# 

# Trabajo Final CNN - Style Transfer

**A-Imágenes de contenido provistas**

# Imagen para estilo

!wget https://upload.wikimedia.org/wikipedia/commons/5/52/La\_noche\_estrellada1.jpg

# Imagen para contenido

!wget https://upload.wikimedia.org/wikipedia/commons/thumb/f/f4/Neckarfront\_T%C3%BCbingen\_Mai\_2017.jpg/775px-Neckarfront\_T%C3%BCbingen\_Mai\_2017.jpg

# Creamos el directorio para los archivos de salida

!mkdir /content/output

**B-Imágenes a utilizar en el trabajo**

IMAGEN CONTENIDO:

IMAGEN ESTILO



# Respuestas

# 1) En base a lo visto en el paper ¿Qué significan los parámetros definidos en la siguiente celda?

**1-Seteo variables pesos**

**\*-Total\_variation\_weight:**

Este peso está destinado al término, total\_variational\_weight\_looss (cuánto va a pesar en la función de costo total).

La total variational loss, es una función de costo que evalúa la variación entre vecinos cercanos (en términos prácticos la que esa es tomar una fila y restarla a la fila superior, y lo mismo entre columnas restando la del lateral). Consiguiendo que la imagen final tenga bordes más suaves.

Más abajo se explica cual es el objetivo de esta loss (ver punto 5)

Esta variación se calcula sobre el contenido (no sobre las matrices de gram). En nuestra función de costo total (en la versión original de la notebook entregada), le estamos indicando que va a tener un peso del 0.1.

Si este peso es muy alto, la imagen de contenido reduce la diferencia entre píxeles vecinos a tal punto que se pueden perder las divisiones y mezclar los colores (En la sección 5, genere una imagen con este peso en 100 para ver el resultado).

**\*-Style\_weight:**

Es el peso que le voy a dar en la función de costo total, al término de la loss que trata de minimizar el error en estilo, contra la imagen de ruido blanco.

Lo que se minimiza es el MSE entre las matrices de gram de los features a la salida de las capas de estilo (en este caso son 5 capas) de la imagen de estilo y la imagen aleatoria creada para ir ajustando. (se adjunta en el punto 5 las curvas de loss con diferentes pesos, y las imágenes de resultado con diferentes pesos)

**\*-Content\_weight:** es el peso que le voy a dar en la función de costo total, al término de la loss que trata de minimizar el error en el contenido, contra la imagen de ruido blanco

**2-Seteo de escala imágenes**

Luego establece el tamaño de las imágenes que se usarán en la función img\_to\_array.

Para esto asigna 400 al alto o filas (esto se asigna directamente)

En la línea siguiente se calculan las columnas que tendrá la imagen (o el ancho). Dicho ancho se saca calculando el ratio de la altura asignada 400 sobre la altura original (400/altura original) \* ancho\_original. Así consigue el nuevo ancho, que mantiene proporción con el alto asignado.

Ejemplo del cambio de dimensiones para ver la transformación

print(height,width)

599 775 ->Original

print (img\_nrows,img\_ncols)

400 517 ->Transformada

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

**1-np.expand\_dims**: Le agrega otra dimensión a la numpy array en la posición indicada. Esto se hace porque la entrada de la red, está dada por (batch Size, alto, ancho, canales).

En este caso cuando cargo la imagen solo tengo (alto, ancho y los canales). Para poder conseguir el formato de entrada que admite la función preproces\_input. Aplicó **expand\_dim** y paso de esto (400, 517, 3) a esto (1, 400, 517, 3)

Ejemplo del cambio de shape

print("antes",img.shape)

antes (400, 517, 3)

img = np.expand\_dims(img, axis=0)

print("despues",img.shape)

después (1, 400, 517, 3)

**2-preprocess\_input**: La función preprocess\_input está destinada a adecuar una imagen al formato que requiere el modelo.

Algunos modelos utilizan imágenes con valores que van de 0 a 1 (sería la versión normalizada). Otros, de -1 a +1. El que usamos en este momento que es el **vgg19** "caffe" (papper página 9 **“The model is publicly available and can be explored in the caffe-framework.24”**), no está normalizado, sino centrado a cero.

Entonces esta función preproce\_image hace lo siguiente:

* Recibe el path de la imagen
* Carga la imagen en memoria
* Convierte la imagen en un array.
* Le agrega la dimensión del batch (como se explicó antes)
* Aca se invoca a **preprocess\_input,** y le pasamos el array obtenido antes, y nos retorna el formato adecuado para entrar en la red.

# 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?

**Objetivo función deprocess\_image:**

La VGG trabaja retornando imágenes BGR y centradas a cero.

Para poder ver la imagen o guardarla como jpg necesito pasarla a RGB con valores de píxel de [0,255].

El objetivo de esta función es hacer esta conversión de lo que retorna la VGG (RGB centrado a cero) a BGR con valores [0,255]

**Como lo hace**

**1-Quitamos el centrado a cero:** Para realizar esto sumamos esas medias (que son números fijos 103.939;116.779;123.68)

El origen de estos números viene del entrenamiento de la red original. La red se entrenó con imágenes, y cuando se calculó la media por canal tomando todos las imágenes arrojó esos valores.

Entonces de las imágenes que resultan a la salida, asumimos que sumándole la media la media por canal, volverán al rango original (remover el centrado a cero). Para eliminar los casos donde después de sumar la media no quede en el rango correcto [0.255], aplicamos un numpy.clip(0,255)--> los valores por debajo de cero pasan a cero y los que superen 255 quedan en 255.

Luego tengo que revertir el orden de los canales pasar de BGR → RGB, para reasigno la variable x, recorriendo el último eje (el de los canales) de atrás para adelante.

\*Ejemplo de numpy clip:

###Ejemplo: Clip

ejemplo\_ff= np.arange(12)

print("antes de clip ",ejemplo\_ff)

ejemplo\_yy=np.clip(ejemplo\_ff, 3, 6)

print("despues de clip",ejemplo\_yy)

antes de clip [ 0 1 2 3 4 5 6 7 8 9 10 11]

despues de clip [3 3 3 3 4 5 6 6 6 6 6 6]

# 4) En la siguientes celdas:

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

**1-Matriz de gram:** Nos mide la correlación entre vectores, en este caso la usamos para ver cuán similares son los features maps a la salida de una capa entre sí.

Si la multiplicación de los vectores da un número alto entonces están altamente correlacionado, una número bajo poca correlación.

La idea es que sea cual sea la relación entre features maps, la imagen random busque imitar esa correlación.

**2-Para que se usa:**

La matriz de Gram es una matriz de covarianza empírica.

Para la transferencia de estilo, calcularemos las matrices gram de los features maps en un conjunto de capas inferiores de la red (las destinadas a estilo).

Para explicar esto podemos tomar una sola capa, suponga que tiene 32 mapas de características. La matriz de gram es la covarianza entre cada uno de los mapas de características.

Lo que se mide es si, en una posición de píxel en particular, la característica #X tiende a coincidir con la característica #Y.

Por ejemplo, si en una imagen, siempre que tenga la característica #15 iluminada en un punto en particular, la característica #8 también se ilumina. Vamos a buscar que en la imagen destino el patrón de coincidencia de características coincida.

**3-Porque se permuta:** El objetivo es recibir una imagen y retornar la gram matrix de (vectores features por vectores features). Para eso necesitamos hacer una matriz de feature, cuyos lados sean los vectores de features maps que tenga a la salida de la capa convolucional. Donde cada vector tendrá una longitud = el alto\*el ancho, del feature map.

Pero primero tenemos que transformar el feature mas que es un Tensor con este formato ((Height, Weight, Feature = features maps)) → a un tensor que tenga número de feature map en la primer dimensión y vector de 1 con el feature map aplanado en la segunda dimensión, quedaría con este shape: (Feature Map, Height \* Weight)

**Proceso**

Primero hacemos una permutación de las dimensiones, poniendo como primer eje, el de los features, y dejando luego el alto y el ancho detrás.

Nosotros recibimos en el input un tensor con este shape (H,W,F = features maps) y lo tenemos que convertir a --> (F=Features maps, H\**W)*

*Entonces al permutar, nos queda atras altura y ancho, y podemos hacer un flaten de estos dos, sin tocar el eje que corresponde a features. O sea el resultante es un vector, de shape (Features, alto\**ancho).

Luego resta hacer la transpuesta para que los mismos vectores que en una matriz son fila en la otra sean columna y se pueda hacer la multiplicación de todos contra todos.

**Ejemplo:**

Supongamos que tengo 10 mapas de caracteres, de 4x4, entonces para armar la matriz de gam me quedan 10 vectores de 16 de largo. La matriz de gram va a ser de 10 x 10, que son los vectores que tengo.

Poniendo un print en la función vemos el resultado, acá se puede ver los distintos shape desde que recibimos X hasta que generamos la gram matrix en la función.

**Resultados del cálculo:**

x recibido gram: (400, 656, 64) → Así recibo los feature maps

X permutado gram: (64, 400, 656) → Permuto y pongo delante los FM

x flatten gram; Tensor("Reshape:0", shape=(64, 262400), dtype=float32) → genero un vector 2D (Feature maps, altor\*ancho)

x flatten transpuesta gram (262400, 64) → Hago transpuesta para poder luego multiplicar vector contra vector

matriz gram resultado (64, 64) → Matriz resultado

**Ejemplo simple de cálculo de gram**

#Ejemplo:

#Matriz de gram

#Caso A: tengo un vector en la primer fila y el otro ortogonal, o sea hay poca correlación, en la segunda fila la correlacion es maxima

print("caso 1 TV")

aa = np.ones((2,2), dtype=float)

aa[0]=1

aa[1][0]=-1

aa[1][1]=1

print("matriz aa",aa)

bb=np.transpose(aa)

print("trasponemos aa ",bb)

cc=np.dot(aa,bb)

print("resultado producto punto",cc)

#Como vemos los vectores ortogales dio cero el producto y para los paralelos duplico su modulo

caso 1 TV

matriz aa

[[ 1. 1.]

[-1. 1.]]

trasponemos aa

[[ 1. -1.]

[ 1. 1.]]

resultado producto punto [[2. 0.]

[0. 2.]]

# 5) Losses:

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

Rta:

\*Style\_loss:

Es la función de costo para ajustar el estilo.

Lo hace calculando el error cuadrático medio entre las matrices de gram de los features maps a la salida de cada una de las capas seleccionadas para estilo, para la imagen con el estilo (desde la cual quiero tomar el estilo pero no el contenido) y la imagen aleatoria (o de ruido blanco como la referencia el paper).

En este caso son 5 matrices para una imagen fuente del estilo (Fuente) y 5 para la imagen destino (Destino), entonces el cálculo será: MSE entre ([Gram(Fuente1)-Gram(Destino1)]+....+[Gram(Fuente5)-Gram(Destino5)]).

Las capas que se utilizan, son las capas menos profundas de la red. Aca se marcan cuales son:

**block1\_conv1 (3, 3, 3, 64) -->Style**

block1\_conv2 (3, 3, 64, 64)

**block2\_conv1 (3, 3, 64, 128) -->Style**

block2\_conv2 (3, 3, 128, 128)

**block3\_conv1 (3, 3, 128, 256**) **-->Style**

block3\_conv2 (3, 3, 256, 256)

block3\_conv3 (3, 3, 256, 256)

block3\_conv4 (3, 3, 256, 256)

**block4\_conv1 (3, 3, 256, 512) -->Style**

block4\_conv2 (3, 3, 512, 512)

block4\_conv3 (3, 3, 512, 512)

block4\_conv4 (3, 3, 512, 512)

**block5\_conv1 (3, 3, 512, 512)-->Style**

block5\_conv2 (3, 3, 512, 512)-->Contenido

block5\_conv3 (3, 3, 512, 512)

block5\_conv4 (3, 3, 512, 512)



**\*content\_loss:**

Aca la función de costo es más simple, es un error cuadrático medio de un feature map contra otro feature map (sin realizar cálculos con el contenido del feature map). Entonces tomó el feature map a la salida de la capa seleccionada para contenido de: En primer lugar la imagen aleatoria (la cual es input de la red, y es este input el que se entrena, acá entrenamos o ajustamos la imagen no los pesos de la red, para que su contenido en la próxima pasada reduzca el error), en segundo lugar la imagen fuente de la cual voy a copiar el contenido (pero no el estilo). Con estos dos features maps, resto uno al otro y calculo el error cuadrático medio.

Las capas que se usan acá son las más cercanas a la salida. En este ejemplo usamos la “block5\_conv2”

**\*Total variational weight:**

Esto es un agregado que no está en el paper. Lo que calcula es la variación dentro de la misma imagen entre, en primer lugar las filas comparando la filas desde la posición 0 a la posición filas totales - 1, con las que van de la posición 1 hasta el final (filas totales) esto hace que compare cada fila con la que está justo encima obteniendo la variación entre las filas cercanas.

Luego hace lo mismo con el ancho (que serían las columnas).

Con lo cual termina obteniendo dos matrices con la diferencia entre vecinos cercanos. Luego las suma y les aplica una exponenciación, y suma todos los resultados y así se consigue un escalar que representa las diferencias absolutas para los valores de píxeles vecinos en la imagen de entrada.

Esto estaría, de alguna forma, midiendo cuánto ruido hay en las imágenes o cuán fuerte son las transiciones.

Si el número obtenido es es grande hay mucha variación, si es cero son vecinos idénticos.

Al agregar la variación total a la función de costo del entrenamiento la imagen resultante se ve mucho más suave.

(Se realizaron pruebas variando el peso de esta loss y se adjuntan en el repo, con el sufijo tv\_xx, donde xx es el peso asignado)

**\***A continuación un código de prueba donde se verifica el cálculo explicado con matrices pequeñas para su comprensión, mostrando el caso de un matriz donde los vecinos varían mucho (loss grande) y otra donde no hay variación (loss pequeña):

##Ejemplo:

##Se realiza un ejemplo simple de lo que hace la Total Variational loss

#Caso A: la diferencia entre vecinos es grande entre filas

print("caso 1 TV")

aa = np.ones((3,3), dtype=float)

aa[1]=2

aa[2]=5

print("Matriz A vecinos muy distintos",aa)

tt=aa[:2, :] - aa[1:, :]

jj=aa[:,:2] - aa[:,1:]

print("Diferencia filas",tt,"Diferencia Columnas",jj)

ss=np.square(tt)

zz=np.square(jj)

totalxx=K.sum(K.pow(ss, 1.25))

print("total variation A:",K.eval(totalxx))

print("caso 2 TV")

#Caso B: la diferencia entre vecinos es pequña entre filas

aa = np.ones((3,3), dtype=float)

aa[1]=1.1

aa[2]=1.2

print("Matriz b vecinos parecidos",aa)

tt=aa[:2, :] - aa[1:, :]

jj=aa[:,:2] - aa[:,1:]

print("Diferencia filas",tt,"Diferencia Columnas",jj)

ss=np.square(tt)

zz=np.square(jj)

totalxx=K.sum(K.pow(ss, 1.25))

print("total variation B:",K.eval(totalxx))

##Fin Ejemplo

caso 1 TV

Matriz A vecinos muy distintos [[1. 1. 1.]

[2. 2. 2.]

[5. 5. 5.]]

Diferencia filas [[-1. -1. -1.]

[-3. -3. -3.]] Diferencia Columnas [[0. 0.]

[0. 0.]

[0. 0.]]

**total variation A: 49.76537180435969**

caso 2 TV

Matriz b vecinos parecidos [[1. 1. 1. ]

[1.1 1.1 1.1]

[1.2 1.2 1.2]]

Diferencia filas [[-0.1 -0.1 -0.1]

[-0.1 -0.1 -0.1]] Diferencia Columnas [[0. 0.]

[0. 0.]

[0. 0.]]

**total variation B: 0.018973665961010265**

\*A continuación los gráficos de la loss con diferentes entrenamientos donde se asignaron distintos pesos, para entender como la magnitud de cada número es diferentes, cuan rápido converge cada una al mínimo, y de esta forma asignar pesos en ratios que favorezcan la convergencia de la loss total.

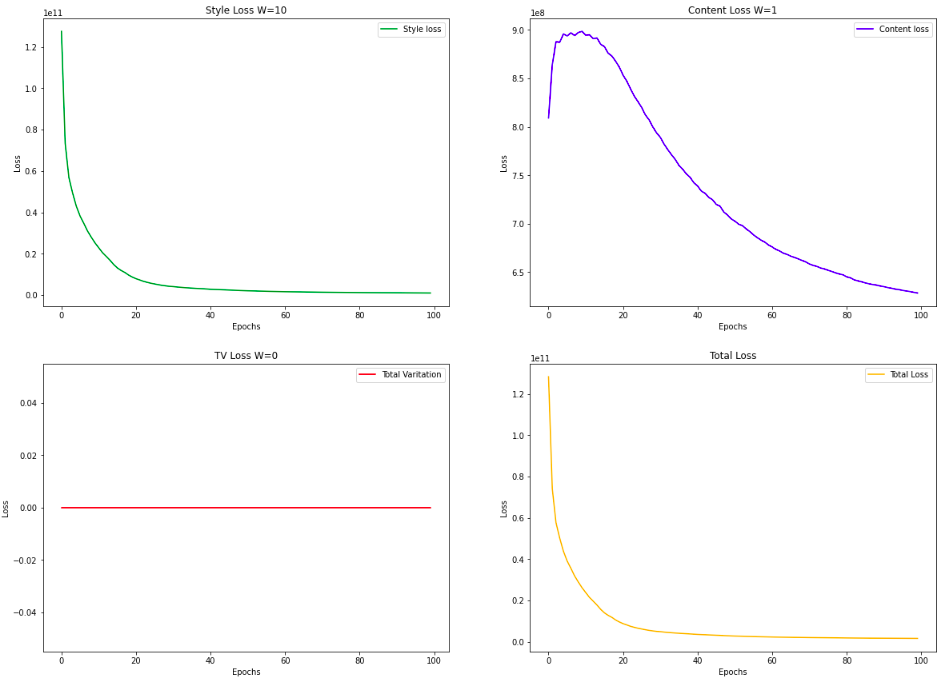
Según lo mostrado en los gráficos, para mi la distribución de pesos del caso C, es la mejor, porque iguala la magnitud de los escalares de cada loss, todos quedan cerca de 1xe10 aprox. En el caso D vemos como bajando la diferencia entre vecinos (Total variational loss con un alto pesos), se terminan por mezclar los bordes.

IMAGEN CONTENIDO:



IMAGEN ESTILO

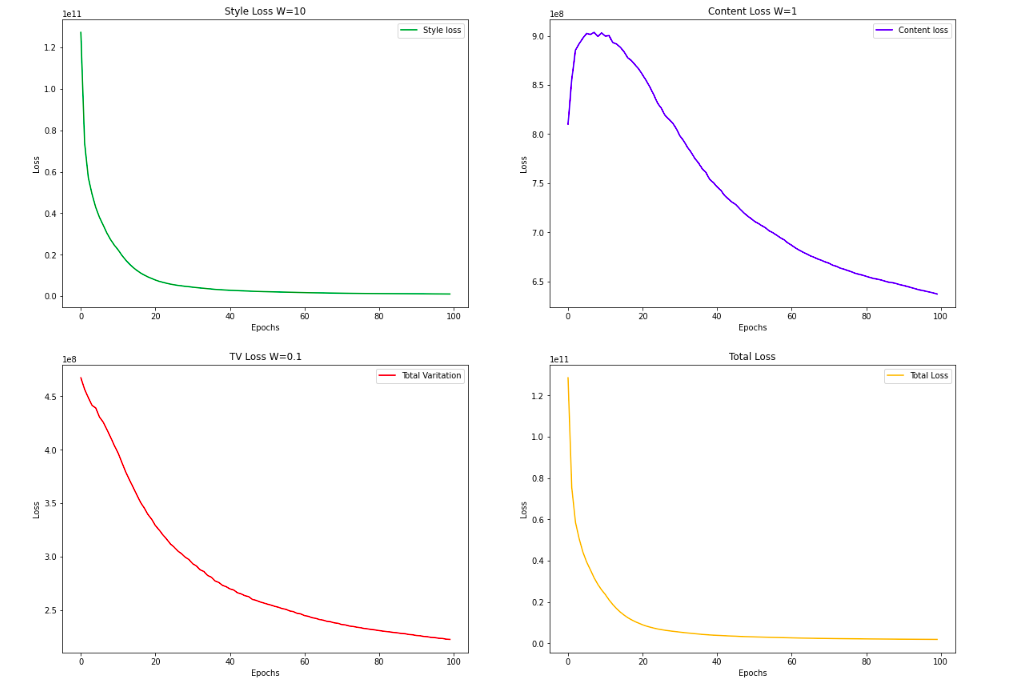


A-PESOS (Style=10, Content=1, Total Varitatinal=0) 



**Observación:** Con la total variational en cero se puede ver en el contorno del cuerpo de la windsurfista que la transición es bien marcada con el agua. El estilo al estar con un peso de 10 ganó preponderancia y borró algunos detalles de contenido como la cara de la persona.

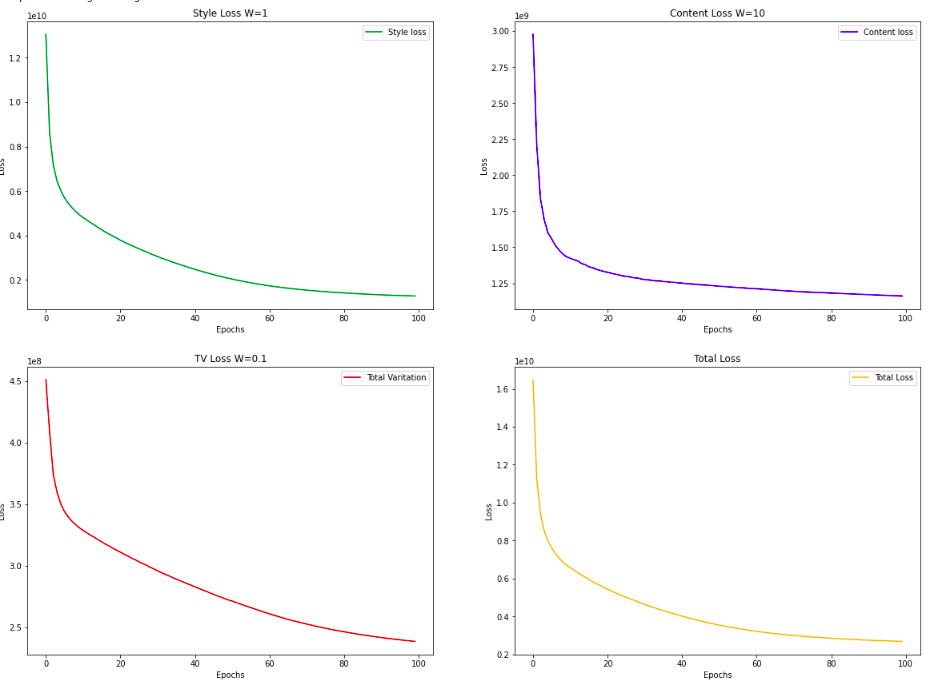
**B-PESOS (Style=10, Content=1, Total Varitatinal=0.1)**





**Observación:** Con la total variational en 0.1, no difiere mucho del caso A

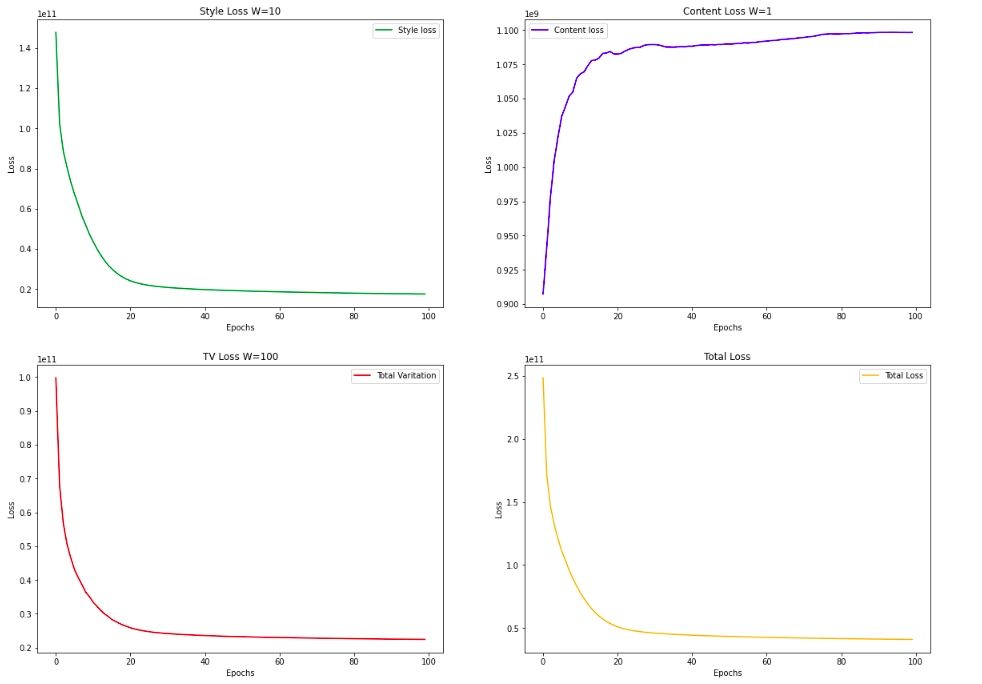
**C-PESOS (Style=1, Content=10, Total Varitatinal=0.1)-->Mejor**





**Observación:** Con un peso de 10 para el contenido y 1 para el estilo, obtenemos la mejor versión, tenemos los colores del estilo, pero se puede distinguir mejor la cara, las rodillas y la punta de la tabla.

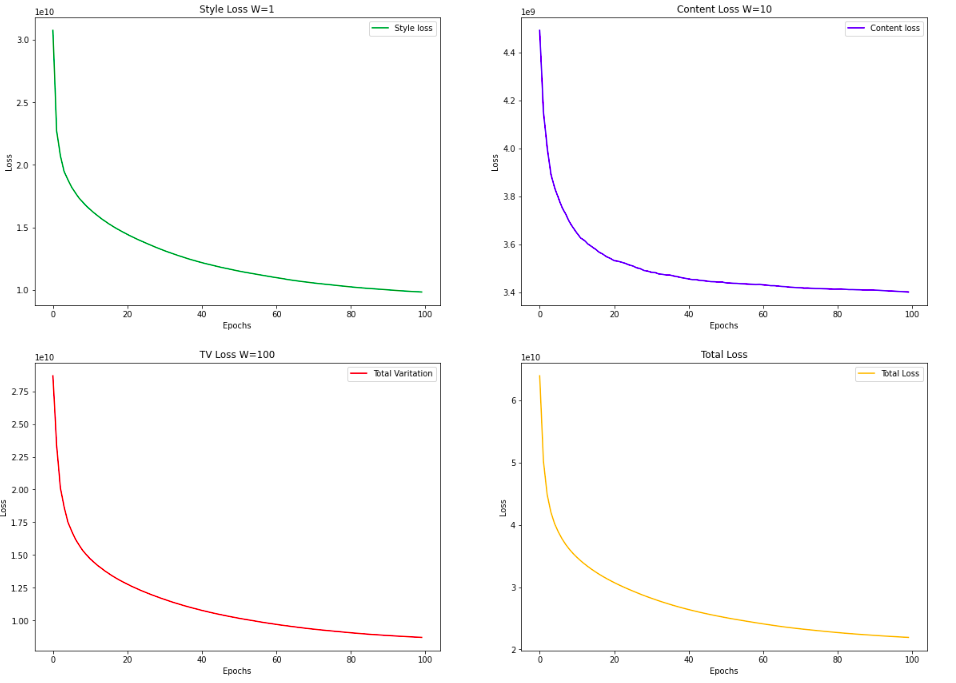
**D-PESOS (Style=10, Content=1, Total Varitatinal=100)**

****



**Observación:** Con la total variational en 100 al eliminar la variación entre vecinos cercanos, vemos como por ejemplo en el borde que limita la espalda con el agua, se crea un borde fantasma donde se mezclan colores y se pierde nitidez. Y esto se puede comprobar en la loss del contenido que al tener poco peso, si bien la loss total mejora con las iteraciones, las loss de contenido por si sola empeora.

**E-PESOS (Style=1, Content=10, Total Varitatinal=100)**

****

****

**Observación:** Con la total variational en 100 se pierde nitidez como antes, pero acá, con el peso invertido entre estilo y contenido, los bordes se mantienen un poco más nítidos (por el peso mayor del contenido) y los colores se mezclan.

# 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?

Respuesta:

Antes de explicar las funciones requeridas, es necesario entender el objetivo que perseguimos con esta implementación.

Lo que buscamos es actualizar iterativamente nuestra imagen de salida de modo que minimice nuestra función de costo. A diferencia de otras implementaciones, acá no actualizamos los pesos asociados con nuestra red, sino que entrenamos nuestra imagen de entrada.

**1- La funcion fmin\_l\_bfgs\_b:**

Lo que hace esta función es, buscar un mínimo para la función que pasamos por parámetro usando el algoritmo L-BFGS-B. (Búsqueda de mínimos o de raíces, lo que hace por ejemplo regula falsi)

Le estamos pasando: *la funcion que quiero minimizar (función de la cual quiero encontrar un input X que me de un mínimo*

*Le pasamos entonces, el input actual (el X) donde estoy parado (esperamos que luego de aplicar la función retorne un nuevo input, que minimice el resultado de la loss)*

\*A continuación un ejemplo de cómo encuentra un mínimo para funciones simples

##EJEMPLO:

##Como funciona la busqueda de minimos de fmin\_l\_bfgs

#Aca tenemos un ejemplo corto de como minimiza la funcion F(x)=x^2, aca sabemos que el minimo es cero. Tiene que buscar el x que mas se aproxime a este valor

#Le podria pasar args los args son constantes no los optimiza, pero me permiten por ejemplo:

#si uso una recta tener diferentes pendientes en cada llamada, o si buscmo minimizar error cuadratico, le puedo pasar el par de Y\_TRUE, para que vaya probando la resta

#'x' y 'x0' son los parametros que estoy optimizando - el resto es todo argumento

#<Caso A: X²>:

def func(x, \*args):

print("x ", x)

return (x\*\*2) + 1

initial\_values = np.array([10])

print("initial\_values ", initial\_values)

tt, min\_val, info =fmin\_l\_bfgs\_b(func, x0=initial\_values, args=(0,0), approx\_grad=True)

print("x:",tt, "minval: ", min\_val, "info :",info)

y\_true =(-7.10542736e-15)\*\*2 + 1

print("comprobacion del resultado",y\_true)

#</Caso A: X²>

initial\_values [10]

x [10.]

x [10.00000001]

x [9.]

x [9.00000001]

x [-7.10542736e-15]

x [9.99999289e-09]

x: [-7.10542736e-15] minval: **[1.]** info : {'grad': array([0.]), 'task': b'CONVERGENCE: NORM\_OF\_PROJECTED\_GRADIENT\_<=\_PGTOL', 'funcalls': 6, 'nit': 2, 'warnflag': 0}

comprobacion del resultado **1.0**

#<Caso B: MSE>

x\_true = np.arange(0,10,0.1)

print("valores x ",x\_true)

m\_true = 2

b\_true = 1.0

y\_true = m\_true\*x\_true + b\_true

print("valor de y\_true",y\_true)

def func(params, \*args):

#print("params ", params)

#print("args ", args)

x = args[0]

y = args[1]

m, b = params

y\_model = m\*x+b

error = y-y\_model

return sum(error\*\*2)

initial\_values = np.array([0, 0])

print("initial\_values ", initial\_values)

tt, min\_val, info =fmin\_l\_bfgs\_b(func, x0=initial\_values, args=(x\_true,y\_true), approx\_grad=True)

print("x:",tt, "minval: ", min\_val, "info :",info)

#Comprobacion del resultado

y\_calculado = tt[0]\*x\_true + tt[1]

error = y\_true-y\_calculado

print("comprobacion del resultado",sum(error\*\*2))

#</Caso B: MSE>

valores x [0. 0.1 0.2 0.3 0.4 …hasta…. 20.2 20.4 20.6 20.8]

initial\_values [0 0]

x: [1.99999998 1.00000008] minval: **2.3347348270214455e-13** info : {'grad': array([4.8017570e-07, 7.3179247e-08]), 'task': b'CONVERGENCE: NORM\_OF\_PROJECTED\_GRADIENT\_<=\_PGTOL', 'funcalls': 24, 'nit': 5, 'warnflag': 0}

comprobacion del resultado **2.3347348270214455e-13**

#Fin test de la funcion

**2-Diferencia:**

La diferencia con el paper es la variational loss, en el paper no se implementa esta función, que como explique anteriormente sirve para suavizar las transiciones.

**3-Alternativa:**

Podemos usar diferentes alternativas para minimizar la función de costo en vez de usar “**fmin\_l\_bfgs\_b**”.

Por ejemplo el “**AdamOptimizer”.** Se adjunta una notebook donde se utilizó este optimizador para buscar el mínimo.

Básicamente tendría que declarar como se ve a continuación este optimizador, y aplicarlo a la imagen

opt = tf.train.AdamOptimizer(learning\_rate=10.0)

grads, all\_loss = eval\_loss\_and\_grads(x)

opt.apply\_gradients([(grads, init\_image)])

Y de esta manera obtener init\_image minimizada

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

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

Se explicó en el punto sobre la loss (Ver punto 5)