Generating New Pokemon with the Wasserstein Generative Adversial Network

Eileen Wang

I. Introduction

Generative adversial networks (GANs) are a type of artificial intelligence algorithm where two models are trained simultaneously. The algorithm usually involves two networks, the generator and the discriminator competing against each other in a zero-sum game framework. The main purpose of GANs is to learn the distribution of the training data in order to create new features and generate high-quality images. This report specifically focuses on implementing a type of GAN called the Wasserstein Generative Adversial Network (WGAN) based on the Wasserstein distance with the aim of generating new Pokemon from a training dataset of 919 Pokemon images. The implementation was done in python using the Tensorflow package. More specifically, the discriminator was implemented using a 5 layer convolutional network while the generator was built using a 6 layer deconvolution network with feature map dimensions $8 \times 8 \times 512$, $8 \times 8 \times 256$, $16 \times 16 \times 128$, $32 \times 32 \times 64$, $64 \times 64 \times 32$ and $128 \times 128 \times 3$. Both discriminator and generator had a learning rate of 0.0002. The model was then trained for 1100 epochs on Google Colab's GPU. Some challenges faced throughout the project was designing the architecture of the network, training the network and ensuring that problems such as mode collapse were mitigated, and also the lack of data which we addressed through employing data augmentation techniques.

II. Introduction to GANs

Generative adversial networks (GANs) were developed in 2014 by Ian Goodfellow, a machine learning researcher at Google Brain. Unlike traditional convolutional networks, GANs involve two entities or adverseries competing against each other so that the network can learn to generate from a training distribution. The two networks are called the generator and discriminator. The goal of the generator is to create realistic-looking images while the discriminator's job is to determine which of the generated images are fake. In order to generate the most realistic images, it is necessary to have a good generator and discriminator. If the generator is not good enough, it will not be able to trick the discriminator into classifying the image as real, resulting in a model that will never converge. Similarly, if the model has a bad discriminator, then images which are fake will be classified as authentic, thus resulting in a model that never produces the desired output. For this reason, it is often difficult to train and stabilise a GAN as it is required to train two networks from a single backpropagation.

Generative vs. Discriminative Algorithms

Before discussing how GANs work in detail, it is important to understand how discriminative and generative algorithms work. Disiminative algorithms take in some features, x and attempt to predict its label, y. Expressed mathematically, they try to predict $p(y|x)$ which is the probability of a label y given the features x. That is, discriminative algorithms are primarily concerned with mapping given features to certain labels. Conversely, generative algorithms behave in the opposite way and attempt to predict features given a specific label i.e. the probability of x given y. In general, discriminative algorithms try to learn the boundary between classes while generative models attempt to model the distribution of individual classes (*Kanaujia, 2010*).

How GANs Work

In GANs, the discriminator network is built as a traditional convolutional network that aims to categorise inputted images into certain labels. In contrast, the generator is constructed as a deconvolutional network (also known as a transposed convolutional network) which is the inverse of a standard convolutional network starting at the fully connected layer. While a normal convolutional network downsamples an inputted image to produce a probability, a deconvolutional network does the opposite by taking random data and upsampling it

to create an image (*Gauthier, 2015*). *Figure 1* and *Figure 2* highlight the differences between a standard convolutional neural network and a deconvolutional network.

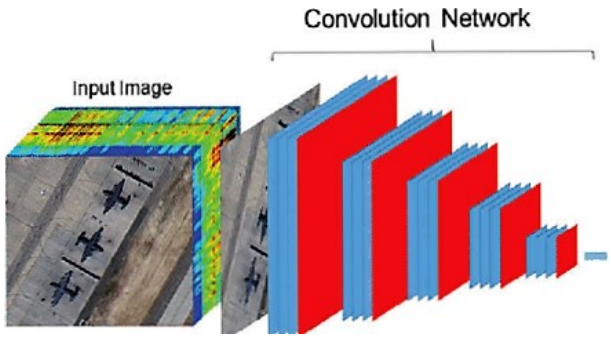


Fig. 1. Architecture of a convolutional neural network showing how an inputted image is downsampled.

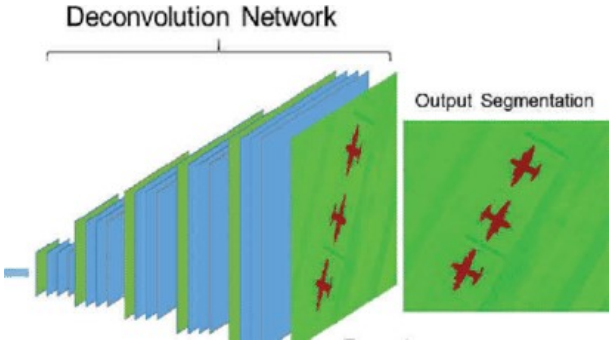


Fig. 2. Architecture of a deconvolutional neural network showing how an inputted image is upsampled.

The generator first works by accepting random noise and attempts to create an image from this random data. The image that is generated is then passed into the discriminator with a number of real images extracted from the actual dataset. The discriminator then uses these fake and real images to calculate a probability between zero and one, representing how authentic the image is. Zero probability means that the discriminator has predicted the image as fake while a probability of one means that the discriminator predicted the image as real. *Figure 3* below displays a graphical representation of this process.

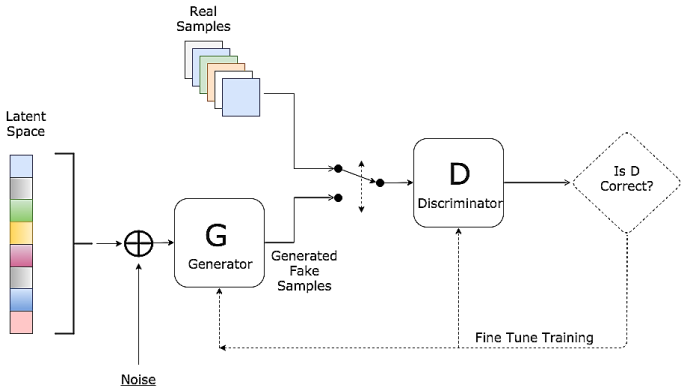


Fig. 3. Diagram of how a typical GAN works.

As shown in the diagram, the discriminator and generator are consistently providing feedback to each other so that each network can fine-tune their own respective parameters. While the generator is learning to pass on false images, the discriminator is learning to detect them.

GAN Loss Function

Let us now focus on the loss functions of the generator and discriminator.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

Fig. 4. Loss function for GAN discriminator.

Figure 4 shows the loss function for the discriminator, where the delta symbol represents the gradient, the subscript θ_d means that we only apply the gradient to the discriminator, m is the number of samples (or batch size), and D and G denote the discriminator and generator respectively. The first term denotes the log probability of the real data, x. We want to optimise the discriminator such that it maximises the log probability of predicting a 1 for x since x are the authentic images. The second term is the log probability of the generated data, z and we want to optimise the discriminator such that it rates these poorly since z are the fake images. More specifically, for each sample data, x_i and z_i, we take the difference of the first and second term, sum all the differences up for each $i \in [1, m]$ and divide by m to get the average difference over all samples. Overall, we desire to maximise the equation from *Figure 4* through gradient ascent since we want the first term to be high and the second term to be

low. Note that the reason for using the log of the probabilities is because such a method results in a more well-scaled gradient (*Eghbal-zadeh and Widmer, 2017*).

Additionally, let us examine the loss of the generator which is shown below:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} -\log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

Fig. 5. Loss function for GAN generator.

Here, the subscript θ_g means that we apply the gradient only to the generator this time. Unlike the discriminator's loss function, *Figure 5* clearly shows that the generator is trying to minimise the log probability of the discriminator being correct. Thus, this becomes a minimisation or gradient descent problem.

III. Wasserstein GAN

There are several types of GANs that one can consider. In this report, we will specifically focus on implementing the Wasserstein GAN (WGAN) which uses a loss function that represents the distance between the distribution of the data and the generated data, calculated using the Wasserstein distance. The Wasserstein distance is the cheapest cost in converting a data distribution q to a different data distribution p (*Villani, 2008*). Intuitively, suppose that each distribution can be viewed as a pile of 'dirt'. The Wasserstein distance is the minimum cost of turning one pile of dirt into another where the cost would be calculated by the amount of dirt required to be moved, multiplied by the mean distance to be moved. For this reason, the Wasserstein distance is also known as the 'Earth Mover's Distance'. A simplified calculation of the Wasserstein distance based on Kantorovich-Rubinstein duality (*Basso, 2015*) is shown below:

$$W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

Fig. 6. Wasserstein distance.

where *Pr* is the real data distribution, *Pg* is the generated data distribution and finally, f represents the 1-Lipschitz function which is defined to have the constraint:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|.$$

This Lipschitz function is what the discriminator in a WGAN learns during training. In other words, the discriminator in a WGAN is designed to learn some weights that will find a good Lipschitz continuous function so that we can compute the Wasserstein distance. As the loss function in *Figure 6* decreases, the Wasserstein distance diminishes, meaning that the outputs produced by the generator grows closer to the real data distribution. The only remaining problem is that we must ensure that the continuity constraint defined above is maintained during training. To enforce this, WGANs apply a simple clipping method to the weight values in f after every gradient update so that the discriminator weights always stay within a certain range (*Arjovsky et al. 2017*).

Note that implementing the discriminator in WGANs is similar to implementing the discriminator in the standard original GAN. The only difference is that the outputs are not inserted into the sigmoid function. Thus, rather than the outputs being probabilities, they will be scalar scores used to determine how good the input images are (not whether the images are fake or real). For this reason, the discriminator in WGANs are often called *"critics"* since they are not being trained to classify (*Arjovsky et al. 2017*).

Why WGANs are better than GANs

Traditional GANs perform poorly when the two distributions are located in separate union spaces without overlaps. Note that when two distributions, p and q are the same, their divergence is zero. However, as the mean of q increases away from p, the divergence increases and the gradient of the divergency diminishes to zero, resulting in a generator that is unable to learn (*Hui, 2018*).

In order to solve this problem, WGANs were proposed as an alternative. To put it simply, this is because Wasserstein distance has the capability of providing a meaningful and smooth representation of the in-between distance even when two distributions are far apart. This is because the Wasserstein distance has two important properties: firstly, the function is continuous everywhere and secondly, the function is almost everywhere differentiable i.e. the gradient of

the function is almost everywhere (*Arjovsky et al. 2017*). Such a function means that we can strongly train the discriminator and provide useful gradients to the generator before updating it whereas in original GANs, pre-training the discriminator was not possible since this would result in vanishing gradients.

Furthermore, WGANs are also robust to mode collapse, a problem whereby generators produce samples with extremely low diversity (*Thanh-Tung et al. 2018*). This usually occurs when data distributions are multimodal. Consider for example the MNIST dataset (a large database for handwritten digits) which contains 10 major modes since there are 10 digits from '0' to '9'. It is possible that the generator can learn to fool the discriminator every time by only learning to generate the same single digit. This issue where only a few 'modes' are generated was an extremely common issue that was present in original GANs.

WGAN Loss Function

Finally, we finish off this section by discussing the discriminator and generator loss functions for WGANs. The loss functions for the discriminator and generator are shown in *Figure 7* below where the first function corresponds to the discriminator's loss while the second corresponds to the generator.

$$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f\left(x^{(i)}\right) - f\left(G\left(z^{(i)}\right)\right) \right]$$

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -f\left(G\left(z^{(i)}\right)\right)$$

Fig. 7. Discriminator and Generator WGAN loss functions.

As shown, the loss functions are quite similar to the loss functions of the original GAN as presented in *Figure 4* and *Figure 5*. The only difference is that instead of using the log probabilities from the discriminator, we use the 1-Lipschitz function since the discriminator output is now a scalar score instead of a probability. More specifically, *Figure 7* says that we want to optimise the discriminator such that it maximises the score when rating the real data, *x* and minimises the score when rating the fake data, *z*. Conversely, the generator aims to achieve the opposite. Analogous to original GANs, the discriminator is trying to maximise its gradient whereas the generator is trying to minimise its own one.

IV. Method

The implementation of our WGAN was done in Python using Tensorflow, an open-source software library for building and training deep learning models. In this section, we will describe the implementation of the discriminator, generator and our 'train' function in detail.

Discriminator Function

The discriminator function is a convolutional network which first takes in a number of 128×128 RGB images. The first layer of the network is a convolutional layer which uses a 5×5 kernel, initialised from a truncated normal distribution. This is the recommended initialiser for neural network weights and filters as this initialiser ensures that values more than two standard deviations from the mean are discarded and re-drawn. Furthermore, a step-size of 2 was used.

The outputs of the convolutional layer were then normalised using batch normalisation. This technique is used to adjust and scale the activations to ensure zero mean and unit variance. More specifically, normalisation is first done by subtracting the batch mean and batch standard deviation from each input, *x*. After normalising *x*, the inputs are then inserted into a linear function with hyperparameters γ and β. These parameters are learnable during training, thus giving the ability for the network to either keep the mean and variance as zero and one respectively or learn any other distribution that might be better (see *Figure 8* for more details on batch normalisation).

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Fig. 8. Batch Normalisation.

The main benefits of the normalisation process is so that we can scale input features that are extremely different. In turn, this reduces oscillations during gradient descent so that the network can converge faster to the minimum point (*Ioffe and Szegedy, 2015*).

Finally, the normalised outputs were inputted into the leaky ReLU (rectified linear unit) function. Given an input x, the leaky ReLU is an activation function that produces $\alpha \times x$ for negative inputs and x itself when values are positive (see *Figure 9*). The α chosen in our implementation was 0.2.

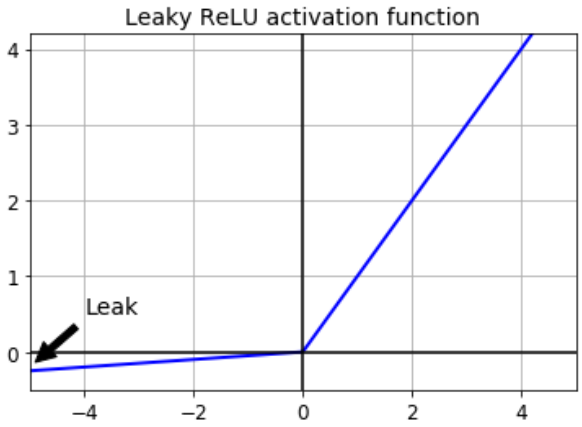


Fig. 9. Leaky ReLU function.

ReLU functions are alternatives to traditional activation functions like logistic sigmoid and hyperbolic tangent where the derivatives of such functions approach zero as the absolute value of x becomes large. This results in the 'vanishing gradient problem', whereby the gradient in the loss function is extremely small, thus slowing down the training process or even halting the process entirely. Conversely, ReLU functions have the benefit that the gradient has a constant value (which is different depending on whether x is positive or negative), resulting in faster learning and no vanishing gradient (*Agarap, 2018*). A further advantage of ReLU functions in general is that it is easy to compute their derivatives, whereas functions like sigmoids need to perform excessive exponential operations.

The result from the first activation is a feature map with dimensions $64 \times 64 \times 64$. The second to fourth layers were constructed in the same way, resulting in feature maps of dimensions $32 \times 32 \times 128$, $16 \times 16 \times 256$ and $8 \times 8 \times 512$. Finally, the last layer is the fully connected layer where the output from the fourth activation is flattened into one dimension. The result is then multiplied by the discriminator weights (initialised from the truncated normal distribution) and a constant bias was then added. *Figure 10* displays a summary of the architecture of our discriminator.

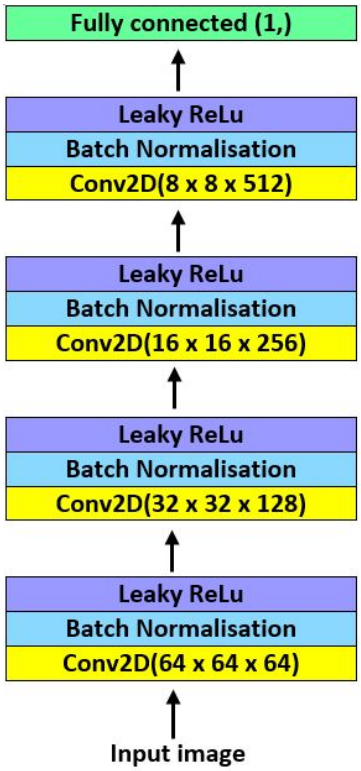


Fig. 10. Architecture of the Discriminator.

Generator Function

The generator function was implemented as a deconvolutional network, starting from the fully connected layer. This function first takes in a number of random samples (100 in our case), where each random sample is generated from a uniform distribution between -1 and 1. The generator then creates a weight variable (initialised from a truncated normal distribution) and a constant bias. The fully connected layer was then constructed by multiplying the generator weights and then adding on the bias. The shape of the output was $8 \times 8 \times 512$ by the number of random samples. These outputs were further normalised using batch normalisation and inserted into the ReLU function, creating the first activation.

After the fully connected layer, five more deconvolution layers were constructed with feature map dimensions of $8 \times 8 \times 256$, $16 \times 16 \times 128$, $32 \times 32 \times 64$, $64 \times 64 \times 32$

and $128 \times 128 \times 3$. More specifically, each layer was constructed by taking the previous activation and applying a 5×5 convolutional filter with a step size of 2, similar to the process mentioned above when describing the discriminator function. Furthermore, batch normalisation was then again applied to each of these outputs and the results were inserted into the ReLU function. This was done until the last layer was reached. Here, the outputs from the sixth (last) convolutional layer was inputted into the hyperbolic tangent activation function which is expressed as:

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

Fig. 11. Hyperbolic tangent activation function.

The final result was a number of RGB images with shape $128 \times 128 \times 3$. *Figure 12* below shows a summary of our generator's architecture.

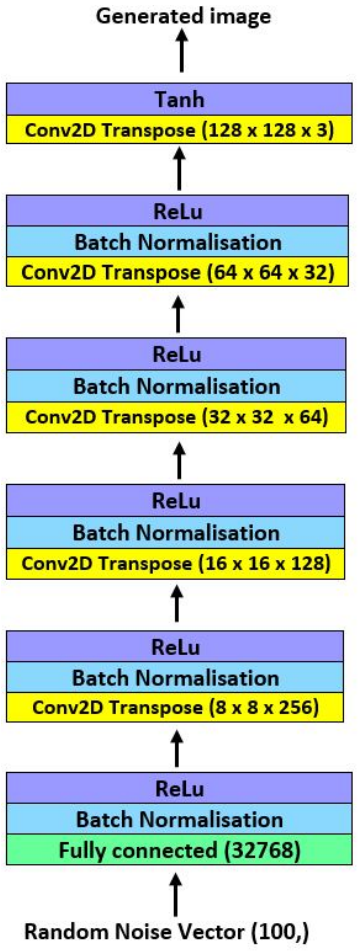


Fig. 12. Architecture of the Generator.

Train Function

The train function first creates the required placeholders necessary to construct our network. A 0.0002 learning rate for our discriminator and generator was used. Next, we calculate the batch number by dividing the total number of images by the batch size. In our case, this was 919 divided by 64 which rounded down to 14. The training loop was then constructed, consisting of three nested loops. For each batch number within each epoch, some random noise from a uniform distribution was generated. In our case, we generated 64 samples of random noise, each of size 100 using the numpy package.

Within the above mentioned loops, we then train the discriminator for 5 times. For each of the 5 times, the discriminator was fed 64 lots of random noise samples and a batch containing 64 images randomly selected from 919 Pokemon images. Before feeding in the data, the properties of the selected images were randomly altered and adjusted by utilising data augmentation techniques which are used to enhance the dataset and increase the amount of data available. Literature has shown that such techniques can improve the model's ability to generalise, prevent overfitting and correctly label new images that may have distortion (*Wang and Perez, 2017*). In total, we used 8 image augmentation techniques which are described as follows:

- Flipping the image horizontally
- Flipping the image vertically
- Adjusting image saturation
- Adjusting image brightness
- Adjusting image contrast
- Rotating the image
- Cropping and then padding the image
- Transposing the image

Each time the discriminator was trained, we also made sure to clip the discriminator weights for the reason described in Section 2 of this report.

The generator was then trained once by feeding random noise to the generator function. The model was also saved every 100 epochs and every 10 epochs, images created from the generator were saved so that we could see the generator's progress. Some of these results are shown in the next section of the report. In total, the algorithm ran for 1100 epochs, meaning that the training loop comprised of 77,000 iterations, calculated by $1100 \times 14 \times 5$. Also note

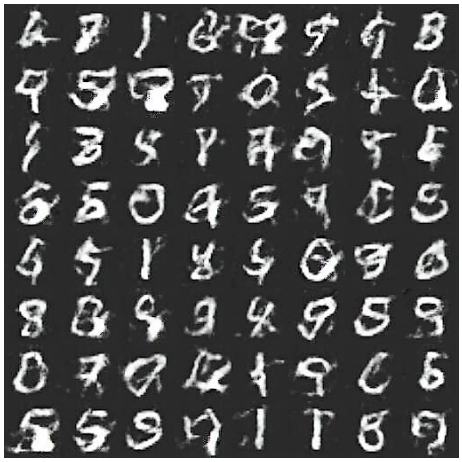
that the discriminator was trained 77,000 times while the generator was trained 15,400 times. That is, the discriminator was trained five times per generator update. According to *Mattyus* and *Urtasun* (2018), the discriminator should be trained more as it serves the purpose of providing useful gradients for the generator.

V. Experiment

In this section, we will discuss the results of our algorithm in detail.

Testing with MNIST Dataset

Before feeding our network the Pokemon images, the capability of our implemented discriminator and generator was first tested using images from the MNIST database. The MNIST dataset is the largest database for handwritten digits which is commonly used for training and testing deep learning models.The images below show the output of the generator after training for 50, 500, and 1000 epochs respectively.

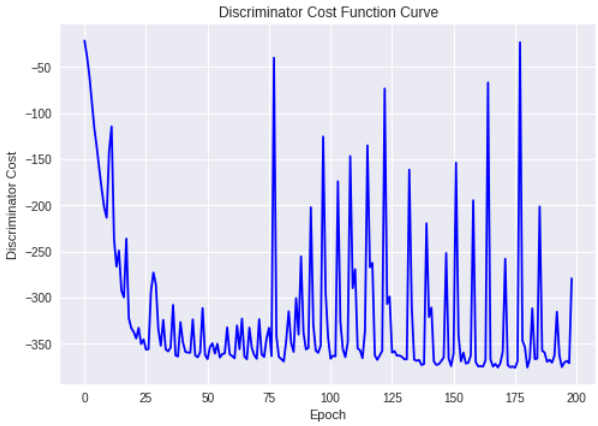


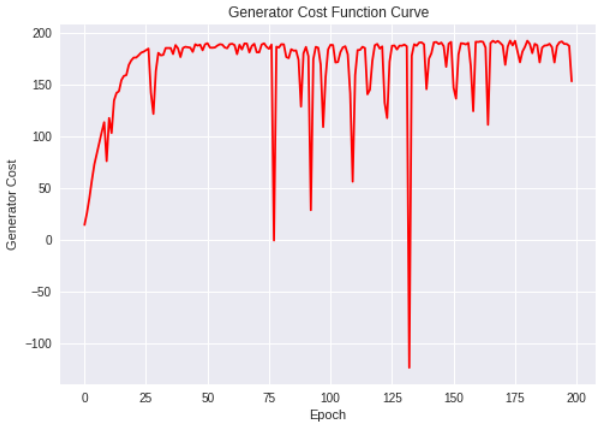
At the 50^{th} epoch, we can see that our generator is beginning to learn to form the shape of the digits. Further, at epoch 500, we begin to see that the generated digits are beginning to look more accurate. Some of the digits can be easily identified, although there is still some blur. Finally, at epoch 1000, we see a bit of improvement as the digits appear to look more distinct and defined.

Testing with Pokemon Dataset

We will now discuss the results from feeding our model images of Pokemon. The Pokemon images have 3 channles (RGB) with dimensions 128×128. A sample of 6 images are shown below:

As previously mentioned, the network was trained for a total of 1100 epochs. The first graph below shows the value of the discriminator's cost (loss) function over the first 200 epochs. Similarly, the second graph shows the generator's cost function in the first 200 epochs.

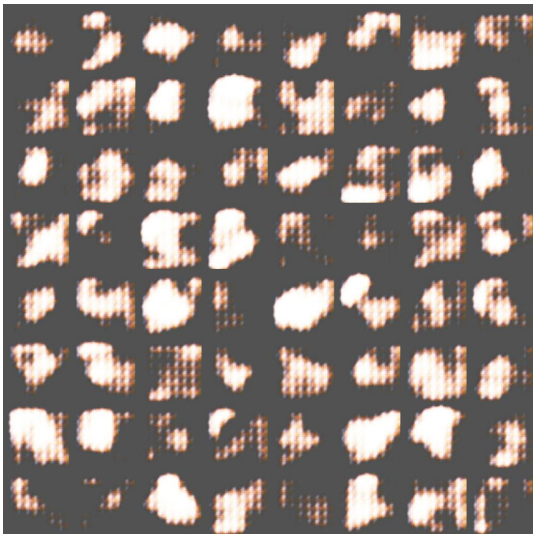




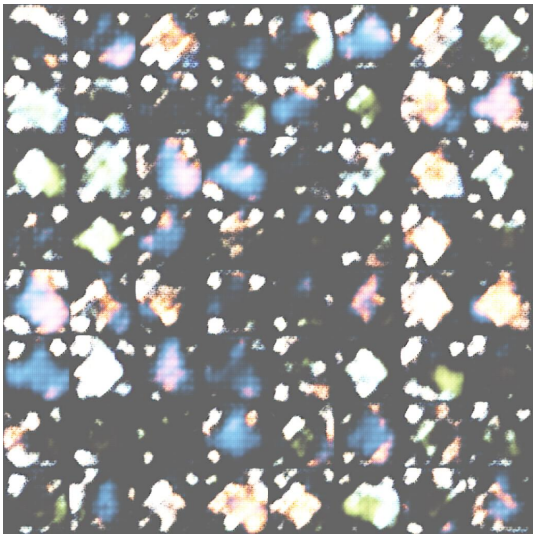
As expected, we see the largest change in the loss values of the generator after the first few epochs. This is due to the fact that the generator is receiving large gradients early in training because it is just beginning to learn how to generate realistic-looking data. Similarly, for the discriminator cost function, we see a steep gradient in the first 20 epochs. During the beginning of the training process, it is evident that the discriminator can easily distinguish between real and fake images. However, as the generator improves, the discriminator starts to make more errors and we start to see more spikes in the cost function.

Also note that it appears that the discriminator and generator losses don't seem to follow a distinct pattern. Unlike standard neural networks, it is often easy to see that the loss decreases when training iterations increase. However, this is not as obvious in the case for GANs due to the fact that with GANs, we have a discriminator and generator competing against each other. Thus, improvement in the generator results in higher loss for the discriminator (and vice versa). Nevertheless, with GANs, we should see that the discriminator and generator losses eventually converge to a certain point. After training our model for awhile, we found that the discriminator and generator for our network both approximately converged to the values -370 and 190 respectively.

Let us now examine the generated images. The picture below displays the results of the generator after the 10^{th} epoch. As depicted, at this point, we can see the generator beginning to learn how to form the shape of the Pokemon against a black background from random samples of noise.

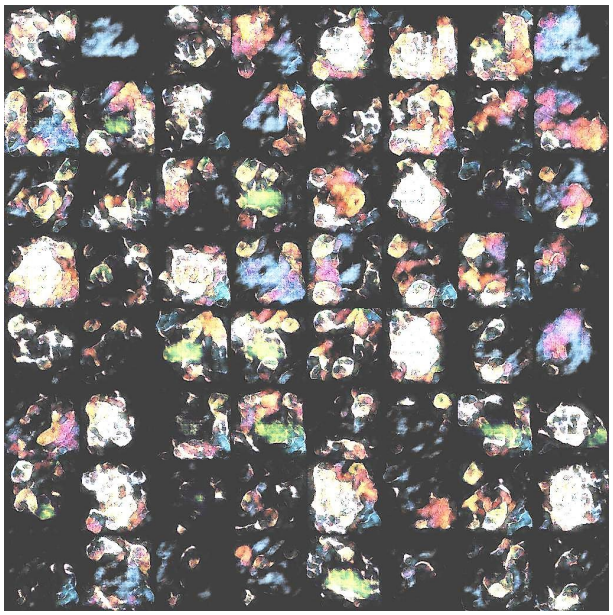


After epoch 70 (see image below), we already begin to see different colours in the generated images. Evidently, the generator has learnt how to add colours, particularly the colours of blue, orange and green.



At epoch 500, we see even more colours have formed

like red and purple. We also start to see a lot more details and different shapes as opposed to the images from epoch 30. Additionally, it appears that the generator has learnt how to add edges and outlines.



Finally, the picture below shows the generated images for epoch 1100.

Here we see more detailed images with smoother textures and more distinct edges and shapes. Additionally, we start to see that our generator is learning how to appropriately colour the image. Note that in epoch 500, we can see that most of the generated image involves a multitude of colours. However, in epoch 1100, some images are only made up of red, orange and yellow (warm colours) while we have some made up of just blue, purple and red. Thus, each image appears to follow a colour scheme, much like how a Pokemon is designed (i.e. water type Pokemon are usually blue while fire types are red). However, it is evident that these generated images still do not look much like Pokemon.

Overall, the results indicated that our WGAN performed well with the MNIST dataset but had trouble generating new Pokemon. We conclude that this is due to the fact that generating Pokemon is a much more complicated task as Pokemon tend to have more complex features in comparison to handwritten digits. We would also like to note that due to our limited amount of GPU usage, we were only able to train the model for 1100 epochs. If we had better access to GPU power, we could train our model for longer which may potentially result in improved images.

VI. Conclusion and Future Work

The aim of this report was to explore the concept behind generative adversial networks, a recently developed artificial intelligence algorithm that is used to learn the distribution of the training data and create new images. More specifically, we discussed a specific type of GAN called the Wasserstein generative adversial network. This newer and improved GAN uses the Wasserstein distance as the loss function and addresses the issues of mode collapse and vanishing gradients - common problems which were present in original GANs.

Our WGAN implementation consisted of five convolutional layers in the generator and six deconvolutional layers in the discriminator while a learning rate of 0.0002 was used. As highlighted in the Experiment section of this report, our network was first trained on the MNIST dataset using Google Colab's GPU for 1000 epochs. The results revealed that our network did quite well in generating new handwritten digits. Although the generated digits were a bit foggy, we could clearly identify the digit in each image. We then trained our WGAN using Pokemon images. This dataset contained 919 different Pokemon which we further enhanced through data augmentation techniques such as flipping, transposing and also by adjusting features such as brightness and saturation.

Our results showed that the algorithm was able to learn the colours and general shape of the Pokemon. However, we were unable to produce detailed images. This was most likely attributed to the complicated nature of the dataset as Pokemon tend to have a diverse range of colours, shapes and features which makes training the network difficult. Conversely, the MNIST dataset is much simplier to learn from because digits tend to have basic and similar shapes.

In order to improve the results, for future work, we can try to alter the architecture of our network and find the optimal parameters. For example, using larger kernels could cover more pixels in the previous layer, resulting in more information. We can also test the results from using different number of filters and convolutional layers in the discriminator and generator. Furthermore, we may also be able to achieve improved results by perhaps using a larger dataset, employing more data augmentation techniques or simply by training the algorithm for a longer period of time.

VII. References

- Agarap, A., 2018, Deep Learning using Rectified Linear Units (ReLU), Adamson University, viewed October 19, 2018, https://arxiv.org/pdf/1803.08375.pdf.

- Arjovsky, M., Chintala, S., and Bottou., L, 2017, Wasserstein GAN, Courant Institute of Mathematical Sciences, viewed October 10, 2018, https://arxiv.org/pdf/1701.07875.pdf.

- Basso, G., 2015, A Hitchhikers guide to Wasserstein distances, viewed October 18, 2018, http://n.ethz.ch/ gbasso/download/A%20Hitchhikers %20guide%20to%20Wasserstein/A%20Hitchhikers %20guide%20to%20Wasserstein.pdf.

- Eghbal-zadeh, H., and Widmer, G., 2017, Likelihood Estimation for Generative Adversarial Networks, Cornell University Library, viewed September 29, 2018, https://arxiv.org/pdf/1707.07530.pdf.

- Gauthier, J., 2015, Conditional generative adversarial nets for convolutional face generation, Symbolic Systems Program, Natural Language Processing Group, viewed October 16, 2018, https://www.foldl.me/uploads/2015/conditional-gans-face-generation/paper.pdf.

- Hui, J., 2018, GAN-Wasserstein GAN WGAN-GP, Medium, viewed October 2, 2018, https://medium.com/@jonathan_hui/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490.

- Ioffe, S., and Szegedy, C., 2015, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Google, viewed September 20, 2018, http://proceedings.mlr.press/v37/ioffe15.pdf.

- Kanaujia, A., 2010, Conditional Models for 3D Human Estimation, Rutgers University New Brunswick, viewed September 25, 2018, https://pdfs.semanticscholar.org/b67a/f2d6023d89 ebc8d7ecb4d6da19c5b25fb7f8.pdf

- Mattyus, G., and Urtasun R., 2018, Matching Adversarial Networks, Uber Advanced Technologies Group and University of Toronto, viewed October 16, 2018, http://openaccess.thecvf.com/content_cvpr_2018/ papers/Mattyus_Matching_Adversarial_Networks _CVPR_2018_paper.pdf.

- Thanh-Tung, H., Tran, T., and Venkatesh, S., 2018, On catastrophic forgetting and mode collapse in Generative Adversarial Networks, Workshop on Theoretical Foundation and Applications of Deep Generative Models, viewed October 16, 2018, https://arxiv.org/pdf/1807.04015.pdf.

- Villani, C., 2008, Optimal Transport: Old and New, Springer Science Business Media, viewed October 2, 2018, http://cedricvillani.org/wp-content/uploads/2012/08/preprint-1.pdf.

- Wang, j., and Perez, L., 2017, The Effectiveness of Data Augmentation in Image Classification using Deep Learning, viewed October 10, 2018, http://cs231n.stanford.edu/reports/2017/pdfs/300.pdf.