COMP3419 Assignment Option 3
Pokemon Generator

By Mirope Yuhao Hu
yuhu5454@uni.Sydney.edu.au
SID: 470139936

October 2018

1 Introduction

Since Turing introduced the theoretical computer science and artificial intelligence[1], the computer vision has been advancing through every fields which humans have discovered or invented before. In the early 2000s, interpretation and analysis of pictures including photos and human artworks have already reaches a bottleneck where there is hardly a game-changing impetus for current related industries to leap into a whole new magnitude of growth. Whereas, in recent years, the well-developed AI pushes the limit to a brand new stage. With the machine learning techniques, AI can now run in a far more sophisticated way and complete tasks such as pattern recognition or deep learning. In this report, I will demonstrate an example of implementing a deep learning technique, which is the Wasserstein Generative Adversarial Network (WGAN), to generate images similar to the Pokemon artworks.

2 Method

2.1 Data Preparation and Augmentation

The Pokemon image dataset I used for this report is retrieved from the website: `https://www.kaggle.com/kvpratama/pokemon-images-dataset [2]`.
The original data set consists of 819 pokemon images, all of them are in 256x256 .png format with transparent background. For better training efficiency and easier feature identification, I have converted all of the images into 128x128 .jpg format with black background as the dataset for my WGAN model to train with.

In order to get a better result, I have Implemented four types of data augmentation techniques to enrich the Pokemon image dataset.

(a) Original.png

(b) Processed.jpg

Figure 1: The original Pokemon image and the processed Pokemon image to train with

- (Random) Vertical flipping and horizontal flipping
- (Random) Cropping
- Rotating to certain degrees
- (Random) Brightness and contrast

(a) Vertical flip

(b) Bright & contrast

(c) Crop

(d) Rotate

Figure 2: Instances of 4 data augmentation techniques I implemented

2.1.1 Vertical & Horizontal Flipping

There are two approaches to flipping the images in the dataset: 1) Flip the original images, save as new and add them into the

database. 2) While the program is running and the images are decoded into tensors, use tensorflow functions to 'flip' the tensors (The function I mentioned here is: `tf.image.random_flip_up_down()`; `tf.image.random_flip_left_right()`) [3]. I used the first method so that I don't need to change the source code too frequently, which makes my experiments easier. Here I used python's **cv2** package, with the function `cv2.flip()`.

2.1.2 (Random) Cropping

Similar to the flipping images in the dataset, there are two approaches to cropping the images as well. I used the first method with a tensorflow function: `tf.random_crop()`[3].

2.1.3 Rotating

There are two approaches to rotating the images as well. I chose the first one with a function: `img.rotate()` in python package **Image** from PIL. And I produced 8 sets of images with clockwise and anti-clockwise of 5, 10, 15, 20 degrees.

2.1.4 Adjusting the Brightness and Contrast

With the first method, I used the tensorflow functions:
`tf.image.random_brightness()`; `tf.image.random_contrast()`[3]. And I produced 2 sets of images with random brightness and contrast.

Supplementary note:
Compared to the method of saving augmented new images directly into dataset, the method of directly augmenting tensors may be more compute intensive when dealing with static dataset in small scale. However, when the dataset is dynamic and enormous in scale (which is the case for industrial and commercial implementation), the second method requires far less digital storage and thus less costs, which indicates its practicality.

2.2 Neural Network Design

I only explain the design in English as simple as possible. All of the details can be discovered within the code delivered.

2.2.1 Discriminator

There are five sets of layers I have implemented for this discriminator:

- First set: A convolution layer with filter size 64, kernel size 5x5 and 2x2 stride; A batch normalization layer and a leaky relu activation layer.
- Second: A convolution layer with filter size 128, rest are the same as the first set of layers.

- Third: A convolution layer with filter size 256, rest are the same as the first set of layers.
- Fourth: A convolution layer with filter size 512, rest are the same as the first set of layers.
- Fifth: Only a readout layer, the details are in the code delivered.

2.2.2 Generator

The design of the generator is quite similar an inverted version of the discriminator, there are five transposed convolution layers embedded in six sets of layers I have implemented for this generator:

- First set: A read-in layer for the input (10 dimensional random vectors), reshaping the input into a 4x4x512 tensor, followed by a batch normalization layer and a leaky relu activation layer
- Second: A transposed convolution layer with filter size 256, kernel size 5x5 and 2x2 stride, a batch normalization layer and a leaky relu activation layer.
- Third: A transposed convolution layer with filter size 128, rest are the same.
- Fourth: A transposed convolution layer with filter size 64, rest are the same.
- Fifth: A transposed convolution layer with filter size 32, rest are the same.
- Sixth: A transposed convolution layer with filter size 3, and compute hyperbolic tangent of the output

2.3 Cost Functions

In 2014, the GAN optimizes the generator and discriminator by stochastic gradient descent[4]. While in 2017, the Wasserstein GAN optimizes them by RMSProp[5].
In mathematical form [6], the stochastic gradient descent is:

$$\theta_{i+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t; x^{(i:i+n)}; y^{(i;i+n)})$$

Where $J(\theta)$ is an objective function parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla^\theta J(\theta)$ w.r.t to parameters. And η is the learning rate, n is the size of the mini batch, $x^{(i;i+n)}$ is the set of training samples and $y^{(i;i+n)}$ is the set of labels.

On the other hand, RMSprop is an adaptive learning rate method proposed by Geoffrey Hinton[7]. The mathematical form of the RMSprop is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2] + \epsilon}} g_t$$

And the mean-square $E[g^2]$ is given by: [7]

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

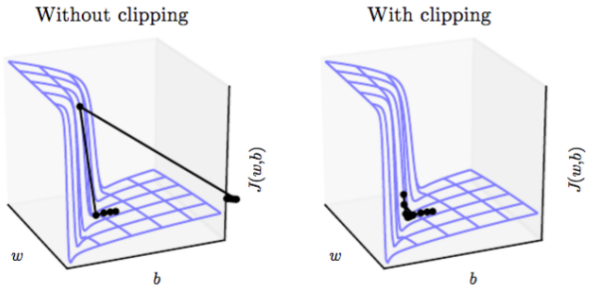
RMSprop divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.
Compared to RMSprop, SGD usually achieves to find a minimum, but it might table significantly longer. It is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. For the fact that GAN needs fast convergence and train a deep or complex neural network, RMSprop is a much better choice.

2.4 Weight/Gradient Clipping

Gradient Clipping is a kind of simple trick which helps a lot on certain occasions. Basically, we prevent gradients from blowing up by recalling them so that their norm is at most a particular value η. I.e., if $||g|| > \eta$, where g is the gradient, we set[8]

$$g \leftarrow \frac{\eta g}{||g||}$$

This biases the training procedure, since the resulting values won't actually be the gradient of the cost function. However, this bias can be worth it if it keeps things stable. The following figure shows an example with a cliff and a narrow valley; if you happen to land on the face of the cliff, you take a huge step which propels you outside the good region. With gradient clipping, you can stay within the valley



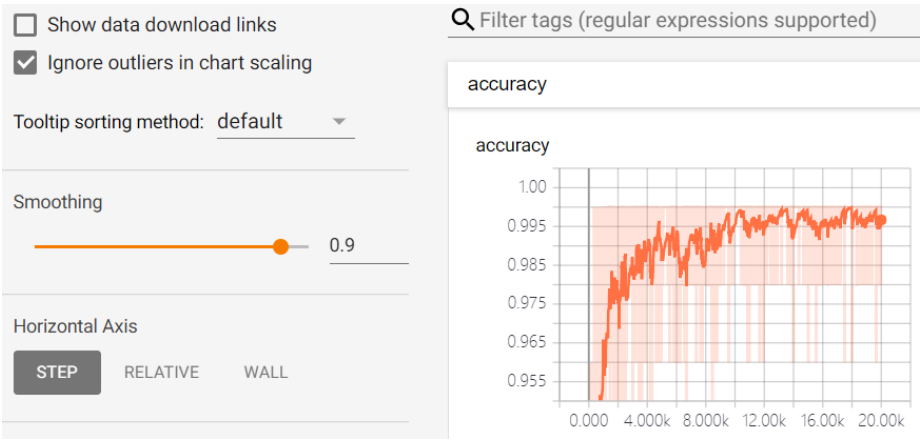
— Goodfellow et al., *Deep Learning*

3 Experiment

With respect to the methods I introduced above and program coded using Tensorflow 1.11, I ran several experiments with the GPU Nvidia GTX1080Ti. The statistics in line plot displayed below was sampled using tensorboard, and the output images are generated during the training process of the model.

3.1 Capability of the Discriminator

In order to test the capability of the discriminator, I modified and trained the GAN's discriminator part with MNIST data set. The modification I made is that I added a fully connected layer between the fourth set of layers and fifth layer, changed the readout layer as well in order to classify the numbers.
The image below is the statistic of the accuracy during the training process:



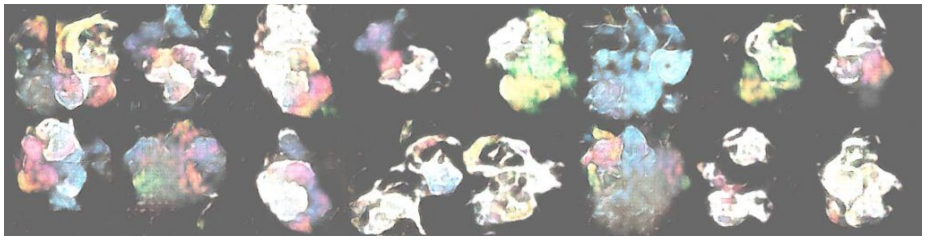
As you can see, after 1000 iterations, the accuracy has already been over 95%; after 10000 iterations, the accuracy is ranging between 99% to 100%. For the testing set, the final accuracy is **around 98.5% to 99%**, which can be consider to be good enough.

3.2 Generated Results of Different Data Augmentation Techniques

For better comparison, every experiment I conducted only ran 300 epoches with a 64 mini-batch. All generated pokemon images are produced in the 300-th epoch. All the smooth rates of the plots are set to 0.95.

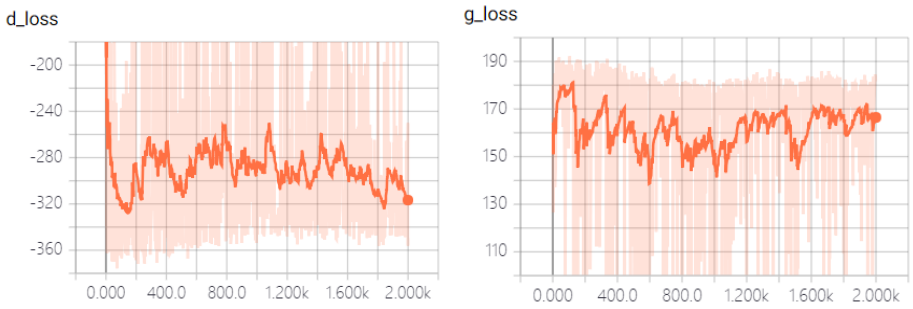
3.3 Without augmentation

In this round of experiment, I set the dataset as the original 819 processed images. The output I got is:



The loss plots of the optimization function of discriminator and generator are:

Apparently, the generated images are pretty unclear on subjects' edges, and



(a) Discriminator Loss

(b) Generator Loss

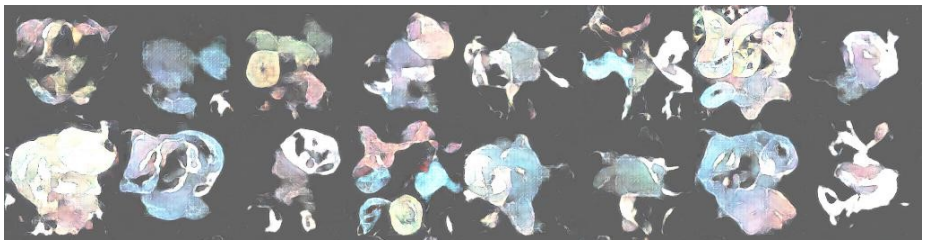
the coloring of the subjects are not quite distinguishable either. For the plots of the loss of generator and discriminators, although in fact that I had ran 2000 epoches, *d_loss* sees no more improvements after 300 epoches, and *g_loss* sees barely improvements during the whole process.

3.4 With different augmentation techniques

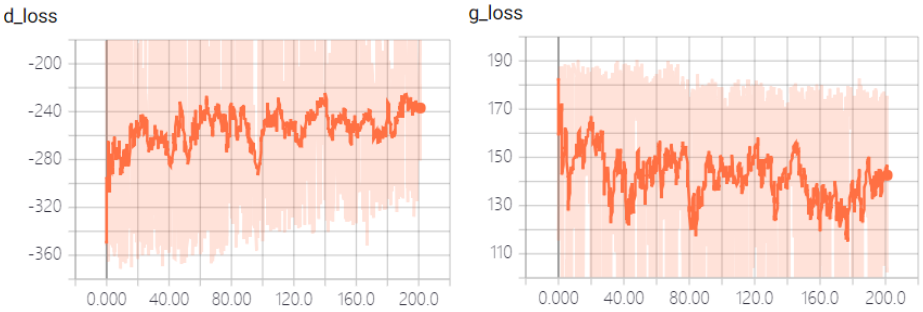
The following 5 experiments are conducted with different combinations of data augmentation techniques, and thus the contents and sizes of database are varying as well.

3.4.1 Combination 1

Database = Original 819 images + Vertical Flipped 819 images + Horizontally flipped 819 images = 2457 images



The loss plots of the optimization function of discriminator and generator are:



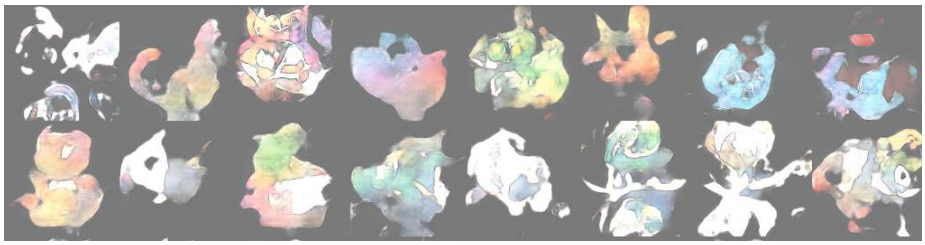
(a) Discriminator Loss

(b) Generator Loss

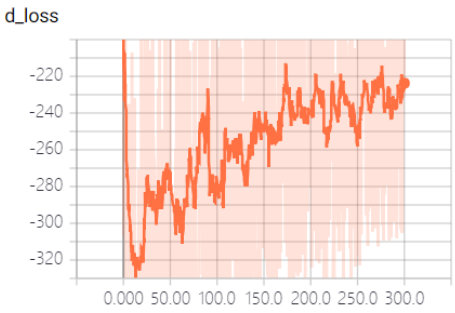
From the generated images, we can see a better clarification on subjects' edges, but the coloring is not quite distinguishable again. In the plots, we can see a relatively better improvement rate for both g_loss and d_loss compared to the experiment without augmentation, although it is slow-paced.

3.4.2 Combination 2

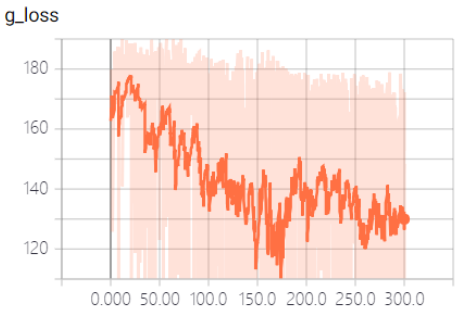
Database = Original 819 images + 1st set Random Cropped 819 images + 2st set Random Cropped 819 images = 2457 images



The loss plots of the optimization function of discriminator and generator are:



(a) Discriminator Loss

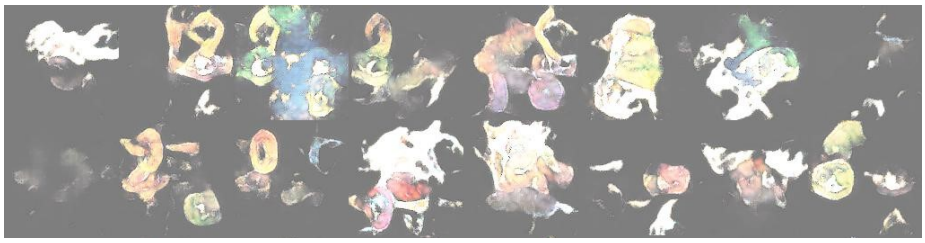


(b) Generator Loss

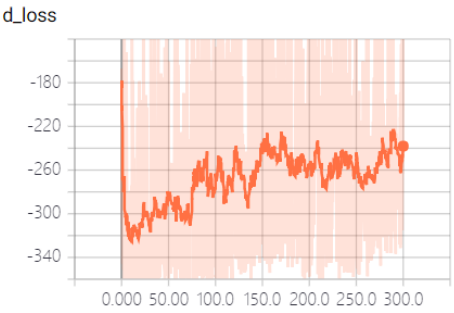
From the generated images this time, we can see a even better clarification on subjects' edges, with a more distinguishable color scheme as well. In the plots, we can see a really good improvement rate for both g_loss and d_loss, with fast pace and moderate fluctuations.

3.4.3 Combination 3

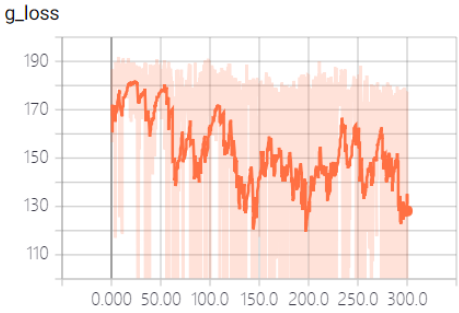
Database = Original 819 images + Clockwise rotated 20 degrees 819 images + Anticlockwise rotated 819 images = 2457 images



The loss plots of the optimization function of discriminator and generator are:



(a) Discriminator Loss



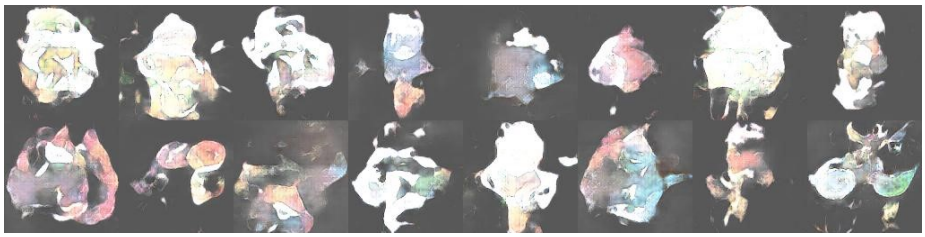
(b) Generator Loss

From the generated images this time, the result seems to be the worst so far,

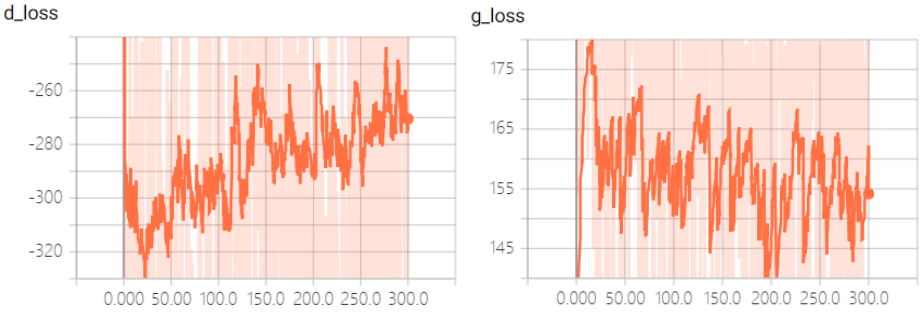
and there are even 2 "vanishing" subjects.Neither rests' edges nor color schemes is satisfying. In the plots, we can see there exits some improvement for both g_loss and d_loss, but slow-paced on improvements.

3.4.4 Combination 4

Database = Original 819 images + 1st set Random brightness and Contrast 819 images + 2st set random brightness and Contrast 819 images = 2457 images



The loss plots of the optimization function of discriminator and generator are:



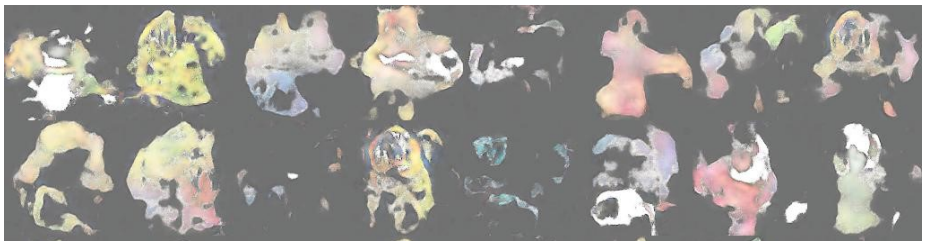
(a) Discriminator Loss

(b) Generator Loss

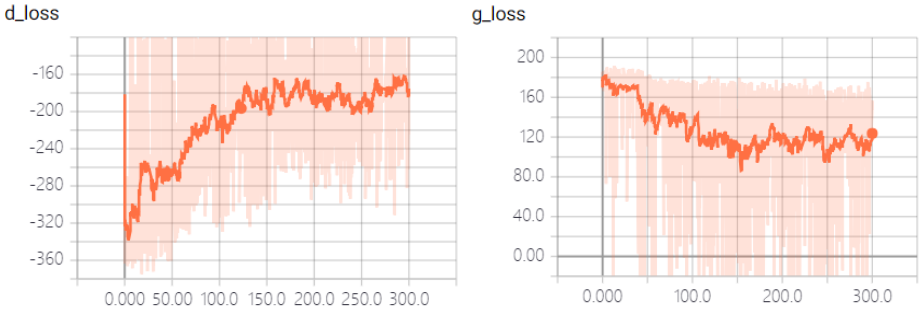
From the generated images, we can see a moderate clarification on subjects' edges, but the coloring scheme is not satisfying. In the plots, we can see some improvement rate for both g_loss and d_loss compared to the experiment without augmentation, but with big fluctuations.

3.4.5 Combination 5

Database = Original 819 images + Vertical/Horizontally flipped 1638 images + Clockwise rotated 20/15/10/5 degrees 3276 images + Anticlockwise rotated 20/15/10/5 3276 images = 9009 images



The loss plots of the optimization function of discriminator and generator are:



(a) Discriminator Loss

(b) Generator Loss

Due to a much bigger size of the dataset, the training time is longer than 9 hours. As a result, we see dedicated edges and relatively good color scheme, but there are still "vanishing" subjects with a even more amount of 4. In the plots, the improvements and fluctuations are really good in the first 150 steps, but for the 150 steps in the latter, the loss values seem to be floating around.

As a summary, the experiment of combination 2 seems to have the best result. That is to say, the random cropping might be the best data augmentation for this WGAN model. At the same time, the rotating technique seems to perform poorly in the experiments of combination 3 and 5. Notice that both combination 3 and 5 have "vanishing" subjects, the larger the dataset is, this troublesome feature occurs to be worse.

4 Conclusion

Actually, before starting off, I had little experience on machine learning. But now in the end of this report, I realize that I have already get my hand wet with a cutting-edge deep learning project. But to be honest, I am not satisfying of the results I get so far. From the experience of my experiments, I think if I established a larger dataset with the cropping data augmentation techniques, better tricks other than weight clipping, and even a improved version of the neural network

of the generator, the images generated by my GAN model might be closer and closer to the human art of Pokemon.
There are still lots of works to do, more ideas to be realized with a broader view, and more time and eagerness to be devoted into it.

References

[1] S. B. Cooper and J. van Leeuwen, "Alan turing: Mathematical machinist," *Alan Turing: His Work and Impact*, pp. 481–485, 2013.

[2] kvpratama, "Pokemon images dataset:dataset of 819 pokemon images." Retrieved from: https://www.kaggle.com/kvpratama/pokemon-images-dataset.

[3] TensorFlow™, "API r1.11 PYTHON."
Retrieved from: https://www.tensorflow.org/api_docs/python/tf.

[4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," 2014.

[5] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," 2017. arXiv:1701.07875v3.

[6] S. Ruder, "An overview of gradient descent optimization algorithms." Retrieved from: http://ruder.io/optimizing-gradient-descent/.

[7] G. Hinton, N. Srivastave, and K. Swersky, "Neural networks for machine learning: Lecture 6a - overview of mini-batch gradient descent,"
Retrieved from:
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[8] R. Grosse, "Lecture 15: Exploding and vanishing gradients,"
Retrieved from:
http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/
readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf.