Generating Pokemon using GANs, machine learning,deep learning, Z-ENG, English or Hungarian

Elo Zsombor
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
elozsombor@gmail.com

Abstract. The goals of this training project laboratory were the following: learning about neural networks, learning about GANs, experiment with WGAN, experiment with augmentation. Using these things i came up with different ideas for improving the learning algorithm, and i would like to present those in the last section. I hope that this work although comes from a total beginner, will contribute somehow to understanding, how can be improved a GAN's learning algorithm.

1 Introduction

Generative Adversial Networks (from now on: GAN) are an unsupervised learning technique for generating new data and discriminating between generated and original data. The former usage of this technology is the one, we try to applicate most of the time, this is our case now as well. A GAN has (usually) two main parts: a Generator and a Discriminator. These two parts are playing a minimax game between them. The Generator tries to fool the Discriminator, the Discriminator tries to discriminate between the generated data, and the original data. The two parts are trained simultaneously, and if our metrics say, that the training ended, we usually discard one of them (usually the Discriminator). I implemented in Google Colaboratory 2 versions of GANs, to try to understand, how the changes between them as well as how the changes in the data sets vary the process and the results . Although I used Google Colaboratory to train (which means my training time and GPU time was limited), I think, I am onto something.

2 Theoretical Background

2.1 WGAN (Wasserstein GAN)

GANs have shown great results in many generative tasks over the past years, but the original process is known to be slow and unstable, therefore in recent

years, many new GANs have been made, to make the process better and more stable. WGAN provides a better solution, to make the training more stable. The problem in the original process is for quantifying the similarity between two probability distributions, which is a measure of our loss function. [1] The original method was the JS (Jensen-Shannon) Divergence, which is based on the KL (Kullback-Leibler) Divergence.

The KL Divergence:

$$D_{KL}(p\|q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx \tag{1}$$

The JS Divergence:

$$D_{JS}(p\|q) = \frac{1}{2} D_{KL}\left(p\|\frac{p+q}{2}\right) + \frac{1}{2} D_{KL}\left(q\|\frac{p+q}{2}\right) \tag{2}$$

Both metrics measures how one probability distribution p diverges from a second expected probability distribution q. The JS Divergence is more useful, then the KL Divergence, because JS Divergence is symmetrical and more smooth then the other one. It is shown in Figure 1.

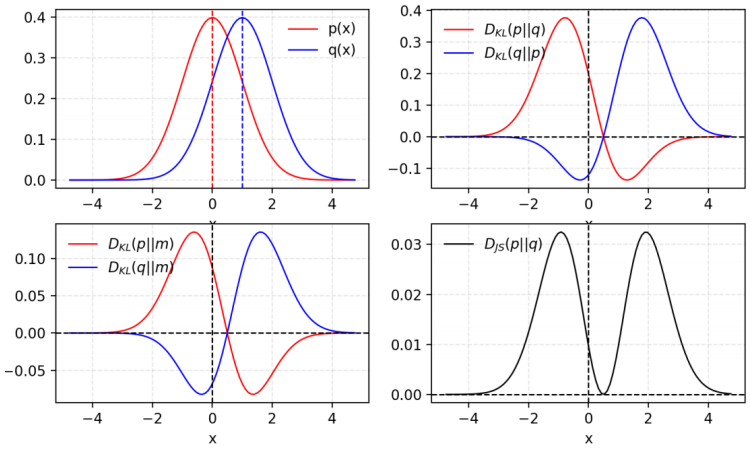

Figure 1: Given two Gaussian distribution, p with mean=0 and std=1 and q with mean=1 and std=1. The average of two distributions is labeled as $m = (p+q)/2$. KL divergence D_{KL} is asymmetric but JS divergence D_{JS} is symmetric.

Instead of the JS Divergence, we will use the Wasserstein Distance. It is called Earth Mover's distance as well, short for EM distance, because it can be interpreted as the minimum energy cost of moving and transforming a pile of dirt in the shape of one probability distribution to the shape of the other distribution. The cost is quantified by: the dirt moved multiplied by far we moved the dirt. A discrete visualization is Figure 2.

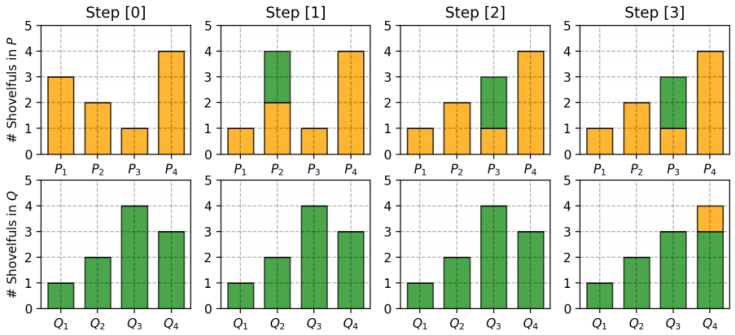

Figure 2: A discrete visualization of Wasserstein Distance.

The next formula shows, how to mathematically interpret the above:

$$W\left(p_r, p_g\right) = \inf_{\gamma \sim \Pi\left(p_r, p_g\right)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{3}$$

In the formula, $\Pi\left(p_r, p_g\right)$ is the set of all possible joint probability distributions between p_r and p_g, where p_r is the data distribution over a real sample and p_g is the generator's distribution over a data.

Wasserstein Distance is better than JS and KL Divergence, because it doesn't have sudden jumps, and differentiable everywhere, even when the two distributions are fully overlapped. It is very helpful, to stabilize the training process using the gradient descents.

If we want to use the Wasserstein Distance as our loss function, then there is a transformed version of it:

$$W\left(p_r, p_g\right) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x\sim p_r}[f(x)] - \mathbb{E}_{x\sim p_g}[f(x)] \tag{4}$$

In the transformed version there is a demand to satisfy. The function f in the new form of Wasserstein metric is demanded to satisfy $\|f\|_L \leq K$, meaning it should be *K-Lipschitz continuous*.

A real-valued function $f : \mathbb{R} \to \mathbb{R}$ is called *K-Lipschitz continuous* if there exists a real constant $K \geq 0$ such that, for all $x1, x2 \in \mathbb{R}$,

$$\left|f\left(x_1\right) - f\left(x_2\right)\right| \leq K\left|x_1 - x_2\right| \tag{5}$$

K is the Lipschitz constant for function $f(.)$. If you want to know more about WGAN, then i recommend [1].

2.2 Image Augmentation

Another big problem with GANs is overfitting. Overfitting happens, when the training causes the model to overfit the training data (the model works only for

the training dataset, but for that almost perfectly) and on the validation data it is not working as well as before. I show you a great plot from [2]:

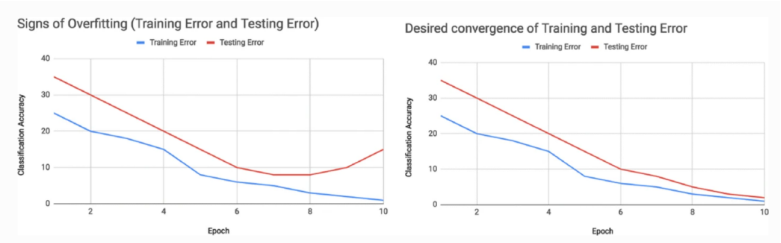

Figure 3: Example of overfitting vs. ideal training

This can be delayed/ceased if we use augmented images as well, because the model don't get fixated/later get fixated on the training data set, so it will perform better, if our estimations are correct.

Another issue is the lack of data, which is a big issue in Deep Learning in general. Because of that, our training can perform poorly more often than not. A great example for what lack of data can cause is in [3]:

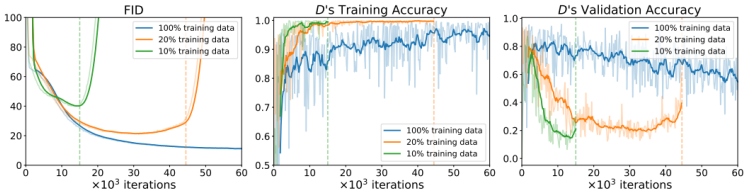

Figure 4: It can be easily seen, that more data we have, the better the model can perform, because it doesn't memorize the training data set that easily.

There are many ways [2] to augment our images, and each case of models demand usually different ways. There are geometric transformations (flipping, cropping rotating, etc.) and photometric transformations (color space transformations). Geometric transformations are great for lessen the positional biases of the training data. Photometric transformations are great for lighting issues faced by the testing data set.

These augmentations can be concatenated as well, to achieve more diversity in the data set, it is highly recommended to use augmentation that way [3].

Both augmentation techniques have of course disadvantages as well. They are memory, time, cost demanding. However it is a price, we have to pay, because more efficient solutions are unknown, or not yet applicable.

Experiments and results can be read in [3] and [4].

2.3 XGBoost

Note: This part is here, because in future works it can be a big part of improving the learning algorithm of any GAN (i suppose).

Image augmentation can solve many problem of ours, however not every augmentations can lead to better results, therefore we have to differentiate between good and bad augmentations, so we can work more efficiently, and we doesn't have to check manually every augmented images.

XGBoost is a Tree Boosting method, which is widely used to solve deep learning problems, because of the its flexibility. In our case, we use it for image classification. The basic thinking behind this method is, to getting closer our predictions to the actual values of the training data set, therefore explore a pattern in the data set, from which we can easily determine, what category we can put into our new data.

Tree Boosting methods use many generated decision trees, thus generating an additive tree model, to get closer and closer to the finish line. Each of them are working differently, in our case the learning algorithm is the Newton tree boosting (NTB)[5]. The method is shown in Figure 5.

Algorithm 1: Gradient boosting

Input : Data set $\mathcal{D}$.
A loss function L.
A base learner $\mathcal{L}_\Phi$.
The number of iterations M.
The learning rate η.

Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg\min_\theta \sum_{i=1}^{n} L(y_i, \theta)$;

for $m = 1,2,..,M$ **do**

$\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;

$\hat{\phi}_m = \arg\min_{\phi\in\Phi,\beta} \sum_{i=1}^{n} \left[\left(-\hat{g}_m(x_i)\right) - \beta\phi(x_i)\right]^2$;

$\hat{\rho}_m = \arg\min_\rho \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \rho\hat{\phi}_m(x_i))$;

$\hat{f}_m(x) = \eta\hat{\rho}_m\hat{\phi}_m(x)$;

$\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;

end

Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$

Figure 5: The Newton Tree Boosting algorithm.

The algorithm can be easily explained in more of a general thinking.

Let's begin with the inputs. The base learner is required, to learn a basis function at each iteration. We need the number of iterations and the learning rate, not to overfit our data, therefore getting closer the predictions step by step.

Before the iterations, we have to get a base value, which we can fit later on. If it is a classification problem, then it's a probability, that our image (for example) is good. That's the first step. Then we make an iteration, in which we will build a decision tree like that in Figure 7:

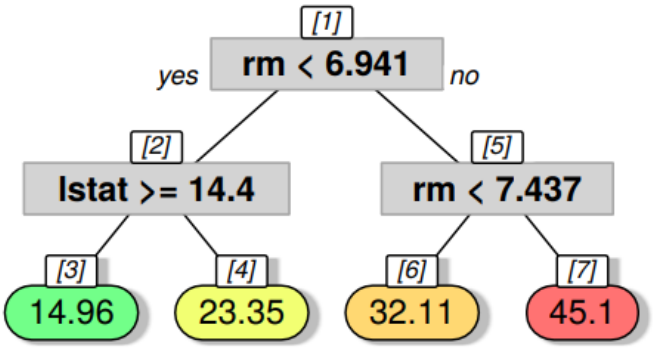

Figure 6: An example of a tree. It is for a regression problem, but the classification ones are very similar to this, only the leafs are between $[-1, 1]$, because those are deteriorations from the real values, which are probabilities.

In the leafs of our tree are deterioration values, how much we deteriorate from our values from before, based on the tree above those. We are talking about an additive tree model, so we do not just simply add the new deterioration values to the model, because it would overfit our model. We add it to the previous deteriorations multiplied by the learning rate instead. That will result in a new prediction, in our classification case, that in which category is our image. And we make these iterations until we reach the number of iteration.

If we choose our parameters well (which is hard to do), we can predict with the trained model the category almost perfectly. When a trained model get a new data to classify, it just iterates through the additive tree model, and makes a prediction.

A great video on the subject can be found here.

3 Experiments

In this section i will explain my work throughout the semester.

3.1 The GANs

After i spent the first couple of weeks with studying, how a GAN works, I came to finally train my first GAN in this Training Project Laboratory, which was followed by one more. Although there are two of them, both of them was trained two ways, with a "normal" data set and an augmented one.

3.1.1 WGAN using Dense Networks

The first model i tried was one, which used Dense Networks. The batches were of 10 images, the images' dimensions were (100, 100, 3). The layers were of Dense layers. This was in a Github repository, which had many notebooks, data sets, and algorithms inside, but they didn't work on Colab yet, which was one of my tasks. This means some adjustments to the saving methods, installing the necessary packages, clone the Github repository in the project. This didn't went that smooth, because recently there was an update in the TensorFlow version, which Google Colaboratory used. Of course after some struggles, I could train it, but in the evaluation part I unfortunately struggled again, because only the inception score worked properly. I could generate some images with that, that will be shown in the Results subsection.

3.1.2 WGAN using Convolutional Network, Gradient Penalty

Because the former model was not working properly with the augmented data set, I made, therefore i had to make working another WGAN (in the last weeks), but fortunately it worked properly from the get-go, although it didn't have that many implemented functions (image generation, evaluation on different basis'). The layers were Convolutional ones, using in the case of the Generator ReLU, in the case of Discriminator LeakyRelu. The batches were of 36 images, the images' dimensions were (100, 100, 3) originally, but when i used the augmented data set, it was (128, 128, 3). In this model the Lipschitz continuous property of the lossfunction was ensured by Gradient Penalty, instead of clipping. This model trained much faster, with shorter epochs, but it trained for many more epochs, than the previous one. More of it in the Results section.

3.2 The Data Sets

In this section i will write about the data sets, that i used for this training project laboratory. The data sets were not just raw png images, they were made into an LMDB data set.

3.2.1 Normal Pokemon Data Set

The first one was a data set consisting of 809 different pokemons. The dimensions of the images were (100, 100), and they were in RGB, so the Tensors' dimensions were (100, 100, 3). This data set was premade for me from 809, (400, 400) images.

3.2.2 Augmented Pokemon Data Set

The other data set that I used, was made from the normal pokemon data set, via augmentation. First I augmented the (400, 400) images 6 ways: shearing, rotating, flipping (top -> bottom, left -> right), cropping and recoloring. After i was done with that, I put the normal and the augmented images together, and resized them to (128, 128) or (100, 100), whichever was the correct size for the model. Then I transformed the data set into a LMDB data set, as both of the models demanded. The size of the data set was 5663 images, and they were consisted of 4553 training images and 1108 validation images. This data set is different than the other data set in scope of diversity, because not only the pokemons were diverse, but the data set itself was diverse as well, because of the augmentation.

3.3 Results

In this section i will write about the results of the training project laboratory.

3.3.1 WGAN using Dense Networks, Normal Data Set

First i trained the WGAN, in which there were dense networks. The training ran unfortunately only for 14 epochs, ending the training with early stopping. The best epoch's generated batch is shown in Figure 7.

With this model i could generate images, therefore i did so. The most interesting one i think was, when i generated random images. In the Figure 8 are shown some examples of the result of it.

As it can be seen, the results of this were not so good. The best epoch's generated batch is very dark and just in some cases can be told, that it looks like a pokemon. The random images are dark as well, in some cases the pokemons are recognizable.

3.3.2 WGAN using Dense Networks, Augmented Data Set

In the last weeks i could make the augmented data set work with this model, therefore i did train it on that. This training session ran for 34 epochs, and ended with early stopping. The best epoch's generated batch is shown in Figure 9.

I did the random image generation here as well. In the Figure 10 are shown some examples of the result of it.

This results are better in my opinion, than it was with the normal data set. The best epoch's generated batch is clearly brighter, and the outlines are much more seeable. Of course it has it's flaws, because the pokemons themselves look like they are standing in some fog, but i think, with some help from image editing tools, they can be corrected much more easier, than the results of the normal one. The random generated images are way better as well, because they are brighter. The problem here is the same, as it was in the case of the best epoch's generated batch though.

Figure 7: The best epoch's generated batch, WGAN using Dense Networks, Normal Data Set.

3.3.3 Dense Results Comparison

I think with the augmented data set the model performed better, than with the normal data set. It didn't get used to the positioning of the pokemons, had more examples of the same pokemon, therefore it could distinguish more what a pokemon looks like in general, not just in some cases, therefore it could generate better images. It can be further improved, but it is a good step in the good direction i think. I used one metric for evaluation, the inception score (mean, sigma). It can be seen here: Table 1

Table 1: Comparison of the Dense results

Used Data Set	IS mean	IS sigma
`Normal`	9.64356898e+140	1.3710762563482616e+284
`Augmented`	1.07400915e+108	3.672973271602897e+216

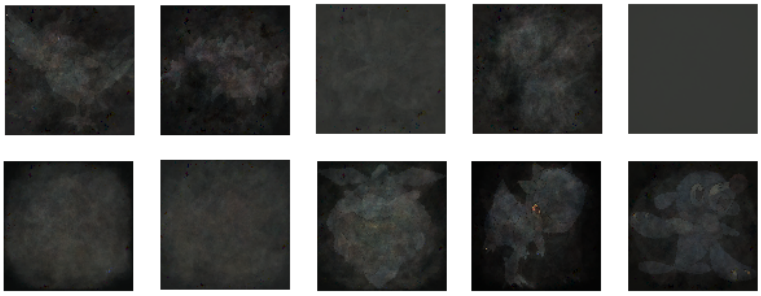

Figure 8: The random generated images, WGAN using Dense Networks, Normal Data Set.

3.3.4 WGAN using Convolutional Networks, Gradient Penalty, Normal Data Set

It was very worrying, that i couldn't figure out (later i did), how to get the augmented data set to work, therefore i got a new model to work with. This model was trained for 261 epochs. It ended, because Google Colab has it's limitations, therefore it terminated the execution at that point. However i think, it's results can be useful though. The best epoch's generated batch is shown in Figure 11.

This model was made in a hurry, so it doesn't have many functionality, so we can work with this only. This results are way better, than it was in the case of dense network, the images are way brighter, the pokemons are recognizable decently. There are some "pixel errors" in the images, but i think with some more training they are correctable.

3.3.5 WGAN using Convolutional Networks, Gradient Penalty, Augmented Data Set

In the last moments of this training project laboratory, i made this model work with my augmented data set. The initial training ran for 436 epochs and ended with termination from the Google Colaboratory. I tried to train it further (it went until the 465th epoch), but i unfortunately did not look closer, and the images generated from it vanished, because i did not copy them to my drive, and when colab terminated, it has been lost. The 436th epochs generated batch is shown in Figure 12

These images are the best ones in my opinion. In contrast to the images from the model with the normal data set, the pokemons are much more distinguishable from the background, have clear outlines. The coloring is much more "pokemonlike", i mean, that it has more uniformly colored areas. There are some cases, where the pokemons are unrecognizable. It can be, because i could accidentaly augmented some images to the point, where they were not recognizable as pokemons. In some cases, i think the model just did not train enough.

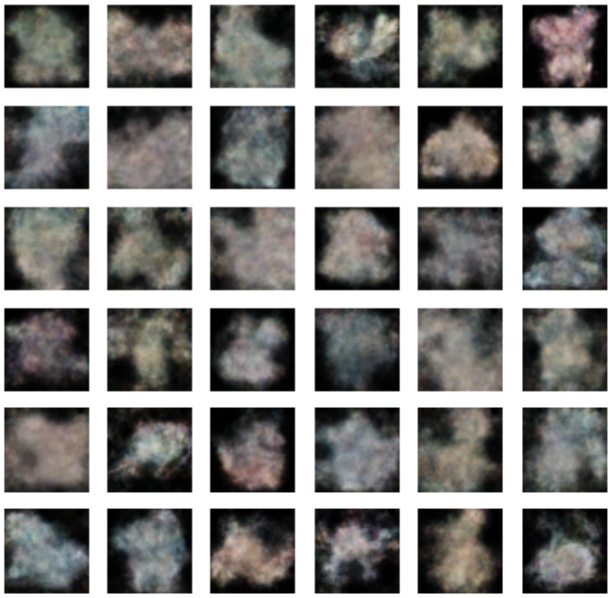

Figure 9: The best epoch's generated batch, WGAN using Dense Networks, Augmented Data Set.

In comparison with the dense networks' (augmented data set) results, i think the images are much more recognizable. Maybe with some more trainings, the dense could have done similarly to the convolutional, but i think, this network is just better for this job.

3.3.6 Convolutional Results Comparison

I can only use the losses from the training, to evaluate the two models. They can be seen Figure 13

As it can be seen, the losses didn't really imrove consistently after some time in both cases, but with more training, they could find the way to improve, i think.

4 Conclusion and Future Work

In this section i would like to conclude this training project laboratory, and give some thoughts, how the results could be improved.

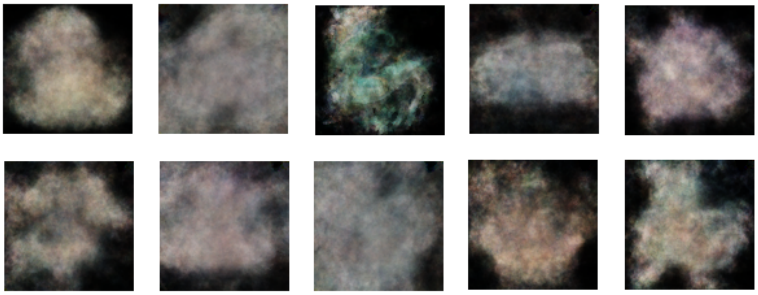

Figure 10: The random generated images, WGAN using Dense Networks, Augmented Data Set.

I think i learned during this training project laboratory a lot. In the first weeks i learned many new things in the field of Deep learning, Neural Networks and especially GANs. The middle of this period was quite hard, because i ran into a lot of problems, which were correctable only by more trained people, than me. However in the end i could produce some results, and i am happy, that i could do so. During the trainings i could learn from real life example, how different two models can work, and how much impact is in changing some minor things. I really enjoyed the work, because i really like the pokemon series, and was always excited to see, that pokemons are generating in front of my eyes.

For future work i have some assumptions, how the results can be improved. One thing could be, to use XGBoost, to distinguish between good and bad augmentations. I think so, because in the results of the augmented cases, there were some images, that aren't recognizable at all. If we used just the recognizable pokemons, the trraining could be faster, better and the results could be better. Thing is, we manually have to label images for XGBoost, to train, therefore it can be very much work. Another solution for the problems in the results can be, to not only just use an augmented data set, but in the training process the generated images should be augmented the same way as the data set, which will consist of augmented images, but augmented the same way (maybe concatenate some augmentation techniques). This way the generator will learn, to generate only pokemons without augmentations.

Acknowledgments

The author would like to express his thanks to Khalid Kahloot for his support as a scientific advisor. This work has been supported by the Department of Automation and Applied Informatics Budapest University of Technology and Economics.

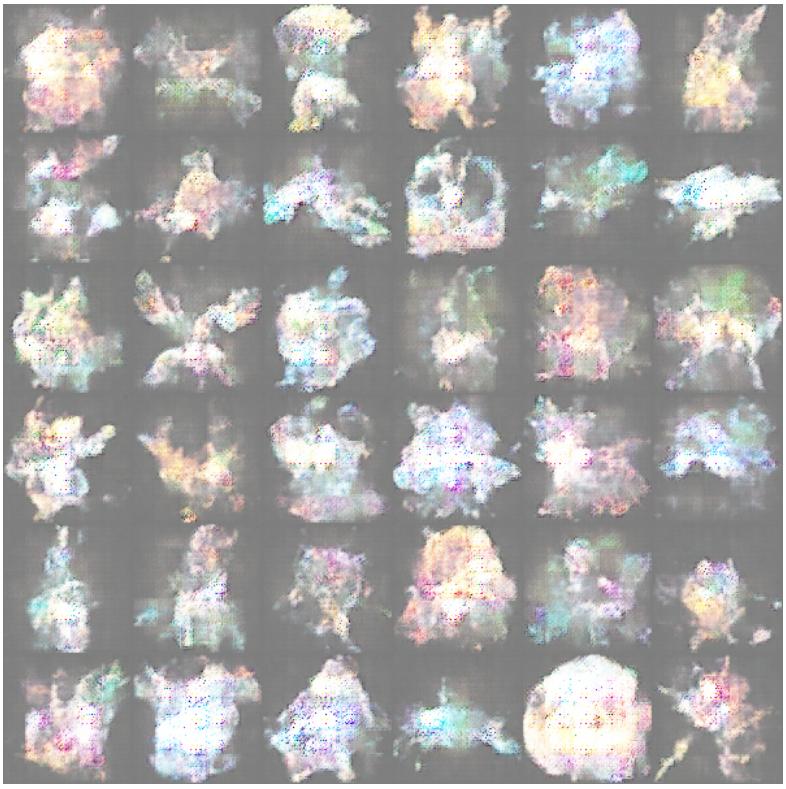

Figure 11: The best epoch's generated batch, WGAN using Convolutional Networks, Normal Data Set.

References

[1] L. Weng, "From gan to wgan." https://arxiv.org/pdf/1904.08994.pdf.

[2] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning." https://link.springer.com/article/10.1186/s40537-019-0197-0.

[3] S. Zhao, Z. Liu, J. Lin, J.-Y. Zhu, and S. Han, "Differentiable augmentation for data-efficient gan training." https://arxiv.org/pdf/2006.10738.pdf.

[4] J. Wang and L. Perez, "The effectiveness of data augmentation in image classification using deep learning." https://arxiv.org/pdf/1712.04621.pdf.

[5] D. Nielsen, "Tree boosting with xgboost." https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2433761/16128_FULLTEXT.pdf.

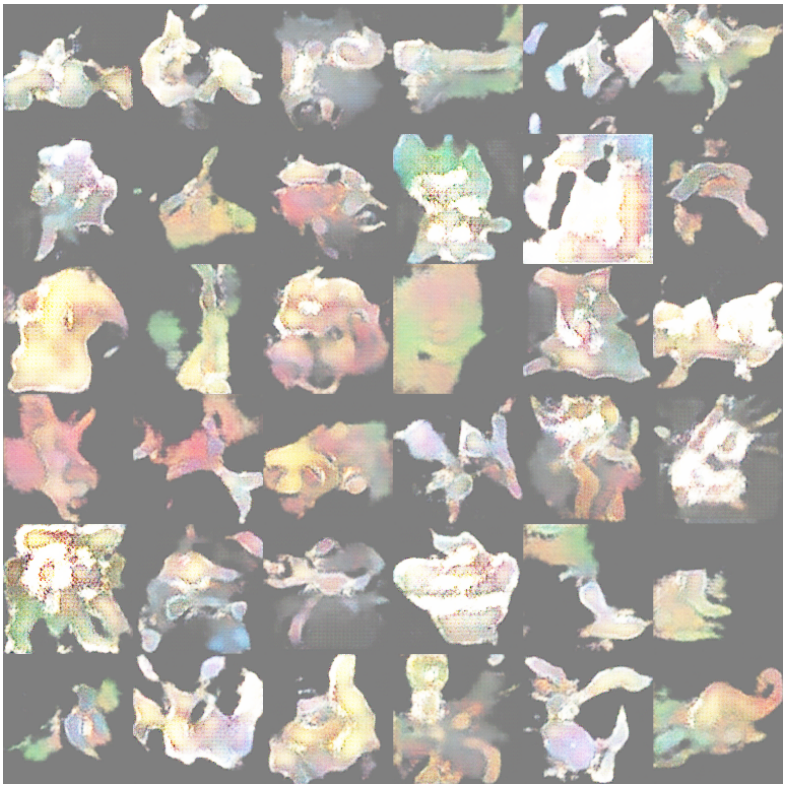

Figure 12: The best epoch's generated batch, WGAN using Convolutional Networks, Augmented Data Set.

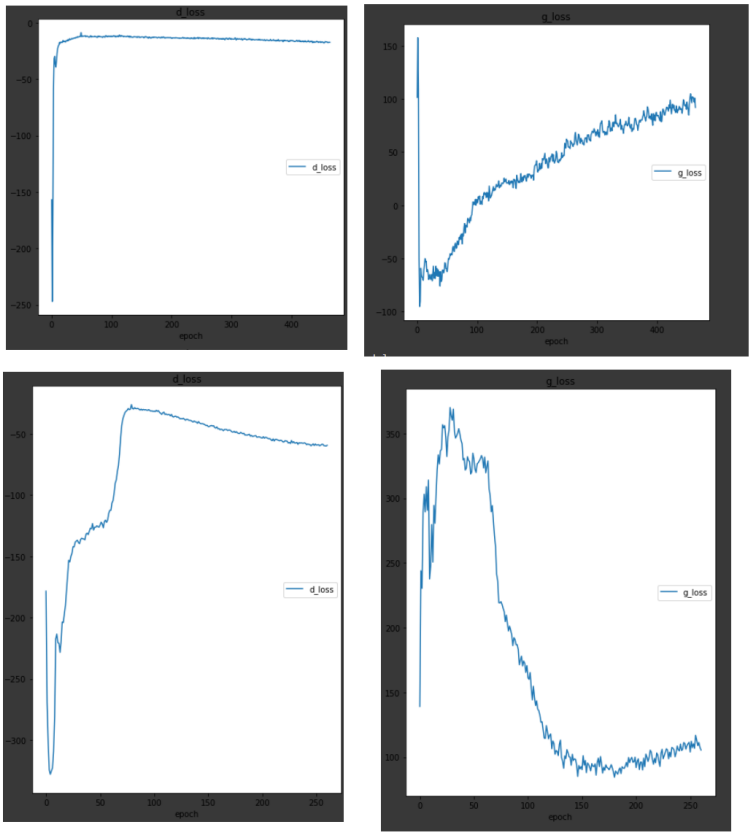

Figure 13: The losses' plots. First row has the augmented plots, the second has the normal plots. The first column has the discriminators, the second has the generators.