

Pokégans

Using generative adversarial networks to create new Pokémons

Michael Hüppe, Nicolai Hermann

March 2021

1 Introduction

Making it's game debut in 1996 with "Pokémon Red and Green" and overall grossing around 368 million on game sells alone the Pokémon series is one of the oldest and most successful franchises of Nintendo. Combining both unique character designs and round based role-playing the first game introduced an entire generation to a distinct game style making it the 7th most successful game of all time. Thus the cultural status the franchise holds today is no surprise. "Fakémons", defining self made Pokémons, have been popularized in the community to the point where they have been featured in entire fan-made games. Therefore the idea of generating new Pokémons with the help of generative adversarial networks was not far away. In the following the methodology for scraping and uniforming the data as well as constructing the network is described. Additionally, we will present some uses of the findings we made. Because of the time limit and the overall scope of this project we weren't able to implement certain ideas which could result in performance gains. However, for the sake of future projects these optimization techniques are discussed at the end of this paper.

2 Data

2.1 Web scraping

The biggest worry we had before beginning the project was the lack of provided data. There are around 900 unique Pokémons each having distinct and easy distinguishable designs making it really hard to find similarities between them and replicating them. Moreover, is there no official image dataset containing more than one of the same rendition of Pokémons. Thus we web-scraped it ourselves. We used the "BeautifulSoup" and "request" package to gain access to the website "pokemondb.net" and parse it into html, find all the attributed tags and use them to load all possible data for each Pokémon. The "national Pokédex" outlined the starting point. Ordered per Generation (here meaning the newly added Pokémons per game) the list gave access to each Pokémon and its current entry. After collecting the links for each entry getting all their accessible images was made easy by the systematic naming convention. The link is constructed as follows: "pokemondb.net/identifier/pokémon". Thereby changing the identifier gave access to

two distinct data sets. While the identifier "artwork" hoards both official and alternative artwork the "sprites" section stores all in-game renditions of the pokémon. In total that provided us with 26,896 (22,825 sprites and 4071 artworks) unique images. Unique however has to be used with caution as we deliberately chose to also include both male and female versions (even though they often differ only by a small detail) and the normal and shiny version (same pokémon but differently color coded). Additionally, event versions like "pikachu" but differently clothed were kept while sprites showing pokémon from behind were excluded.

The code for web-scraping the images can be found [here](#).

Examples for an [entry](#), [their artworks](#) and [sprites](#).



Figure 1: Different renditions of the same Pokémon in various art styles (right to left: Sugimori artwork, Early Sugimori artwork - Red/Green JP, Generation 5 & 6 sprites, Global Link artwork (vector))

2.2 Uniforming

The next problem we encountered was the difference in their background, their image mode and type. At first we resize every image to 64x64. The original artwork and sprites were all transparent ".png", meaning they have no background and therefore a fourth channel encoding the transparency of the given pixel. This fourth channel resulted in partially transparent Pokémon and added unnecessary parameters to the network. Therefore we converted each of the "RGBA" images to uniform "RGB" images with a white background to keep their characteristic black outlining.

As the alternative artworks were collected from a variety of different artists there was no convention resulting in different file types (jpg, jpeg and vector png), backgrounds and image modes. To keep the format we established in the previous step we again removed the additional transparency channel and converted each image to png. Additionally, we detected each image which did not have a white background and adjusted it accordingly. Moreover, was there a difference in coverage of the image. While the artworks used all available space of the image the sprites only used around 10 % therefore exposing a lot of white background. This was apparent after training a few test epochs which resulted in solely white images with a small focus somewhere in the center (which was no surprise as the sprites represented the majority of the dataset). To counteract this, we located the focus point of the image (location of Pokémon) by removing each row/column only containing white pixels. Then to avoid morphing the Pokémons shape we padded it to a square. At last we again resized the cropped image to get a uniform size of 64x64 pixels.

The code for uniforming can be found [here](#).

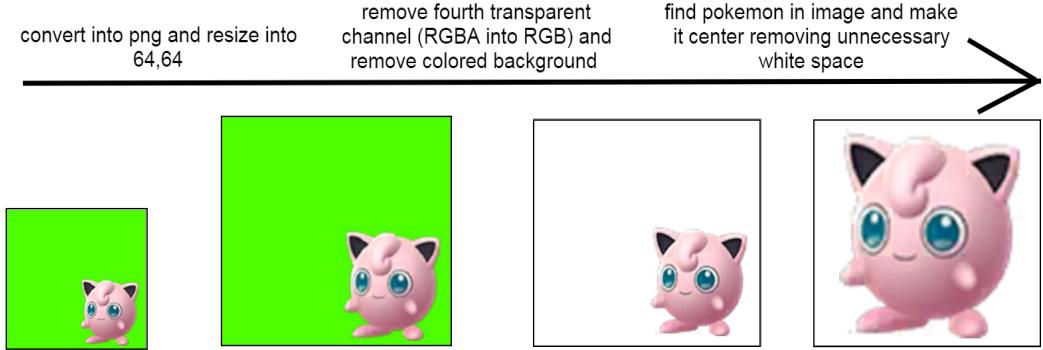


Figure 2: Typical data uniforming process

3 Inspiration

After some research we noticed that generating Pokémons was already an implemented idea. The most common technique being used is a simple generative adversarial network. Examples can be found [here](#) and [here](#). The most common difficulty we saw in these examples was that they only had around 900 images at hand. Thus their results leave room for improvement:

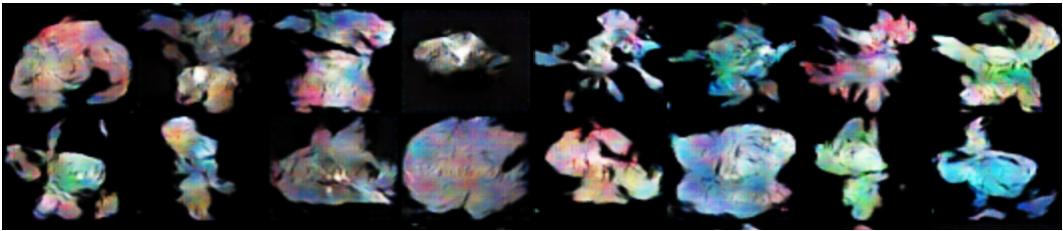


Figure 3: Example of generated Pokémons using a GAN from JOVIAN

4 Model

4.1 Vanilla GAN

The already existing implementations we found tackled the problem with a vanilla or deep convolutional generative adversarial network. A conventional GAN consists of the discriminator and the generator. While the generator aspires to create indistinguishable samples of the given dataset from random noise the discriminator tries to distinguish those from real ones (employing binary classification). Their antagonistic relationship is also apparent in their respective loss function. While the discriminator is penalized for miss-classifying its given samples (classifying real ones as fake and fake as real) the generators loss benefits from false classifications being made. The overall structure the two models form can be seen in the following figure:

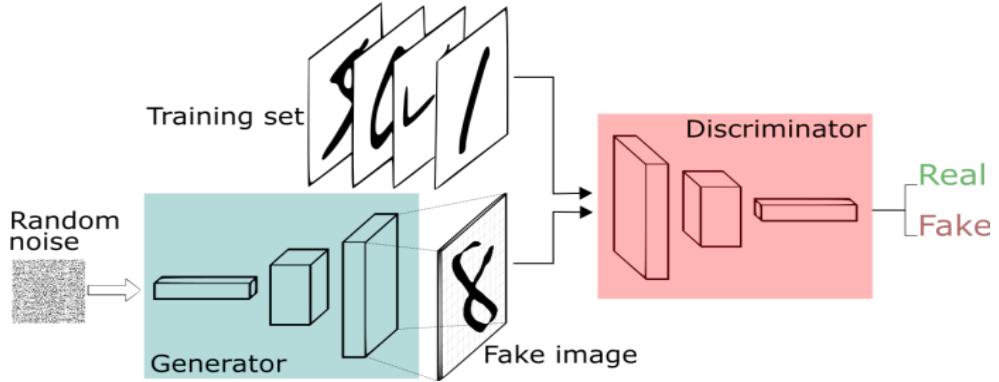


Figure 4: Structure of GAN

4.2 Problems

However the vanilla GAN faces problems which can lead to unsatisfying results. Firstly, GAN's do not typically converge rather aiming for a "Nash-equilibrium" meaning a state where both of them find their respective optimum which is often much harder to achieve. Additionally, the performance of the discriminator has a direct affect on the learning of the generator. If the discriminator classifies over-proportionally good its loss gets closer to zero, resulting in vanishing gradients and no or slow learning for the generator. However, a bad performance leaves the generator with no meaningful representation to replicate. Mode collapse, meaning that instead of learning the underlying sample representation the generator only focuses on a small space essentially replicating the same samples with no variety, can arise. At last GAN's typically lack a good evaluation metric indicating the quality of results or improvement of learning thus making model comparison and hyperparameter adaption harder.

4.3 Solutions

4.3.1 WGAN-GP

The vanilla GAN typically makes use of the KL or JS divergence for similarity measure. In contrast, the Wasserstein GAN employs the "Wasserstein distance" improving the in-between distance representation. Under the synonym of "Earth-Movers Distance" the idea of a cost function, defining the minimum distance it takes to convert one probability distribution into another, becomes apparent. However to adequately minimize the distance between the samples and the generated output (or rather maximize the original function of $W(p_x, G(z)) = \frac{1}{K} \sup_{\|f\|_L \leq K} E_{x \sim p_x}[f(x)] - E_{x \sim G(z)}[f(x)]$) Lipschitz continuity has to be used. Usually this is done by applying weight clipping, meaning to set boundaries for updating the weights. However, this can still cause slow convergence or vanishing gradients if the clipping window is either to large or to small. Therefore one can use gradient penalty instead. The idea here is that the interpolated points between the real and generated data is supposed to have a gradient norm of 1 for the differentiable function f . Therefore the model is penalized if its gradient norm differs from 1. After combining

both the more valuable evaluation metric presented by the Wasserstein distance and using gradient penalty to minimize weight clipping problems, higher training stability is given.¹

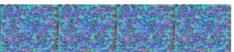
DCGAN	LSGAN	WGAN (clipping)	WGAN-GP (ours)
Gated multiplicative nonlinearities everywhere in G and D			
			
tanh nonlinearities everywhere in G and D			
			
101-layer ResNet G and D			
			

Figure 5: Performance of WGAN-GP

4.3.2 AEGAN

Typically vanilla GANs learn to map some latent space to the given sample space i.e. transforming random noise into indistinguishable (tho unseen) samples from the dataset. Thus each point in the sample space Z ought to map to a sample in the dataset X . Though injectivity is often given surjectivity is not. This is apparent in the appearance of mode collapse where multiple points in the latent space map to the same point in the dataset. Therefore the model tends to replicate only a specific sample while disregarding others.

To ensure bijectivity between both the latent and the sample space the AEGAN architecture makes use of the generator function $G : Z \rightarrow X$ and the additional encoder function $E : X \rightarrow Z$. While the generator function serves the same purpose as in the vanilla GAN (replicating realistic samples from the latent space) the encoder is meant to do the opposite thus replicating latent space points from the given samples. Differentiation between real and generated samples/latent points is established by training two separate discriminators thereby completing the four network architecture. Taken from the original paper [3] the following visualization illustrates the relation between the four networks. The generator G , the encoder E and both Discriminator D_x and D_z are depicted with a square while their input ² and output values ³ are illustrated with a circle. The losses are visualized using diamonds where L1 and L2 respectively stand for the image and latent vector reconstruction loss (former being measured with the Manhattan distance and latter with the Euclidean distance) and the red/yellow GAN for the adversarial image/latent vector loss. The colored arrows represent the structure of the their relationship. (red = vanilla GAN, blue = image autoencoder, yellow = latent vector generator, latent vector autoencoder).

¹refer to "Improved Training of Wasserstein GANs" (2017) [2] for more performance comparison

² x, z representing the dataset and random noise i.e. the sample and latent space

³ \hat{x}, \hat{z} portraying the produced samples/latent points

\tilde{x}, \tilde{z} denote the reproductions from \hat{x}, \hat{z} meaning $G(E(x))$ and $E(G(z))$

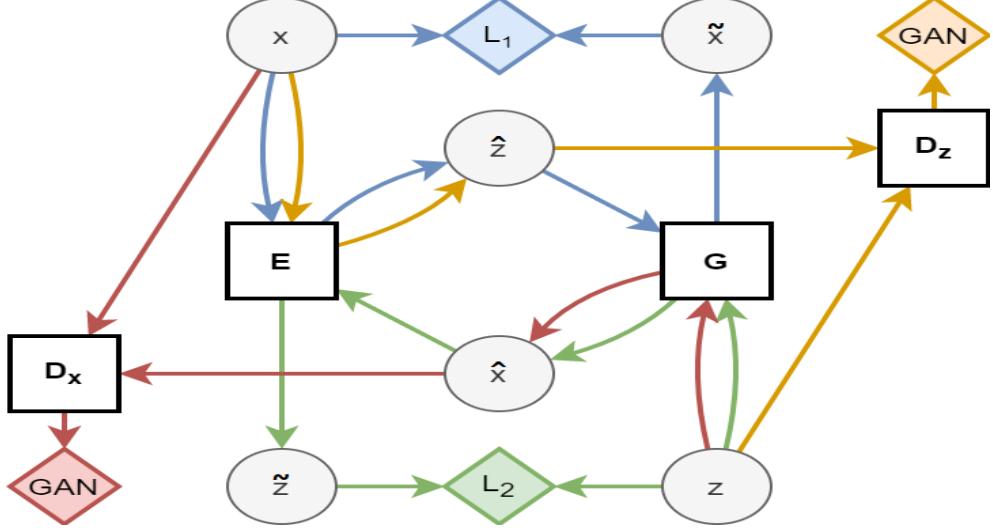


Figure 6: Performance of WGAN-GP

5 Training

Training the AEGAN and WGAN-GP are both based on the training algorithm of the vanilla GAN. Typically, both individual parts of the GAN are trained separately where a training step is defined for each network respectively. Therefore the usual approach is to train the discriminator for a number of epochs on both real and generated samples and then train the generator with the discriminator as its fixed loss function.

Though the coarse structure for the AEGAN is kept understanding the concept as a whole can be quite complex. To avoid confusions we will analyse each component individually. One training step includes eight updates per phase (16 in total). Each Discriminator is trained on two real data batches ($2x$ images x , $2x$ noise z) aiming to predict them as real and the following as fake. In addition the image discriminator D_x is trained on one batch of reconstructed images ($\hat{x} = G(E(x))$) and one batch of generated images ($\hat{x} = G(z)$). Accordingly the latent discriminator D_z gets two extra updates on one batch of reconstructed latent vectors ($\hat{z} = E(G(z))$) and one batch of embedded real images ($\hat{z} = E(x)$). The loss function applied is binary-cross-entropy. However, grasping the concept of the generative phase can be more difficult as it incorporates three loss functions.

Building an intuition for this phase might be easier with the graph given in the appendix. In four rounds the model is fed both a batch of noise z and one of real images x . Therefore, model returns six outputs in total. The first one being reconstructions of the real input images and the second one a reconstruction of the input noise/ latent vectors. Although having the opposite goal (fooling the discriminator) the remaining four resemble the outputs of the discriminator phase. When computing the overall prediction error for the model, the image reconstruction loss is determined by the mean absolute difference to its original and weighted with a factor of 10. The mean squared error of the reconstructed latent vector is weighted with a factor of 5. The

remaining ones are computed with BCE again and are not further weighted. Noteworthy here is that the opposing networks (D_x , D_z vs. G , E) are decoupled in their respective phase, meaning one is not updated while the other one trains. Considering an ablation study we used a batch size of 32 and a latent space of 70. While changing the batch size, learning rate (0.0005) or optimizer had no apparent effect increasing the latent space resulted in early collapse of the generator only generating blank images. However, a too small latent space leaves no valuable representation for the generator thus not being able to find clear shapes to replicate. We trained the AEGAN for roughly 384 000 training steps and 72 hours in total using a tensorflow with a local GPU (GTX 1650). Training the WGAN, though similar to the vanilla training step, makes following changes: Firstly train the generator and the discriminator and get both of their respective losses. One has to note that the original paper [2] suggests to perform 5 discriminator training steps for each generator training step. Then by taking the norm of the gradient calculate the gradient penalty. After multiplying this gradient penalty with a constant weight factor it's added to the discriminator loss. Finally update the weights of the respective network with the given optimizer. Moreover, does the original paper recommends to use RMSProp rather than ADAM but we did not see any specific performance gain with either of them so we chose ADAM with a learning rate of 0.0002. Additionally, we specified the latent space as 128 and trained with a batch size of 32. Here the results of the ablation study of the AEGAN is also applicable meaning that only changes in latent space were noticeable. Here we trained for around 68 hours also using a local GPU.

6 Implementation

For this project Tensorflow and its package Keras was used.

For the WGAN-GP we made usage of [this implementation](#) provided by Aakash Kumar Nain on the official Keras site [5]. To employ the .compile and .fit function the training function was overridden and the two sub models are defined via functional models. The discriminator makes use of a convolution block and the generator employs an upsampling block. Here the number of filters, activation, kernel and stride size, padding and the usage of batch normalization or dropout can be specified. Although this implementation was of great help regarding the training step it was created for the Fashion-MNIST dataset thus its architecture was only suited for a simpler gray scale images. Therefore, we modified the overall architecture of the two networks to fit our RGB dataset.

The AEGAN paper offered a code, constructed for the 'anime faces' dataset [3] therefore giving us a reference to work with. However, due to the unconventional coding style we had to reimplement everything only keeping the rough architecture of the model. Because of this complex model design this reimplementation was arguably our biggest challenge. Relying more on sub-class models was a big mistake as saving them, while using custom layers, is almost impossible. Thus we had to switch back and re-implement all of them as functional models. The train_step function of the Aegan was overwritten to enable the ".fit" and ".compile" functions for more structure, compatibility, speed and the possibility of employing custom callbacks. Additionally, we adapted the architecture to produce 128x128 images but ultimately reverting back to 64x64 as we constantly ran out of memory and the training time increased immensely.

The code for all models can be found [here](#).

7 Results

In the following the three implemented models are compared with respect to progress made over different epochs and evaluated based on characteristics of quality. At last we will compare the results we deem as best from each model. After researching the design of Pokémons we narrowed it down to five measurements of good value, meant to represent how good the generator learned the underlying composition of a Pokémon.

The data normalization mentioned previously gave us a sixth measurement. Therefore the quality measures from least to most difficult⁴ to replicate are: white background, black lining, darker shadows on lower half of the Pokémon , lighter highlight on upper half of the Pokémon, eyes and distinct body shape and limbs.



Figure 7: Quality measurements based on design characteristics found in Pokémons

7.1 DCGAN

To get a base line for comparison we trained a deep convolutional GAN to better evaluate the progress our new architectures made. Implementation wise we used the same approach as we did with the WGAN. Therefore based on [this implementation](#) [1] we modified the training step to fit our purposes. The following figure illustrates the progress made by the DCGAN over 191 epochs. The complete animated training process of 900 epochs can be found [here](#).

⁴supposedly after observing the training process and prominent features in the dataset

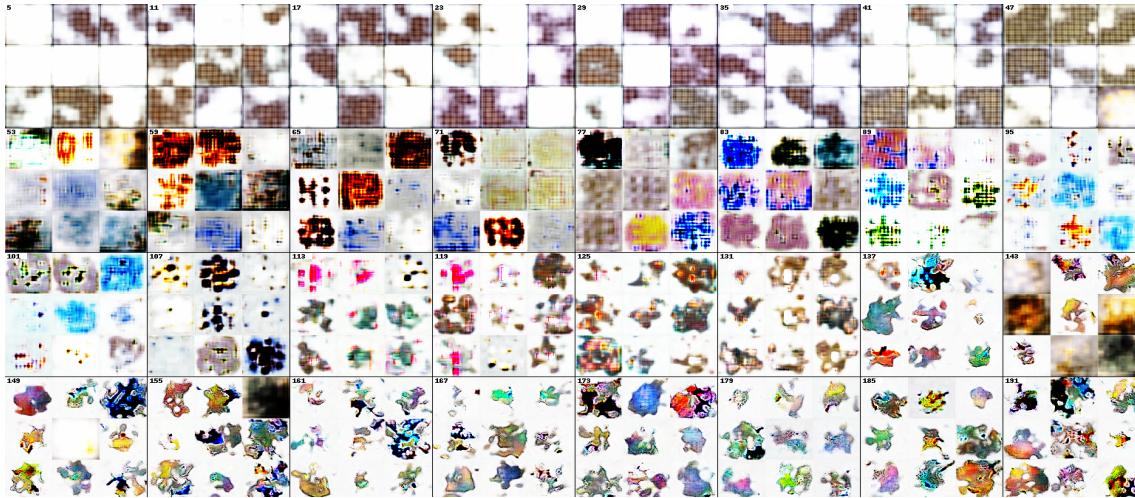


Figure 8: Progress of image quality of the DCGAN

In the first 50 epoch it can be seen that the DCGAN tries to generate a white background while not focusing on any color. The following 50 epochs represent the opposite. Although colorful, the DCGAN fails to generate images with a strong focus point. However, in the next 50 epochs both a white background and a colorful focus point is prominent. In addition to this the checkerboard-esque look of the previous images decreases, resulting in more uniform colored spaces. Nevertheless, a black lining cannot be seen until the continuing 30 epochs. This however fades with the remaining 20 epochs. The quality of the images kept declining after the 190th epoch. Moreover, eyes, limbs or proper shading/highlighting were not observed in multiple training runs.

7.2 WGAN-GP

The WGAN-GP is supposed to ensure training stability and decrease the risk of mode collapse. The following figures illustrate the progress and quality measurements in images generated by the WGAN-GP architecture until the 46 epoch.



Figure 9: Training progress of WGAN GP

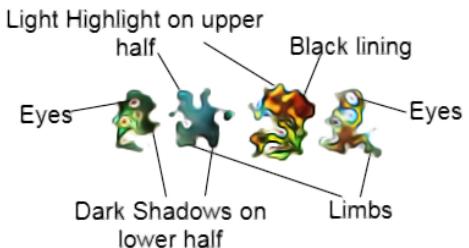


Figure 10: Observed features from the WGAN-GP (epoch 46)

Evident in the training progress of the WGAN-GP is that the first 3 criteria of quality are already met in the sixth epoch. In addition to the white background, black lining and shadowing, more uniform color palettes and lighter highlights start to arise after epoch 20. Moreover, eyes (though often multiple instead of the typical pair) can be observed after epoch 36. However, a clear body shape or distinct limbs are barely noticeable and are only hinted at after epoch 50.

7.3 AEGAN

The WGAN-GP is supposed to ensure training stability and decrease the risk of mode collapse. Additionally to more training stability the AEGAN architecture promises a decrease in mode collapse due to its bijective mapping. Figure 11 describes the progress made until the 75th epoch.

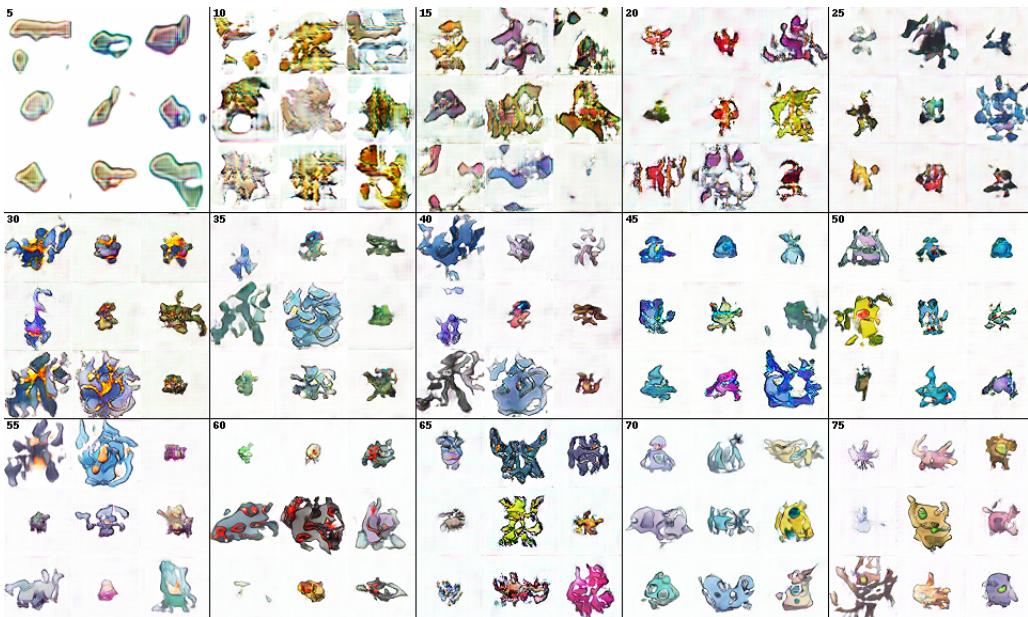


Figure 11: Progress of image quality of the AEGAN

One can observe that a white background and noticeably darker lining can already be found in the 5th epoch. While shading becomes more prominent after the 30th epoch eyes can not be seen until the 70th epoch. However, in contrast to the two previous models body shapes are seen much earlier in training. Due to the bijective mapping gained access to the interpolated images of two generated samples. Therefore we were able to create an animation for an evolution of one Pokémon into another. An interactive notebook for this can be found [here](#).

7.4 Comparison

Figure 12 visualizes the quality difference between the three models by taking samples after the 53th epoch. Apparent here is that both WGAN-GP as well as AEGAN have more distinct

features than the DCGAN at this point. Figure 13 illustrates our favorite results of each model.



Figure 12: Quality difference between the model in epoch 53



Figure 13: Best generated Pokémon of each model.

It can be observed that the AEGAN learned to represent all quality measures including a clear body shape and limbs (epoch 220). Though focusing on eyes the WGAN also managed to bring comprehensive results (epoch 190). Although the DCGAN eventually managed to achieve some results with a clear body shape the appearance thereof was quite spar and only after training around 900 epochs (epoch 921).

In conclusion it can be said that both the WGAN-GP aswell as the AEGAN enhance the quality of the generated images in respect to a more simpler DCGAN. Moreover, do the more complex model

8 Outlook

The last thing for us was to evaluate what we could do with the results that we got and how to optimize them. To get better results we thought of implementing a "color-coder" inspired by this implementation [4]. Based on the fact that Pokémons are color coded, meaning that they have a specific color scheme for their design, Lazarou implemented 3 additional layers for each color channel encoding its usage. Therefore the network learned to disregard the other color channels if one is already strongly activated thus the Pokémons get a variation of green shades rather than a more (unpleasing) variation of all colors.

Additionally, did we intent to combine both the WGAN-GP and the AEGAN but due to time restrictions a working implementation wasn't finished. The last optimization we tried to employ was to combine both the WGAN-GP and the AEGAN with a class conditional GAN to achieve training stability and more distinct and common features. However, due to the complex and varying designs of the Pokémons it is hard to find common similarities worth generalising for.

Therefore we present following approaches:

Categorizing them based on their type might be beneficial as "water" types commonly have fins or "grass" Pokémons are often green. The idea behind this was that the model easily learns that a grass Pokémon is green and can then focus on more details (adding flowers for instance). However, after grouping the Pokémons there were several problems with this technique:

Firstly, almost all Pokémons have more than one type, so one would have to distinguish between the influence both types have on the Pokémons and make a decision whether or not these attributes align more with one type or another.

Secondly, the types themselves often do not have any specific characteristic that defines them.

The type "normal" features no apparent color coding or lacks any uniformity in body shape. At last there are 18 different types thus each type would have roughly 1400 unique images which for the variety in appearance isn't enough.

Therefore we switched to a different strategy which was grouping them based on their shape alone. Conveniently "Pokéwiki" tags every Pokémon with a shape indicating whether or not it is bipedal, has a tail, consists only of its head, has wings etc..



Figure 14: Different shapes of Pokémons. left to right: only head, serpentine body, with fins, head and arms, head and base, bipedal with tail head and legs, quadruped body, single wings, tentacles, multiple bodies, bipedal tailless, multiple wings, insecticide body

This categorization results in fourteen different classes minimizing variation in their body shape while ensuring enough differences in their color and other features.

The code for categorizing the Pokémons can be found [here](#).

Although, pleased with the results we still have to admit that these are not yet fully game worthy. However, they do give you a sense of shape and idea what a new Pokémon could be. Therefore they could offer inspiration for an artist aspiring to fully conceptualize them. Additionally, was there an idea to make a classifier able to recognise patterns between different types (such as "fire-type" Pokémons being red or "flying-type" Pokémons having wings) and assign them types and attacks (move-sets). However, due to the complex implementation which costed us a lot of time we weren't able to realise this idea. We do have to admit that we are not the best artist however this is something our results might bring:



Figure 15: Possible real world application for results

8.1 Conclusion

In conclusion it can be said that both of our implemented models achieve satisfying results and optimize the outcomes of a simple GAN. Never having worked with BeautifulSoup or any kind of Web scraping software being able to accumulate the largest (at least which is publicly available) Pokémon image data is one of our proudest achievements.

Additionally, the research intensity the AEGAN and WGAN-GP brought and the successful implementation of these models is more than enough to leave us satisfied with this project. At last we hope that this rather nostalgic review of modern image generation brought you an interesting view on GANs. Saying that we would like to present the proudest achievements of this project all generated via AEGAN or WGAN-GP:



Figure 16: Favourite results of both AEGAN and WGAN-GP

References

- [1] François Chollet. Dcgan-gp overriding model.train_step. 2020.
- [2] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. 30, 2017.
- [3] Conor Lazarou. Autoencoding generative adversarial networks. 2020.
- [4] Conor Lazarou. I generated thousands of new pokémon using ai. 2020.
- [5] Aakash Kumar Nain. Wgan-gp overriding model.train_step. 2020.