Generative Latent Optimization

Yasmin Heimann, 311546915

January 14, 2021

1 Preface

Generative Latent Optimization (GLO) is a method to optimize the latent space of generative models.

GLO learns to map learnable latent vectors to samples in a target dataset by minimizing a reconstruction loss.

During the traning phase, optimizing the parameters of the generator and tuning the corresponding latent vectors are alternatively performed. When converged, the model is able to generate novel samples given latent vectors sampled from the distribution.

The model build in this report is based on the GLO variation described in Yedid Hoshen & Aviv Gabbay paper (attached in the references), using a simplified version due to runtime concerns.

2 Method and Specifications

The GLO presented in the code is applying a generative linear and convolutional neural network on a concatenation of embedded class and content latent codes, to learn how to reconstruct images from the MNIST dataset (2000 images are taken for the learning process).

The latent code size is 25 each, and 50 in total after the content and class latent codes' concatenation.

The Adam optimizer was used to train the generator with $Loss = L_1 + L_2$, which is calculated on the difference between the network output and the related MNIST image.

The hyperparametrs used are batch size of 32 and epocs number of 50, as was recommended for a reasonable runtime.

Other parameters will be analysed and discussed in the results section, and will be tuned according to the empirical results.

3 Results

3.1 Regularization Effects

Latent optimization over the class embeddings must ensure that no content information is present in the class representation. In the training set, a class embedding is shared across multiple images, which prevents the embedding from including content information. To ensure that class information does not leak into the content representation, we regularize the content code to enforce minimality of information.

Using the approach presented in the article, we regularize the content code with an additive Gaussian noise of a fixed variance, and an activation decay penalty.

3.1.1 Gaussian Noise Tuning

I have added a Gaussian noise vector generated with $\mu = 0$ and different fixed variance at each run in a range of $\sigma \in [0.0001, 5]$.

The different variance values change the network's image reconstruction significantly.

- Larger noise values makes the content latent code random and unreliable, thus the learning process will rely on the class latent code - by putting all the weight of the learning process on the the class latent code. This scenario then results in images generation that looks like the average case of the class predicted. In our case, the digits will look the same for every different content code of the same class, and will look like an average digit writing, that follows the average cases in this class.

- Relatively meduim-small noise values, have created the separation effect between the class and content latent codes, and have successfully learned different digits with multiple characters, thus different content latent codes of the same class will result in differently reconstructed images.

- A noise with very low σ values, may decrease the separation and make it less effective by causing leaking, as it will be comparable to no noise addition at all. We will see that on the MNIST data, the small noise addition does not make a significant change in the results, which is a result of the simplicity of the data, and thus a range of relatively low values will work fine.

We shall find the right balance to get the maximal separation between the content and class latent codes, which will be discussed and examined next.

I have examined the different variance values' implications on the output of the model by looking at the loss convergence value, which is shown below:

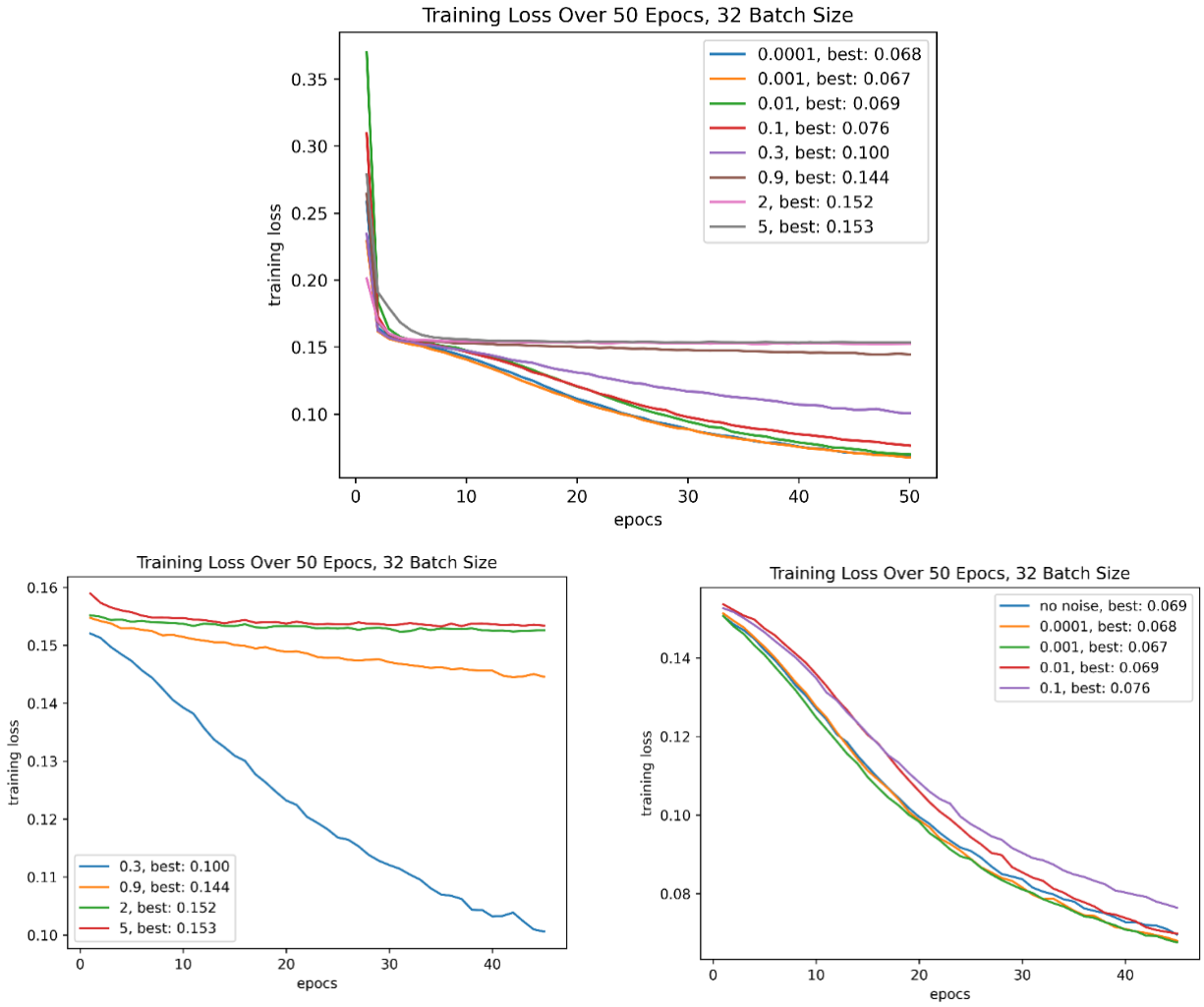

Figure 1: **The Upper Graph:** Loss Convergnce over 50 epocs, using Gaussian Noise addition, with $\mu = 0$ and $\sigma = [0.0001, 0.001, 0.1, 0.3, 0.9, 2, 5]$. **The Lower Graphs:** show a separate analysis of relatively low and high σ values.

We see that the high σ values tend to get "stuck" at a 0.15 loss after only few iterations, and thus are not able to improve from this point and on. We can assume that these runs just converge to the loss of the class only (the 'average case'), as it does not consider the content latent code which is too noisy.

More precisely, when $\sigma \geq 0.9$, we converge to a relatively high loss of ~ 0.15, and when we use a value of $\sigma \leq 0.3$ we converge to ~ 0.07 loss.

Thus, we can determine that a Gaussian noise addition with $\mu = 0$, $\sigma \geq 0.9$ is a high noise value for the model, which makes the content vector too noisy and will enforce it to rely on the class latent code, which will learn the 'average' digit image, and practically fail the test. This 'average' case is probably found at the 0.15 loss line.

Alternatively, a Gaussian noise with $\mu = 0$, $\sigma \leq 0.3$ shows better convergence and thus expected to learn a variety of digit image attributes, when at $\sigma = 0.3$ we can already see a significant improvement.

Ineresting to see is the curve with no noise at all, which achieves the same results as the small noise ranges, which will

next exhibit nice separation. This can point on the simplicity of the MNIST dataset, which results in almost no need for noise addition for the separation.

Now, we shall explore the results of the image reconstruction from the last epoc in the learning process, which will confirm our conclusions on the noise's σ addition.

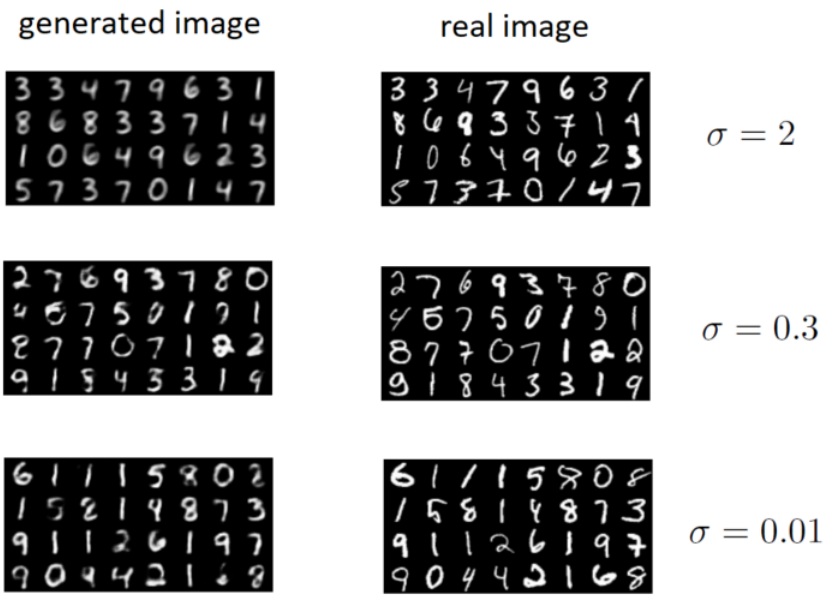

Figure 2: The images generated in the last training epoc for different σ values vs. the real image

We can see clearly that for $\sigma = 2$, all the digits look the same. For example, the 7 digit looks the same in every generated image, even though the real 7 looks different in the real images.

For $\sigma = 0.3$, we can finally see the separation effect, and we are able to generate quite nice images which differ from each other, and relate to the original one.

For $\sigma = 0.01$, we get the same separation effect, and it is almost impossible to determine by eye the difference between the σ values that are less than 0.3 . We can somewhat see some difference in the learning processes for 0.3 and 0.01, though in general in this data set it does not seem to be a significant change, as regularization is less crucial due to its characteristics.

3.1.2 Weight decay

I have regularized only the content code by adding weight decay of 0.001 to its optimizer, to ensure no leaking from the content to the class embeddings.

This approach have had almost no effect on the results.

In the general case, disentanglement will not happen without regularization as the content code can learn all the image attributes (both known and unknown) if unregularized. Though, in practice, due to the special properties of latent optimization, sometimes things work without it on very simple datasets like the MNIST we train on.

I have used a gaussian noise with $\mu = 0$, $\sigma = 0.3$, and recontsructed the images using different weight decay values. The loss curve of the learning is shown below:

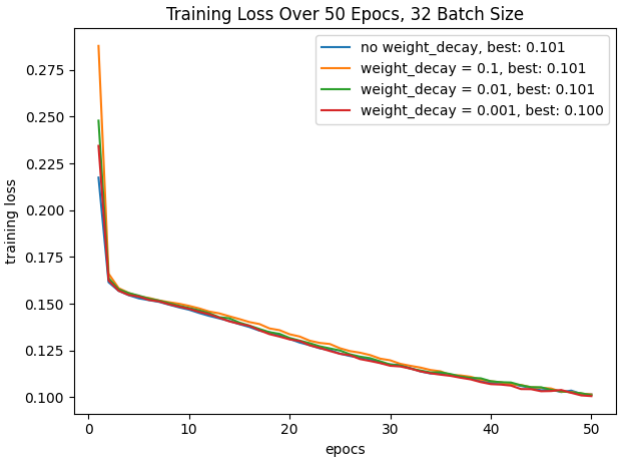

Figure 3: Weight decay values of $0.1, 0.01$ and 0.001, over 50 epocs, 32 batch size

We can see that the regularization parameter on the content vector did not do much change to the learning process, with nor without any regularization value. This result is expected due to the MNIST characteristics and simplicity, though for good practice I have used a weight decay of 0.001 throughout the report. Also, when looking at the images samples there was no siginificant change.

3.2 Loss Variants

Due to limited computational power, we were recommended to use a more simple loss than the perceptual loss. Thus, I have examined the implication of the L1 and L2 loss, and decieded, also as recommended, to use the $L1 + L2$ loss.

We can see below the implication of each loss separately and together, on the images generated by the network:

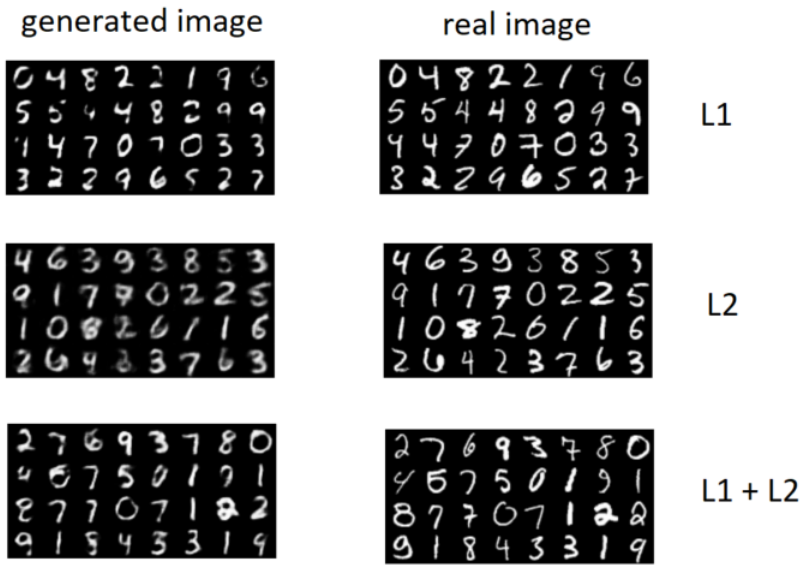

Figure 4: The reconstructed images using different Loss functions

The $L2$ loss produces more blurry digit images, where the $L1$ is much sharper but is missing more attributes and is less accurate.

The $L1 + L2$ seems to minimize the downsides of both losses, and gives a balanced output that keeps both attributes accuracy with relatively sharp images.

4 Discussion

The generator alongwith the right hyperparameter choices, had shown nice learning process and results.

Overall for this task of reconstructing images from MNIST, with the limitation of computational power, we got very nice results, using these final parameters that maximized the results and thus were used widely in the report:

1. 32 Batch Size and 50 Epocs, which is enough for fine results
2. $L1 + L2$ loss to minimize the faults of each loss alone
3. Gaussuan Noise Addition with $\mu = 0, \sigma = 0.3$, which gives a medium-low level of noise addition that enforces disentanglement.
4. Adam optimizer with a weight decay of 0.001 for regularization purposes and good practice even when the results were not much affected by it.

A sample from the final model results is as follows:

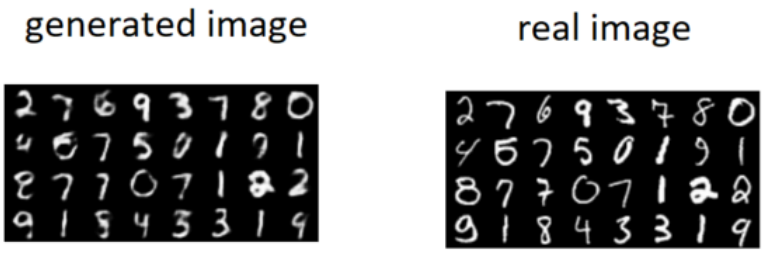

Figure 5: Recostructed images using the best final parameters

5 Further Work

The report examines a simple GLO variation, compatible with standart PC computing power, thus further work can include:

1. Applying the GLO method on a more varied and complicated dataset
2. Applying a better class and content latent code combination for the generator's input, rather than a simple concatenation

6 References

- "Demystifying inter-class disentanglement" by Yedid Hoshen and Aviv Gabbay. https://arxiv.org/abs/1906.11796
- GLO algorithm using TensorFlow. https://github.com/clvrai/Generative-Latent-Optimization-Tensorflow
- GLO algorithm using PyTorch. https://github.com/tneumann/minimal_glo/blob/master/glo.py