

Visão por Computador e Processamento de Imagen

(1º ano de Mestrado em Computação Gráfica)

Generative Adversarial Networks

Relatório

Rui Armada
(PG50737)

7 de julho de 2023

Resumo

O presente relatório detalha o desenvolvimento e aplicação de *GANs*, ou *Generative Adversarial Networks*, para a transformação de ilustrações, realizadas por *Claude Monet*, para fotografias realistas. Em adição, será realizada uma discussão dos resultados obtidos por este modelo.

Conteúdo

1	Introdução	2
2	Problema	3
2.1	Descrição	3
2.2	Estrutura do dataset	3
3	Preparação dos modelos e tratamento de dados	4
3.1	Estratégia geral	4
3.1.1	Pré-Processamento	4
3.1.2	Mapping	6
3.1.3	Visualização de Imagens	6
4	Generator & Discriminator	8
4.0.1	Generator	8
4.0.2	Discriminator	8
5	Loss Functions	11
5.1	Generator loss	11
5.2	Discriminator loss	11
5.3	Reconstruction Loss	12
5.4	Identity loss	12
6	Treino	13
7	Resultados obtidos	18
8	Conclusões e análise	22

Capítulo 1

Introdução

O presente trabalho prático, desenvolvido no âmbito da unidade curricular de Visão por Computador e Processamento de Imagem, visa apresentar sucintamente a aprendizagem derivada do contexto prático e teórico lecionado, principalmente a exploração de *GANs* e *Autoencoders*, assim como a exploração de *loss* e manipulação ou combinação de *datasets*.

Desta forma, foi dada a liberdade de escolher um problema. Após uma breve pesquisa, foi escolhido um *dataset* do `tensorflow_datasets`, o *cycle_gan/monet2photo*. Com isto o problema deste trabalho prático é: **transformar uma obra de arte numa fotografia**.

Portanto, as ferramentas que foram utilizadas para a realização deste trabalho prático foram as mesmas utilizadas durante o decorrer da UC, isto é, ferramentas como o **Tensorflow**, entre outros.

Capítulo 2

Problema

2.1 Descrição

O problema escolhido para este trabalho prático será então: conseguir transformar uma pintura, de *Claude Monet*, numa foto realista com o auxílio de *GANs* e *Autoencoders*.

2.2 Estrutura do dataset

Portanto, e como foi referido na introdução, o *dataset* escolhido pode ser encontrado no package `tensorflow_dataset` ou no seguinte link.

Com isto, o dataset encontra-se dividido em quatro partes cada uma contendo um número variado de imagens:

- **trainA:** 1072 imagens de pinturas
- **trainB:** 6287 fotografias
- **testA:** 121 imagens de pinturas
- **testB:** 751 imagens de fotografias

Capítulo 3

Preparação dos modelos e tratamento de dados

3.1 Estratégia geral

GANs é uma classe de *machine learning* que surgiu em 2014. Esta consiste em dois modelos que serão treinados ao mesmo tempo, e que irão contestar-se reciprocamente, o **generator** aprende a criar imagens que parecem reais, enquanto o **descriminator** aprende a distinguir quais são as imagens reais e quais são as imagens falsas, por outras palavras, o ganho de um modelo é a **loss** do outro. No início, quando o treino começa, o *Generator* irá criar imagens falsas que serão óbvias para o *Discriminator* rapidamente aprender que estas são imagens falsificadas, mas à medida que o treino avança, o *Generator* irá aproximar-se gradualmente de uma imagem capaz de enganar o *Discriminator*.

Dito isto, foi então carregado o *dataset*, procedendo a um *split* adequado entre *test* e *train sets*, da seguinte forma:

```
1 DATASET = 'cycle_gan/monet2photo'  
2  
3 dataset, metadata = tfds.load(DATASET, with_info=True, as_supervised=True)  
4  
5 train_monet, train_photo = dataset['trainA'], dataset['trainB']  
6 test_monet, test_photo = dataset['testA'], dataset['testB']
```

Listing 3.1: Carregamento de dados

3.1.1 Pré-Processamento

Feito isto, foi necessário fazer um pré-processamento dos dados para prevenir o *overfitting* do modelo. Portanto, foram desenvolvidas as seguintes funções:

- **random_crop**: efetua um *crop* na imagem, isto serve para o modelo conseguir aguentar imagens com diferentes dimensões

```

7 def random_crop(image):
8     cropped_image = tf.image.random_crop(
9         image, size=[IMG_HEIGHT, IMG_WIDTH, 3])
10    return cropped_image

```

Listing 3.2: Função random_crop

- **normalize:** Altera a escala do *pixel value range* de [0, 255] para [-1, 1]

```

11 def normalize(image):
12     image = tf.cast(image, tf.float32)
13     image = (image / 127.5) - 1
14    return image

```

Listing 3.3: Função normalize

- **random_jitter:** *jittering* refere-se a uma técnica de pré-processamento de dados utilizada para aumentar a diversidade e a robustez dos conjuntos de treino. Aqui também é feito um *mirroring* das imagens para oferecer ainda mais robustez ao conjunto de treino.

```

15 def random_jitter(image):
16     image = tf.image.resize(image, [286, 286],
17                           method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
18
19     image = random_crop(image)
20     image = tf.image.random_flip_left_right(image)
21    return image

```

Listing 3.4: Função random_jitter

Para concluir o pré-processamento, foram definidas duas *parent functions* para incorporar as três funções mencionadas anteriormente.

```

22 def preprocess_image_train(image, label):
23     image = random_jitter(image)
24     image = normalize(image)
25    return image
26
27 def preprocess_image_test(image, label):
28     image = normalize(image)
29    return image

```

Listing 3.5: Parent Functions

É importante notar que o `random_jitter` é apenas aplicado ao conjunto de dados de treino para o modelo conseguir fazer um esforço extra para tentar resolver o problema proposto.

3.1.2 Mapping

Com o pré-processamento já definido, é preciso aplicá-lo ao conjunto de dados. Para este efeito, foi utilizada a função `map` para aplicar o processamento. A função `map` recebe dois parâmetros, o primeiro é a função que vai ser mapeada e o segundo é o *data sample*. Aqui também é feito um `shuffle` para que o pré-processamento seja aplicado de forma aleatória à *data sample*.

```
30 train_monet = train_monet.map(
31     preprocess_image_train, num_parallel_calls=AUTOTUNE).cache().shuffle(
32     BUFFER_SIZE).batch(1)
33
34 train_photo = train_photo.map(
35     preprocess_image_train, num_parallel_calls=AUTOTUNE).cache().shuffle(
36     BUFFER_SIZE).batch(1)
37
38 test_monet = test_monet.map(
39     preprocess_image_test, num_parallel_calls=AUTOTUNE).cache().shuffle(
40     BUFFER_SIZE).batch(1)
41
42 test_photo = test_photo.map(
43     preprocess_image_test, num_parallel_calls=AUTOTUNE).cache().shuffle(
44     BUFFER_SIZE).batch(1)
```

Listing 3.6: Mapping

3.1.3 Visualização de Imagens

Para confirmar se os dados foram carregados corretamente e se o pré-processamento também funcionou corretamente, foram verificadas as imagens da seguinte forma:

```
45 plt.subplot(121)
46 plt.title('Train_monet')
47 plt.imshow(sample_monet[0] * 0.5 + 0.5)
48
49 plt.subplot(122)
50 plt.title('Train_monet with random jitter')
51 plt.imshow(random_jitter(sample_monet[0]) * 0.5 + 0.5)
```

Listing 3.7: Imagens do train_monet

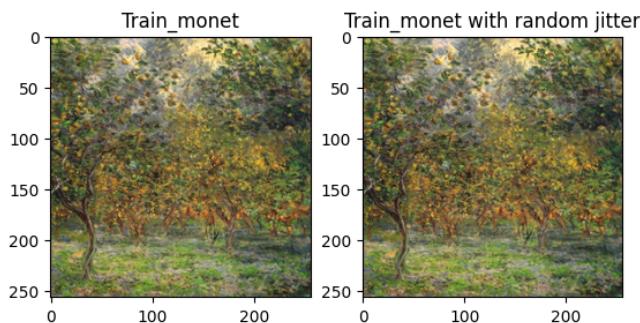


Figura 3.1: Imagens do train_monet: output

```
52 plt.subplot(121)
53 plt.title('Train_photo')
54 plt.imshow(sample_photo[0] * 0.5 + 0.5)
55
56 plt.subplot(122)
57 plt.title('Train_photo with random jitter')
58 plt.imshow(random_jitter(sample_photo[0]) * 0.5 + 0.5)
```

Listing 3.8: Imagens do train_photo

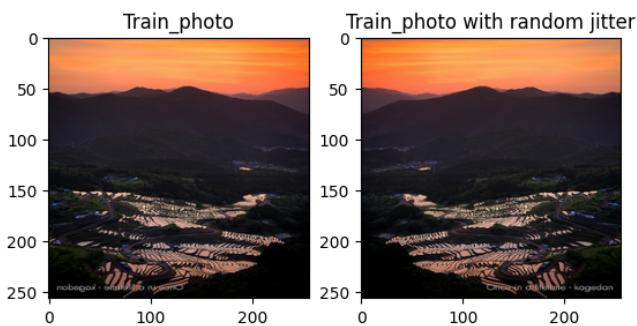


Figura 3.2: Imagens do train_photo: output

Capítulo 4

Generator & Discriminator

4.0.1 Generator

Portanto, a arquitetura do **generator** é dividida entre um *encoder* e um *decoder* que são bastante similares a uma arquitetura de *U-Net* modificada:

1. Cada bloco do *encoder* terá as seguintes *layers*: **Convultion layer**, **Batchnorm layer** e **Leaky ReLU layer**;
2. Cada bloco do *decoder* terá as seguintes *layers*: **Transposed Convolution layer**, **Batchnorm layer**, **Dropout layer**(somente os para os primeiros 3 blocos para evitar *overfitting*) e **ReLU layer**;
3. Um *encoder* é usado para *Downsampling* e para compreender a essencia da imagem. Contrariamente, o *decoder* é usado para *Upsampling* da imagem proveniente da imagem *encoded*;
4. Em adição, existem *skip connections* entre o *encoder* e *decoder* similar ao *U-Net*.

4.0.2 Discriminator

1. O *discriminator* é um *PatchGAN* em que cada *patch* de 30×30 de uma classe de saída classifica uma porção de 70×70 de uma imagem de entrada.
2. Cada bloco no *discriminator* terá as seguintes *layers*: **Convolution layer**, **BatchNorm layer**, **Leaky ReLU layer** e a forma de saída após a última camada é *batch_size* $30 \times 30 \times 1$.
3. Portanto, o *Discriminator* possui dois parâmetros:
 - A imagem de entrada e a imagem alvo, que o *Discriminator* deve classificar como real;
 - A imagem de entrada e a imagem gerada, que o *Discriminator* deve classificar como falsa.

No caso de uma *Cycle GAN*, usa-se normalização por instância em vez de normalização por *batch* (*batch normalization*), e também pode-se usar um gerador baseado em *ResNet* em vez de *U-Net*. Uma *Cycle GAN* requer dois *Generators* (g e f) e dois *Discriminators* (x e y) a serem treinados.

- O *Generator* g aprende a converter a imagem X em Y.
- O *Generator* f aprende a converter a imagem Y em X.
- O *Discriminator* x aprende a diferenciar entre a imagem original X e a imagem gerada X ($f(Y)$).
- O *Discriminator* y aprende a diferenciar entre a imagem original Y e a imagem gerada Y ($g(X)$).

Com isto, foi utilizado os *generators* e *discriminators* do *Tensorflow* que se encontram implementados no *pix2pix*.

```

59 generator_g = pix2pix.unet_generator(OUTPUT_CHANNELS, norm_type='instancenorm')
60 generator_f = pix2pix.unet_generator(OUTPUT_CHANNELS, norm_type='instancenorm')
61
62 discriminator_x = pix2pix.discriminator(norm_type='instancenorm', target=False)
63 discriminator_y = pix2pix.discriminator(norm_type='instancenorm', target=False)

```

Listing 4.1: Generator e Discriminator

Feito isto, e por razões de teste, foi testada a geração de imagens sem treinar o modelo.

```

64 to_photo = generator_g(sample_monet)
65 to_monet = generator_f(sample_photo)
66
67 plt.figure(figsize=(8,8))
68 contrast = 8
69
70 imgs = [sample_monet, to_photo, sample_photo, to_monet]
71 title = ['Monet', 'To Photo', 'Photo', 'To Monet']
72
73 for i in range(len(imgs)):
74     plt.subplot(2, 2, i+1)
75     plt.title(title[i])
76     if i % 2 == 0:
77         plt.imshow(imgs[i][0] * 0.5 + 0.5)
78     else:
79         plt.imshow(imgs[i][0] * 0.5 * contrast + 0.5)
80 plt.show()

```

Listing 4.2: Untrained Generator

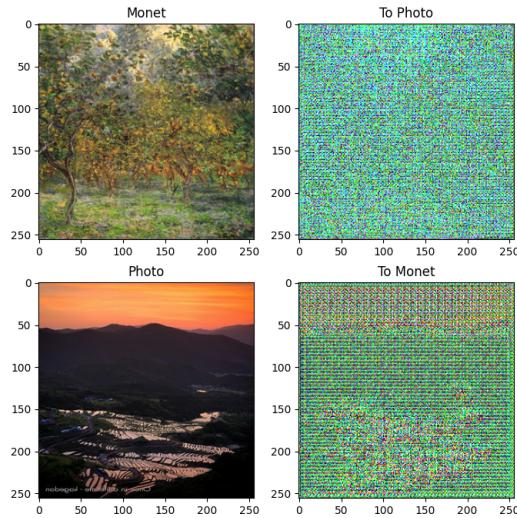


Figura 4.1: Imagens geradas pelo *Generator* sem qualquer tipo de treino.

```

81 plt.figure(figsize=(8,8))
82
83 plt.subplot(121)
84 plt.title('Is it a real photo?')
85 plt.imshow(discriminator_y(sample_photo)[0, ..., -1], cmap='inferno')
86
87 plt.subplot(122)
88 plt.title('Is it a real monet painting?')
89 plt.imshow(discriminator_x(sample_monet)[0, ..., -1], cmap='inferno')
90
91 plt.show()

```

Listing 4.3: Untrained Discriminator

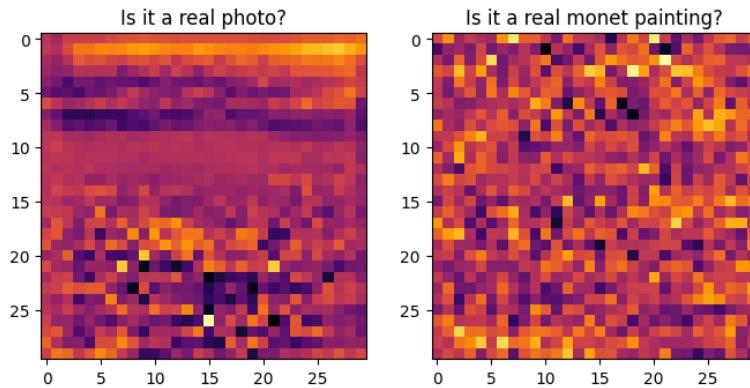


Figura 4.2: Imagens geradas pelo *Discriminator* sem qualquer tipo de treino.

Portanto, dá para observar que sem qualquer tipo de treino apenas é possível obter um *noise* que à primeira vista parece aleatório. Para resolver isto é necessário desenvolver funções de *loss* para conseguir obter um *output* adequado.

Capítulo 5

Loss Functions

5.1 Generator loss

Aqui, é calculada uma perda de entropia cruzada sigmoidal das imagens geradas e uma matriz de uns, para permitir que as imagens geradas se tornem estruturalmente semelhantes às imagens alvo.

5.2 Discriminator loss

Esta função recebe dois *inputs*, por exemplo, recebe as imagens reais e as imagens geradas.

- A `real_loss` é uma perda de entropia cruzada sigmoidal das imagens reais e uma matriz de uns (porque são imagens reais), e a perda `generated_loss` é uma perda de entropia cruzada sigmoidal das imagens geradas e uma matriz de zeros (porque são imagens falsas).
- a `total_loss` é a soma da `real_loss` e da `generated_loss`.

```
92 def discriminator_loss(real, generated):
93     real_loss = loss_obj(tf.ones_like(real), real)
94
95     generated_loss = loss_obj(tf.zeros_like(generated), generated)
96
97     total_disc_loss = real_loss + generated_loss
98
99     return total_disc_loss * 0.5
100
101 def generator_loss(generated):
102     return loss_obj(tf.ones_like(generated), generated)
```

Listing 5.1: Discriminator e Generator loss

Portanto, agora o *Discriminator* consegue diferenciar se a saída é um quadro ou uma fotografia, mas não pode determinar a similaridade entre as imagens de entrada e as imagens geradas.

Para resolver esse problema, será usada uma perda de reconstrução. Com isto, o que é uma perda de reconstrução?

Supõem-se que se traduziu uma frase de inglês para português. Agora, quando se converte de volta a frase em português para inglês, ela deve ser a mesma que a declaração inicial. Isso é chamado de **Cycle consistency**.

5.3 Reconstruction Loss

- Supondo que a imagem X seja passada pelo *generator* g , que produz a imagem gerada \hat{Y} .
- A imagem gerada \hat{Y} é passada pelo *generator* f , que produz a imagem reconstruída \hat{X} .
- A perda de reconstrução agora é calculada como a média do erro absoluto entre X e \hat{X} (a diferença entre a imagem original e a imagem reconstruída).

Aqui, é necessário um novo parâmetro chamado `lambda`, que pode ser ajustado e determina o quanto similar a saída é em comparação com a entrada.

```
103 def calc_cycle_loss(real_img, cycled_img):  
104     loss = tf.reduce_mean(tf.abs(real_img - cycled_img))  
105  
106     return LAMBDA * loss
```

Listing 5.2: Reconstuction loss

5.4 Identity loss

Posteriormente, é necessário definir a **Identity loss**. O *generator* g é responsável por traduzir a imagem X para Y . A perda de identidade é quando a imagem Y é alimentada para o *generator* g , ele deve produzir a imagem real Y ou algo semelhante à imagem Y .

```
108 def identity_loss(real_image, same_image):  
109     loss = tf.reduce_mean(tf.abs(real_image - same_image))  
110     return LAMBDA * 0.5 * loss
```

Listing 5.3: Identity loss

Capítulo 6

Treino

Portanto, após esta preparação do modelo é necessário treiná-lo. Com isto, o treino que foi utilizado para este problema consiste em 4 passos cruciais:

1. Determinar as previsões
2. Calcular a perda
3. Calcular os gradientes
4. Aplicar os gradientes aos otimizadores

```
111 @tf.function
112 def get_predictions(real_x, real_y):
113
114     with tf.GradientTape(persistent=True) as tape:
115
116         fake_y = generator_g(real_x, training=True)
117         cycled_x = generator_f(fake_y, training=True)
118
119         fake_x = generator_f(real_y, training=True)
120         cycled_y = generator_g(fake_x, training=True)
121
122     # same_x and same_y are used for identity loss.
123     same_x = generator_f(real_x, training=True)
124     same_y = generator_g(real_y, training=True)
125
126     disc_real_x = discriminator_x(real_x, training=True)
127     disc_real_y = discriminator_y(real_y, training=True)
128
129     disc_fake_x = discriminator_x(fake_x, training=True)
130     disc_fake_y = discriminator_y(fake_y, training=True)
```

Listing 6.1: Funções ilustrativas dos passos de treino: get_predictions

```

131 @tf.function
132 def get_loss(disc_fake_x, disc_fake_y, real_x, real_y, cycled_x, cycled_y, same_x
133   , same_y):
134
135   gen_g_loss = generator_loss(disc_fake_y)
136   gen_f_loss = generator_loss(disc_fake_x)
137
138   total_cycle_loss = calc_cycle_loss(real_x, cycled_x) + calc_cycle_loss(real_y
139   , cycled_y)
140
141   total_gen_g_loss = gen_g_loss + total_cycle_loss + identity_loss(real_y,
142   same_y)
143   total_gen_f_loss = gen_f_loss + total_cycle_loss + identity_loss(real_x,
144   same_x)
145
146   disc_x_loss = discriminator_loss(disc_real_x, disc_fake_x)
147   disc_y_loss = discriminator_loss(disc_real_y, disc_fake_y)
148
149   disc_real_x = discriminator_x(real_x, training=True)
150   disc_real_y = discriminator_y(real_y, training=True)
151
152   disc_fake_x = discriminator_x(fake_x, training=True)
153   disc_fake_y = discriminator_y(fake_y, training=True)

```

Listing 6.2: Funções ilustrativas dos passos de treino: get_loss

```

150 @tf.function
151 def get_gradients(total_gen_g_loss, generator_g, total_gen_f_loss, generator_f,
152   disc_x_loss, discriminator_x, disc_y_loss, discriminator_y):
153
154   generator_g_gradients = tape.gradient(total_gen_g_loss,
155                                         generator_g.trainable_variables)
156   generator_f_gradients = tape.gradient(total_gen_f_loss,
157                                         generator_f.trainable_variables)
158
159   discriminator_x_gradients = tape.gradient(disc_x_loss,
160                                             discriminator_x.trainable_variables)
161   discriminator_y_gradients = tape.gradient(disc_y_loss,
162                                             discriminator_y.trainable_variables)
163
164   disc_fake_x = discriminator_x(fake_x, training=True)
165   disc_fake_y = discriminator_y(fake_y, training=True)

```

Listing 6.3: Funções ilustrativas dos passos de treino: get_gradients

```

165 generator_g_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
166 generator_f_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
167
168 discriminator_x_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
169 discriminator_y_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
170
171 (...)

172
173 @tf.function
174 def grad_optimizer(generator_g_gradients, generator_f_gradients, generator_g,
175     generator_f, discriminator_x_gradients, discriminator_x,
176     discriminator_y_gradients, discriminator_y):
177
178     generator_g_optimizer.apply_gradients(zip(generator_g_gradients,
179                                             generator_g.trainable_variables))

180     generator_f_optimizer.apply_gradients(zip(generator_f_gradients,
181                                             generator_f.trainable_variables))

182     discriminator_x_optimizer.apply_gradients(zip(discriminator_x_gradients,
183                                                 discriminator_x.
184                                                 trainable_variables))

185     discriminator_y_optimizer.apply_gradients(zip(discriminator_y_gradients,
186                                                 discriminator_y.
187                                                 trainable_variables))

```

Listing 6.4: Funções ilustrativas dos passos de treino: grad_optimizer

Para verificar se o treino correu da melhor forma, foi desenvolvida a seguinte função que coloca lado a lado a imagem *input* e a imagem que foi gerada em cada época.

```

187 def generate_images(model, test_input):
188     prediction = model(test_input)
189
190     plt.figure(figsize=(12, 12))
191
192     display_list = [test_input[0], prediction[0]]
193     title = ['Input Image', 'Predicted Image']
194
195     for i in range(2):
196         plt.subplot(1, 2, i+1)
197         plt.title(title[i])
198         plt.imshow(display_list[i] * 0.5 + 0.5)
199         plt.axis('off')
200     plt.show()

```

Listing 6.5: Função generate_images

Combinando todos os passos de treino, referidos anteriormente, obteve-se a seguinte função representativa do *loop* principal de treino do modelo desenvolvido.

```

201 @tf.function
202 def train_loop(real_x, real_y):
203

```

```

204     with tf.GradientTape(persistent=True) as tape:
205
206         fake_y = generator_g(real_x, training=True)
207         cycled_x = generator_f(fake_y, training=True)
208
209         fake_x = generator_f(real_y, training=True)
210         cycled_y = generator_g(fake_x, training=True)
211
212         same_x = generator_f(real_x, training=True)
213         same_y = generator_g(real_y, training=True)
214
215         disc_real_x = discriminator_x(real_x, training=True)
216         disc_real_y = discriminator_y(real_y, training=True)
217
218         disc_fake_x = discriminator_x(fake_x, training=True)
219         disc_fake_y = discriminator_y(fake_y, training=True)
220
221         gen_g_loss = generator_loss(disc_fake_y)
222         gen_f_loss = generator_loss(disc_fake_x)
223
224         total_cycle_loss = calc_cycle_loss(real_x, cycled_x) + calc_cycle_loss(real_y,
225             , cycled_y)
226
227         total_gen_g_loss = gen_g_loss + total_cycle_loss + identity_loss(real_y,
228             same_y)
229         total_gen_f_loss = gen_f_loss + total_cycle_loss + identity_loss(real_x,
230             same_x)
231
232         disc_x_loss = discriminator_loss(disc_real_x, disc_fake_x)
233         disc_y_loss = discriminator_loss(disc_real_y, disc_fake_y)
234
235         generator_g_gradients = tape.gradient(total_gen_g_loss,
236                                         generator_g.trainable_variables)
237         generator_f_gradients = tape.gradient(total_gen_f_loss,
238                                         generator_f.trainable_variables)
239
240         discriminator_x_gradients = tape.gradient(disc_x_loss,
241                                         discriminator_x.trainable_variables)
242         discriminator_y_gradients = tape.gradient(disc_y_loss,
243                                         discriminator_y.trainable_variables)
244
245         generator_g_optimizer.apply_gradients(zip(generator_g_gradients,
246                                         generator_g.trainable_variables))
247
248         generator_f_optimizer.apply_gradients(zip(generator_f_gradients,
249                                         generator_f.trainable_variables))
250
251         discriminator_x_optimizer.apply_gradients(zip(discriminator_x_gradients,
252                                         discriminator_x.
253             trainable_variables))
254
255         discriminator_y_optimizer.apply_gradients(zip(discriminator_y_gradients,
256                                         discriminator_y.
257             trainable_variables))

```

```
trainable_variables))
```

Listing 6.6: Função train_loop

Feito isto, inicia-se finalmente o treino do modelo.

```
255 for epoch in range(EPOCHS):
256     start = time.time()
257
258     n = 0
259     for image_x, image_y in tf.data.Dataset.zip((train_monet, train_photo)):
260         train_loop(image_x, image_y)
261         if n % 10 == 0:
262             print ('.', end='')
263         n += 1
264
265     clear_output(wait=True)
266
267     generate_images(generator_g, sample_monet)
268
269     if (epoch + 1) % 10 == 0:
270         ckpt_save_path = ckpt_manager.save()
271         print ('Saving checkpoint')
```

Listing 6.7: Treino do modelo

Capítulo 7

Resultados obtidos

Assim sendo, nesta versão do trabalho prático foi primeiro efetuado um treino com 25 épocas. Foram realizados treinos com 5 e 10 épocas, mas devido ao curto período de aprendizagem o modelo não era capaz de divergir o suficiente da imagem original. Já a partir de 25 épocas já se começava a notar alguma transformação da imagem numa fotografia.



Figura 7.1: Resultado com 25 épocas de treino na imagem *sample*.

Após uma breve análise, dá para reparar que a imagem possui um nível de realismo mais elevado do que a imagem original. Aqui, com 25 épocas, o modelo consegue simular a neblina da cidade, com base nas fotos do *train_photo* e começa a melhorar o realismo da água. É necessário relembrar, que o objetivo deste modelo é transformar pinturas em imagens realistas representativas do mundo real. Posteriormente, foi aumentado o número de épocas para um total de 85 obtendo-se seguintes resultados.

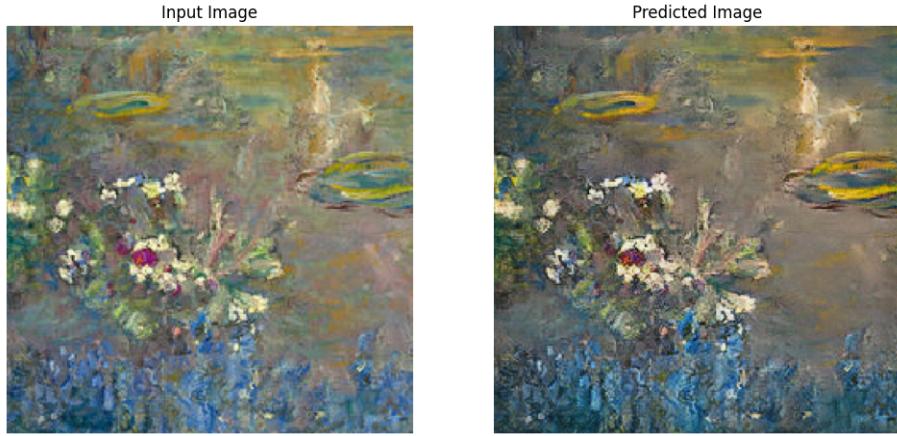


Figura 7.2: Resultado com 85 épocas de treino na imagem *sample*.

Aqui a imagem *sample* é diferente, mas de qualquer das formas dá para analisar os resultados com base no que foi dito anteriormente. Nesta fase do treino o modelo é capaz de transformar os efeitos de luz na água, da imagem original, para uma versão que se aproxima bastante da realidade. Porém, pode-se afirmar que quanto mais estilizado o traço da forma é, mais difícil é para o modelo a converter. Por exemplo, as flores no quadro são bastante estilizadas, isto pode ser devido às técnicas que o artista utilizou, sendo que o modelo ainda não as consegue transformar em flores realistas.



Figura 7.3: Resultado com 185 épocas de treino na imagem *sample*.

Com 185 épocas, o modelo consegue prever quase com exatidão a água presente na pintura. Também já começa a conseguir transformar os lírios de forma a que fiquem realistas.

Com isto, deu para perceber que de forma a obter resultados super-realistas é necessário fazer um pouco de “*brute-force*” e correr o modelo com um número exorbitante de épocas de treino. Pode-se dizer que quanto mais tempo de treino mais tempo o modelo tem para analisar as fotografias, aprender a essência delas e começar a converter as pinturas com tudo o que aprendeu das fotografias. Portanto, com o treino feito pode-se testar a *GAN* com o *testset* para verificar o desempenho que este obteve.

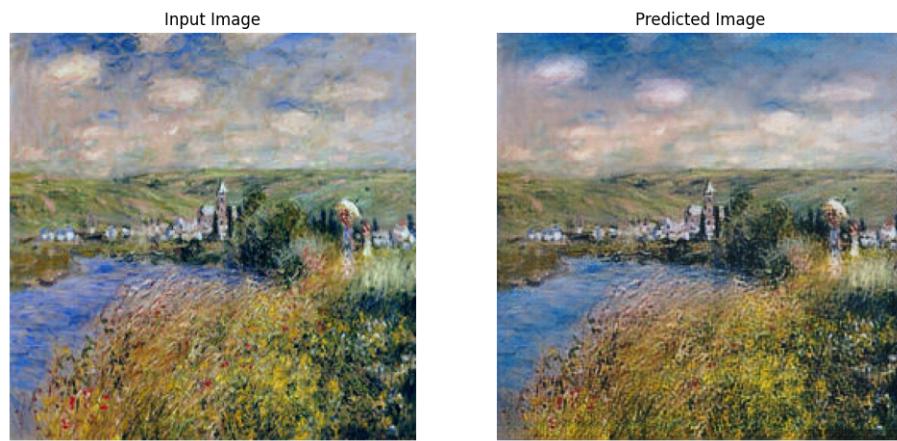


Figura 7.4: Resultado das previsões no *testset* com o modelo treinado com 25 épocas.



Figura 7.5: Resultado das previsões no *testset* com o modelo treinado com 85 épocas.



Figura 7.6: Resultado das previsões no *testset* com o modelo treinado com 185 épocas.

Aqui foram escolhidos os melhores resultados, basicamente nesta parte das previsões escolheu-se dar *display* 10 imagens e foram retiradas as melhores conversões. Pode-se observar que o mesmo que foi mencionado acima, quanto mais épocas de treino melhor o modelo é a transformar as pinturas. É de realçar a figura 7.6 que foi das capturas que mais se aproximou à realidade possuindo um elevado realismo no céu, no mar e no terreno em si pelo que já começa a parecer uma fotografia tirada de um telemóvel. Portanto, conclui-se que nesta versão do modelo é necessário um treino extensivo dos dados para conseguir gerar resultados ainda melhores dos que foram obtidos e apresentados neste documento.

Capítulo 8

Conclusões e análise

Ao longo deste trabalho prático, explorou-se as *Generative Adversarial Networks* capazes de gerar imagens novas com base em imagens pré-existentes. Como referido no início, este trabalho tinha como objetivo transformar as pinturas de *Claude Monet* em versões realistas como fotografias. Aqui, consideram-se os resultados obtidos como satisfatórios, apesar de ser necessário usar números enormes de épocas de treino para atingir resultados bastante próximos da realidade.

De forma a explorar soluções melhores, e como trabalho futuro, pode ser interessante explorar técnicas diferentes de geração de imagens como as *VAE* e *WGAN* ou até mesmo *StyleGan* e até mesmo construir funções de *loss* novas para verificar se o desempenho do modelo é beneficiado ou não. Também podem ser exploradas formas diferentes de pré-processamento das imagens como a inserção de rotações ou até mesmo alterações da tonalidade das mesmas o que iria reforçar ainda mais o conjunto de dados e melhorar a geração das novas imagens.