[-0.1, 0.1] + mean latent vector



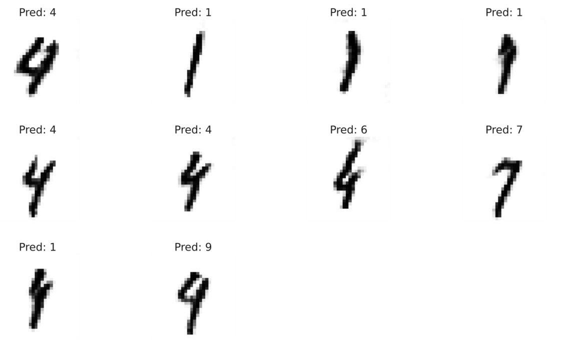
(b) Noisy uniform distribution sampled from[-0.3, 0.3] + mean latent vector



(c) Noisy uniform distribution sampled from[-0.5, 0.5] + mean latent vector



(d) Noisy uniform distribution sampled from[-0.7, 0.7] + mean latent vector



(e) Noisy uniform distribution sampled from[-0.9, 0.9] + mean latent vector

FIGURE 11 – Noise for exploration