

Code Files:

For this exercise we broke down the code into multiple files:

- `ex2.py` - here is the code for the GAN training the AE (including the novel case)
- `ex2_2.py` - here we implemented the inverse GAN, GAN interpolation, AE novel tester, AE interpolation
- `models.py` - here we have the different models we tried for the MNIST and celebs dataset and for the auto encoder. In addition, we added here our implementation for the Non-Saturated Cross Entropy loss function.

Last but not least, in this file we implemented a class to represent an enum in order to choose dynamically the type of Models we would like to train. (See `argos -d` and `-g` in the "Running the code" section)

- `common.py` - here we have code that is shared for `ex2.py` and `ex2_2.py`
- `utils.py` - Other utility functions

Running the code:

- `ex2.py`: to run the GAN training, run `'python ex2.py'`. There are many possible arguments that can be given to the script. We urge you to run `'python ex2.py --help'` to see the possible arguments. In particular, the most important flags are:
 - `--dataset`: defines if to train on the MNIST or Celeb dataset
 - `-d`: this enables one to choose the model for the discriminator. As can be seen in `models.py`, there are around 18 different models for the Celebs dataset and 5 for the MNIST data set. The syntax is by `-d M<n>` where `n` is the number of the model, e.g. `-d M3`.
 - `-g`: same as `-d` but for the generator.
 - `-l` (lowercase L): defines the loss function for the training. Legal arguments are:
 - `"mse"`
 - `"cross_entropy"`
 - `"non_saturated"`
 - `-k`: the number of unrolling iterations to use
 - `-gs`: the number of iterations that the generator is trained over a single batch
 - `-ds`: the number of iterations that the discriminator is trained over a single batch
 - `-e`: number of epochs
 - See `--help` for the rest.

A typical run would be:

- `python ./ex2.py -g "M1" -d "M3" -b 24 -l "non_saturated" -k 4 -e 50 -z 50 --dataset "MNIST"`
 - `python ./ex2.py -g "M11" -d "M12" -b 128 -l "non_saturated" -k 2 -e 50 -z 100 -gs 5 -ds 2 --dataset "celeb"`
- `ex2_2.py`: Here the arguments are:
 - `-i`: number of iterations for converging the search of the latent vector `z`
 - `-s`: number of different random starting vectors `z` when searching the best overall
 - We suggest not modifying the rest of the arguments in this file.

In this code, running the default code with no arguments, that is, “python ./ex2_2.py”
In order to run the Inverse GAN, we are adding a “.pkl” file to our submission corresponding to the trained generator model. For the code to run, please leave the file **in the same directory as ex2_2.py** (and not in the data directory for instance). Thanks.

Generative Adversarial Networks - Q1

In this section, we implemented the models for both the MNIST data set and the Celebs dataset. In the training of the generator and discriminator we added the following functionality in order to converge to the best possible model:

- The option to have multiple training optimization steps for the **generator** in a single batch
- The option to have multiple training optimization steps for the **discriminator** in a single batch
- The option to have **Unrolling steps** as mentioned in class, in order to avoid Mode Collapse. The way we implemented this was by first “fake training” the discriminator for a given ‘k’ amount of steps, then training the generator and finishing by rolling back to the state of the discriminator before the “fake training”. This approach was taken from the code [here](#), as it was unclear by reading the original paper on the unrolling GAN how to implement the unrolling concept.

MNIST dataset: We tried implementing the suggested structures for the generator and discriminator but with no real success. The best results were given when we used the MSE as a loss function and we were stuck with mode collapse on the digit 1. In addition, we noticed that for both the generator and discriminator, the sigmoid activation function did not manage to yield any results.

After that, we found a design for GAN which implements both the generator and discriminator with only FC layers. For the most part this structure worked, but we noticed we were having a lot of salt and pepper noise in the generated images where it should be completely black. To fix that problem, we figured that having convolutional layers would help since conv layers give weight to spatial details, which led us to the idea of having a FC setup for the discriminator (with some dropouts), while for the generator a model that is constructed of 3 FC layers followed by 3 ConvTranspose.

In addition, we managed to implement the unrolling steps concept for which with $k \geq 2$ gave very nice results indeed.

The described above models are model M1 for the generator and M3 for the discriminator which can be found in the Models.py file

Lastly, we would like to add that the total number of parameters learned in each models are:

G: M1. Trainable params: 675025

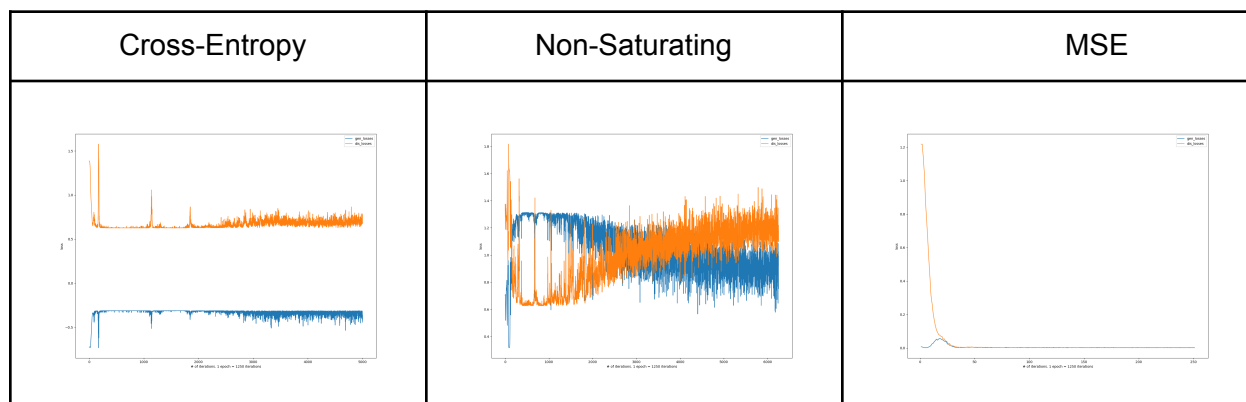
D: M3. Trainable params: 1460225

Which is definitely unexpected since we would expect the generator to be more expressive than the discriminator. As an opposite argument, given that the best result turned out when using unrolling steps, we might say that the generator is using that expressiveness of the discriminator at his advantage without the need for more data to converge.

Loss Functions - Regarding the difference between different loss functions:

- **Cross-Entropy**: As a rule of thumb, this loss function achieved worse results than the Non-Saturating loss. This was a result of either vanishing gradients or attenuated gradients preventing the nets from training enough. This mainly happened when we ran the net with multiple steps for the decoder and a single one for the generator.
- **Non-Saturating**: This mostly achieved the best results. It's still far from bulletproof and when running the net with multiple steps for the discriminator as opposed to a single step for the generator it still got to a vanishing gradient state, but in general we hardly got vanishing gradients with this loss whenever the normal Cross-Entropy loss didn't, making this a preferred choice. Now, the reason this loss function manages better to avoid saturation is because instead of trying to avoid making fake images it focuses on rewarding making real like images and therefore even when the discriminator is more confident of itself it is less likely for the gradient of G to disappear.
- **MSE**: Except in rare attempts, the MSE did not produce anything. Probably because our hyper-parameters were fine tuned to work better with the other loss function but in general it converged to a vanishing gradient state incredibly fast, preventing the nets to train. We should mention though that in early attempts we did achieve to train the nets with the MSE loss but it only generated the digit 1 (every time!), and when we tried to get rid of the mode collapse it stopped working.

Comparison between the different losses:

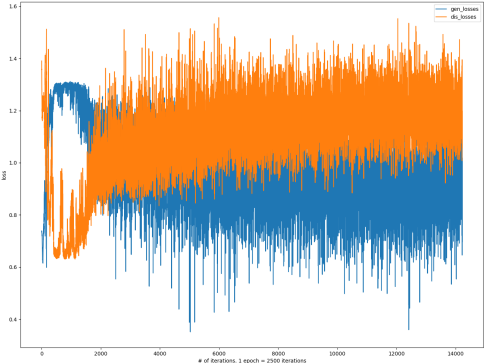
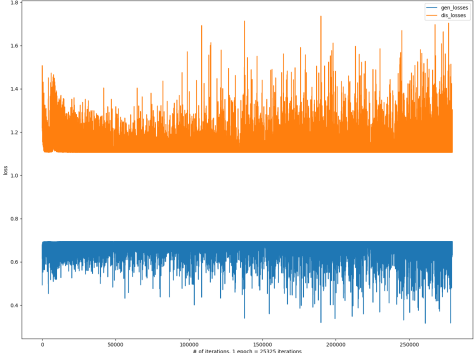


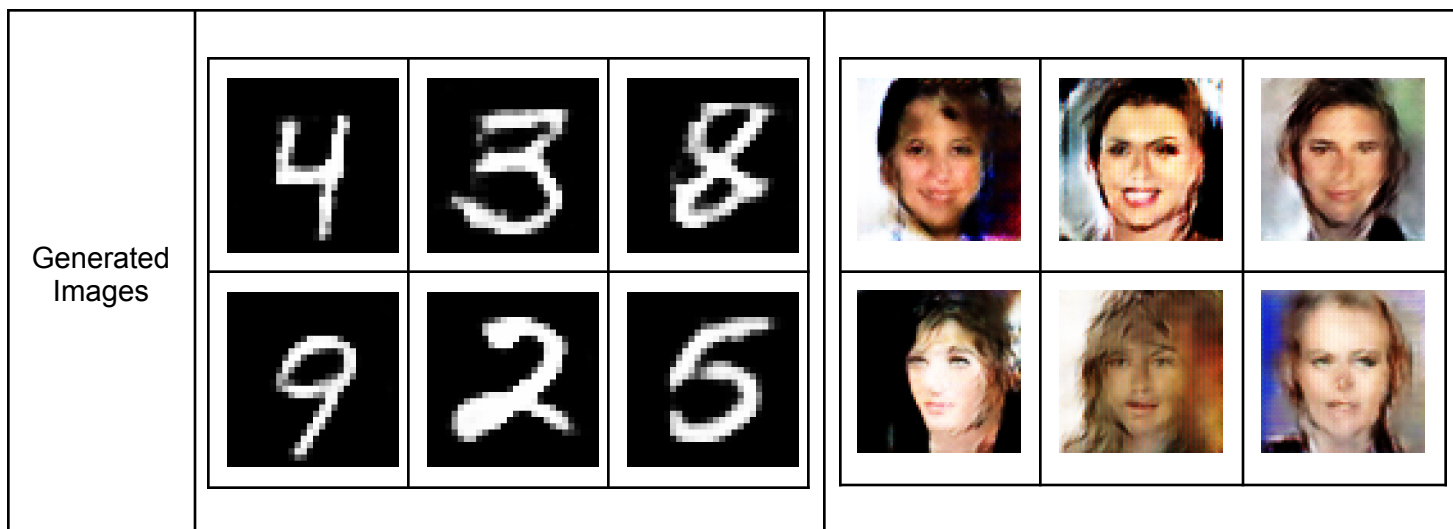
Celeb dataset: With this dataset we struggled to converge since the training took so much time to run. Due to that, it was very difficult to fine tune the models and all models are approximately the same. The real differences were with the number of channels in the conv layers. Eventually

we found the models that were promising, and so instead of optimizing the design we chose to find the best running parameters.

A significant struggle we had was dealing with vanishing gradients. As opposed to the MNIST dataset in which we found a running configuration that removed vanishing gradients, with the celeb dataset we needed to tweek many more parameters to achieve a non saturated train. For example, we started mixing the number of training steps for the discriminator and generator for a single batch, fine tuning the unrolling steps parameter k to avoid mode collapse, having a different learning rate for the discriminator and generator optimizers (eventually what worked was to have 0.0001 for the generator and 0.0004 for the discriminator), having a very small batch of 8, and last but not least instead of computing the loss of the discriminator with a matrix of 1s for a real image, it was multiplied by a factor of 0.9 to avoid making the discriminator too sure of itself. All these changes made it possible for us to train a generator with this dataset.

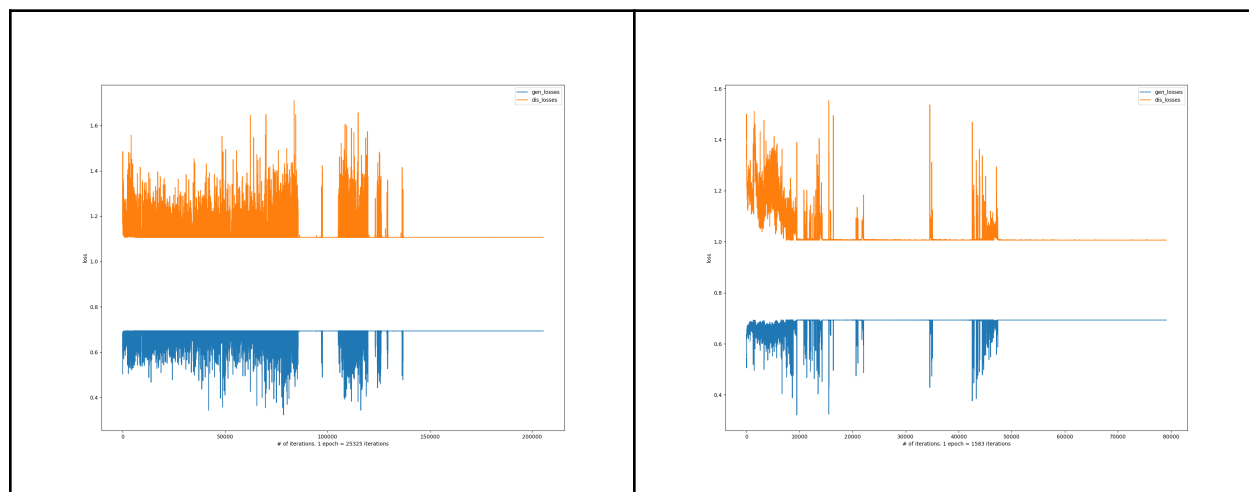
Following is a table of the MNIST dataset and celebs dataset with their best results and loss plots. Do note that as opposed MNIST, in the case of the celebs dataset there isn't as much of a battle between the nets. This might be due to the 0.9 factor mentioned before.

	MNIST: in the plot we see the clear battle between the generator and the discriminator which in turn successfully trains a working generator.	Celebs: this plot was also achieved using the non saturating version of the loss. We managed to
Loss Plot	 A line plot showing the training loss for the MNIST dataset. The x-axis is labeled '# of iterations, 1 epoch = 2500 iterations' and ranges from 0 to 14000. The y-axis is labeled 'loss' and ranges from 0.4 to 1.6. There are two data series: 'gen_loss' (blue line) and 'dis_loss' (orange line). The generator loss starts around 1.3, drops to 0.6 at 1000 iterations, and then fluctuates between 0.4 and 0.8. The discriminator loss starts around 1.3, drops to 0.6 at 1000 iterations, and then fluctuates between 0.8 and 1.5.	 A line plot showing the training loss for the Celebs dataset. The x-axis is labeled '# of iterations, 1 epoch = 25125 iterations' and ranges from 0 to 25000. The y-axis is labeled 'loss' and ranges from 0.4 to 1.6. There are two data series: 'gen_loss' (blue line) and 'dis_loss' (orange line). The generator loss is very low, fluctuating between 0.4 and 0.5. The discriminator loss is high, fluctuating between 1.2 and 1.5.



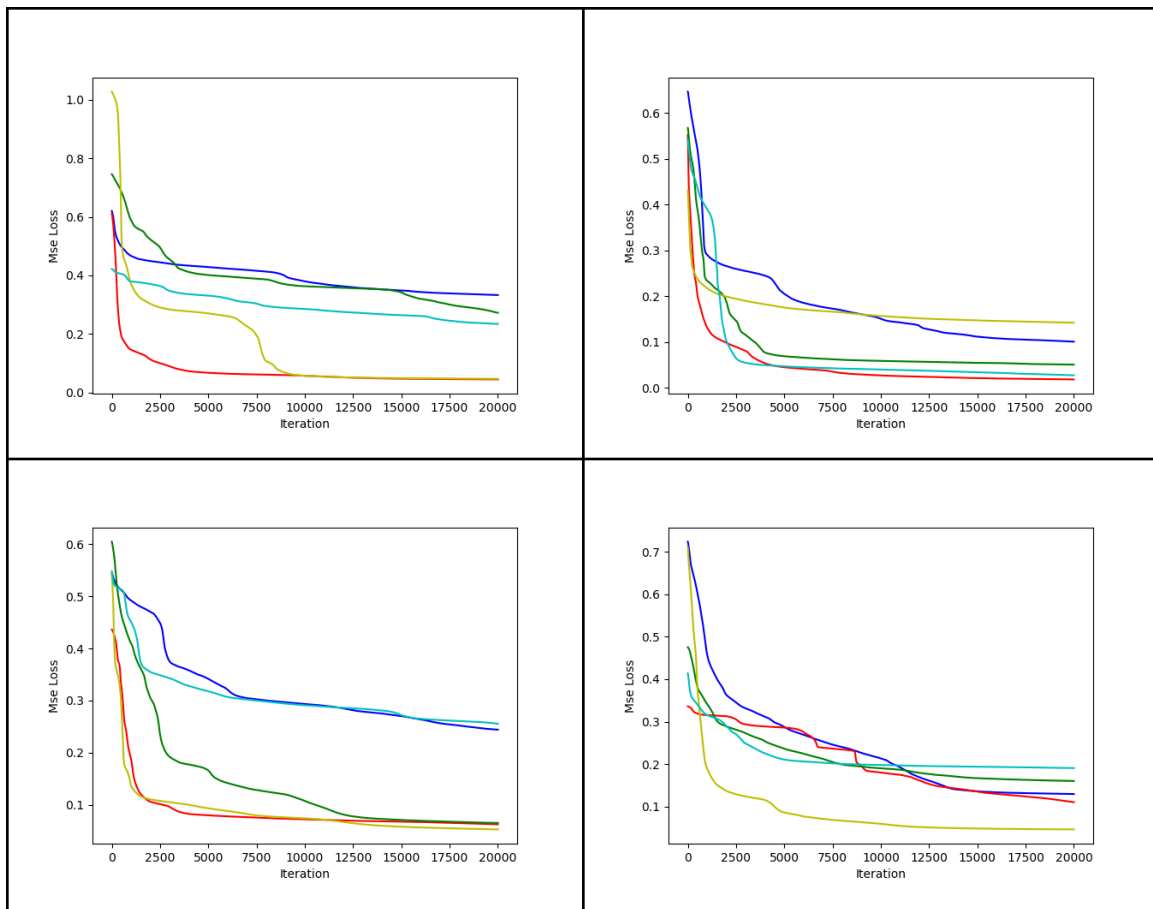
Do note that the upscaling of the images in the case of the celebs dataset make them look worse than they actually are. (they were upscaled by 5x).

In the following table we show two plots of the loss functions with the celeb dataset before achieving not to saturate. The point in which a graph stops changing is when we enter a vanishing gradient state and the nets stop improving. As can be seen, at random the vanishing gradients appear and disappear. These plots are before we managed to train the



For this section we first trained the Generator from Q1 and used it to find the model inversion given an input image.

The search goes as follows: given an image, we randomly pick a vector z from a normal distribution with size of the latent space and set it as input to a SGD optimizer. At this point, for a fixed number of iterations we compute the MSE loss between a generated image from the Generator model to the input image, and then step the optimizer which optimizes the z vector. Since the SGD algorithm might converge to a local minimum and hence the initialization of z might affect the final results, we improved our results by performing the process on 5 different random (from normal distribution) z vectors, and choose the z whose final loss was the smallest. This process improved our results tremendously, as can be seen in the following 4 loss figures:



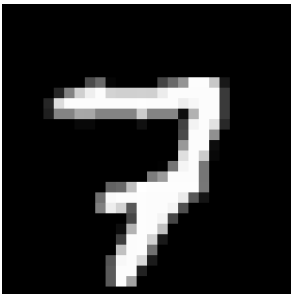
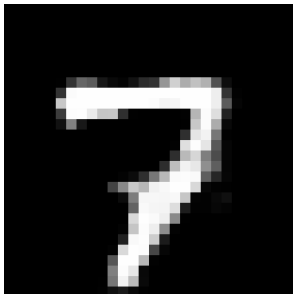
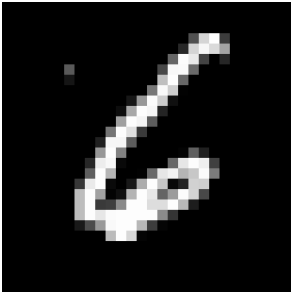
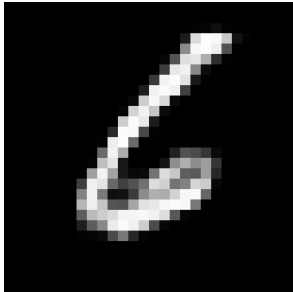


In all the figures, each color shows the loss plot for a different initial z while going through the optimization process. At the end, the z with the lowest loss is picked. As can be seen, if we would perform the process on a single initial z , with high probability we would not converge fast enough or even at all.

When comparing the input image with the image given by the found vector z , the feature that was preserved in all our tests was the size of the digit. In most cases the output image contained the same digit as the input image, occasionally with a slightly different shape. In

addition, in some cases small details that weren't strictly present in all images of the category weren't preserved. For instance, in the following table, the figure 7 generated did not fully contain the middle stipe which is not present in all the dataset 7's. Another example can be seen with the figure 6, where in the original input image there is a white dot to its upper left which is removed in the generated image.

Examples:

Input image of the inverse GAN:	Generated images:
	
	
	

Generative Adversarial Networks - Q3




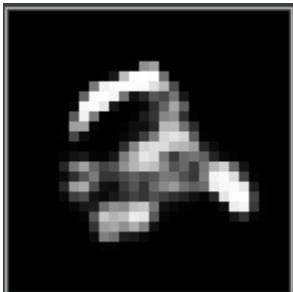


We implemented 3 different masks:

- Pepper - randomly blackens 30% percents of the pixels
- Crosswalk - blackens every second row.
- Square - randomly picks a 8x8 patch in the image and blackens it.

Our implementation relies on the inverse GAN model - we apply the chosen mask on the target image and then give it as input to the inverse GAN and display the results.

The best restoration was achieved for images masked by the pepper mask, and the worst for images masked by the crosswalk mask. In the case of a square mask, it depends on the precise location of the square (and how much of the image it covers), but in most cases it did better than crosswalk but worse than pepper.

In the next table can be seen the results for the different masks.

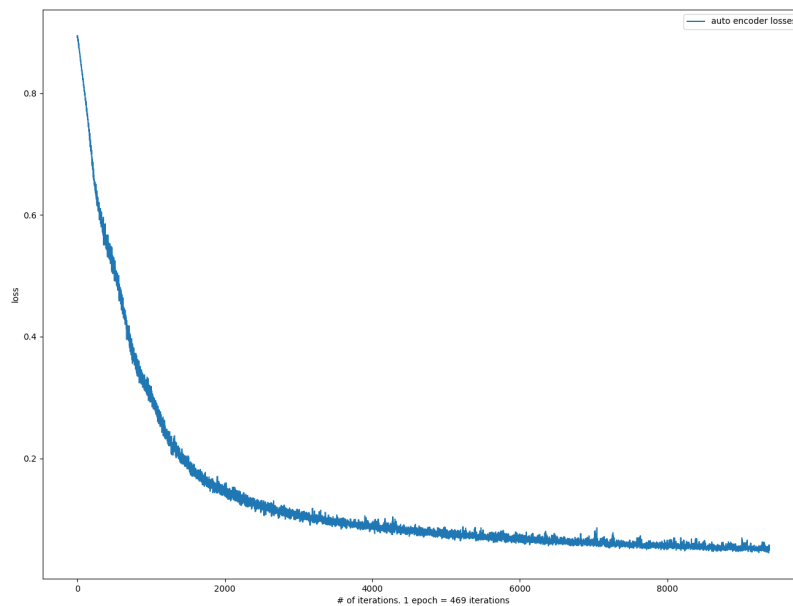
	Masked target image	Restored image
<u>Pepper Mask</u>		
<u>Crosswalk Mask</u>		
<u>Square Mask</u>		

Non-Adversarial Networks - Q1



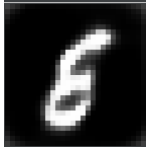
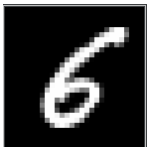
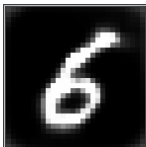
We've trained the autoencoder on MNIST dataset with mse_loss function (L2) , latent space of size 100 (dimension (100,1)). The encoder contains 3 convolution layers and 1 fully connected layer (and we used relu activation and batch normalization). The Decoder contains 3 transposedConvolution layers and one fully connected, and relu as activation. 20 epochs were enough to get very good results, as can be seen below:





















And the loss plot:



In order to be able to return the z created by the encoder and then use it to perform the intermediate task, our autoencoder returns both the output (generated image) and the latent space vector z . We've tried the intermediate experiment multiple times on random images with different a values using the given formula $G(a \cdot z_1 + (1-a) \cdot z_2)$. As expected, as a is closer to 1 and it holds that $a > 1-a$, the output image of $G(a \cdot z_1 + (1-a) \cdot z_2)$ is more similar to $G(z_1)$ (and vice versa). In addition, for similar digits in similar size and shapes, for example 6 and 8 or 1 and 7, the intermediate images were smoother and clearer, and for images with very different shapes the results were less smooth. For example:

	Original	autoEncoder(image)	Intermediate image for $\alpha = 0.5$
Image 1			
Image 2			

In the following table are a couple of examples of the interpolation output for $\alpha = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1]$:





































	$\alpha = 0$	$\alpha = 0.125$	$\alpha = 0.25$	$\alpha = 0.375$	$\alpha = 0.5$	$\alpha = 0.625$	$\alpha = 0.75$	$\alpha = 0.875$	$\alpha = 1$
Interpolation 6 to 8									
If we look at digits with very different shapes (or sizes), the transition is a bit harsher.									

In the second case, since the 8 and 7 digits are so different, the image generated when $\alpha = 0.5$ is more likely to be labeled by an entirely different category (in that case 3) than by one of the edges categories.

When repeating the interpolation experiment with the trained GAN, most tries were successful though many times the starting image did not resemble so much the original image before the GAN inversion which is why the GAN interpolation did less good compared to the AE interpolation in general.

Additionally, a surprising result was that a significant number of tries yielded an unexpected result where during the interpolation a third type of number appeared. This is closely compared to the AE case, but the intermediate digit looks much better than in our case (see below).

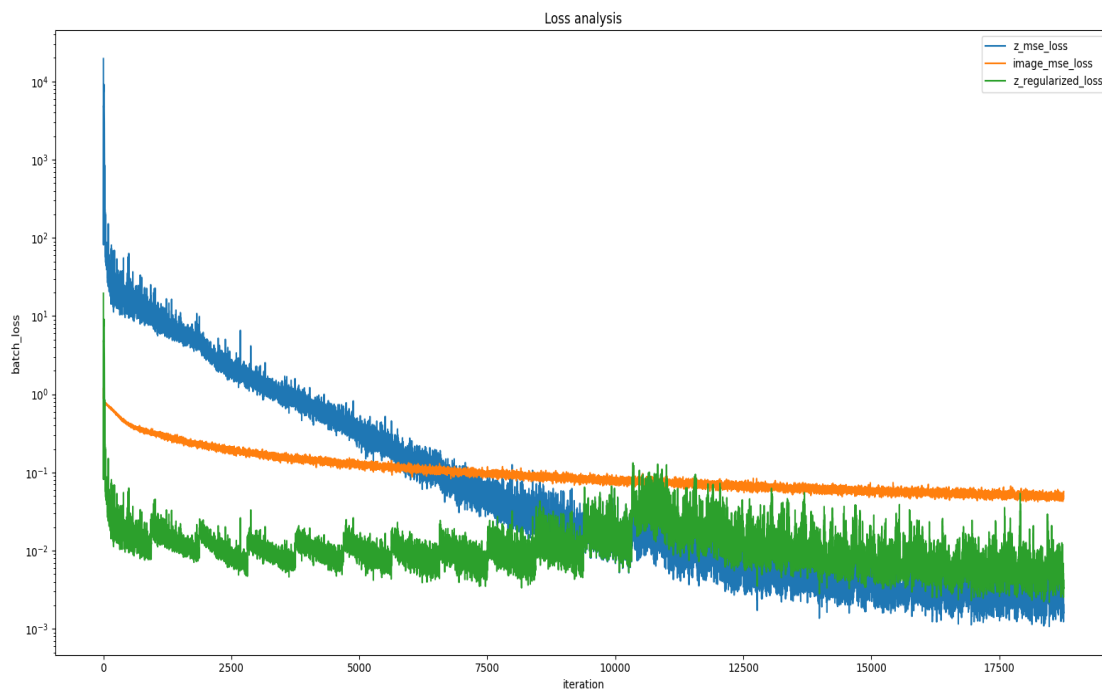
In the following table can be seen all the results:

	$\alpha = 0$	$\alpha = 0.125$	$\alpha = 0.25$	$\alpha = 0.375$	$\alpha = 0.5$	$\alpha = 0.625$	$\alpha = 0.75$	$\alpha = 0.875$	$\alpha = 1$
Interpolation from an unclear 4 to 2									
A good interpolation from 8 to 5									
A very unexpected interpolation from 1 to 7 where in the middle appeared a perfect 4!!									
Again a very unexpected 5 in the interpolation from 4 to 3!									

Non-Adversarial Networks - Q2

In order to force the latent space into the given distribution, during the training we calculated the z 's mean, variance and kurtosis, and then computed the z_mse_error which is calculated to $mse([mean_z, variance_z, kurtosis_z], [0,1,3])$. Then we added the z_mse_error multiplied by a factor (on which we will elaborate later on) to the regular mse error of the autoencoder output and the target image ($image_mse_error$). Since we didn't apply any activation on the output of the encoder (z), its values aren't bounded at all, so on the first few iterations the z_mse_error is very large and might cause gradient explosion. In order to regulate it, we multiply the z_mse_loss by the factor mentioned above, before adding the two losses. On the first epoch the factor is initialized to 0.001, and is multiplied by 2 every epoch (exponential growth) until it reaches epoch #12 and then it stops growing (the maximal factor allowed is 2.048).

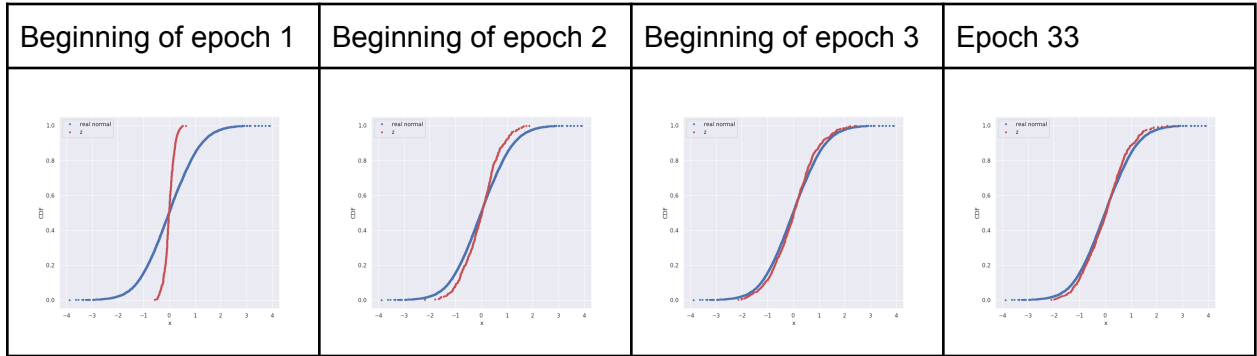
This way, on the first few epochs the z_mse_error is still relatively high (even though the factor is very low), but after those few epochs the z_mse_error drops, and the factor is still relatively small, meaning that at this phase of the training the image mse loss is larger in relation to the z_mse_error and as a result most of the efforts are utilized to minimize the $image_mse_error$. After few epochs, the factor is large enough to give the z_mse_error enough weight to count in the minimization process. On the figure below, you can see the behavior of the loss described above:



The y axis is in log scale for visualization reasons. Note that the x axis is the total number of iterations (not epochs). The blue plot shows the z_mse_loss , which as can be seen is very large on the beginning and drops very fast on the first few iterations. The orange plot shows the $image_mse_loss$, which decreases in relatively constant frequency. The green plot presents the

z_mse_loss multiplied by the factor, and each small peak is a beginning of a new epoch (which increases the factor). Note that the blue plot describes the actual z_mse_loss while the green plot is the z_mse_loss seen by the SGD optimizer.

Additionally, we saved the CDF function of the latent space z generated by the encoder during the training of the AE to see the change over time. Additionally we added the CDF of a truly normal distribution with 10,000 samples for comparison. The results are shown in the following table: (since the legend is small, the blue CDF is of the true normal distribution and the red of the latent space)



In comparison to the GAN results, the autoencoder (with forcing latent space to the given distribution) results are a lot sharper (less blur) and as a result more similar to the original data they are trying to generate.

Theoretical questions:

- Each image in ImageNet belongs to 1 out of 1000 classes. Assume the labels of the different classes are represented by numbers from 1 to 1000. Denote N a classification network trained on Imagenet, denote $I \in ImageNet$, denote l be the label of I . Since l is the true label of I , $\forall k \in [1000]$, $k \neq l$ if $N[I] = k$, the network will be "punished" with higher loss. Meaning, $\forall k \in [1000]$, $k \neq l$ the image I adds the constraint $N[I] \neq k$, which leads to 2 algebraic constraints: $N[I] > k$ and $N[I] < k$. Therefore, the total number of algebraic constraints each image adds is $999 \cdot 2 = 1998$.

- Assume f dims are $d \times d$, I dims are $n \times n$ and assume for simplicity that the dimensions of $C(I)$ are the same as I 's. Denote w_1, \dots, w_{d^2} the weight of f (meaning

$$\forall 0 \leq i, j < d, f_{i,j} = w_{i \cdot d + j}) \text{ If so, } \forall 0 \leq i, j < n, C(I)_{i,j} = \sum_{r=0}^{d-1} \sum_{c=0}^{d-1} f_{r,c} I_{i - \text{floor}(\frac{d}{2}) + r, j - \text{floor}(\frac{d}{2}) + c}$$

- The gradient of $C(I)$ with respect to I is $\left[\frac{\partial C}{\partial I_{i,j}} \right] \in \mathbb{R}^{n \times n}$ ($0 \leq i, j < n$). Using the definition of $C(I)_{i,j}$ from above, we get that from the chain rule

$$\frac{\partial(C)}{\partial(I_{i,j})} = \sum_{k,t} \frac{\partial(C)}{\partial(C(I)_{k,t})} \frac{\partial(C(I)_{k,t})}{\partial(I_{i,j})} = \sum_{i,j \text{ s.t. } 0 \leq i-k < \text{floor}(\frac{d}{2}), 0 \leq j-t < \text{floor}(\frac{d}{2})} \frac{\partial(C)}{\partial(C(I)_{k,t})} f_{i-k,j-t} \text{ which}$$

amounts to $\nabla_I C = \nabla_{C(I)} C * f$

- b. The gradient of $C(I)$ with respect to f is $[\frac{\partial(C)}{\partial(f_{i,j})}] \in \mathbb{R}^{n \times n}$ ($0 \leq i, j < d$). In order to compute $\frac{\partial(C)}{\partial(f_{i,j})}$ we need to sum up all entries that share the weight $f_{i,j}$ what

$$\text{leads to } \frac{\partial(C)}{\partial(f_{i,j})} = \sum_{k,t=0}^{\text{floor}(\frac{d}{2})} I_{i+k,j+t} \frac{\partial(C)}{\partial(C(I)_{k,t})}, \text{ which amounts to } \nabla_f C = I * \text{flip}(\nabla_{C(I)} C),$$

where $\text{flip}(x)$ denoted reversing the order of entries in x .

In order to make the procedure more efficient we can use the fact that activation functions are applied on each element separately, and therefore back propagation through activation functions can be implemented using element-wise multiplication.

In addition, Relu activation for instance only propagate positive gradients which allows us to set to zero some of the weights. In case of backpropagation gradients in convolution layers, it could be implemented using transposed convolution.

The following figure we found online (can be found [here](#)) shows how the jacobian matrix can be simplified in linear layers and activation layers:

Backpropagation through activationfunction

$$\begin{bmatrix} J_{\mathbf{x}}^E \\ J_{\mathbf{x}}^{h^{(1)}} \end{bmatrix} \begin{bmatrix} h' \\ h' \\ h' \\ h' \\ h' \\ h' \end{bmatrix} = \begin{bmatrix} J_{\mathbf{x}}^E \end{bmatrix} * \begin{bmatrix} h' h' h' h' h' h' \end{bmatrix}$$

↑
elementwise multiplication

Backpropagation through linearlayers

$$\begin{bmatrix} J_{\mathbf{x}}^E \end{bmatrix} \begin{bmatrix} \mathbf{x} & \mathbf{x} & \dots & \mathbf{x} & \mathbf{x} \end{bmatrix} J_{\mathbf{w}}^{\text{linear}(\mathbf{x}, \mathbf{w}, \mathbf{b})} = \begin{bmatrix} J_{\mathbf{x}}^E \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix}$$

needs to be flattened

3. LargeGAN

- a. Main changes:

- i. Increasing the batch size till 2048. This showed to achieve a better Inception Score and Frechet Inception Distance.

- ii. Increasing the number of channels in each layer by 50% which also achieved better Inception Score and Frechet Inception Distance results, though this change was effective only after modifying their different residual block structure.
- iii. Since they were using multiple cores for the training (128 to 512 cores to be precise), they also computed the batch normalization across all devices, rather than per-device as was done previously.
- iv. They let the random vector z affect multiple layers of the generator directly instead of only the first layer so not only the first layer is directly affected by the latent space vector. They call it skip- z .
- v. Additionally, using a different distribution for the training and sampling after the training. This is explained further in the next section about Truncation Trick.
- vi. In practice, the training of BigGAN according to the authors needed early stopping to avoid training collapse.
- vii. Finally, in the appendix the authors elaborate in more detail the architectural design of the model. In particular they say that as a basis they used the ResNet GAN architecture with some modifications. I believe that these details are off the scope of this question so we shall finish here.

- b. **“Truncation Trick”:** The truncation trick simply says that one should generate an image from an already existing GAN generator from a truncated distribution instead of the original latent space distribution which if used during training, that is, if the original latent space is the normal distribution (as suggested in the paper), then the sampling for generating a trained net will be from the same distribution but avoiding the tails given some truncation value.

This makes sense since the net was trained for vectors generated that were closer to 0 than of those at the extremes. In addition this is a neat trick since it does not require any modification to an existing GAN but just changing the post-training sampling.

- c. **Inception Score (IS):** This was originally defined by Tim Salimans, et al. in their 2016 paper “Improved Techniques for Training GANs.”. This score is computed from a general trained deep network which gives a score according to how much a generated image resembles an object in other real images, and the diversity of the images generated by the generator. Linking this to the “Truncation Trick”, by avoiding generating images from part of the latent space you automatically decrease the diversity of the generator, but in turn it might increase the quality of the images being generated.

Frechet Inception Distance (FID): As opposed to the Inception Score, the Frechet Inception Distance computes its score by the distance between the generated images and the training images of the GAN, while the Inception Score only took into consideration the distribution of the generated images.

