Adversarial Networks for Image Generation

Sélim Ben Abdallah

Abstract

This study aims to understand how Generative Adversarial Networks (GANs) [2] and Deep Convolutional GANs (DCGANs) [7] work. In the first section, I study GANs and train a generative model on the MNIST dataset to produce images of handwritten digits. In the second section, I explore the contributions of DCGANs and how they improve upon the original GAN architecture. In the third and final section, I briefly introduce other variants of GANs and discuss possible directions for further work.

Note: I wrote this article myself and used ChatGPT-4o to help correct sentences and improve the clarity of certain sections.

1 Generative Adversarial Networks

In this section we will see how Generative Adversarial Networks (GANs) work, and how we can use them to generate realistic images with the MNIST dataset.

1.1 Introduction to Generative Adversarial Networks

GANs [2] were introduced as a new framework for estimating generative models via an adversarial process, in which two models, discriminator and generator, are trained simultaneously.

The discriminator D is trained to distinguish between real data samples, drawn from the true data distribution, and fake samples produced by the generator G.

Through this adversarial process, both models are expected to improve over time: the Discriminator becomes better at detecting fakes, while the Generator learns to produce increasingly realistic images (in other words, the fool ends up being fooled).

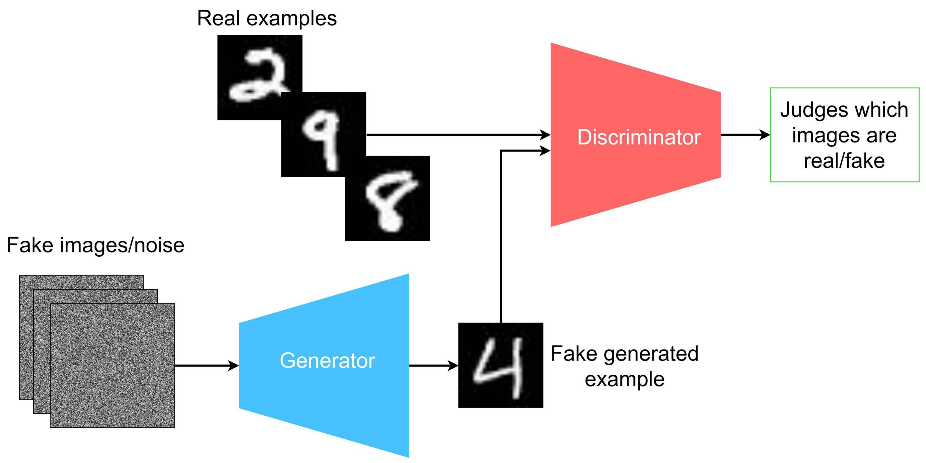


Figure 1: GAN architecture (Image source : IBM Developer)

1.2 Generative modeling

Let x be a sample from the real data distribution, such that $x \sim p_{\text{data}}(x)$, where $p_{\text{data}}(x)$ represents the unknown distribution of the real data.

In probabilistic terms, we want to learn a generative model $p_\theta(x)$, where θ represents the model parameters. The goal is to optimize θ such that the model distribution $p_\theta(x)$ approximates the true data distribution $p_{\text{data}}(x)$. But with high-dimensional data (images for instance), directly optimizing $p_\theta(x)$ is intractable (we rarely have an explicit model of the real data or, simply cannot assign a likelihood value to x).

1.3 The Generator

To overcome this issue, [2] introduces a lower-dimensional latent variable z, drawn from a known distribution $p_z(z)$, such as a gaussian distribution. $p_z(z)$ are defined as "noise variables" by Goodfellow.

The idea now, is to define the generative model implicitly by learning a function $G(z, \theta_g)$. G is a differentiable function represented by a multilayer perceptron with parameters θ_g, that maps z to the data space : $x_{\text{fake}} = G(z)$, $z \sim p_z(z)$.

This defines the generator's distribution p_g over x. The generator learns a function $G(z)$ that transforms z into data-like outputs, in other terms, it learns to produce samples that look like they came from $p_{\text{data}}(x)$.

1.4 The Discriminator

Now, the discriminator, which is also a multilayer perceptron $D(x, \theta_d)$, receives as input either a real data sample $x \sim p_{\text{data}}(x)$ or a generated sample $G(z)$, and outputs single scalar $D(x)$, that represents the probability that x came from the real data rather than from p_g. $D(x) \in [0, 1]$ by definition.

1.5 Adversarial Training

The generator and the discriminator are trained simultaneously in a two-player min-max game with the value function $V(D, G)$.

D is trained to maximize the probability of assigning the correct label to both training examples x and samples from G (i.e. G(z), i.e. x_{fake}). In other terms, the discriminator tries to correctly classify inputs as real or fake.

G is trained to minimize $log(1 - D(G(z)))$. In other terms, the generator tries to produce outputs that are indistinguishable from real data. If the generator becomes so good that $x_{\text{real}} \approx x_{\text{fake}}$, then as $G(z) = x_{\text{fake}}$, we have $D(G(z)) \approx D(x)$. By definition $D(x_{\text{real}}) \approx 1$, which minimizes $log(1 - D(G(z)))$.

We can express the two-player min-max game as follows :

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

1.6 Training procedure (theory)

The training alternates between the discriminator and the generator. To update D, we fix G and train D to distinguish real from fake samples. To update G, we fix D and train G to fool D.

Goodfellow points out that in practice we can train G to maximize $log(D(G(z)))$ rather than training G to minimize $log(1 - D(G(z)))$. The reason is that "early in the learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data, which leads $log(1 - D(G(z)))$ to saturate, and so the original min-max formulation may not provide sufficient gradient for G to learn well."

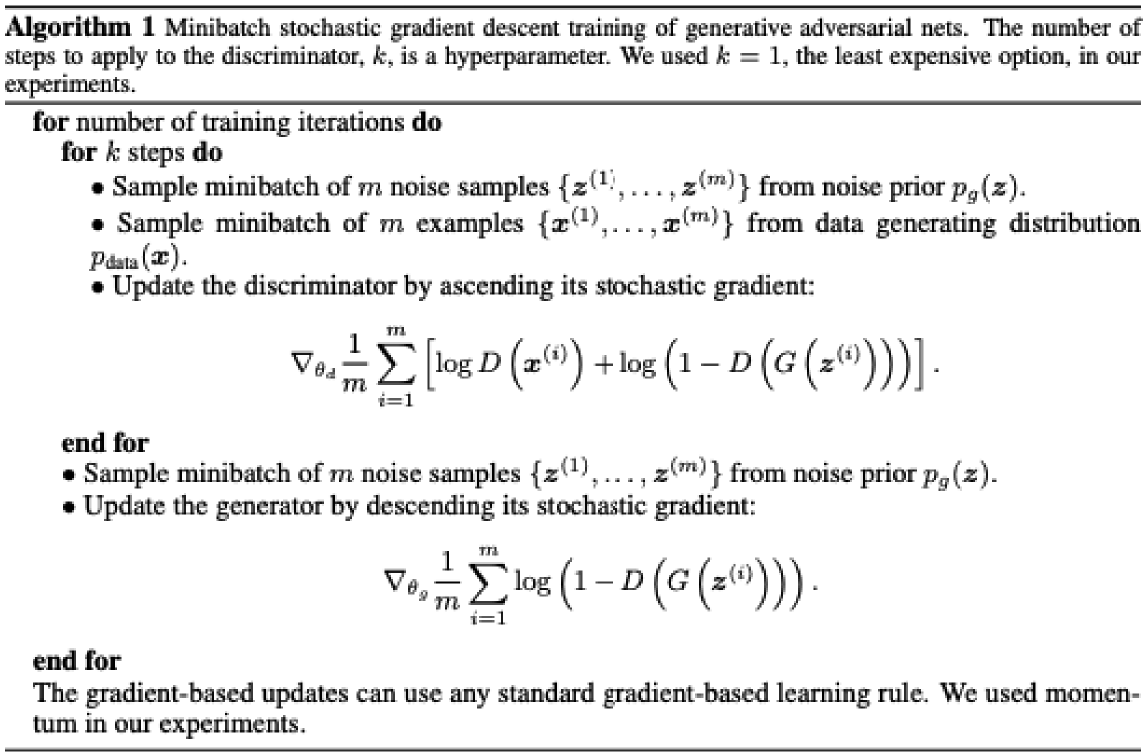
So in practice, the two-player min-max game is like a two-player "min-min" game, as both networks minimize their own separate loss function ; maximizing $log(D(G(z)))$ is same as minimizing -$log(D(G(z)))$. But that's a heuristic and not the original formulation of the two-player min-max game ...

1.7 Optimal discriminator

The optimal D is obtained when G captures the real data distribution $p_{\mathrm{g}}(x) = p_{\mathrm{data}}(x)$. At equilibrium, D is maximally confused, meaning it cannot distinguish real samples from fake ones, and $D(x) = 0.5$, for all x (Proof in [2] if interested).

1.8 Training procedure (in practice)

In practice, GANs rely on (mini-batch) Stochastic Gradient Descent to train the generator and the discriminator. Instead of computing gradients over the entire dataset, training is done on smaller randomly chosen subset of the data (called mini-batches).



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of $m$ examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
              $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right].$
    end for
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Update the generator by descending its stochastic gradient:
          $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$
end for
```

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 2: GAN training algorithm (Image source : GAN paper [2])

1.9 Image generation with a simple GAN model

I have trained a simple GAN to generate handwritten digits. I have used the MNIST dataset [5] to train my model. The model architecture is as follows :

Parameters	**Value**
Learning rate	0.0002
Batch size	128
Noise vector size (z)	100
Loss (criterion)	Binary Cross-Entropy
Optimizer	Adam
Beta1 (Adam)	0.5
Beta2 (Adam)	0.999
LeakyReLU slope (G,D)	0.2
Number of epochs	100
Image size	28x28
Nb. of channels	1

Table 1: Parameters used for training G and D - GAN

I trained the model for 100 epochs, with the noise vector z set to a dimension of 100. Since the ultimate goal of D is to determine whether the received samples are real or fake, I used the Binary Cross-Entropy loss function for training and the Adam optimizer for stochastic gradient descent.

The architecture of G is as follow :

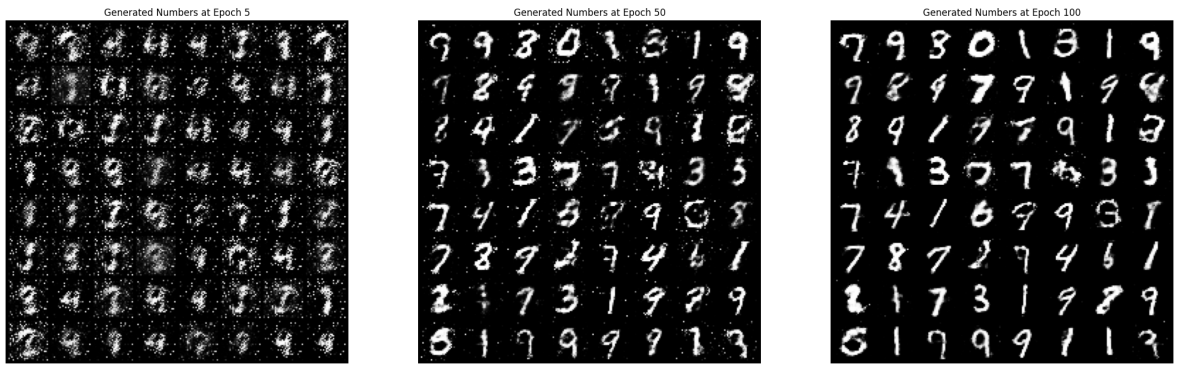
```
Generator(
  (G): Sequential(
    (0): Linear(in_features=100, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Linear(in_features=256, out_features=784, bias=True)
    (5): Tanh()
  )
)
```

G takes the latent noise vector z as input and generates a flattened image that will be the input for D. The architecture of G is relatively modest: it is composed of three fully connected linear (FCN) layers, two LeakyReLU activation layers, and one Tanh layer. The fully connected linear layers gradually upscale the dimension of the input noise vector z to match the dimension of the MNIST images ($1 \times 28 \times 28$). LeakyReLU layers serve as non-linear activation functions to reduce the linearity introduced by the FCN layers. LeakyReLU is preferred over regular ReLU to avoid the "dying ReLU" issue. Finally, the Tanh layer maps the pixel values of the last FCN layer to the range $[-1, 1]$.

The architecture of D is as follow :

```
Discriminator(
  (D): Sequential(
    (0): Linear(in_features=784, out_features=256, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Linear(in_features=128, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

D's architecture is quite similar to that of G. It is composed of three FCN layers, two LeakyReLU activation layers, and one Sigmoid layer. The FCN layers gradually downscale the dimension of G's output to produce a single scalar, which is then passed through the Sigmoid function to map it to the $[0, 1]$ range. This value represents the probability that the input image is real rather than generated (fake).


Generated Numbers at Epoch 5
Generated Numbers at Epoch 50
Generated Numbers at Epoch 100

(a) Images at 5 epochs (b) Images at 50 epochs (c) Images at 100 epochs

Figure 3: Generated digits at 5, 50 and 100 epochs (GAN)

At first, I thought that 100 epochs wouldn't be sufficient to generate realistic samples, but it appears that even after 50 epochs, the results are quite satisfactory, as we can already distinguish clearly some digits at that step.

In this section, we explored how GANs work in theory and presented a simple architecture to train a GAN on the MNIST dataset for generating images of handwritten digits. As we can see, most of the generated digits are quite realistic, but some remain noisy and blurry. In the next section, we will examine another architecture called Deep Convolutional GANs (DCGANs) [7]. DCGANs replace the fully connected layers used in standard GANs with deep convolutional and convolutional-transpose layers. This architecture is designed to produce sharper and more realistic images, enable more stable training, better exploit the spatial structure of images, and scale efficiently to higher resolutions.

2 Deep Convolutional GANs

2.1 What's new ?

DCGAN [6] introduced several architectural innovations that improved the quality and stability of GAN training on image data.

Instead of traditional pooling layers, DCGAN replaces them with strided convolutions in the discriminator and fractional-strided (or transposed) convolutions in the generator, allowing the model to learn its own spatial upsampling and downsampling.

To stabilize training, batch normalization is applied in both the generator and discriminator, reducing internal covariate shift.

Also, DCGAN eliminates fully connected hidden layers in deeper networks, encouraging the use of only convolutional operations to better preserve spatial structure.

Activation functions are carefully chosen, the generator uses ReLU in all layers except the output, which uses Tanh to match image normalization. The discriminator uses LeakyReLU throughout to prevent the dying ReLU problem and ensure better gradient flow.

2.2 Image generation with DCGAN

I have also trained a DCGAN model to generate handwritten digits, on the same MNIST dataset [5] as for GANs. The model architecture is as follows :

Parameters	**Value**
Learning rate	0.0002
Batch size	128
Noise vector size (z)	100
Loss (criterion)	Binary Cross-Entropy
Optimizer	Adam
Beta1 (Adam)	0.5
Beta2 (Adam)	0.999
LeakyReLU slope (G,D)	0.2
Number of epochs	100
Image size	28x28
Nb. of channels	1

Table 2: Parameters used for training G and D - DCGAN

I trained the model for 100 epochs, with the noise vector z set to a dimension of 100, as for GANs. I also used the Binary Cross-Entropy loss function for training and the Adam optimizer for stochastic gradient descent.

The architecture of G is as follows :

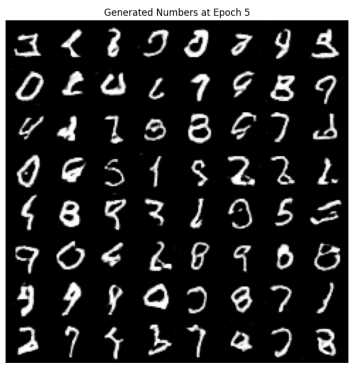
```
DCGAN_Generator(
  (G): Sequential(
    (0): ConvTranspose2d(100, 256, kernel_size=(7, 7), stride=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(128, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): Tanh()
  )
)
```

G takes a latent noise vector z and generates an image using transposed convolutional layers to progressively upscale spatial dimensions. The architecture includes three ConvTranspose2d layers that expand the input from 1×1 to 28×28, with decreasing channel depth (from 256 to 1). Batch normalization is applied after the first two convolutional layers to stabilize training, and ReLU is used as the activation function throughout, except in the final layer where Tanh maps pixel values to the range [1, 1], matching the normalized image data.

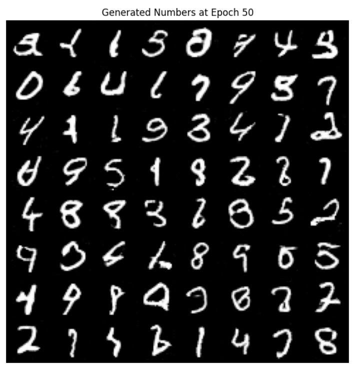
The architecture of D is as follows :

```
DCGAN_Discriminator(
  (D): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Flatten(start_dim=1, end_dim=-1)
    (6): Linear(in_features=6272, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
```

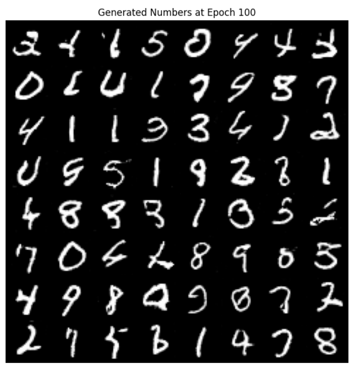
D takes an image as input and outputs a probability indicating whether the image is real or generated. It uses two Conv2d layers to progressively downsample the image from 28×28 to 7×7 while increasing feature depth (from 64 to 128). Each convolution is followed by a LeakyReLU activation to maintain gradient flow, and BatchNorm2d is applied after the second convolution to stabilize learning. The resulting feature map is flattened and passed through a Linear layer that reduces it to a single output value, which is then passed through a Sigmoid to produce a probability in [0, 1].



(a) Images at 5 epochs



(b) Images at 50 epochs



(c) Images at 100 epochs

Figure 4: Generated digits at 5, 50 and 100 epochs (DCGAN)

At 5 epochs, the generated images are already realistic and not as noisy as those produced by the original GAN. This highlights the effectiveness of DCGAN's architecture, particularly the use of transposed convolutions, batch normalization, and spatially-aware feature learning—in generating clearer and more structured outputs early in training.

3 Brief overview of other GAN-based architectures

Several other architectures have been introduced following the development of GANs and DCGANs. Vanilla GANs use the Jensen-Shannon divergence to measure the similarity between the generated and real data distributions. In contrast, Wasserstein GANs (WGANs) [1] replace this with the Wasserstein distance, which provides a more meaningful and smoother loss metric. This leads to more stable training and helps mitigate issues like mode collapse. WGANs are especially useful in applications that require consistent and high-quality image generation.

StyleGAN [4] introduces a novel style-based generator architecture that enables fine-grained control over image synthesis at different levels of detail, such as coarse structures, facial features, and textures. It is achieved by separating the latent space from the image synthesis process, allowing manipulation of image attributes in a more interpretable way. It finds applications in photorealistic human face generation or dataset augmentation.

SRGAN [6] is designed for image super-resolution. It enhances the resolution of input images using a perceptual loss function based on high-level feature maps. It combines an adversarial loss with a perceptual loss for realistic textures.

CycleGAN [8] learns to translate images between two domains without requiring paired training data. It uses cycle consistency to ensure that an image translated to the target domain and back remains unchanged. CycleGANs are useful for style transfer and are even used for medical imaging and photo enhancement.

Pix2Pix [3] is a conditional GAN trained with paired images. It learns a mapping from input images to output images, such as sketch-to-photo, using a generator and discriminator setup with L1 loss. Pix2Pix can be used for supervised image-to-image translation.

4 Conclusion

I explored the concepts behind GANs and DCGANs, implementing both architectures to generate handwritten digits using the MNIST dataset. I observed how DCGANs improve the quality and stability of generated images compared to standard GANs. This study provided valuable insight into how adversarial training can be leveraged for image generation and sets the stage for deeper exploration into advanced variants like CycleGAN and StyleGAN in future work.

5 Code

The code for the GAN and the DCGAN architecture is available on github : **https://github.com/selim-ba/computer-vision/**

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[2] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[3] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1125–1134, 2017.

[4] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4401–4410, 2019.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[7] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[8] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2223–2232, 2017.