Final Project

Ivan Lima do Espirito Santo
Rosa Yuliana Gabriela Paccotacya Yanque
Thiago Gomes Marçal Pereira

***Abstract*—Image-to-Image translation refers to an area of Machine Learning that aims taking images from one domain and transforming them into images of another domain in a way that the original images preserve their essence but change the style. Some images can be easily related to each other, like satellite photo and map drawings, horses and zebras, landscape photos and Monet landscape paintings, oranges and apples. In all of these pairs of domains, there are similarities but also each one of the pair has its particular characteristic.**

One approach to the problem is the paired image-to-image translation via Deep Neural Network. This approach uses pairs training data which each pair of image represents the same image in the two different styles (domains). This method requires a lot of labeled data (matched pairs) which can be time-consuming and expensive to acquire. It is not easy finding situations in which pairs of images are at same position and location.

Our proposal in this project is to address the image-to-image translation from a domain X to a domain Y without paired images. Our translation is based on Cycle-Consistent Adversarial Networks [3], that uses adversarial loss to make the mapping $G : X \rightarrow Y$ such that distribution of images from $G(x)$ is as close as possible to the distribution of Y. And also uses the inverse mapping F that maps Y to X to be able to define a cycle consistency loss that enforces $F(G(X)) \approx X$, constraining the range of possible solutions.

To evaluate our implementation we performed two experiments: Horses to Zebras and landscape photos to Monet paintings.

I. Image to Image Translation

Consider a problem of creating a image. For instance, we could draw a rough representation of an object and try to create a more accurate image from this rough representation. For this approach to be attainable, we need first a dataset with a certain amount of these rough images in pairs with their accurate images. Once these pairs of images are available for a lot of different objects (or people, animals, landscape, etc) the problem of creating a new image from a new rough representation is a matter of learning the translation function (see [2] and [4]). However, this approach relies on data labeling, which is expensive and not desirable. On the other hand, consider the problem of creating images, but this time you do not have pairs of them to train the translation function. All you have is different images, with different styles. Satellite images and google map drawings, horses and zebras, landscape photos, and landscape paintings are all similar in some sense. We can posit they differ in style. Therefore, we can think of image translation as an attempt to creating a new image by applying the style of other images. For instance, we can translate a photo image of a landscape into a famous painting style. Because the images do not need to be available in pairs, this approach is known as unpaired. One way to achieve unpaired image-to-image translation is through Generative Adversarial Networks (see appendix A). In summary, the trick is learning to translate an image from X to Y by mapping $G : X$ to Y in a way the distribution of images from $G(X)$ becomes indistinguishable from the distribution Y. Moreover, the inverse mapping $F : Y$ to X is possible and achieved as well.

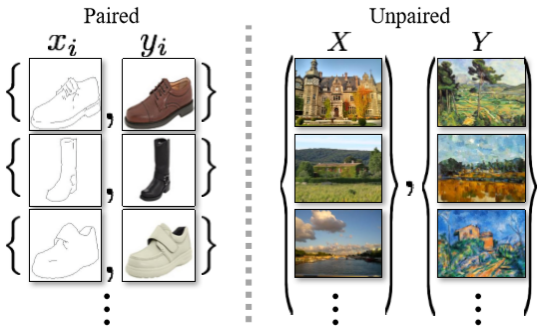


Fig. 1: Comparison between pair (left) and unpaired (right) approach.

Notice in figure 1 extracted from [3] that while in the paired approach we need matched pairs of images in the different domains, in the unpaired approach there is no such requirement. We just have to have random images in the different styles (domains).

II. Unpaired Image-to-Image Translation

Based on is explained before, GAN approach appears to be suitable to learn to translate between different domains without images pairs. When we have two sets of images X and images Y, with GAN we may find a mapping G that maps X to Y as close as they end up indistinguishable. However, this translation is not sufficient to guarantee that an individual x and an output y is a meaningful pair once there are many possible G. Furthermore, another practical problem is common during the optimization of the adversarial objective when all the images maps to the same output images due to mode collapse.

These problems claim for some improvements in the GAN approach to make it indeed suitable for the task of unpaired image-to-image translation. Aiming to address these issues, [3] implemented the concept of cycle consistency. When a translation G maps the images from X to images Y, also a translation F maps them backwards, i.e. from Y to X. In other words, instead of seeking any translation G, we seek

RA: 956694 - i956694@g.unicamp.br
RA: 263068 - r263068@dac.unicamp.br
RA: 189691 - t189691@g.unicamp.br

for one that has an inverse F. The existence of F is assumed, therefore both G and F are training simultaneously. The cycle consistency loss is then defined to make $F(G(x)) \approx x$ and $G(F(y)) \approx y$. Accordingly, for the unpaired image-to-image translation we have the final objective by combining one adversarial loss for G, one adversarial loss for F and the cycle consistency loss.

III. Formulation

Given two domains X (images of horses, for instance) and Y (images of zebras, for instance), our aim is to learn a mapping function G that maps X to Y and learn a mapping function F that maps Y to X. For each of them, we use two adversarial discriminators, Dx and Dy. While Dx aims to discriminate between images x and translated images $F(y)$, $D(x)$ aims to discriminate between images y and $G(x)$. See figure 2.

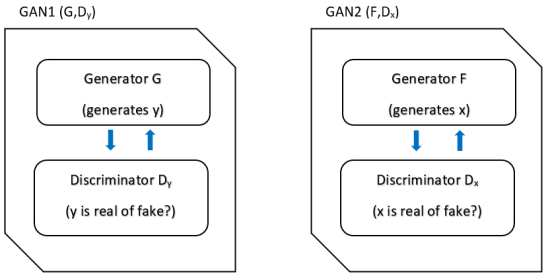


Fig. 2: Two GANs formulation scheme.

Besides the two adversarial loss for each of them, which push the generated images towards the target images, we need the cycle consistency loss to make G and F consistent with each other (avoid contradiction).

Adversarial loss is core idea behind GANs. If we take the equation 7 (see appendix A) and write it for the map $G : X$ to Y, and the discriminator D_y as shown in left side of figure 2, we have expressions 1

$$min_G max_{D_y} \ell_{GAN}(G, D_y, X, Y) = E_{y \sim p_{data}(y)}[\log D_y(y)] + E_{x \sim p_{data}(x)}[\log(1 - D_y(G(x))] \quad (1)$$

G tries to generate images similar to Y while D_y tries to recognize if they are real or fake. Real images come from y and fake images are generated by $G(x)$. At the same time G tries to minimize, D_y tries to maximize.

The same happens in the right side of the figure 2, but now for the map $F : Y$ to X, as shown in the expression 2

$$min_F max_{D_x} \ell_{GAN}(F, D_x, Y, X) = E_{x \sim p_{data}(x)}[\log D_x(x)] + E_{y \sim p_{data}(y)}[\log(1 - D_x(F(y))] \quad (2)$$

Each of these two adversarial losses represents the minmax game between the generator and the discriminator.

However, adversarial losses are not sufficient to guarantee that G and F found in the Nash equilibrium of the minmax game would map a individual input x to a desired output y. In fact, with enough capacity, a network can map a same set of input images to any aleatory combination of images in the target domain. Therefore, we constrain the possible mapping functions making them cycle-consistent. G and F should be each other inverse.

$$x \to G(x) \to F(G(x)) \approx x \quad (3)$$

$$y \to F(y) \to G(F(y)) \approx y \quad (4)$$

3 and 4 are known as forward cycle consistency and backwards cycles consistency respectively. Each image x from domain X, the image translation cycle should be able to bring x back to the original image. Each image y from domain Y, the image translation cycle should be able to bring y back to the original image. To achieve this we define the cycle consistency loss as following:

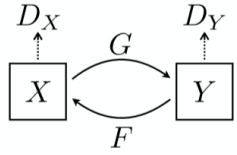
$$\ell_{cyc}(G, F) = E_{x \sim p_{data}(x)}\left[\|F(G(x)) - x\|_1\right] + E_{y \sim p_{data}(y)}\left[\|G(F(y)) - y\|_1\right] \quad (5)$$

Where the quantity inside the brackets is the L1 norm.

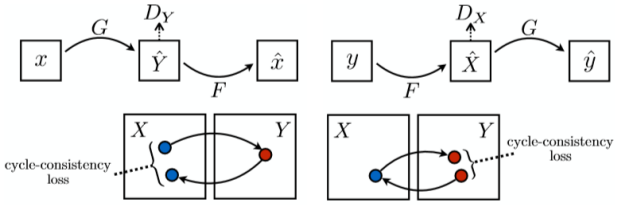
Summing all three losses we have the expression 6 for the full objective.

$$\ell(G, F, D_x, D_y) = \ell_{GAN}(G, D_y, X, Y) + \ell_{GAN}(F, D_x, Y, X) + \lambda \ell_{cyc}(G, F) \quad (6)$$

where the parameter λ is used to balance the weight of the two objectives.



(a) Mapping Functions



(b) Forward Cycle Consistency (c) BAckward Cycle Consistency

Fig. 3

Figure 3 provides a visualization for two cycles consistency loss by two dots distance. As close as dots are, lower the the consistency loss.

IV. Architecture

We can consider our task as jointly training two autoencoders, where each one map an image to itself via an intermediate representation - translation of the image to another

domain. It is a special case of adversarial autoencoders which in order to train the bottleneck layer to match an arbitrary target distribution, we use the adversarial loss. In other words, the target distribution for the X to X is that of the domain Y.

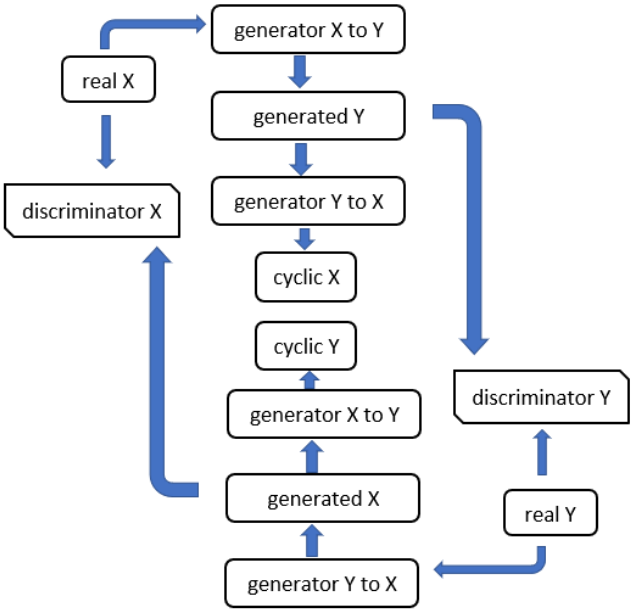


Fig. 4: simplified scheme of the architecture

A simplified representation of the model is shown in figure 4. An image from the domain X is fed to the generator that generates an image Y. This generated image Y passed to the discriminator, which judged if it is real (1) or fake-generated (0). This same image generated Y is the input of generator Y to X, to convert it back to the original image X. The same process occurs for the Y domain. Both discriminators receive two inputs: one generated and one real. The discriminator's task is to distinguish these images and assign them as false if they come from the adversary, the generator. The generator, in its turn, tries to defeat the discriminator generating images as indistinguishable as possible from the original images.

The building blocks of the implementation are mainly the generator, figure 5, and the discriminator, figure 7. The generator is divided in decoder, transformer and encoder.

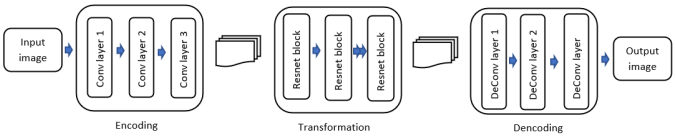


Fig. 5: Generator

Table I shows the parameters normally defined in image processing and some typical values. The first step is to apply a convolution network to extract the features of the image. The convolution network take a image and move the filter

TABLE I: Common Parameters

description	value
number of filters in the first layer the generator	32
number of filters in the first layer the discriminator	64
size of batch	1
poll size	50
Input image width	256
Input image height	256
RGB color format	3

over the image respecting the stride size (how many rows and columns of pixies to skip when moving along the picture), performs the element-wise multiplication between the filter and the images correspondent region and store the values. The typical parameters used on the first layer encoding is shown in table II

TABLE II: Parameters used in the first encoding layer

parameters	value
width of the filter window	7
height of the filter window	7
width of the stride	1
height of stride	1

The parameters shown in table II, with the image input and the number of filters are normally the set of input parameters of a generator. The output tensor is carried on to the next convolution layer to extract features in deeper level.

The transformation block, figure 6, combines different close features of a image and based on the features decides to transform the feature vector encoding of an image from one domain to the other. We use layers of resnet blocks for this purpose. Resnet block is a neural network layer that has two convolution layers where the residual of an input is added to the output. The goal is to enforce that the properties of previous layers inputs are available for later layers, therefore their output do not diverge to much from the original input. Residual network serves well of for the purpose to retain the characteristic of the original input in our transformation.

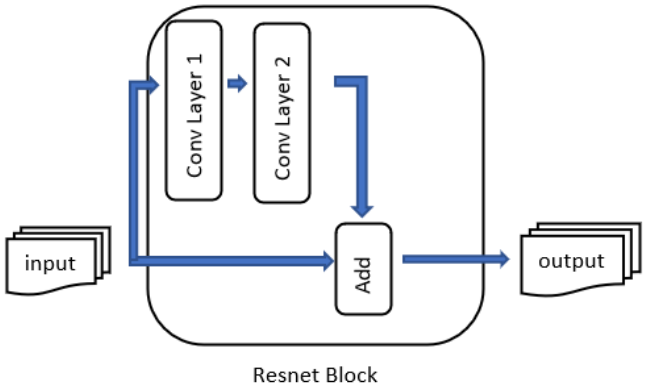


Fig. 6: Resnet Block

We input a feature vector from on domain into the transformation and output a feature vector of the other domain.

Afterwards, it comes the decoding phase. We take the low level features vector output from the transformation and pass throw a deconvolution layer to recover the original dimension. Summarizing the generator process, we take images from the domain X with dimension $[256, 256, 3]$ pass to the encoding to extract the features and output a feature tensor of dimension $[64, 64, 256]$. We input the encoding output into a transformation block of resnets that keep the dimension and transform the image to the Y domain. The output of the transformation block is input into a decoding block that recover the original dimension in the Y domain.

For adversarial training we build the discriminator as shown in the figure 7

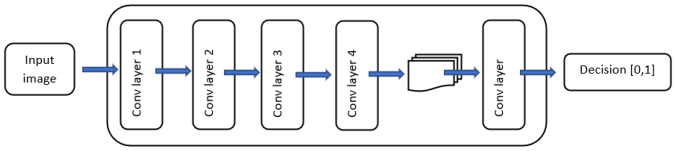


Fig. 7: Discriminator

The discriminator, figure 7, is a sequence of convolution network to extract the feature which the last convolution layer produce one dimension output used for decision making.

As the model works in both direction, domain X to Y and Y to X, we need generators and discriminators in both direction.

To build the model we define first the $input\_X$ and $input\_Y$ as a function of the batch size, the image width, the image height and the image layer. Subsequently we build the generators and discriminators following the structure shown in table III.

TABLE III: model content

name	input	purpose
gen_Y	$input\_X$	generate X to Y
gen_X	$input\_Y$	generate Y to X
dec_X	$input\_X$	distinguish real/fake X
dec_Y	$input\_Y$	distinguish real/fake Y
dec_gen_X	gen_X	distinguish real/fake X
dec_gen_Y	gen_Y	distinguish real/fake Y
cyc_X	gen_Y	generate Y to X
cyc_Y	gen_X	generate X to Y

Regarding the discriminators dec_X and dec_Y, they must assign 1 if the image is real, so we train them by minimizing the square difference to 1 for both discriminators. At the same time dec_gen_X and dec_gen_Y should predict 0 for images generated, thus we train them by minimizing the square of these discriminators. The discriminator final loss is the average of these two losses.

The generator should also be enforced to convince the discriminator about its images authenticity, therefore dec_gen_X and dec_gen_Y square difference to 1 is used to train the model as well. Next we define the cycle loss as the difference between the original image and the cyclic image. The total generator loss is then defined as the sum between the loss to fool the discriminator and the cycle loss times a scale factor λ which can be tun to balance the weight of each loss in the learning process.

After all the losses being defined, we train the model by minimizing all of them.

V. Implementation

Our implementation uses Keras and TensorFlow as shown in table IV. Most of the training time was performed in Google Colab due GPUs advantage. Once the model had trained we saved it to be able to transfer it to Gitlab.

TABLE IV: version

tool	version
Keras	2.3.1
TensorFlow	1.15.2

For our experiments and initial training, we used Horse2Zebra and Monet2Photo images available at https: $people.eecs.berkeley.edu/\sim taesung\_park/CycleGAN/$ which is the same dataset of the original CycleGAN paper.

For data augmentation, we implemented simple image manipulations, listed below:

- Add poisson noise (using Numpy's random.poisson)
- Random Color Change (by changing image's channels)
- Mirroring Image
- Flipping image Vertically
- Randomly erasing some part of the image

The discriminator contains 5 convolutions and uses $LeakyReLU$ with $2,764,743$ total trainable parameters. The generator is a more complex structure, once it contains 24 convolutions with $tanh$ at the output and resnet with $LeakyReLU$ as activation functions. The Generator has $2,850,609$ total trainable parameters. See appendix B for more details.

For the loss minimization we use ADAM algorithm, which is a stochastic gradient descent optimizer commonly used in Deep Neural Networks.

For the initial 500 epochs at the training phase, we set a learning rate to 2E-4 for both Generator and Discriminator. Afterwards, for comparison with different dataset, we reduced the Generator Learning rate to 2E-7, maintaining the Discriminator's learning rate at 2E-4, such that it might learn faster how to validate data from other datasets.

For training the model, datasets are divided in training and test, in groups A and B, that would refer to transformations $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and applied for both generators and discriminators, as shown in figure 4. As mentioned before, we used Google Colab to train our models, but as it has 12 hours limit per execution session, we had to implement a solution for saving the network weights. For implementation simplicity, we considered only the cycle_loss, and not the generator and discriminator losses individually, for saving both weights. At end of every epoch, we verified the losses and saved the weights.

Moreover, at the end of every epoch, we select images randomly from test dataset and save their outputs. The images were saved by columns, where the top row is the the original image, middle row is the transformation from domain X to

domain Y ($G : X \rightarrow Y$), and the bottom row is the inverse transformation of the generated image $F : Y \rightarrow X$. It was done for both groups (domains), and a few images can be seem, depending on the number of batches used.

After the timeout, we restart execution, loading previously saved weights and run again. Due to our time constraints, we opted to train about 600 epochs. We see that the final results can still be improved with further training.

Due to Gitlab's restrictions, the training part is commented and the code just executes the test phase on CPU. Furthermore, as our implementation requires specific Keras and Tensorflow versions, we use a different docker image (tensorflow/tensorflow:1.5.1-py-3), as adnrv/opencv uses python 3.7.3 that has some known incompatibilities with Tensorflow 1.x. The dataset added to Gitlab is also reduced to prevent memory issue due to image size that could interrupt the Python execution. Moreover, weights are pre-loaded and test images are converted.

VI. Results

At this first image shown at figure 8, we see that the transformations do not present visual meaningful outputs, being just random noise.

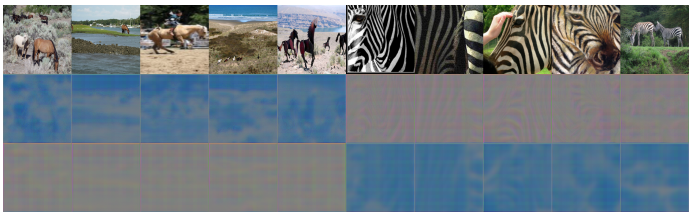


Fig. 8: horse2zebra 1 epoch

After some epochs, we see more meaningful data, but still not style transfers, mostly color changes. We can already detect something that looks like an "image segmentation", because some horses and zebras can be easily differentiated in the transformations. Figure 9 shows the output for 50 epochs.

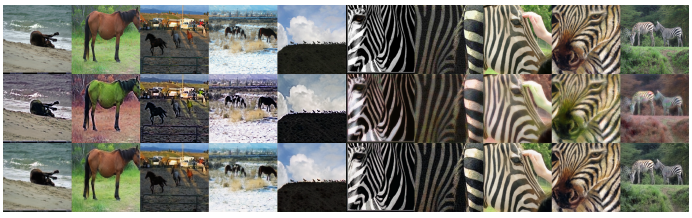


Fig. 9: horse2zebra 50 epochs

The final results from our experiments presents better transformation, similar to the output in Gitlab execution.

Notice that figure 9 shows pretty impressive results for the inverse.

In order to test our network, we decided to try it against Monet2Photo Dataset. The model tries to transform Monet paintings style into real landscape photos and vice versa.

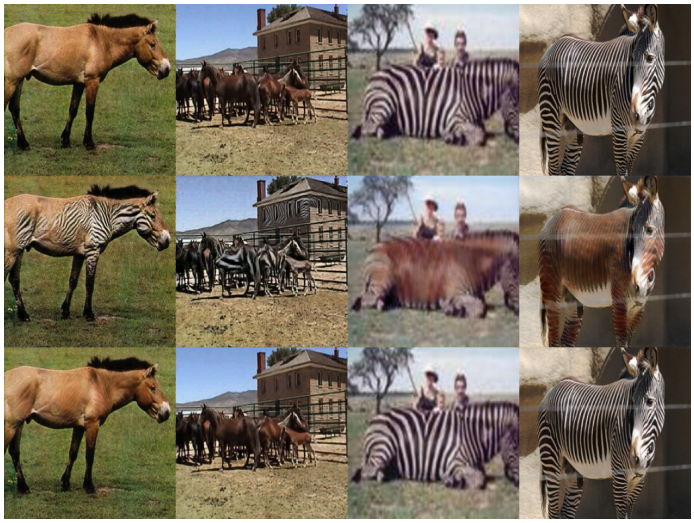


Fig. 10: horse2zebra 113 epochs

Due to time constraints, our approach was to apply the weights from training horse2zebra dataset. From initial epochs, this approach did not provide good translation, specially for paintings as input. As we can see in figure 11, there are some "zebra patterns" spread over the image.

Fig. 11: Monet2photo 9 epochs

Figure 12 highlights with red mark some zebra patters found at low epochs translations.

After 80 epochs, these patterns vanish, although no style transfer was seen in both ways. The learning rate adopted was too small. This option, with the transfer learning did not produce the desired results. See figure 13.

Next, we decided to re-run transfer learning, but reducing learning rate of the generator to cross-check results. Changing the learning rate for the generator to 1E-4 allowed the network to learn the style faster, and with a few epochs. It is possible to see some improvements in the images. See figures 14 and 15. Also figure 16 for a closer view.

The image translation is more discernible from photo to Monet painting style, specially in sky areas. Notice that, in the image 15 example, the sea/tree painting gets well converted on photo style, and the trees seem more natural.

Regarding the Monet/photo experiments, we can say that the transfer learning of the rates showed reasonable results, even though the images domains being so different. We believe that with more epochs, results will improve. This would be done in future work, as the time is constrained.

Fig. 12: Zoom in the image 11 to show zebra patterns.

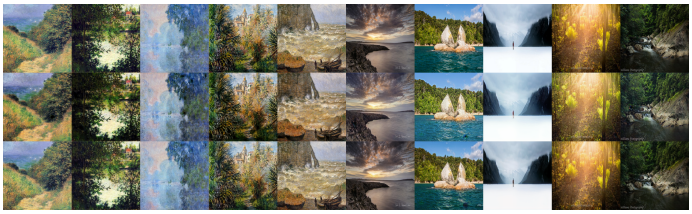


Fig. 13: Monet2photo 80 epochs

VII. Conclusion

Although we see promising results, there is room for a lot of improvement in our implementation. Saving the loss' decay over the learning process would be one of them. Unfortunately, we ended up overwriting outputs to save space. Knowing the behavior of the adversarial losses and the cyclic losses will give us insight into how to adjust the model. We would have educated guess on how to tun the λ factor, the learning rate, or which building block should be modified to achieve better performance. Tracking all losses is our priority for future work.

Testing different domains is also crucial for further improvements. Although zebras and horses do not share the same habitat, they are resembling animals. We can say the same for landscape photos and landscape paintings. Trying the response of our model to completely different domains would give us insightful knowledge about its strengths and weakness.

Even though the choices of images domains presented in this work seem recreational and not challenging at all, GANs concept and in particular CycleGANs concept were well explored. The adversarial loss combined with cyclic loss proved to be a powerful concept to domains translation. It transcends image applications.

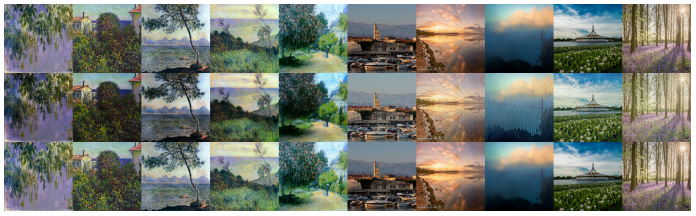
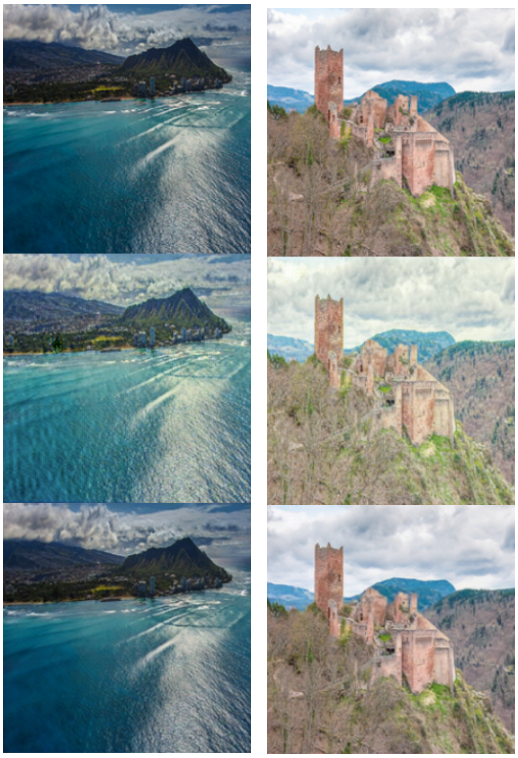


Fig. 14: Monet2photo 125 epochs and generator learning rate 1E-4

Fig. 15: Monet2photo 160 epochs and generator learning rate 1E-4



(a) (b)

Fig. 16: Zoom at figure 15

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial networks. *ArXiv*, abs/1406.2661, 2014.

[2] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5967–5976, 2017.

[3] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[4] J.-Y. Zhu, R. Zhang, D. Pathak, T. Darrell, A. A. Efros, O. Wang, and E. Shechtman. Toward multimodal image-to-image translation. *ArXiv*, abs/1711.11586, 2017.

APPENDIX A
GENERATIVE ADVERSARIAL NETWORKS

A. GANs' concept

GANs' creates a generative models via an adversarial process, that simultaneously trains two models: a generative model G that captures the data distribution and therefore generates samples, and a discriminative model D that estimates the probability that a sample came from the training data rather than generated by G. In other words, the discriminative model tries to predict if the samples is real (from the training data) or fake (generated by the generative model), while the generative model tries to fool the the discriminative model generating samples as similar as possible to real ones. This setup is equivalent to a minimax two-players game. The training rule for G is to maximize the probability of D making a mistake. The samples generated in this way end up very similar to real samples as one can see the left face (real) and the generated face in figure 17. [1].

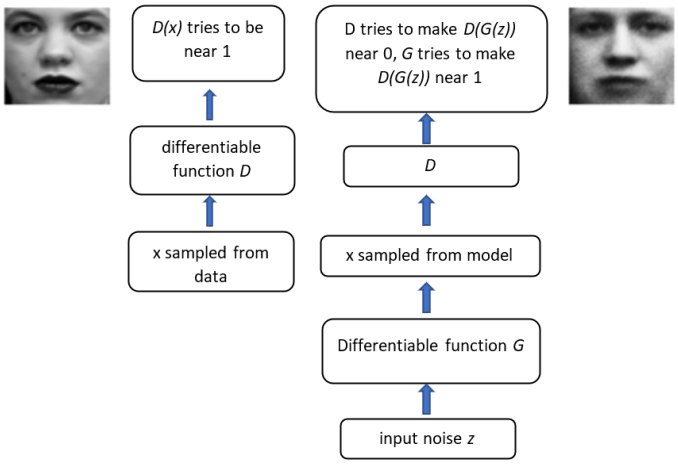


Fig. 17: Figure from NeurIPS 2016 GAN Tutorial (Goodfellow)

B. GANs' Math

We can mathematically describe the adversarial relationship between the generative model and the discriminative model with an equation of format 7.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}\,(x)}[\log D(x)] + \mathbb{E}_{z\sim p_x(z)}[\log(1-D(G(z)))] \tag{7}$$

$G(z)$ generates x images. Meanwhile the discriminator tries to classify (discriminate) which of x are real versus generated. $\mathbb{E}_{x\sim p_{\text{data}}}$ is the real data distribution. The framework is such as it tries maximize the log probability under the discriminator which can have 0 or 1 sigmoid output. Discriminate will try to drive the output to 1 if it is real. Simultaneously, $\mathbb{E}_{z\sim p_x(z)}$ does the opposite. It tries to drive the discriminator to 0 because it is not real. In other words, the discriminator is trying to assign 1 to real and 0 to fakes as accurate as possible. If we reach the Nash Equilibrium of this game, none of them can improve anymore, the generator should generate things that looks real. The discriminator at that point outputs $50/50$. The game is defined to try to generate things as realistic as possible. The GANs algorithm presented in [1] is as follows:

C. The Optimal Discriminator

From equation 7, using the integrals and changing the variable to write both integrals with respect to x we have the following expressions:

$$\begin{aligned} V(G,D) &= \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1-D(G(z)))] \\ &= \int_x p_{\text{data}}\,(x)\log D(x)dx + \int_z p(z)\log(1-D(G(z)))dz \\ &= \int_x p_{\text{data}}\,(x)\log D(x)dx + \int_x p_g(x)\log(1-D(x))dx \\ &= \int_x \left[p_{\text{data}}\,(x)\log D(x) + p_g(x)\log(1-D(x))\right]dx \end{aligned} \tag{8}$$

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 $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

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

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

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

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

By calculation the derivative with respect to $D(x)$ and set it equal to zero we have:

$$\begin{aligned} \nabla_y [a \log y + b \log(1-y)] &= 0 \\ \implies y^* = \frac{a}{a+b} \quad \forall \quad [a,b] &\in \mathbb{R}^2 \backslash [0,0] \end{aligned} \tag{9}$$

Therefore, the optimal discriminator will take the form of equation 10.

$$D^*(x) = \frac{p_{\text{data}}(x)}{(p_{\text{data}}(x) + p_g(x))} \tag{10}$$

Inserting equation 10 in the original objective expression 7, we have expression 11:

$$\begin{aligned} V(G, D^*) &= \mathbb{E}_{x \sim p_{\text{data}}} \left[\log D^*(x)\right] + \mathbb{E}_{x \sim p_g} \left[\log\left(1 - D^*(x)\right)\right] \\ &= \mathbb{E}_{x \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}\right] + \mathbb{E}_{x \sim p_g} \left[\log \frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)}\right] \\ &= -\log(4) + KL\left(p_{\text{data}} \,\|\, \left(\frac{p_{\text{data}} + p_g}{2}\right)\right) + KL\left(p_{\text{g}} \| \left(\frac{p_{\text{data}} + p_g}{2}\right)\right) \end{aligned} \tag{11}$$

where the KL terms are the Jensen-Shannon Divergence (JSD) of p_{data} and p_g, which is ≥ 0. Notice that $V(G^*, D^*) = -\log(4)$ when $p_g = p_{\text{data}}$.

APPENDIX B
DISCRIMINATOR AND GENERATOR STRUCTURE

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 256, 256, 3)	0
random_crop_1 (RandomCrop)	(None, 70, 70, 3)	0
general_conv2d_0 (Conv2D)	(None, 35, 35, 64)	3136
leaky_re_lu_1 (LeakyReLU)	(None, 35, 35, 64)	0
general_conv2d_1 (Conv2D)	(None, 18, 18, 128)	131200
instance_normalization_1 (In	(None, 18, 18, 128)	2
leaky_re_lu_2 (LeakyReLU)	(None, 18, 18, 128)	0
general_conv2d_2 (Conv2D)	(None, 9, 9, 256)	524544
instance_normalization_2 (In	(None, 9, 9, 256)	2
leaky_re_lu_3 (LeakyReLU)	(None, 9, 9, 256)	0
general_conv2d_3 (Conv2D)	(None, 5, 5, 512)	2097664
instance_normalization_3 (In	(None, 5, 5, 512)	2
leaky_re_lu_4 (LeakyReLU)	(None, 5, 5, 512)	0
general_conv2d_4 (Conv2D)	(None, 3, 3, 1)	8193

Total params: 2,764,743
Trainable params: 2,764,743
Non-trainable params: 0

Fig. 18: Discriminator Structure

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 256, 256,3)	0	
reflection_padding2d_1 (Reflect	(None, 262, 262,3)	0	input_3[0][0]
general_conv2d_0 (Conv2d)	(None, 256, 256, 32)	4736	reflection_padding2d_1[0][0]
instance_normalization_7 (Insta	(None, 256, 256, 32)	2	general_conv2d_0[0][0]
activation_1 (Activation)	(None, 256, 256, 32)	0	instance_normalization_7[0][0]
general_conv2d_1 (Conv2d)	(None, 128, 128, 64)	18496	activation_1[0][0]
instance_normalization_8 (Insta	(None, 128, 128, 64)	2	general_conv2d_1[0][0]
activation_2 (Activation)	(None, 128, 128, 64)	0	instance_normalization_8[0][0]
*			
*			
*			
general_deconv2d_22 (Conv2dTran	(None, 256, 256, 32)	18464	activation_22[0][0]
instance_normalization_29 (Inst	(None, 256, 256, 32)	2	general_deconv2d_22[0][0]
activation_23 (Activation)	(None, 256, 256, 32)	0	instance_normalization_29[0][0]
general_conv2d_23 (Conv2d)	(None, 256, 256, 3)	4707	activation_23[0][0]
activation_24 (Activation)	(None, 256, 256, 3)	0	general_conv2d_23[0][0]

Total params: 2,850,609
Trainable params: 2,850,609
Non-trainable params: 0

Fig. 19: Generator Structure