Pokemon image generation

Lucia Baldassini (s2775468) Andreas Pentaliotis (s3667537)
Li Meng (s2651513) Avinash Pathapati (S3754715)

Deep Learning
Faculty of Science and Engineering
University of Groningen

1 Introduction

In this paper we compare two methods for generating Pokemon images, Deep Convolutional GAN (DCGAN) and Wasserstein GAN (WGAN). For this project we used two datasets, which will be introduced in Section 2 of this paper. As it will be shown later, the first dataset produces poor results compared to the second one. Section 3 and Section 4 provide an overview of the architectures used for DCGAN and WGAN along with some of the results obtained. We conclude this paper with a short conclusion and points of discussion.

2 Datasets

Both datasets were preprocessed using the same technique. The images were resized to 128x128 pixels and normalized. Moreover, the images were augmented by shifting the width and the height in the range [-0.2,0.2], by allowing random rotations in the range of 10 degrees and by allowing a random zoom in the range [1-0.2,1+0.2]. Finally, the images were normalized in the range [-1,1].

2.1 Pokemon Generation One

The first dataset we used is the Pokemon Generation Dataset [1]. It contains over 10,000 images of First Generation Pokemon divided into 151 classes. Each class contains around 60 images of the same Pokemon. However, after accurate inspection, we had to manually remove many images because they poorly represented the classes (for example, they contained only noise). In the end, we were left with 40% of the original data, which turned out to be too little to obtain good results, as it will be shown below. For this reason, we decided to train the model on a second dataset.

2.2 Mgan

The second dataset that we used is called Mgan [7] and it contains around 14,774 images of different sizes, not divided into classes.

3 DCGAN

3.1 Overview

The deep convolutional generative adversarial network (DCGAN) was first introduced by Radford et al [6] in 2015 and it combines the concept of generative adversarial networks [3] with convolutional neural

[1] https://www.kaggle.com/thedagger/pokemon-generation-one

networks. As convolutional neural networks are appropriate for dealing with images (see for example [2]), the DCGAN was appealing for our task.

Our DCGAN implementation followed the architectural guidelines for stable DCGAN implementations provided by Radford et al [6]. Although we experimented with several hyperparameters such as optimizer learning rate, leaky ReLU alpha parameter and number of kernels in convolutional layers we found that, in general, the hyperparameter values mentioned in the guidelines provided the best results. In this section, we first describe our final architecture for the generator and discriminator and then we present the results for both datasets.

3.2 Discriminator

Our discriminator takes as input the preprocessed images of size 128x128 pixels, which are comprised of three colour channels and have pixel values that range from -1 to 1. The input images are then passed through a convolutional layer with 128 kernels of size 5x5. We pad each input image with zeros before we apply the convolution operation, but we also use a stride of 2 to shrink the image height and width while creating the feature maps. After the convolution operation, we apply batch normalization to the feature maps and then we pass them through a leaky ReLU activation layer with an alpha of 0.2. The leaky ReLU activation function is given by

$$g(x, a) = \max(0, x) + a \min(0, x) \tag{1}$$

where, in our case, a = 0.2. This allows for gradient descent even when the input x is negative. We then apply this pattern three more times. The only change in the pattern is the number of kernels used by the convolutional layer. We gradually increase the kernels from 128 to 256, then to 512 and then to 1024. Consequently the flattened feature maps are fed to a dense layer with only one unit that implements the sigmoid activation function which is given by

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

This is appropriate for the task of determining whether an image is real or fake because it squashes the input x to a number between 0 and 1, and this number can be seen as the probability that the image is real. The objective of the discriminator is to minimize the binary cross-entropy loss function using the Adam optimizer with a learning rate of 0.0002 and a β_1 of 0.5 - we keep the default values for the rest of the parameters. As a result, it should distinguish real from fake images.

3.3 Generator

Our generator can be seen as operating in the opposite direction of the discriminator. Specifically, the generator is given a 100-dimensional Gaussian noise vector as input. Then, the input vector is fed to a dense layer of 161984 units, which is immediately reshaped to have dimensions 4x4x1024. We then use fractionally strided convolutions implemented with the `conv2DTranspose` function provided by the `Keras` library to upsample the reshaped noise gradually. The first layer after the reshaping operation implements the transposed convolution operation with 512 kernels, while the kernel size, strides and padding parameters are the same as in all the convolutional layers of the discriminator. The resulting feature maps are then passed through a batch normalization layer followed by a ReLU activation layer. The ReLU activation function is given by

$$R(x) = \max(0, x) \tag{3}$$

and allows for gradient descent only when its input x is positive. This pattern is then repeated two more times with the kernels in each transposed convolution layer decreasing by 512 to 256, and then to 128. After that, we use one final transposed convolution layer with 3 kernels in order to upsample the feature maps to dimensions 128x128x3. In this layer, the kernel size, strides and padding parameters are also kept the same as in all the convolutional layers of the discriminator. Finally, the feature maps are passed through a layer that implements the tanh activation function that is given by

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4}$$

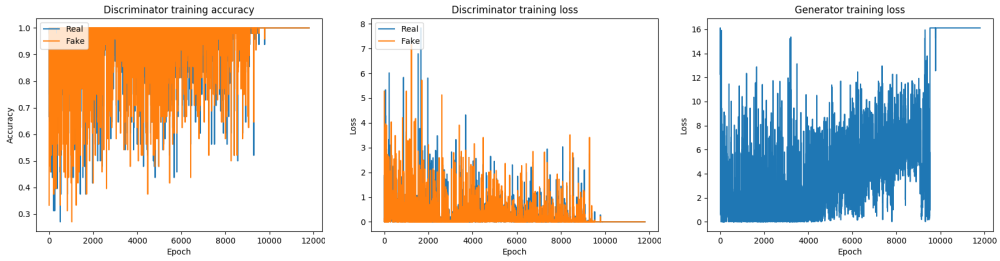
and squashes its input x to a number between -1 and 1, effectively generating a fake image that has the same size and range of pixel values as the preprocessed real images. The objective of the generator is to create fake images from noise, associate them with real labels and use them to minimize the binary cross-entropy loss function using the Adam optimizer with the same parameter values as the ones used by the discriminator. As a result, it should fool the discriminator into thinking that the fake images are real.

3.4 Results

3.4.1 Pokemon Generation One dataset

We trained the model on different classes. Some examples can be seen in Figure 2. First, we started by generating Pokemons from only one class (first row of the figure). Although we can clearly recognize the class of the Pokemon, the images are a bit deformed so that no human would consider it as real Pokemon images. We also generated types of Pokemon, such as water and fire Pokemon (second row of the figure). In the last example, we see that the model generated an image with blue colors, as a water Pokemon should look like. However, the generated image is of poor quality, the shape of the Pokemon is not well defined and we see some blobs of colors that should not be present. Lastly, we generated Pokemon from all classes (last row of the figure). Those are the poorest generated ones: the shapes are again not well defined and we see blobs of colors or white areas that should not be present.

The bad quality of the images is reflected in the graphs that were generated during training. For brevity, we only show a couple of examples, see Figure 1. In general, we see that the training process is very unstable, as can be seen by the large amount of spikes in the plots shown. We also see that at around 9000 epochs the training saturates, after which the algorithm does not learn anymore.



(a) Training accuracy of discriminator (b) Training loss of discriminator (c) Training loss of generator

Figure 1: Training accuracy and loss of discriminator and training loss of generator for a sample training interval of 12000 epochs.

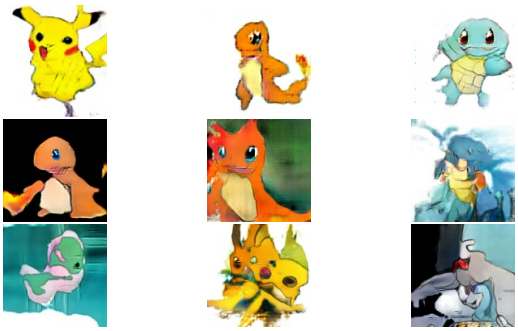
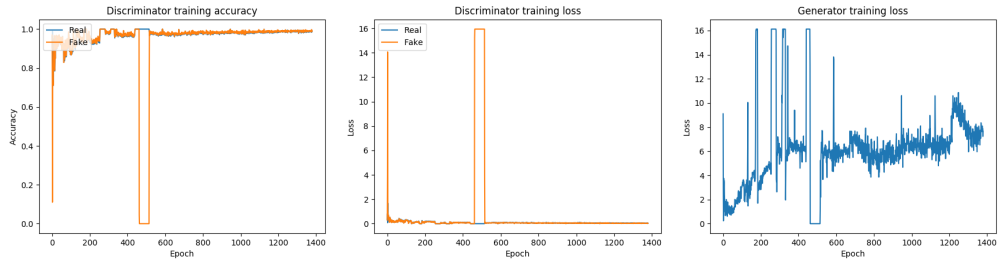


Figure 2: Sample of generated images. First row: example of one-class images. Second row: examples of generated types of Pokemon. Third row: example of generated Pokemon from all classes.

3.4.2 Mgan dataset

We also trained on the Mgan dataset without changing any hyperparameter in our DCGAN architecture. Although we still had the unstable training issue, the results were better. In Figure 5 we can see the training procedure for a sample run of 1460 epochs. We can see that there are some spikes in the generator loss and the discriminator manages to outperform the generator for a brief interval between epochs 400 and 600. However, this is not enough to prevent our generator from learning to create decent fake images in the long run. In Figure 4, we can see a sample of 10 images that were generated at random from the generator of epoch 1460. One drawback is that we get the same image twice. Most of the images resemble the shape of a Pokemon despite the fact that some of them may still be considered to be blobs.



(a) Training accuracy of discriminator (b) Training loss of discriminator (c) Training loss of generator

Figure 3: Training accuracy and loss of discriminator and training loss of generator for a sample training interval of 1460 epochs.

Figure 4: Sample of 10 generated images from the generator of epoch 1460.

4 WGAN

4.1 Overview

We also explored WGAN [1] on our datasets, an algorithm expected to have better learning stability than traditional GANs. Traditional GANs have the problem that the discriminator tends to saturate. The loss of the generator, in some cases, can be unstable or stop decreasing due to vanishing gradients. WGAN was proposed to achieve smoother gradients using Earth-Mover (EM) distance (Wasserstein Distance) as its cost function. EM distance is the minimum cost of transporting mass when converting two data distributions. Applying the Kantorovich-Rubinstein duality, we get EM distance by

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \frac{1}{K} \sup_{||f||_L \leq K} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)] \tag{5}$$

where $\mathbb{P}_r, \mathbb{P}_\theta$ are data distributions, the supremum is over all the K-Lipschitz functions $f : X \to R$

$$|f(x_1) - f(x_2)| \leq K||x_1 - x_2|| \tag{6}$$

$K = 1$ for 1-Lipschitz functions.

WGAN achieves the Lipschitz constraint $||f||_L \leq K$ by simply using weight clipping. A constant c is introduced so that $w_i \in [-c,\ c]$ for all parameters w_i. Weight clipping is applied after the original gradient descent step of the optimizer. Therefore, the algorithm is capable of achieving better performance in terms of stability and mode collapse comparing to traditional GANs. On the other hand, the performance of this algorithm is very sensitive to the setting of hyperparameter c [4].

4.2 Discriminator and Generator

We trained WGAN with network architectures and parameter settings similar to our DCGAN. The weight clipping constant c is set to the suggested 0.01 and the discriminator is trained 5 times per each generator training in an epoch as mentioned in [1].

In the discriminator, the number of kernels is set to 64, 128, 256, 512 for each convolutional layer. We also used He's initializer [5] for the fed dense layer, which selects samples from the truncated normal distribution with standard deviation $\sqrt{2/fan}$ (fan is the number of input units inside the weight tensor) centering on 0.

In our generator, the input vector is fed to a reshaped dense layer with dimension 4x4x512 instead. We also added an extra convolutional layer with the same kernel size, stride, padding, and with batch normalization. The number of kernels is set to 256, 128, 64, 32 for each layer.

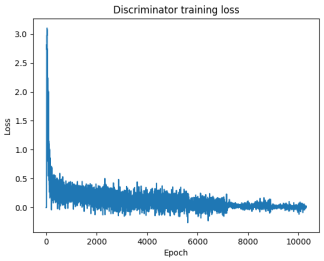
4.3 Results

4.3.1 Pokemon Generation One dataset

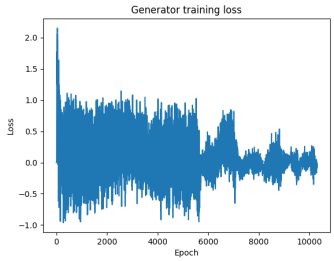
We chose a specific class from this dataset and trained our WGAN architecture with weight clipping. Particularly, we chose Pikachu class images to train our WGAN. The generated images are shown in Figure 5a. We can see that the generated image looks like a Pikachu but the image is not clear. They are of poor quality compared to the ones generated by DCGAN. The reason is that some of the images in the dataset are not related to Pikachu as shown in Figure 5b and also WGAN takes relatively very large number of epochs to generate high quality images. WGAN with gradient penalty instead of weight clipping can provide a slight improvement in the quality for the same number of epochs we trained but will not be as good as the images generated in Figure 2.

(a) Pikachu Generated image

(b) Incorrect Pikachu image



(c) Training loss of discriminator



(d) Training loss of generator

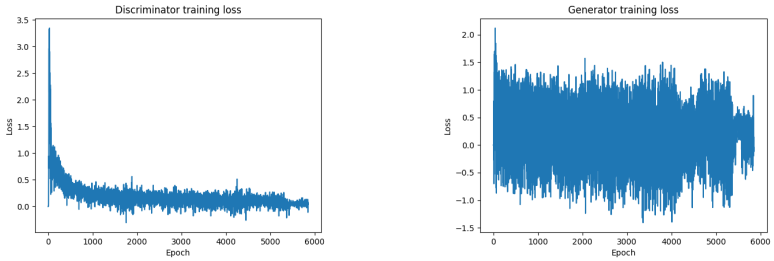
Figure 5: Generated Images and loss functions after training for 10300 epochs

The main advantage of WGAN is loss convergence. It is clear from Figure 5c that the discriminator loss is reduced with the increase in the number of epochs. Also, the image quality improved with the decrease in loss. Thus, there is a correlation between the loss and the image quality which confirms the findings in [1], unlike normal GAN. One can also see that the discriminator loss function changes smoothly and is quite stable compared to the loss in DCGAN.

4.3.2 Mgan dataset

First, we started training the same WGAN architecture on all the images from the Mgan dataset. Since it was taking really long for each training epoch to finish, we trained on randomly chosen 200 images from the dataset. The generated images after 10k epochs are shown in the Figure 6. The quality of the generated images improved a lot compared to the images in Figure 5a, but it is still worse than the images generated in Figure 4. To get the similar quality obtained in DCGAN, the training of WGAN has to be extended for a large number of epochs.

Figure 6: Sample of 6 generated images from the WGAN generator of epoch 5850.



(a) Discriminator loss

(b) Generator loss

Figure 7: Discriminator and Generator loss WGAN-Mgan dataset

We can clearly see from Figure 7 that the discriminator loss is reduced with the increase in number of epochs whereas the image quality improved. So there is a correlation between the loss and the image quality in this case as well, thereby we can conclude that the training process of WGAN is quite stable in general compared to DCGAN.

5 Conclusion and Discussion

In this paper, we show two models to generate images using GANs: DGAN and WGAN. Overall, we think that the rather disappointing results can be attributed to mainly two factors. First, the quality of the dataset plays an important role. When training with the second clean dataset, the results were much better compared to the first dataset. This in turn shows how important the choice of a good dataset is in Deep Learning. Secondly, GANs are inherently hard to train. Besides the fact that they take a long time to train, it was very difficult to have a stable training. Very often, the training saturated even after a couple of hundreds of epochs because the discriminator was too successful, the gradient of the generator vanished and only noise was produced.

Regarding WGAN, there is also an algorithm Wasserstein GAN with gradient penalty (WGAN-GP) [4], which improves the WGAN by applying gradient penalty instead of weight clipping to achieve 1-Lipschitz constraint. The main advantage to make Lipschitz constraint close to 1 by gradient penalty in WGAN-GP is that weight clipping introduces optimization problems when the weights are often clipped into the maximum or minimum. Therefore, the discriminator of WGAN is considered to be overly simplified. In our experiment, however, the training of the WGAN-GP was computationally too expensive as it takes time to calculate the gradient penalty.

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[4] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5767–5777, 2017.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[6] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 15(1):1929–1958, 2015.

[7] Y. Sher. Monster gans: create monsters for your game, 2017. https://medium.com/@yvanscher/using-gans-to-create-monsters-for-your-game-c1a3ece2f0a0.