

TP RNII - Style Transfer - Juan Francisco Sbruzzi

Al final terminé reacomodando las imágenes en el repositorio de manera poco óptima, así que elimino la parte de descargarlas.

```
In [ ]: from tensorflow.python.framework.ops import disable_eager_execution
disable_eager_execution()
from tensorflow.keras import backend as K

from tensorflow.keras.preprocessing.image import load_img, save_img, img_to_array
from tensorflow.keras import layers
import numpy as np
from scipy.optimize import fmin_l_bfgs_b
import time
import argparse
import matplotlib.pyplot as plt

from tensorflow.keras.applications import vgg19
from pathlib import Path
```

```
In [ ]: # Definimos las imágenes que vamos a utilizar, y el directorio de salida

base_image_path = Path("./grito-pajaro/output_1000_10000_1/output_at_iteration_195.png")
style_reference_image_path = Path("./grito-pajaro/style.jpg")
result_prefix = Path("./grito-pajaro/output_1000_10000_1_cont")
iterations = 400
```

(1) ¿Qué significan los parámetros definidos en la siguiente celda?

`style_weight` se refiere al peso del estilo de la imagen de referencia, lo que en el paper es denominado β , cuanto mayor sea este valor relativo respecto a α más se priorizará el estilo durante la optimización.

`content_weight` se refiere al peso del contenido de la imagen original, lo que en el paper es denominado α , cuanto mayor sea este valor relativo respecto a β más se priorizará el contenido de la imagen original.

`total_variation_weight` se hablará más en detalle cuando veamos la variation loss, pero en principio es el peso de un término que regulariza y suaviza el resultado final, penalizando variaciones bruscas entre pixeles aledaños.

```
In [ ]: total_variation_weight = 10 # peso de la variation loss, regulariza
style_weight = 10000 # peso del estilo, beta
content_weight = 1 # peso del contenido, alpha
```

```
In [ ]: # Definimos el tamaño de las imágenes a utilizar
width, height = load_img(base_image_path).size
img_nrows = 400
img_ncols = int(width * img_nrows / height)
```

(2) Explicar qué hace la siguiente celda. En especial las últimas dos líneas de la función antes del return. ¿Por qué?

```
img = load_img(image_path, target_size=(img_nrows, img_ncols))
```

Se carga la imagen desde el path especificado, escalándola al tamaño indicado por `target_size`. La imagen se carga en formato PIL.

```
img = img_to_array(img)
```

Se convierte la imagen a un array de numpy, tridimensional, con shape `(height, width, channels)`: ya que no se provee el argumento `data_format`, los canales ocupan el último lugar del array (ver [código fuente](#)).

```
img = np.expand_dims(img, axis=0)
```

Se agrega una dimensión al array en el índice cero. Esto se debe a que el modelo [utiliza capas Conv2D](#) de Keras que necesitan una entrada con shape `(batch_size, height, width, channels)`.

```
img = vgg19.preprocess_input(img)
```

Los modelos de Keras Application esperan que las entradas sean preprocesadas de una manera particular, definida para cada modelo. En el caso de VGG19, la transformación realizada es [convertir las imágenes de RGB a BGR y eliminar la media respecto al dataset de ImageNet en cada canal](#). Esto se implementó en general para todas las Keras Applications mediante una única función `_preprocess_numpy_input` definida en `imagenet_utils` que en el caso de VGG19 es llamada con el argumento `caffe`. En la [definición](#) se verifica el

comportamiento, es notable que las medias de ImageNet estén hardcodeadas en el código. También es notable que pareciera que esta función se puede llamar antes o después del `expand_dims`, en base a lo que veo de la implementación.

```
In [ ]: def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_nrows, img_ncols))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return img
```

(3) Habiendo comprendido lo que hace la celda anterior, explique de manera muy concisa qué hace la siguiente celda. ¿Qué relación tiene con la celda anterior?

Hace la operación inversa a la celda anterior, recupera la información de la imagen en el formato original:

- Recupera la shape original
- Restaura las medias eliminadas en el preprocess de VGG19 (del source code de `imagenet_utils`, `mean = [103.939, 116.779, 123.68]`, afortunadamente coincide con lo que se puso acá)
- Pasa el orden de los canales de BGR a RGB,
- Castea a `uint8` (con "saturación", mediante el `clip`) ya que en el medio se realizaron operaciones con floats (como sumar las medias, en esta celda).

```
In [ ]: def deprocess_image(x):
    x = x.reshape((img_nrows, img_ncols, 3))
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

```
In [ ]: # get tensor representations of our images
# K.variable convierte un numpy array en un tensor, para
base_image = K.variable(preprocess_image(base_image_path))
style_reference_image = K.variable(preprocess_image(style_reference_image_path))
```

```
In [ ]: combination_image = K.placeholder((1, img_nrows, img_ncols, 3))
```

```
In [ ]: # combine the 3 images into a single Keras tensor
input_tensor = K.concatenate([base_image,
                            style_reference_image,
                            combination_image], axis=0)
```

```
In [ ]: # build the VGG19 network with our 3 images as input
# the model will be loaded with pre-trained ImageNet weights
model = vgg19.VGG19(input_tensor=input_tensor,
                     weights='imagenet', include_top=False)
print('Model loaded.')

# get the symbolic outputs of each "key" layer (we gave them unique names).
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])

model.summary()
```

Model loaded.
Model: "vgg19"

Layer (type)	Output Shape	Param #
=====		
input_4 (InputLayer)	[3, 400, 533, 3]	0
block1_conv1 (Conv2D)	(3, 400, 533, 64)	1792
block1_conv2 (Conv2D)	(3, 400, 533, 64)	36928
block1_pool (MaxPooling2D)	(3, 200, 266, 64)	0
block2_conv1 (Conv2D)	(3, 200, 266, 128)	73856
block2_conv2 (Conv2D)	(3, 200, 266, 128)	147584
block2_pool (MaxPooling2D)	(3, 100, 133, 128)	0
block3_conv1 (Conv2D)	(3, 100, 133, 256)	295168
block3_conv2 (Conv2D)	(3, 100, 133, 256)	590080
block3_conv3 (Conv2D)	(3, 100, 133, 256)	590080
block3_conv4 (Conv2D)	(3, 100, 133, 256)	590080
block3_pool (MaxPooling2D)	(3, 50, 66, 256)	0
block4_conv1 (Conv2D)	(3, 50, 66, 512)	1180160
block4_conv2 (Conv2D)	(3, 50, 66, 512)	2359808
block4_conv3 (Conv2D)	(3, 50, 66, 512)	2359808
block4_conv4 (Conv2D)	(3, 50, 66, 512)	2359808
block4_pool (MaxPooling2D)	(3, 25, 33, 512)	0
block5_conv1 (Conv2D)	(3, 25, 33, 512)	2359808
block5_conv2 (Conv2D)	(3, 25, 33, 512)	2359808
block5_conv3 (Conv2D)	(3, 25, 33, 512)	2359808
block5_conv4 (Conv2D)	(3, 25, 33, 512)	2359808
block5_pool (MaxPooling2D)	(3, 12, 16, 512)	0
=====		
Total params:	20,024,384	
Trainable params:	20,024,384	
Non-trainable params:	0	

(4) En la siguientes celdas:

- ¿Qué es la matriz de Gram? ¿Para qué se usa?
- ¿Por qué se permutan las dimensiones de x?

La matriz de Gram es una matriz construida a partir de los productos punto de los feature maps correspondientes a una determinada capa de una CNN. De cierta manera, esto permite obtener una medida de la correlación entre los filtros de la capa. En el contexto de style transfer, la matriz de Gram se usa para capturar información sobre la textura de cierta imagen, ya que en vez de analizar la configuración espacial de las features se trabaja con la correlación. Lo que se hará luego es generar una imagen cuya matriz de Gram tenga la menor distancia posible (MS) a la matriz de Gram de la imagen de referencia de estilo/textura.

Razonamiento: el argumento que se le pasa a esta función, que calcula la matriz de Gram, es un tensor de shape `(height, width, filters)`. La permutación deja un tensor de shape `(filters, height, width)`. Al hacer `batch_flatten` se aplana todas las dimensiones excepto por la primera, de manera que `features` tiene shape `(filters, height*width)`. De esta manera, al hacer la multiplicación mediante `dot`, la matriz resultante es de dimensión `(filters, filters)`, que es lo que permite tener la correlación entre los distintos filtros de la capa, que es lo que queríamos. La permutación permite que el resultado esté dado como el producto de las componentes del feature map de la manera que especifica el paper:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

```
In [ ]: def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram
```

(5) Losses

Explicar qué mide cada una de las losses en las siguientes tres celdas.

- Style loss: mide la distancia (mean-squared) entre la matriz de Gram de la imagen siendo generada y la de la imagen de referencia. Es decir, mide qué tan distintos son los estilos de la imagen generada y la de referencia.
- Content loss: mide la distancia entre las features de la imagen original y las de la imagen generada, es decir, mide qué tan distinta es la información contenida en ambas.
- Variation loss: viendo lo que hace la función particularmente, veo que penaliza el hecho de que dos pixeles contiguos sean muy distintos, ya que se hace mayor cuanto mayor es la diferencia que haya, tanto en x como en y, a una distancia de 1 pixel. Por lo tanto, mide qué tan "discontinua" es la imagen pixel a pixel, *infiero* que para penalizar cambios bruscos en los valores de pixeles y suavizar el resultado final.

```
In [ ]: def style_loss(style, combination):
    assert K.ndim(style) == 3
    assert K.ndim(combination) == 3
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return K.sum(K.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))
```

```
In [ ]: def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

```
In [ ]: def total_variation_loss(x):
    assert K.ndim(x) == 4
    a = K.square(
        x[:, :img_nrows - 1, :img_ncols - 1, :] - x[:, 1:, :img_ncols - 1, :])
    b = K.square(
        x[:, :img_nrows - 1, :img_ncols - 1, :] - x[:, :img_nrows - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

```
In [ ]: # Armamos la loss total
loss = K.variable(0.0)
layer_features = outputs_dict['block5_conv2']
base_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss = loss + content_weight * content_loss(base_image_features,
                                              combination_features)

feature_layers = ['block1_conv1', 'block2_conv1',
                  'block3_conv1', 'block4_conv1',
                  'block5_conv1']
for layer_name in feature_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    loss = loss + (style_weight / len(feature_layers)) * sl
loss = loss + total_variation_weight * total_variation_loss(combination_image)
```

```
In [ ]: grads = K.gradients(loss, combination_image)

outputs = [loss]
if isinstance(grads, (list, tuple)):
    outputs += grads
else:
    outputs.append(grads)
```

```
f_outputs = K.function([combination_image], outputs)
```

(6) Explique el propósito de las siguientes tres celdas

¿Qué hace la función fmin_l_bfgs_b? ¿En qué se diferencia con la implementación del paper? ¿Se puede utilizar alguna alternativa?

La función `fmin_l_bfgs_b` implementa un optimizador BFGS (Broyden–Fletcher–Goldfarb–Shanno) con menor huella de memoria que BFGS, necesario debido a la enorme cantidad de variables del problema. BFGS es una optimización que se basa en estimar la curvatura del gradiente con las derivadas de segundo orden, efectivamente haciendo una estimación de algo que se llama "matriz hessiana", siendo un método "de segundo orden". La implementación del paper menciona particularmente "gradient descent", con lo que entiendo que utilizaron SGD, que sería un método de "primer orden" ya que trabaja directamente con el gradiente.

En la página de [PapersWithCode](#) del paper investigué los algoritmos de optimización más utilizados en las implementaciones de este estilo. Resulta que casi todos los más "starreados" en github permiten elegir entre la utilización de lbfsgs o adam, o usan directamente uno de estos dos. En particular, adam es un algoritmo de "primer orden" pero utiliza información pasada del gradiente para llegar a una performance similar con un costo computacional que puede ser menor, por lo que tiene sentido de que dependiendo del tamaño de los datos a tener en cuenta sea conveniente uno u otro.

Sobre las celdas en particular: ya que se usa la implementación de scipy de lbfsgs, se define una clase que permite calcular gradientes y losses en una sola pasada pero que se le pueda pasar a scipy como funciones separadas, de manera que se aprovechen las optimizaciones de la librería, luego se itera varias veces sobre el algoritmo de optimización para obtener resultados en busca de un mínimo en las losses. La mejor salida final sería la obtenida al terminar la iteración.

```
In [ ]: def eval_loss_and_grads(x):
    x = x.reshape((1, img_nrows, img_ncols, 3))
    outs = f_outputs([x])
    loss_value = outs[0]
    if len(outs[1:]) == 1:
        grad_values = outs[1].flatten().astype('float64')
    else:
        grad_values = np.array(outs[1:]).flatten().astype('float64')
    return loss_value, grad_values

# this Evaluator class makes it possible
# to compute Loss and gradients in one pass
# while retrieving them via two separate functions,
# "loss" and "grads". This is done because scipy.optimize
# requires separate functions for loss and gradients,
# but computing them separately would be inefficient.
```

```
In [ ]: class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        loss_value, grad_values = eval_loss_and_grads(x)
        self.loss_value = loss_value
        self.grads_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grads_values)
        self.loss_value = None
        self.grads_values = None
        return grad_values
```

(7) Ejecute la siguiente celda y observe las imágenes de salida en cada iteración.

```
In [ ]: evaluator = Evaluator()

# run scipy-based optimization (L-BFGS) over the pixels of the generated image
# so as to minimize the neural style loss
x = preprocess_image(base_image_path)
```

```

save_img(result_prefix / 'output_at_iteration_0_base.png', deprocess_image(preprocess_image(base_image_path)))
save_img(result_prefix / 'output_at_iteration_0_style.png', deprocess_image(preprocess_image(style_reference_image_path)))

for i in range(iterations + 1):
    print('Start of iteration', i)
    start_time = time.time()
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x.flatten(),
                                       fprime=evaluator.grads, maxfun=20)
    print('Current loss value:', min_val)
    # save current generated image
    img = deprocess_image(x.copy())
    fname = result_prefix / ('output_at_iteration_%d.png' % i)
    end_time = time.time()
    if(i % 10 == 0):
        save_img(fname, img)
        print('Image saved as', fname)
    print('Iteration %d completed in %ds' % (i, end_time - start_time))

```

```

In [ ]: import os, re

def natural_sort(l):
    convert = lambda text: int(text) if text.isdigit() else text.lower()
    alphanum_key = lambda key: [convert(c) for c in re.split('([0-9]+)', key)]
    return sorted(l, key=alphanum_key)

def print_iterations(folder, columns=3, figw=5, figh=4):
    extensions = ['webp', 'jpg', 'jpeg']
    img_paths = natural_sort(os.listdir(folder))
    amount = len(img_paths)
    contains_base = False
    contains_style = False
    for imgname in os.listdir(folder):
        if('base' in imgname):
            contains_base = True
            amount = amount - 1
            img_paths.remove('output_at_iteration_0_base.png')
        if('style' in imgname):
            contains_style = True
            amount = amount - 1
            img_paths.remove('output_at_iteration_0_style.png')
    amount_total = amount + 2
    rows = np.ceil(amount_total/columns).astype(np.uint16)
    fig, ax = plt.subplots(rows, columns, figsize=(columns*figw, rows*figh))
    if(not contains_style):
        parent_paths = os.listdir(os.path.dirname(folder))
        for ext in extensions:
            if(f'style.{ext}' in parent_paths):
                style_path = f'style.{ext}'
        style_img = deprocess_image(preprocess_image(os.path.join(os.path.abspath(os.path.dirname(folder)), style_path)))
    else:
        style_img = load_img(os.path.join(os.path.abspath(folder), 'output_at_iteration_0_style.png'))
    if(not contains_base):
        parent_paths = os.listdir(os.path.dirname(folder))
        for ext in extensions:
            if(f'base.{ext}' in parent_paths):
                base_path = f'base.{ext}'
        base_img = deprocess_image(preprocess_image(os.path.join(os.path.abspath(os.path.dirname(folder)), base_path)))
    else:
        base_img = load_img(os.path.join(os.path.abspath(folder), 'output_at_iteration_0_base.png'))

    for i in range(rows*columns):
        if(len(ax.shape) > 1):
            curr_ax = ax[int(i/columns)][i%columns]
        else:
            curr_ax = ax[i]
        if(i == 0):
            curr_ax.imshow(style_img)
            curr_ax.set_title('style')
        if(i == 1):
            curr_ax.imshow(base_img)
            curr_ax.set_title('base')
        if(i > 1 and i < amount_total):
            curr_ax.imshow(load_img(os.path.join(os.path.abspath(folder), img_paths[i-2])))
            curr_ax.set_title(img_paths[i-2])
        curr_ax.axis('off')

```

```
In [ ]: print_iterations('estrellada-neckarfront/output')
```



(8) Generar imágenes para distintas combinaciones de pesos de las losses. Explicar las diferencias. (Adjuntar las imágenes generadas como archivos separados.)

En todos los casos, la generación se almacenó en una carpeta llamada `output_{variation}_{beta}_{alpha}`, no voy a aclarar con títulos cada muestra porque se embarraba la notebook, así que hay que ver el nombre de la carpeta que printeo. En el repositorio dejé los resultados más notables junto con algunas iteraciones para ver el avance de la optimización.

Inicialmente, el resultado no me convenció (probablemente en parte se debió a no dejar terminar la optimización ya que estaba llevando mucho tiempo en Colab, eventualmente pasé a hacerlo localmente y fue más rápido). Por esto, y viendo los resultados del paper, disminuí la relación β/α .

```
In [ ]: print_iterations('estrellada-neckarfront/output_0.1_100_1')
```

(5, 3)



output_at_iteration_1.png



output_at_iteration_2.png



output_at_iteration_3.png



output_at_iteration_4.png



output_at_iteration_5.png



output_at_iteration_6.png



output_at_iteration_7.png



output_at_iteration_8.png



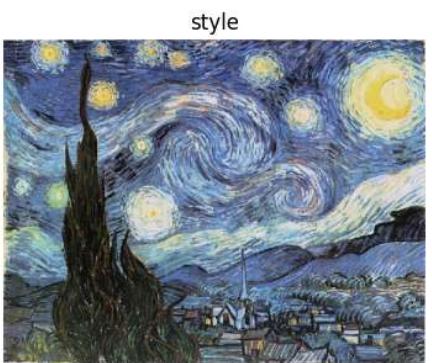
output_at_iteration_9.png



output_at_iteration_10.png



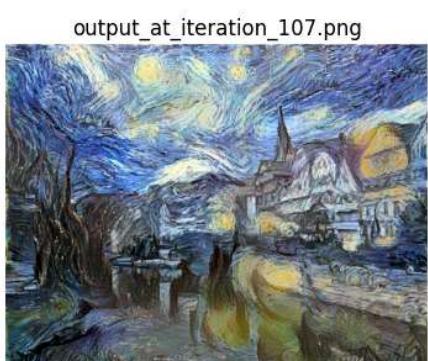
In []: `print_iterations('estrellada-neckarfront/output_0.1_1000_1')`



In []: `print_iterations('estrellada-neckarfront/output_1_1000_1')`



```
In [ ]: print_iterations('estrellada-neckarfront/output_0.1_10000_1')
```



Conclusiones

Primero me engañé por estar entrenando poco debido a una restricción temporal imaginaria que tenía en mi cabeza. Eventualmente me acordé de que no hace usar siempre Colab y que tampoco hace falta apagar la computadora a la noche, con lo que pude dejar corriendo varias "tiradas" y obtener buenos resultados, como se verá en el siguiente punto. Sobre las imágenes originales lo que se puede notar es:

- Al elevar la relación entre estilo sobre contenido, efectivamente se pierde más información de la imagen original. Esto lo noto especialmente en el remolino que aparece más marcado y deforma el límite entre la construcción y el agua, al priorizar el estilo este remolino difumina ese límite irrecuperablemente, mientras que al reducir la relación se mantiene una línea distintiva marcada. Esto se nota incluso en etapas comunes para todos los entrenamientos
- No creo que sea perceptible en la notebook, con lo que recomiendo abrir `output_1_1000_1/output_at_iteration_20.png` y `output_0.1_1000_1/output_at_iteration_14.png` y hacer zoom para notar la diferencia al elevar la variation loss. Claramente los pixeles terminan teniendo un color un poco más homogéneo, con menos outliers y artefactos.

```
In [ ]: fig, ax = plt.subplots(1, 2, figsize=(11,5))
var_weight_low = load_img('estrellada-neckarfront/output_0.1_1000_1/output_at_iteration_14.png').crop((200, 200, 300, 300))
var_weight_high = load_img('estrellada-neckarfront/output_1_1000_1/output_at_iteration_20.png').crop((200, 200, 300, 300))
ax[0].imshow(var_weight_low)
ax[0].axis('off')
ax[0].set_title('Variation weight = 0.1')
ax[1].imshow(var_weight_high)
ax[1].axis('off')
ax[1].set_title('Variation weight = 1')
```

Out[]: Text(0.5, 1.0, 'Variation weight = 1')

Variation weight = 0.1



Variation weight = 1

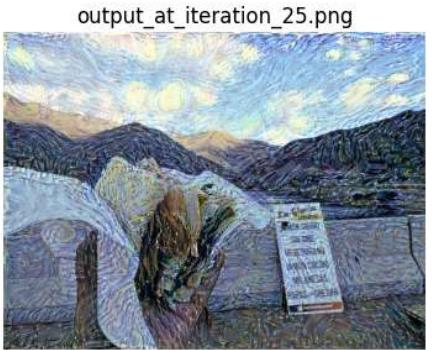
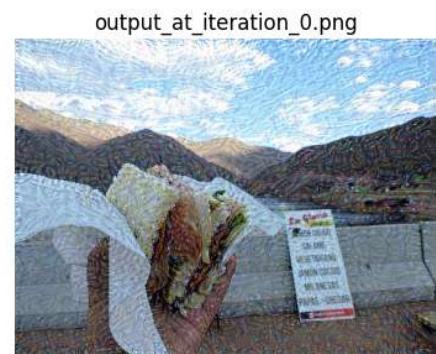


(9) Cambiar las imágenes de contenido y estilo por unas elegidas por usted. Adjuntar el resultado.

Voy a marcar algunos de los resultados más interesantes de cada combinación que procesé, pero no creo que sea la totalidad de lo que hice. Todas las imágenes de base de esta sección son de mi autoría, me parecía más divertido trabajar sobre algo que generé yo.

Sánguche de jamón crudo + noche estrellada

```
In [ ]: print_iterations('estrellada-sanguche/output_0.1_1000_1')
```

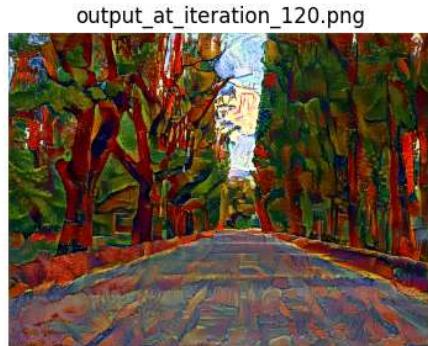
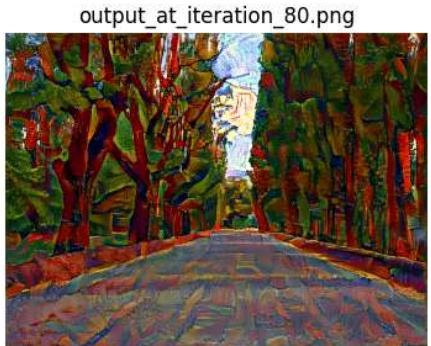
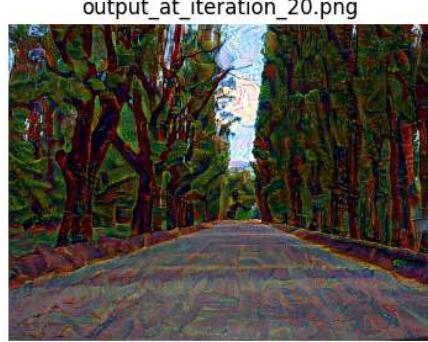
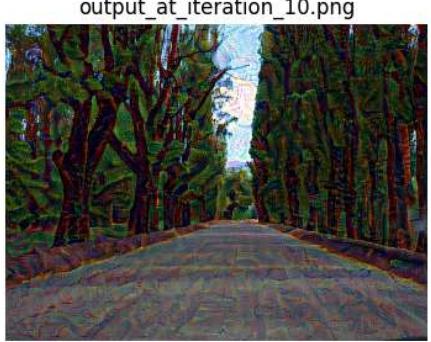
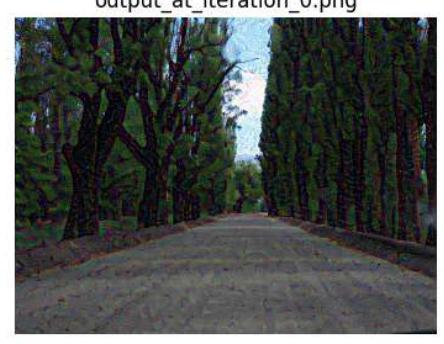
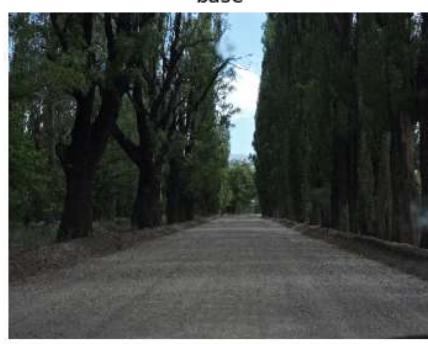
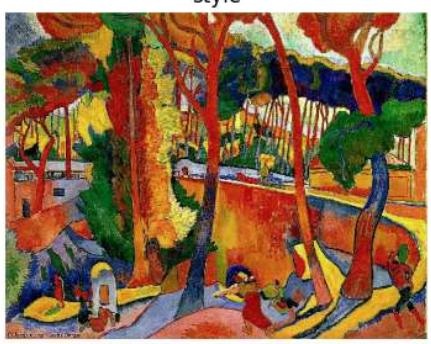


Queda un poco fuera de tono el contenido de la imagen, las características del sánduche dejan de notarse (lo cual era esperable por reducir tanto la resolución). El trabajo que hizo sobre el paisaje es notable, me gustó mucho (el sánduche también).

Ruta + The Turning Road (L'Estaque)

Este es uno bastante interesante, veamos cómo salió:

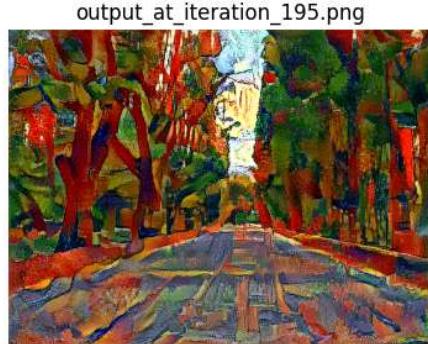
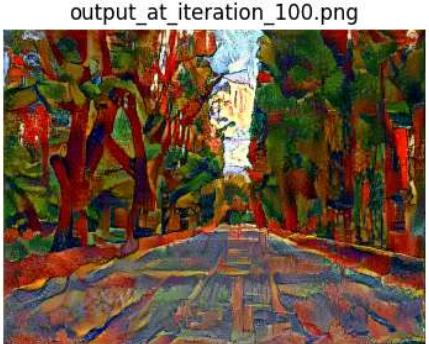
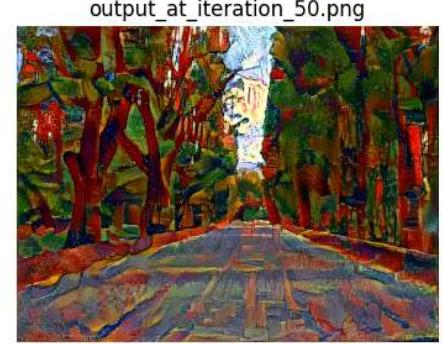
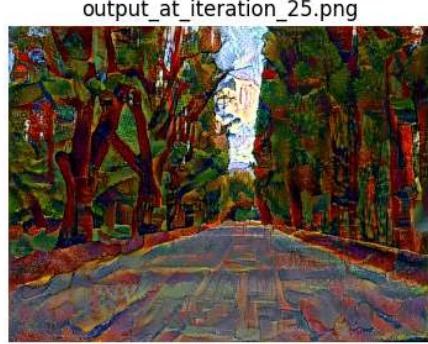
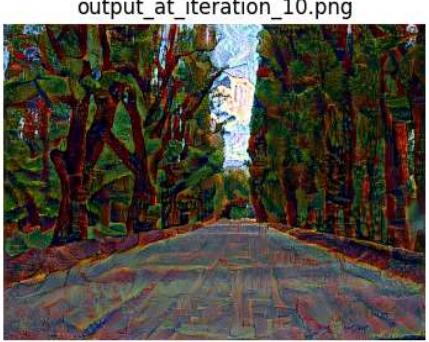
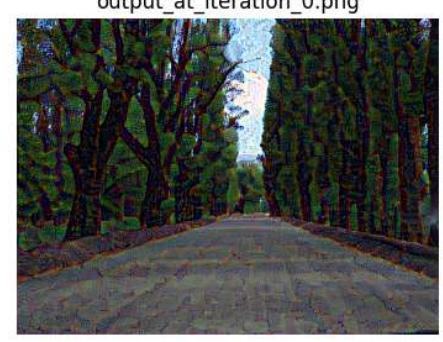
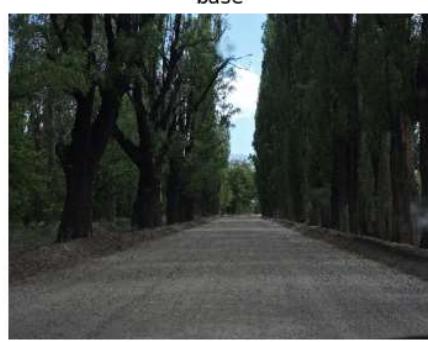
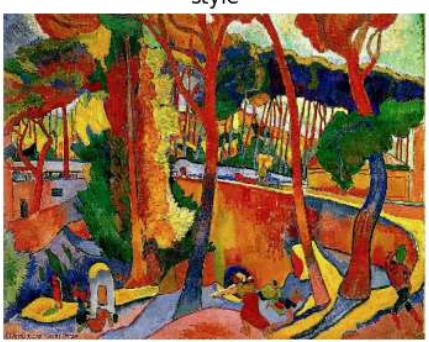
```
In [ ]: print_iterations('turningroad-ruta/output_0.1_1000_1')
```



Visto así chiquito tiene bastante buena pinta, pero al abrir la imagen se notan muchos artefactos que no me gustaron, con lo que entrené con más fuerza de estilo y más variation weight.

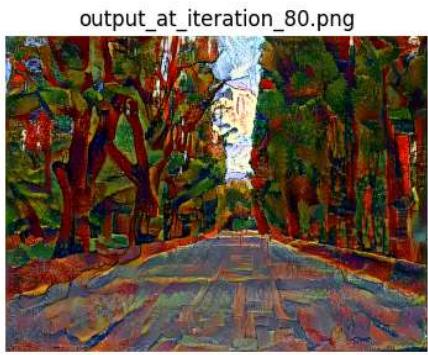
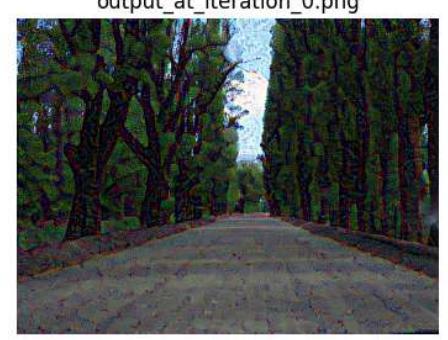
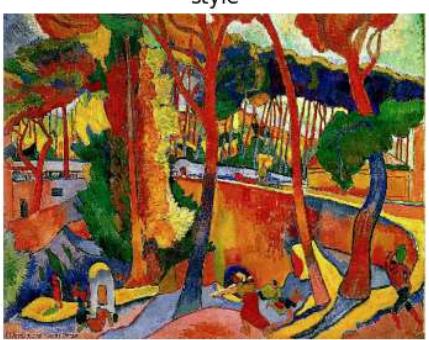
Acá cometí un error crítico: voy a elevar x10 el style weight pero x3 el variation weight, con lo que el efecto final es que al final tengo MENOS variation loss respecto del que más loss me va a poner, que es el estilo. Tardé demasiado en darme cuenta de esto, siendo "esto" el hecho de que debería haber escalado el variation_weight **junto** con el peso del estilo si quería mantener algún tipo de proporción entre ellos. Al dejar quieto el var weight y elevar tanto el style weight, efectivamente se reduce el efecto de optimizar para disminuir la variation.

```
In [ ]: print_iterations('turningroad-ruta/output_0.3_10000_1')
```



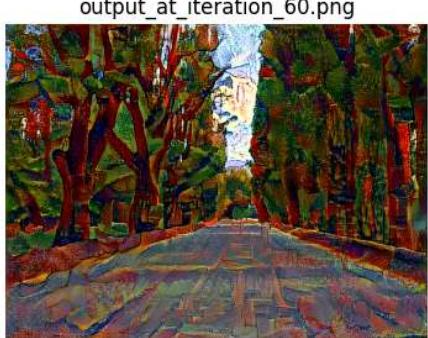
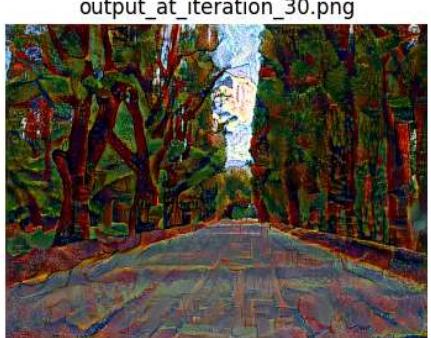
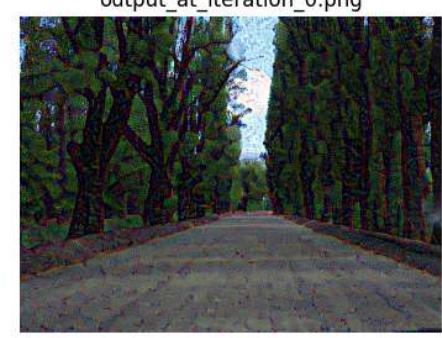
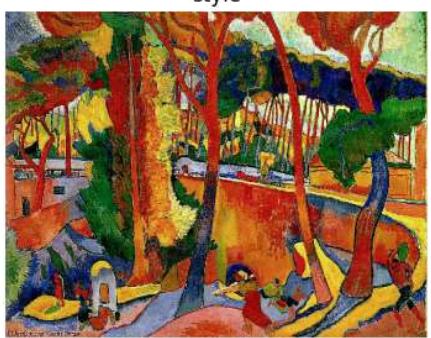
Está un poco más colorido, pero el cielo está bastante polémico. Sin embargo, me agrada el efecto que terminó teniendo. A partir de acá incremento la variation para ver cómo afecta los artefactos que no me gustaban.

```
In [ ]: print_iterations('turningroad-ruta/output_5_10000_1')
```



Corté el entrenamiento porque estaba muy igual a lo anterior y necesitaba usar la PC para otras cosas.

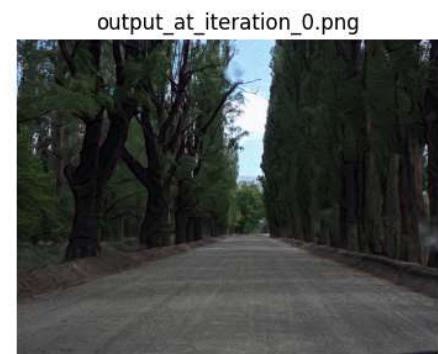
```
In [ ]: print_iterations('turningroad-ruta/output_15_10000_1')
```



Creo que de cierta manera termina más colorido, pero no se ataca el problema que quería atacar: los artefactos del cielo.

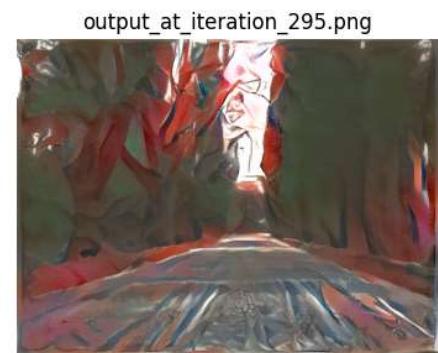
Ruta + Demoiselles D'Avignon de Picasso

```
In [ ]: print_iterations('demoiselles-ruta/output_1_10000_1')
```



Continuando este entrenamiento:

```
In [ ]: print_iterations('demoiselles-ruta/output_1_10000_1_cont')
```

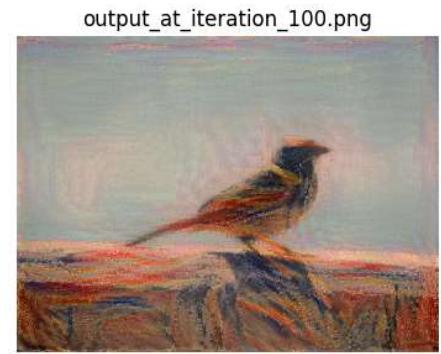
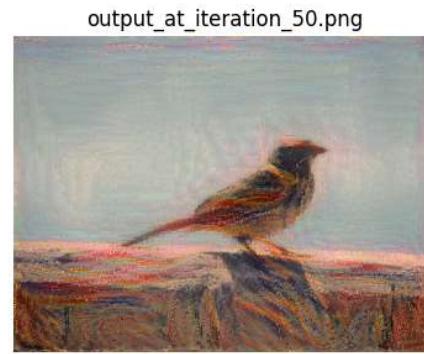


Me agrada bastante el estilo que quedó, pero creo que habría que jugar más con los pesos para llegar a un resultado más convincente.

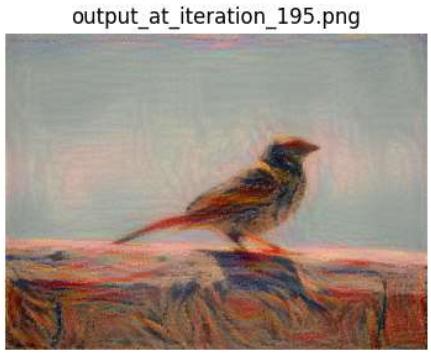
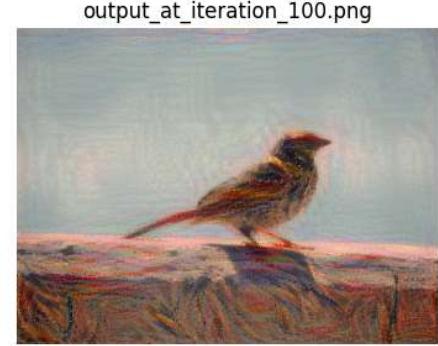
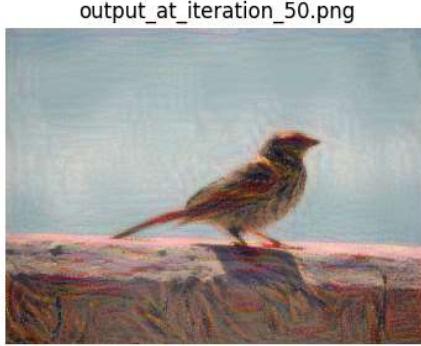
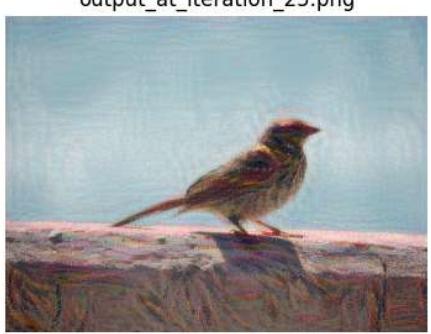
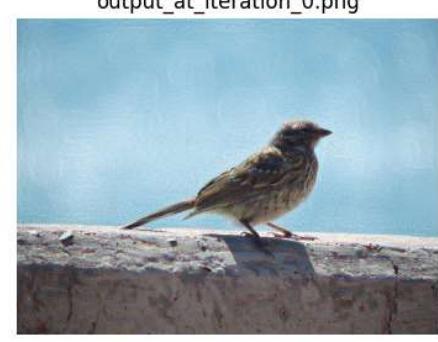
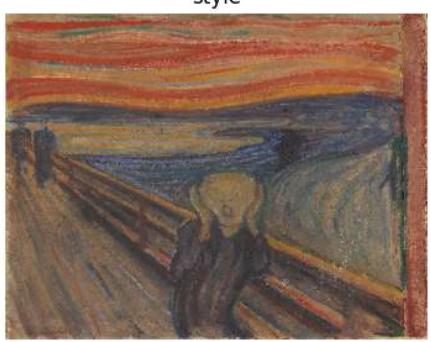
Pájaro + El Grito de Edvuard Munch

Acá se viene otro momento de humildad. No estaba contento con el resultado que estaba teniendo, con muchos pixeles de color fuerte y que generaron una imagen que no me gustaba. Quise subir mucho el variation weight, sin éxito. Eventualmente decidí almacenar la imagen de estilo, y ahí me di cuenta de que al ser tan rectangular en su origen, al hacer el resize El Grito quedó muy comprimido. Esto en conjunto con la pérdida de resolución parece que provocó que la imagen de referencia tuviera esos artefactos que tanto me molestaban al ver el resultado. No había variation weight que me salve, siendo el style weight tanto mayor.

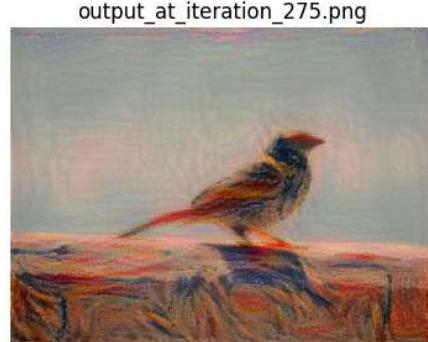
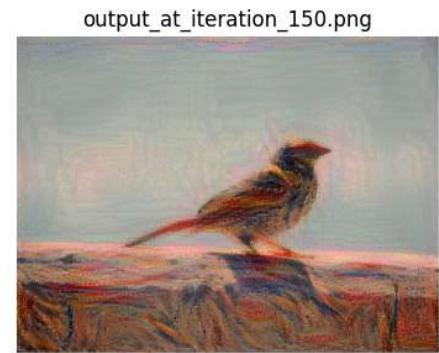
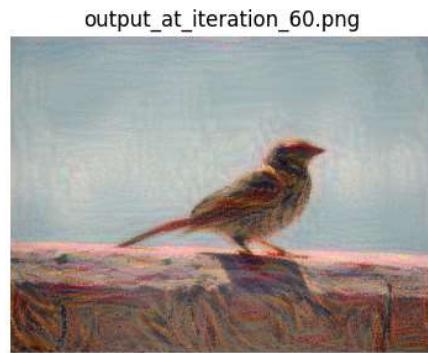
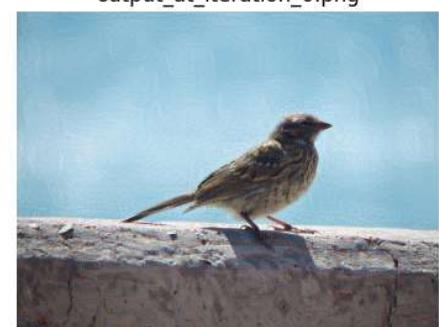
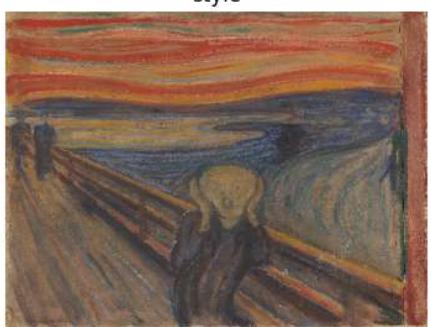
```
In [ ]: print_iterations('grito-pajaro/output_0.1_10000_1')
```



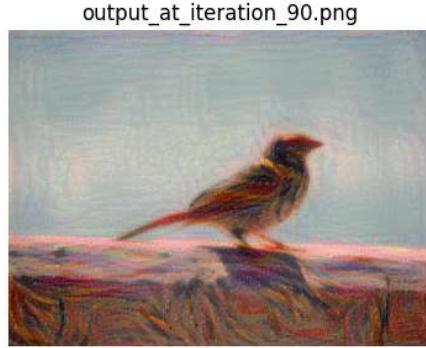
```
In [ ]: print_iterations('grito-pajaro/output_10_1000_1')
```



```
In [ ]: print_iterations('grito-pajaro/output_100_10000_1_hd')
```

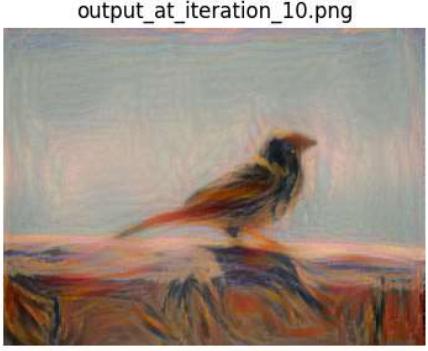


In []: `print_iterations('grito-pajaro/output_1000_10000_1')`



Lo que si rescato de todos estos entrenamientos es lo bien que quedó la parte de abajo de la imagen., se notan los trazos. No del todo satisfecho, decidí correr un entrenamiento a partir de una de estas imágenes pero usando la de referencia cropeada en vez de resizeada:

```
In [ ]: print_iterations('grito-pajaro/output_1000_10000_1_crop')
```

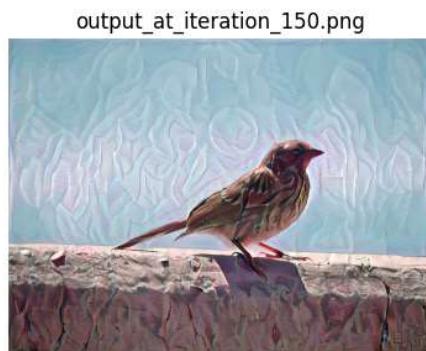
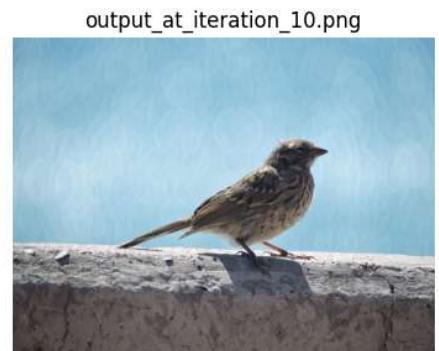


Efectivamente, casi al instante, los artefactos más feos se fueron rápido (notar diferencia con base y la iteración 0). Por desgracia (y por como venían configurados los pesos) la forma del pájaro se perdió mucho más de lo que pretendía, pero a la vez quedó un dibujo que me parece por lo menos interesante.

Pajaro + Demoiselles D'Avignon de Picasso

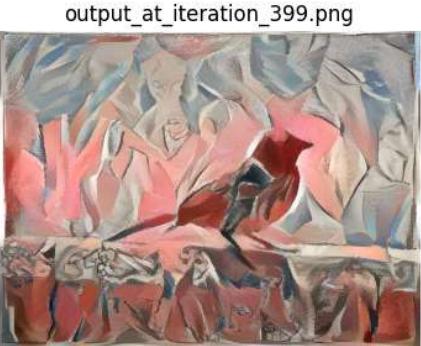
Acá hice el primer experimento en el que dejé que optimice en muchas iteraciones.

```
In [ ]: print_iterations('demoiselles-pajaro/output_paj_demoiselles_10_10000_1_hd')
```



Claramente faltaron iteraciones, por lo que hice una tirada en la que retomé la última iteración y continué la optimización. El resultado final fueron 1000 iteraciones en total, aproximadamente, de las cuales las últimas 300 resultaron en:

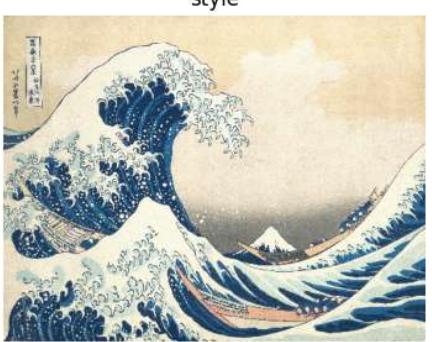
```
In [ ]: print_iterations('demoiselles-pajaro/output_paj_demoiselles_10_10000_1_cont_cont')
```



Puente + The Great Wave off Kanagawa

Acá pasa algo interesante también

```
In [ ]: print_iterations('greatwave-puente/output_kgw_1_10000_1')
```

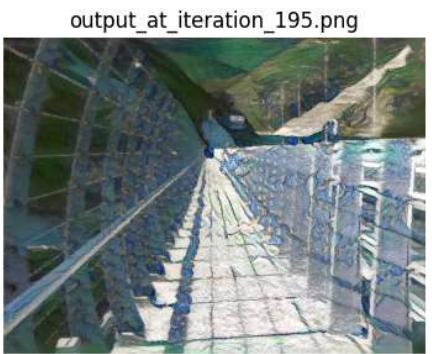
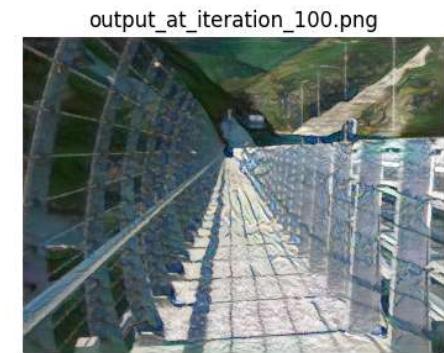


Parece que la parte de estilo que más se quiere transferir es la correlación que provoca el patrón circular del quiebre de las olas. Me queda la curiosidad de cómo se terminaría viendo con más iteraciones, pero tengo la sensación de que no quedaría nada reconocible de la imagen original, lo cual no era mi idea con esta imagen del puente.

Puente + Irises de Van Gogh

Primero lo procesé en baja resolución, como todas las demás:

```
In [ ]: print_iterations('irises-puente/output_0.3_10000_1')
```



Empiezan a aparecer artefactos y colores distonantes correspondientes a la imagen de referencia. Viendo esto, quise probar qué pasaba si subía la resolución de la imagen, buscando que parezca un cuadro pintado a mano con la textura pero sin los colores de la pintura.

```
In [ ]: print_iterations('irises-puente/output_0.3_10000_1_hd', 2, 8, 6)
```

style



base



output_at_iteration_0.png



output_at_iteration_25.png



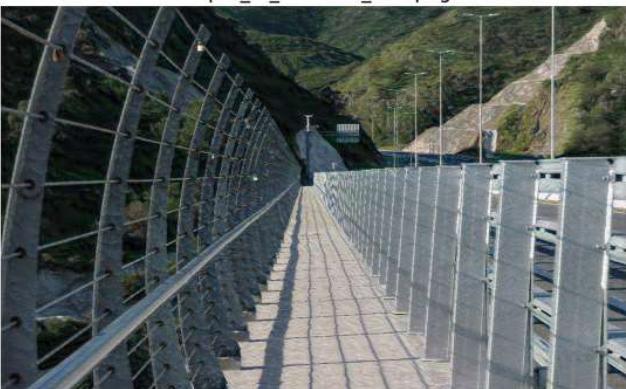
output_at_iteration_50.png



output_at_iteration_75.png



output_at_iteration_100.png



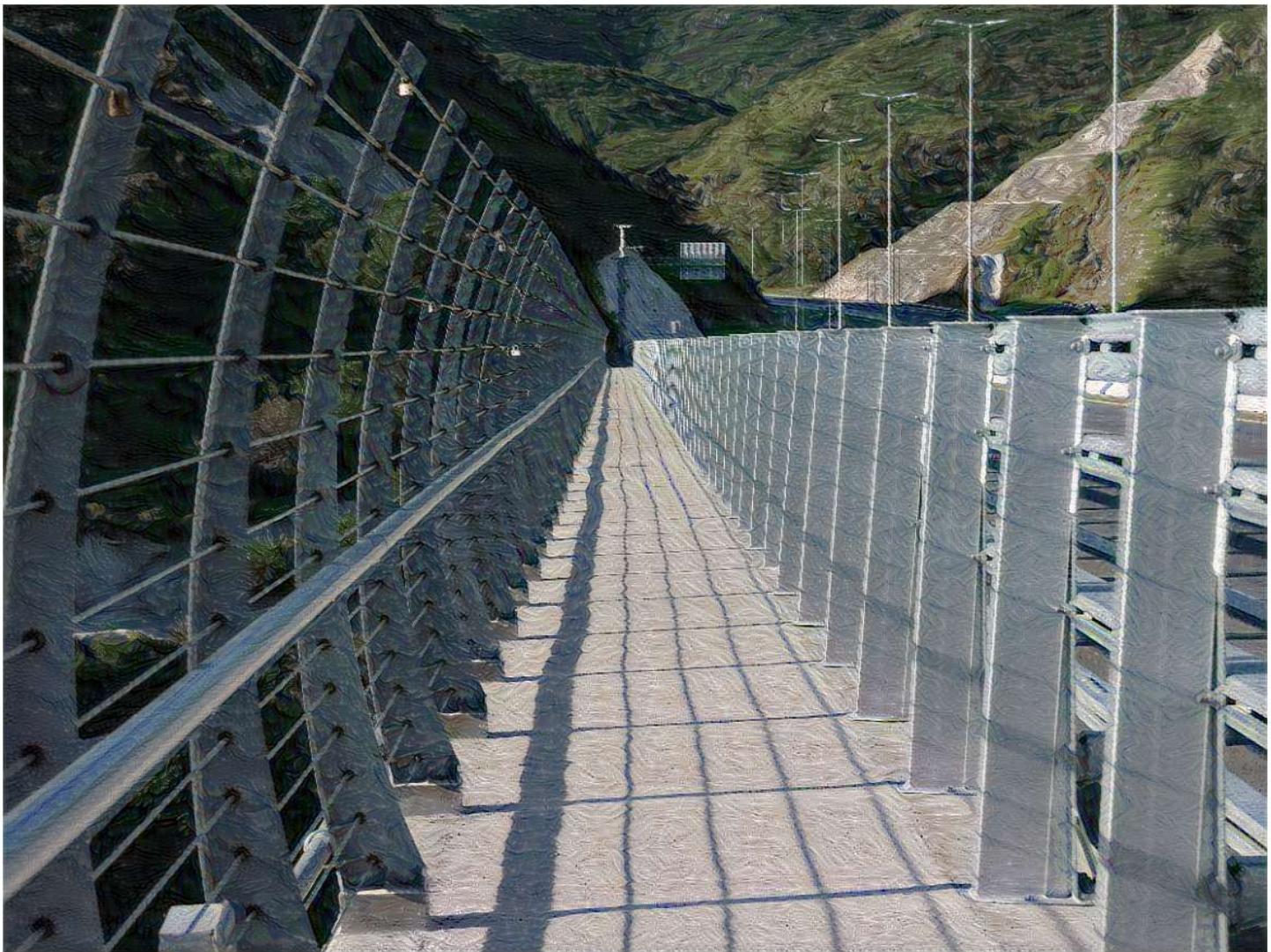


No se nota el detalle como me gustaría, pero es el resultado más bonito que tuve en el TP en mi opinión. La voy a volver a cargar pero mejor y más grande a continuación.

Me puso muy contento ver esto así, realmente. Hasta los candaditos tienen pinta de que se agregaron pintando sobre el fondo. Lo único medio difícil de creer es el aliasing por los detalles finos más lejanos.

Obviamente llevó muchísimo tiempo optimizar esto debido a la resolución, pero valió la pena.

```
In [ ]: plt.figure(figsize=(15, 15))
plt.imshow(load_img('irises-puente/output_0.3_10000_1_hd/output_at_iteration_100.png'))
plt.axis('off')
plt.show()
```



Conclusiones

La transferencia de estilo funciona. Realmente incluso entendiendo cómo funciona me impresiona mucho el efecto que tiene. Me hubiera gustado probar Adam en vez de LBFGS pero no me pude hacer el tiempo, así que quedó así. Costó un poco hacer que corra la notebook, la nueva versión de Keras le movió el arco a muchas cosas que se usan en esta implementación, quise ponerme a actualizar todo pero implicaba reescribir bastante código cuando no me parecía lo prioritario. Si las iteraciones fueran menos lentas se le podría sacar mucho jugo, tuve que tener la PC nonstop casi toda la semana hasta que me anduve demasiado lenta para usarla (tenía 4 optimizaciones en paralelo).