Generative Adversarial Networks and Cycle-GAN

GHERMI Ridouane
ENSAE Paris
`ridouane.ghermi@ensae.fr`

1 Introduction

In this assignment, we'll implement and train several GANs. First, we give an introduction of what is a GAN. Then, we summarize two main improvements of GANs: DC-GAN and Wasserstein GAN. Finally, we consider a more advanced architecture, called Cycle-GAN, which combines two GANs in order to perform image translation between two domains (e.g. from natural images to painting in the style of Monet).

2 Generative Adversarial Network (GAN)

A GAN is a framework for estimating generative models via an adversarial training, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the case where G and D are defined by multi-layer perceptrons, the entire system can be trained by backpropagation.

The generator's distribution p_g is computed from a prior distribution p_z (often a multivariate Gaussian distribution) on input noise z and a mapping $G(z; \theta_g)$, where G is a MLP with parameters θ_g. We also define a second MLP $D(x; \theta_d)$ with parameters θ_d that outputs a single scalar: $D(x)$ represents the probability that x came from the data rather than p_g. Let note p_{data} the data distribution.

We train D as a classifier, to maximize the probability of assigning the correct label to both training examples x and samples generator's samples $G(z)$. We simultaneously train G to minimize $\log(1 - D(G(z)))$. In other words, D and G play the following two-player minimax game with value function $V(D, G)$:

$$min_G max_D V(D,G) = E_{x \sim p_{data}(x)}[log(D(x))] + E_{z \sim p_z(z)}[log(1 - D(G(z)))]$$

In practice, we must implement the game using an iterative, numerical approach. We alternate between k steps of optimizing D and one step of optimizing G. This results in D being maintained near its optimal solution, so long as G changes slowly enough. The procedure is formally presented in Algorithm 1.

In practice, we use the "$logD$" trick: rather than training G to minimize $log(1 - D(G(z)))$, we train G to maximize $log(D(G(z)))$. This results in stronger gradients early in the training.

In the original paper, experiments are performed on three datasets: MNIST (handwritten numbers), CIFAR-10 (natural images in low resolution) and the Toronto Face Database (face images). The method show promising results, being able to generate very plausible fake images. GAN training is controlled through the displayed of losses through epochs (discriminator and generator losses) and of fake images generated from fixed noise variables. The final model is mostly evaluated qualitatively, but can also be used as a feature extractor for downstream task (such as classification).

Algorithm 1 GAN Training

```
1: for it = $1, ..., n_{iter}$ do
2:     # Train the Discriminator (for k steps)
3:     for k steps do
4:         Sample $x_1, ..., x_m$ from $p_{data}$
5:         Sample $z_1, ..., z_m$ from $p_z$
6:         Generate fake images $G(z_1), ..., G(z_m)$
7:         Compute the Discriminator loss: $\frac{1}{m}\sum_{i=1}^{m} \log(D(x_i)) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(G(z_i)))$
8:         Compute gradients and update Discriminator weights (gradient ascent)
9:     end for
10:    # Train the Generator
11:    Sample $z_1, ..., z_m$ from $p_z$
12:    Generate fake images $G(z_1), ..., G(z_m)$
13:    Compute the Generator loss: $\frac{1}{m}\sum_{i=1}^{m} \log(D(G(z_i)))$
14:    Compute gradients and update Generator weights (gradient ascent)
15: end for
```

3 Deep Convolution GAN (DCGAN)

DCGAN is a specific type of GAN designed to process images. The main contribution of this paper is to use and adapt CNNs for both the discriminator and the generator.

The discriminator is a basic CNN for classification, with strided convolutions instead of pooling layers and leaky relu as activation functions for all layers.

The generator is a neural network with a sequence of transpose convolutional layers that progressively upsample the input noise sample to generate a fake image. The original DCGAN uses deconvolutional layers to expand the spatial dimension but they creates artifacts in the generated samples, therefore we use an upconvolutional layer that consists of an upsampling layer followed by a classic convolutional layer. All layers implement a relu acivation function, except for the last layer which uses tanh.

Both D and G include batch normalization and no fully-connected layers.

DCGAN proposes additional qualitative evaluation methods. The latent space (input noise of the generator) is extensively studied:

- we display a random sample of points in 2D thanks to dimensionality reduction algorithms such as t-SNE

- we sample two random points in the latent space, compute linear interpolations between these two points and generate the corresponding fake images to observe the smooth transition between samples

- we try to perform vector arithmetic for visual concepts

4 Wasserstein GAN (WGAN)

We can look at GAN from the Optimal Transport framework. Optimal Transport aims at computing distances between probability distributions. In our case, we want the generator's distribution p_g to approximate the data distribution p_{data}. Minimizing the optimal transport distance between these two distributions (e.g. by backpropagation) would solve the GAN problem.

In WGAN, the generator still maps from the noise distribution p_z to p_g, which aims at approximating p_{data}. However, the discriminator, now called the critic, is here to approximate the Wasserstein distance between the two probability distributions p_g and p_{data}, thus providing relevant gradients to train the generator.

In practice, we can show by the Kantorovich-Rubinstein duality that the Wasserstein distance is equal to

$$W(P, Q) = \sup_{||f||_L \leq 1} E_{x \sim P}[f(x)] - E_{x \sim Q}[f(x)]$$

where the sup is taken over all 1-Lipschitz functions f. Therefore, if we have a parameterized family of functions $(f_w)_w$ that are all K-Lipschitz, we could consider solving the problem

$$W(P, Q) = \max_{w \in W} E_{x \sim P}[f_w(x)] - E_{x \sim Q}[f_w(x)]$$

and if the sup is attained for some $w^* \in W$, this process would yield a calculation of an approximation of $W(P, Q)$ (up to a multiplicative constant). In our case, we want to approximate the distance between p_{data} and p_g, which can be done by

$$W(p_{data}, p_g) = \max_{w \in W} E_{x \sim p_{data}}[f_w(x)] - E_{z \sim p_z}[f_w(g_\theta(z))]$$

where g_θ is the generator (which maps p_z to p_g) and f_w is the critic which allows us to approximate the Wasserstein distance. Furthermore, we could consider differentiating $W(p_{data}, p_g)$ with respect to θ by backpropagation. It can be shown that this follows

$$\nabla_\theta W(p_{data}, p_g) = -E_{z \sim p_z}[\nabla_\theta f_{w^*}(g_\theta(z))]$$

where w^* corresponds to the maximum attained in the Wasserstein distance.

For the result to be correct, we need the functions $(f_w)_w$ to be all K-Lipschitz. However, this is a Neural Network with parameters w. Then, we use weight clipping to enforce these functions to be K-Lipschitz (even if it is not a very optimal way to do that).

The training is implemented as in Algorithm 2. We train the critic to maximize a loss corresponding to the expression of the Wasserstein distance and, after each update of the critic's weights, we clip its weights. Then, after several step of training the critic, we assume that the critic gives a good approximation of the Wasserstein distance between p_{data} and p_g (for the current θ). Then, we train the generator for only one step, using the gradient computed from the critic.

Algorithm 2 WGAN Training

```
 1: for $it = 1, ..., n_{iter}$ do
 2:     # Train the Critic (for $n_{critics}$ steps)
 3:     for $k = 1, ..., n_{critics}$ do
 4:         Sample $x_1, ..., x_m$ from $p_{data}$
 5:         Sample $z_1, ..., z_m$ from $p_z$
 6:         Generate fake images $G(z_1), ..., G(z_m)$
 7:         Compute Critic loss: $\frac{1}{m}\sum_{i=1}^{m} D(x_i) - \frac{1}{m}\sum_{i=1}^{m} D(G(z_i))$
 8:         Compute gradients and update the Critic weights (gradient ascent)
 9:         Clip Critic weights between $(-c, c)$
10:     end for
11:     # Train the Generator
12:     Sample $z_1, ..., z_m$ from $p_z$
13:     Generate fake images $G(z_1), ..., G(z_m)$
14:     Compute Generator loss: $-\frac{1}{m}\sum_{i=1}^{m} D(G(z_i))$
15:     Compute gradients and update the Generator weights (gradient descent)
16: end for
```

WGANs present several advantages over basic GANs. First, we can and should train the critic till optimality: the more we train the critic, the more reliable are the gradients. Therefore, there is far less collapse modes. Second, WGAN training provides a meaningful loss metric, which is the Wasserstein distance between p_{data} and p_g, and tells how close our model is to approximate the data distribution. Thus, we can monitor training just by looking the decreasing loss. Finally, the paper claims that stability is improved, as there is no more tradeoff between training the discriminator or the generator.

5 Cycle-GAN

Say we have a source image domain X and a target image domain Y (e.g. X is the space of natural images and Y the space of paintings from Monet). Converting an image $x \in X$ to $y \in Y$, by mimicking the characteristics of the target domain and applying them to the source image, is the task of image-to-image translation.

Cycle-GAN is a method for image-to-image translation, particularly interesting because it can be used with un-paired training images. This means that in order to train it to translate images from domain X to domain Y, we do not have to have exact correspondences between individual images in those domains.

Cycle-GAN is based on two GANs. The first one consists of a generator $G_{X \to Y}$ which maps from domain X to domain Y, and a discriminator which classifies between fake and real images from Y. The second one is the exact opposite. The specificity of these generators is that they map from an image to another image, as in auto-encoders, instead of mapping from input noise.

Thus, the training is very similar to training two GANs in parallel, using their respective loss functions. However, they could be an infinite way to translate from a domain to another and still satisfies the basic GAN constraints. In order to constrain the model, we add a cycle consistency loss: the idea is that when we translate an image from domain X to domain Y, and then translate the generated image back to domain X, the result should look like the original image that we started with. This is computed by a L1 loss between the original image and the doubly-translated image:

$$\lambda_{cycle} J_{cycle}^{X \to Y \to X} = \lambda_{cycle} \frac{1}{m} \sum_{i=1}^{m} ||x_i - G_{Y \to X}(G_{X \to Y}(x_i))||_1$$

where λ_{cycle} is a hyperparameter balancing the two loss terms. The loss for the $Y \to X \to Y$ cycle is analogous.

Algorithm 3 Cycle-GAN Training

```
1:  for $it = 1, ..., n_{iter}$ do
2:      Sample $x_1, ..., x_m$ from $p_X$
3:      Sample $y_1, ..., y_m$ from $p_Y$
4:      # Train the discriminators
5:      Compute the discriminator loss on real images: $\frac{1}{m}\sum_{i=1}^{m}(D_X(x_i)-1)^2 + \frac{1}{m}\sum_{j=1}^{m}(D_Y(y_j)-1)^2$
6:      Compute the discriminator loss on fake images: $\frac{1}{m}\sum_{j=1}^{m} D_X(G_{Y \to X}(y_i)^2 + \frac{1}{m}\sum_{i=1}^{m} D_Y(G_{X \to Y}(x_i)^2$
7:      Compute gradients and update the discriminators (gradient descent)
8:      # Train the generators
9:      Compute the generator $X \to Y$ loss: $\frac{1}{m}\sum_{i=1}^{m} D_Y(G_{X \to Y}(x_i) - 1)^2 + \lambda_{cycle} J_{cycle}^{X \to Y \to X}$
10:     Compute the generator $Y \to X$ loss: $\frac{1}{m}\sum_{j=1}^{m} D_X(G_{Y \to X}(x_j) - 1)^2 + \lambda_{cycle} J_{cycle}^{Y \to X \to Y}$
11:     Compute gradients and update the generators (gradient descent)
12: end for
```

6 Experiments & Results

6.1 Generate MNIST-like images with DCGAN

We implement a DCGAN to generate images which look like the MNIST dataset (handwritten numbers). Figure 1 shows the loss functions of the discriminator and the generator of the DCGAN through training. However, in DCGAN, these loss functions are not very relevant to monitor training.

In appendix, Figure 6 shows the generation of images throughout the training, from a fixed set of points in the latent space. We can see the improvements of the generator, which perform well at the end of the training.

Figure 2 displays images generated from the linear interpolation between two points in the latent space. We can see that a basic interpolation, here linear, in the latent space generate relevant images after mapping by the generator. It's visually very impressive.

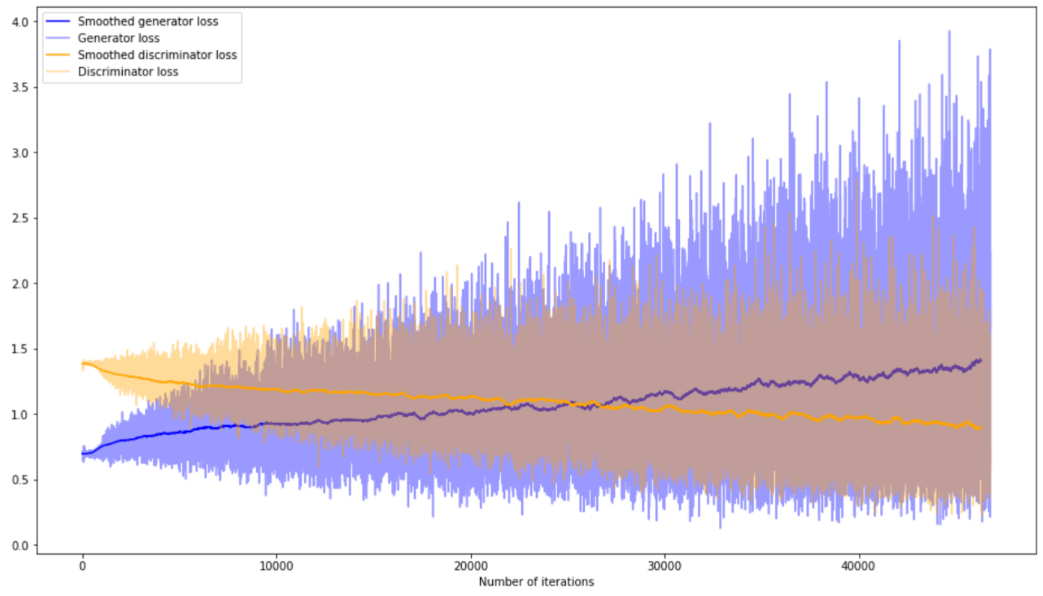


Figure 1: Loss functions of the discriminator and the generator of a GAN throughout training on MNIST.

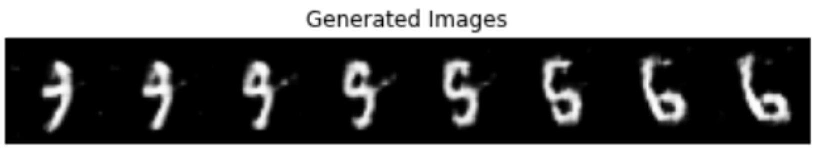


Figure 2: Images generated from the linear interpolation between two points in the latent space.

Finally, we could want to visualize the latent space to get a better understanding of it. Figure 3 shows a dimensionality reduction of the latent space in two dimensions (thanks to t-SNE). We also generate the corresponding images from the generator. We can see that some regions of the latent space corresponds to the generation of images of a specific number. We also note some interpolation between points, as before.

6.2 Image translation between MNIST and USPS with Cycle-GAN

We implement a Cycle-GAN to translate between two image domains: MNIST (handwritten numbers) and USPS (same but much blurrier images). Because Cycle-GAN consists of two discriminators and two generators, Figure 4 shows the loss functions of each of these networks. Generator loss functions include cycle-consistency loss.

Again, loss functions are not very relevant to monitor training. Thus, Figure 5 shows the translation of a fixed set of images throughout training. The first row are original MNIST images, the second row are translation from $G_{X \to Y}$. Same, the third row are original USPS images and the fourth are translation from $G_{Y \to X}$. Each of the six images correspond to an instant in training. We can see that the model improves its generation quality through epochs, reaching a good performance at the end of the training. The model is able to translate between the two domains, in both ways.

Removing the cycle-consistency loss ($\lambda_{cycle} = 0$), the generators collapse and generate images which are plausible for their respective discriminators but not related at all to the original images from the source domain.

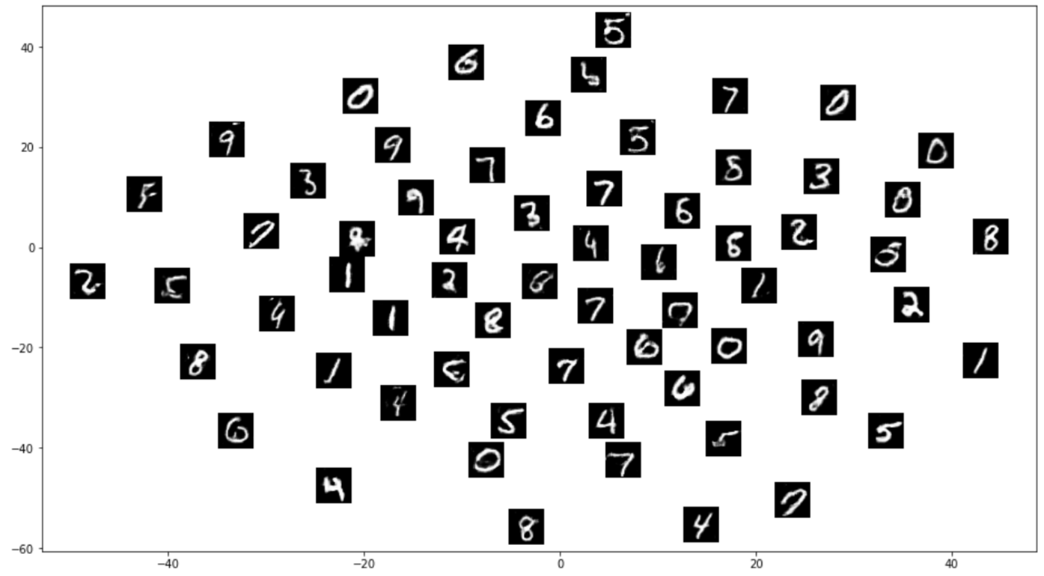


Figure 3: Dimensionality reduction (t-SNE) of a few random points in the latent space and their corresponding generated images.

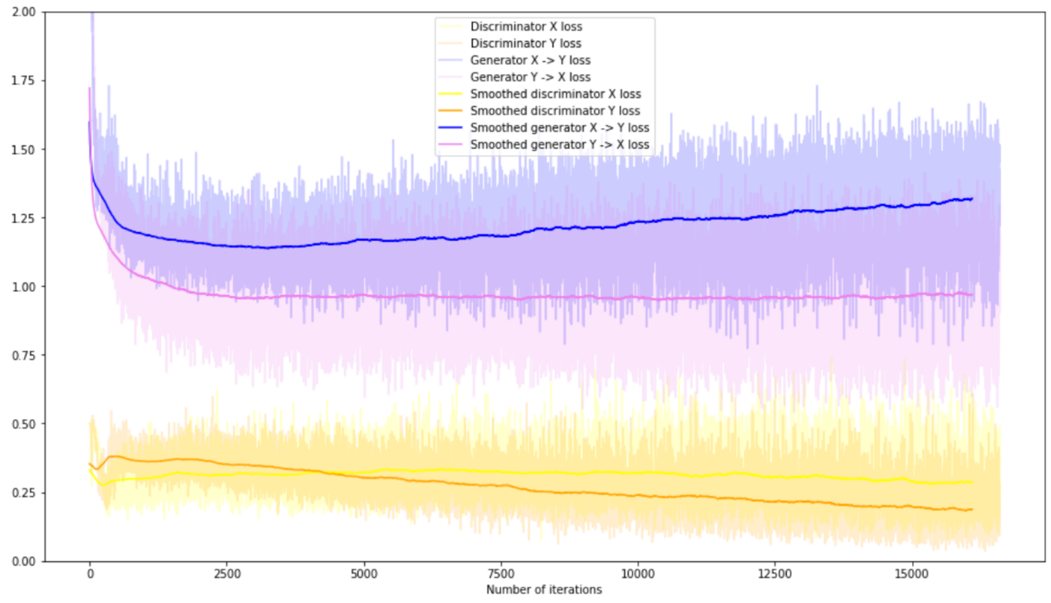


Figure 4: Loss functions of the two discriminators and the two generators of a Cycle-GAN throughout training on MNIST-to-USPS image translation.

7 Conclusion

To conclude, we summarized several key papers in the GAN community: DCGAN, WGAN, Cycle-GAN. We implemented each of these models, showing the promising results they give. In particular, Cycle-GAN is a powerful method, which is able to translate between two very different image domains using datasets of un-paired images. We showed that it works on a basic example of MNIST-to-USPS translation.

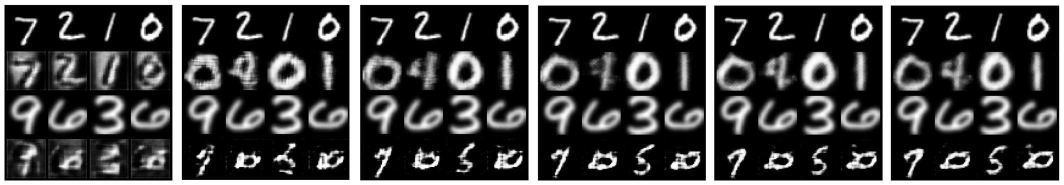


Figure 5: Image translation through training of a fixed set of images from MNIST and from USPS. Row 1: original images x_i from MNIST. Row 2: translation from MNIST to USPS $\hat{y_i} = G_{X \to Y}(x_i)$. Row 3: original images y_j from USPS. Row 4: translation from USPS to MNIST $\hat{x_j} = G_{Y \to X}(y_j)$.

Appendix

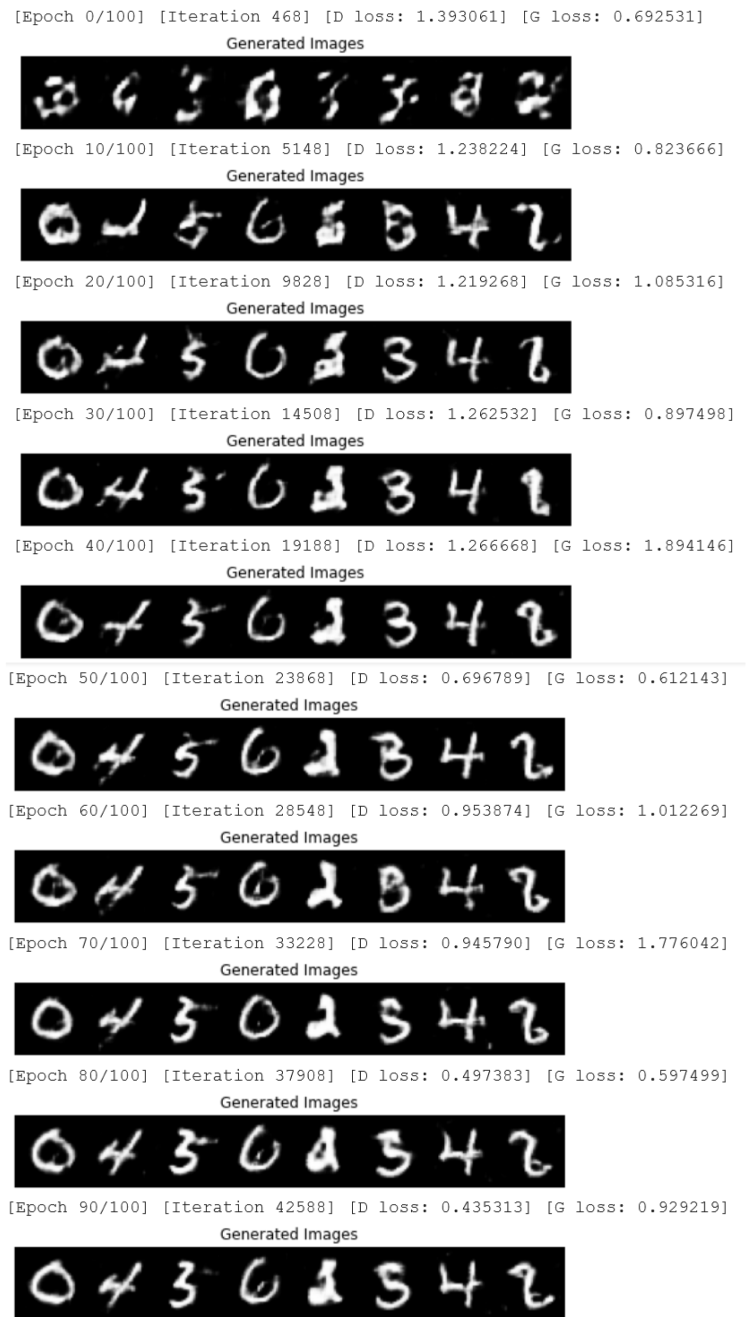
Bibliography

Goodfellow et al. (2014) Generative Adversarial Nets. **Algorithm: GAN**

Radford et al. (2015) Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. **Algorithm: DCGAN**

Arjovsky et al. (2017) Wasserstein GAN. **Algorithm: WGAN**

Zhu and Park (2017) Unpaired Image-to-Image Translation using Cycle-Consistent Generative Adversarial Networks. **Algorithm: Cycle-GAN**



```
[Epoch 0/100] [Iteration 468] [D loss: 1.393061] [G loss: 0.692531]
[Epoch 10/100] [Iteration 5148] [D loss: 1.238224] [G loss: 0.823666]
[Epoch 20/100] [Iteration 9828] [D loss: 1.219268] [G loss: 1.085316]
[Epoch 30/100] [Iteration 14508] [D loss: 1.262532] [G loss: 0.897498]
[Epoch 40/100] [Iteration 19188] [D loss: 1.266668] [G loss: 1.894146]
[Epoch 50/100] [Iteration 23868] [D loss: 0.696789] [G loss: 0.612143]
[Epoch 60/100] [Iteration 28548] [D loss: 0.953874] [G loss: 1.012269]
[Epoch 70/100] [Iteration 33228] [D loss: 0.945790] [G loss: 1.776042]
[Epoch 80/100] [Iteration 37908] [D loss: 0.497383] [G loss: 0.597499]
[Epoch 90/100] [Iteration 42588] [D loss: 0.435313] [G loss: 0.929219]
```

Figure 6: Image generation through training of a fixed set of points in the latent space.