

```

1 import os
2 import sys
3 import scipy.io
4 import scipy.misc
5 import matplotlib.pyplot as plt
6 from matplotlib.pyplot import imshow
7 from PIL import Image
8 import numpy as np
9 import tensorflow as tf
10

```

Neural Style Transfer merges two images, namely: a "content" image (C) and a "style" image (S), to create a "generated" image (G). The generated image G combines the "content" of the image C with the "style" of image S.

Neural Style Transfer (NST) uses a previously trained convolutional network, and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning. Used VGG-19, a 19-layer version of the VGG network. This model has already been trained on the very large ImageNet database, and has learned to recognize a variety of low level features (at the shallower layers) and high level features (at the deeper layers).

```

1 vgg = tf.keras.applications.VGG19(include_top=False,
2                                     input_shape=(img_size, img_size, 3),
3                                     weights='imagenet')
4
5 vgg.trainable = False
<keras.src.engine.functional.Functional object at 0x7a111f397f10>

```

Building the Neural Style Transfer (NST) algorithm in three steps:

- 1) content cost function $J_{content}(C, G)$
- 2) style cost function $J_{style}(S, G)$
- 3) Total Cost $J(G) = \alpha * J_{content}(C, G) + \beta * J_{style}(S, G)$

$$J_{content}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

$$J_{style}^{[l]}(S, G) = \frac{1}{4 \times n_C^2 \times (n_H \times n_W)^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} (G_{(gram)i,j}^{(S)} - G_{(gram)i,j}^{(G)})^2$$

- The shallower layers of a ConvNet tend to detect lower-level features such as edges and simple textures.
- The deeper layers tend to detect higher-level features such as more complex textures and object classes.

We need the "generated" image G to have similar content as the input image C. Suppose you have chosen some layer's activations to represent the content of an image.

- will get the most visually pleasing results if you choose a layer from somewhere in the middle of the network–neither too shallow nor too deep. This ensures that the network detects both higher-level and lower-level features.

forward propagate image C,

- Set the image C as the input to the pretrained VGG network, and run forward propagation.
- Let $a^C(l)$ be the hidden layer activations in the layer l. This will be an $n_H \times n_W \times n_C$ tensor.

forward propagate image G,

- Repeat this process with the image G: Set G as the input, and run forward propagation.
- Let $a^G(l)$ be the corresponding hidden layer activation.

```
1 content_image = Image.open("/content/baby.jpg")
2 content_image
3
```



```

1 #function to compute Content Cost
2 def compute_content_cost(content_output, generated_output):
3     """
4     Computes the content cost
5
6     Arguments:
7     a_C -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image C
8     a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image G
9
10    Returns:
11    J_content -- scalar
12    """
13    a_C = content_output[-1]
14    a_G = generated_output[-1]
15
16    m, n_H, n_W, n_C = a_G.get_shape().as_list()
17
18    # Reshape 'a_C' and 'a_G'
19    a_C_unrolled = tf.reshape(a_C, shape=[m, n_H * n_W, n_C])
20    a_G_unrolled = tf.reshape(a_G, shape=[m, n_H * n_W, n_C])
21
22    J_content = tf.reduce_sum(tf.square(a_C_unrolled - a_G_unrolled))/(4.0 * n_H * n_W * n_C)
23
24    return J_content

```

```

1 example = Image.open("/content/Artwork.jpg")
2 example
3

```



Style Matrix/Gram matrix

- For the sake of clarity, in this assignment G_gram will be used to refer to the Gram matrix, and G to denote the generated image.

Computing Gram matrix G_gram by multiplying the "unrolled" filter matrix with its transpose:

$$G_{\text{gram}} = A_{\text{unrolled}} \times A_{\text{unrolled}}^T$$

```

1 #function to Compute Gram Matrix
2 def gram_matrix(A):
3     """
4     Argument:
5         A -- matrix of shape (n_C, n_H*n_W)
6     Returns:
7         GA -- Gram matrix of A, of shape (n_C, n_C)
8     """
9     GA = tf.matmul(A, tf.transpose(A))
10    return GA

11
12
13
14
15
16
17
18
19
20
21
22
23
24

```

1 #function to compute layer style cost

```

2 def compute_layer_style_cost(a_S, a_G):
3     """
4     Arguments:
5         a_S -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of the image S
6         a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of the image G
7
8     Returns:
9     J_style_layer
10    """
11    m, n_H, n_W, n_C = a_G.get_shape().as_list()
12
13    # Reshape the tensors from (1, n_H, n_W, n_C) to (n_C, n_H * n_W)
14    a_S = tf.transpose(tf.reshape(a_S, shape=[-1, n_C]))
15    a_G = tf.transpose(tf.reshape(a_G, shape=[-1, n_C]))
16
17    # Computing gram_matrices for both images S and G
18    GS = gram_matrix(a_S)
19    GG = gram_matrix(a_G)
20
21    J_style_layer = tf.reduce_sum(tf.square(GS - GG))/(4.0 * ((n_H * n_W * n_C)**2))
22
23
24    return J_style_layer

```

Style Weights

- better results if style costs from several different layers are merged.
- Each layer will be given weights, that reflect how much each layer will contribute to the style.

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S, G)$$

listing the layer names:

```

1 for layer in vgg.layers:
2     print(layer.name)

    input_4
    block1_conv1
    block1_conv2
    block1_pool
    block2_conv1
    block2_conv2
    block2_pool
    block3_conv1
    block3_conv2

```

```
block3_conv3
block3_conv4
block3_pool
block4_conv1
block4_conv2
block4_conv3
block4_conv4
block4_pool
block5_conv1
block5_conv2
block5_conv3
block5_conv4
block5_pool
```

output of layer 'block5_conv4'.the content layer, which will represent the image.

```
1 vgg.get_layer('block5_conv4').output
<KerasTensor: shape=(None, 25, 25, 512) dtype=float32 (created by layer 'block5_conv4')>
```

layers to represent the style of the image and assigning style costs:

```
1 STYLE_LAYERS = [
2     ('block1_conv1', 0.2),
3     ('block2_conv1', 0.2),
4     ('block3_conv1', 0.2),
5     ('block4_conv1', 0.2),
6     ('block5_conv1', 0.2)]
```



```
1 #function to compute combined style cost
2
3 def compute_style_cost(style_image_output, generated_image_output, STYLE_LAYERS=STYLE_LAYERS):
4     """
5         Computes the overall style cost from several layers
6
7     Arguments:
8         style_image_output -- our tensorflow model
9         generated_image_output --
10        STYLE_LAYERS -- A python list containing:
11                    - the names of the layers we would like to extract style from
12                    - a coefficient for each of them
13
14    Returns:
15    J_style
16    """
17
18
19    J_style = 0
20
21    # Set a_S to be the hidden layer activation from the layer we have selected.
22    # The last element of the array contains the content layer image
23    a_S = style_image_output[:-1]
24
25    # Set a_G to be the output of the hidden layers.
26    # The last element of the list contains the content layer image
27    a_G = generated_image_output[:-1]
28    for i, weight in zip(range(len(a_S)), STYLE_LAYERS):
29        # Compute style_cost for the current layer
30        J_style_layer = compute_layer_style_cost(a_S[i], a_G[i])
31
32        # Add weight * J_style_layer of this layer to overall style cost
33        J_style += weight[1] * J_style_layer
34
35    return J_style
```

Total_cost

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

```

1 #function to Compute Total Cost
2 @tf.function()
3 def total_cost(J_content, J_style, alpha = 10, beta = 40):
4     """
5         Arguments:
6             J_content -- content cost coded above
7             J_style -- style cost coded above
8             alpha -- hyperparameter weighting the importance of the content cost
9             beta -- hyperparameter weighting the importance of the style cost
10
11    Returns:
12        J -- total cost
13    """
14    J = alpha * J_content + beta * J_style
15
16    return J

```

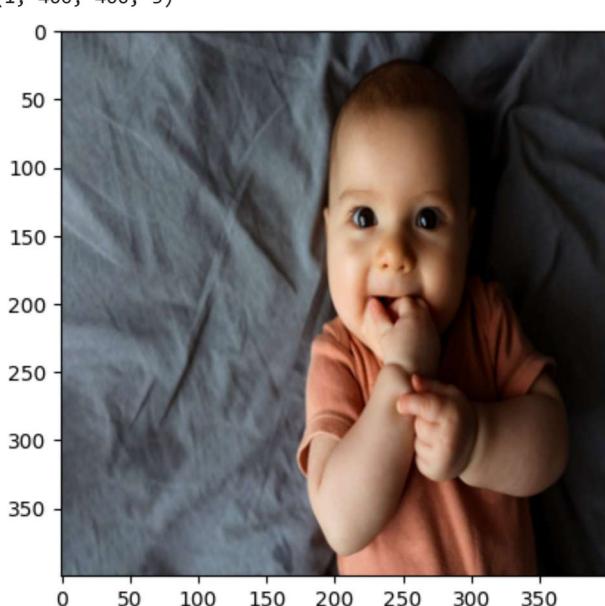
Load the Content Image and Resizing it to (400,400) and converting it to a tensor

```

1 content_image = np.array(Image.open("/content/baby.jpg").resize((400,400)))
2 content_image = tf.constant(np.reshape(content_image, ((1,) + content_image.shape)))
3
4 print(content_image.shape)
5 imshow(content_image[0])
6 plt.show()

(1, 400, 400, 3)

```



Load the Style Image and resizing and converting it to a tensor

```

1 style_image = np.array(Image.open("/content/Artwork.jpg").resize((img_size, img_size)))
2 style_image = tf.constant(np.reshape(style_image, ((1,) + style_image.shape)))
3
4 print(style_image.shape)
5 imshow(style_image[0])
6 plt.show()

```

(1, 400, 400, 3)



Randomly initialize the image to be Generated

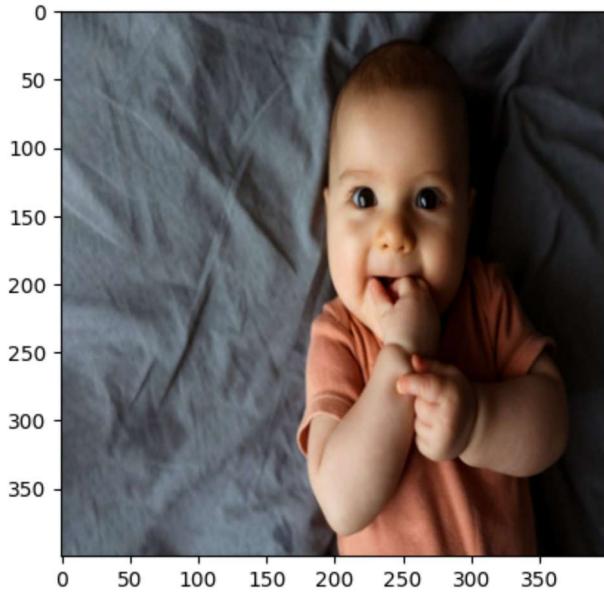
- initializing the "generated" image as content_image.

```

1 generated_image = tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
2 # noise = tf.random.uniform(tf.shape(generated_image), -0.25, 0.25)
3 # generated_image = tf.add(generated_image, noise)
4 # generated_image = tf.clip_by_value(generated_image, clip_value_min=0.0, clip_value_max=1.0)
5
6 print(type(generated_image))
7 imshow(generated_image.numpy()[0])
8 plt.show()

```

<class 'tensorflow.python.ops.resource_variable_ops.ResourceVariable'>



Loading Pre-trained VGG19 Model

```

1 def get_layer_outputs(vgg, layer_names):
2     """ Creates a vgg model that returns a list of intermediate output values."""
3     outputs = [vgg.get_layer(layer[0]).output for layer in layer_names]
4
5     model = tf.keras.Model([vgg.input], outputs)
6     return model

```

defining the content layer and build the model.

```

1 content_layer = [('block5_conv4', 1)]
2
3 vgg_model_outputs = get_layer_outputs(vgg, STYLE_LAYERS + content_layer)

1 content_target = vgg_model_outputs(content_image) # Content encoder
2 style_targets = vgg_model_outputs(style_image)      # Style encoder

```

Compute the Content image Encoding (a_C)

```

1 # Assign the content image to be the input of the VGG model.
2 # Set a_C to be the hidden layer activation from the layer we have selected
3 preprocessed_content = tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
4 a_C = vgg_model_outputs(preprocessed_content)

```

1

Compute the Style image Encoding (a_S)

The code below sets a_S to be the tensor giving the hidden layer activation for 'STYLE_LAYERS' using our style image.

```

1 # Assign the input of the model to be the "style" image
2 preprocessed_style = tf.Variable(tf.image.convert_image_dtype(style_image, tf.float32))
3 a_S = vgg_model_outputs(preprocessed_style)

1 def clip_0_1(image):
2     """
3         Truncate all the pixels in the tensor to be between 0 and 1
4     """
5     return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)

1 def tensor_to_image(tensor):
2     """
3         Converts the given tensor into a PIL image
4     """
5     tensor = tensor * 255
6     tensor = np.array(tensor, dtype=np.uint8)
7     if np.ndim(tensor) > 3:
8         assert tensor.shape[0] == 1
9         tensor = tensor[0]
10    return Image.fromarray(tensor)

```

Train_step

Implementing the train_step() function for transfer learning

- Using the Adam optimizer to minimize the total cost J.
- Using a learning rate of 0.01

- `tf.GradientTape()`:

- Computing the encoding of the generated image using `vgg_model_outputs`. Assign the result to `a_G`.
- Computing the total cost `J`, using the global variables `a_C`, `a_S` and the local `a_G`
- Using '`alpha = 10`' and '`beta = 40`'.

```

1 print("TensorFlow version:", tf.__version__)
TensorFlow version: 2.14.0

1 optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
2 @tf.function()
3 def train_step(generated_image, alpha = 10, beta = 40):
4     # generated_image = tf.Variable(initial_value=generated_image, trainable=True)
5     with tf.GradientTape() as tape:
6         tape.watch(generated_image)
7         # Computing a_G as the vgg_model_outputs for the current generated image
8         a_G = vgg_model_outputs(generated_image)
9         J_style = compute_style_cost(a_S, a_G)
10        # print(J_style.shape)
11        J_content = compute_content_cost(a_C, a_G)
12        # print(J_content.shape)
13        # Computing the total cost
14        J = total_cost(J_content, J_style, alpha = alpha, beta = beta)
15        print(type(J), J.shape)
16        # print(J.shape)
17
18
19        grad = tape.gradient(J, generated_image)
20        print(type(grad), grad.shape)
21        print(type(generated_image), generated_image.shape)
22        optimizer.apply_gradients([(grad, generated_image)])
23        generated_image.assign(clip_0_1(generated_image))
24        # return J

```

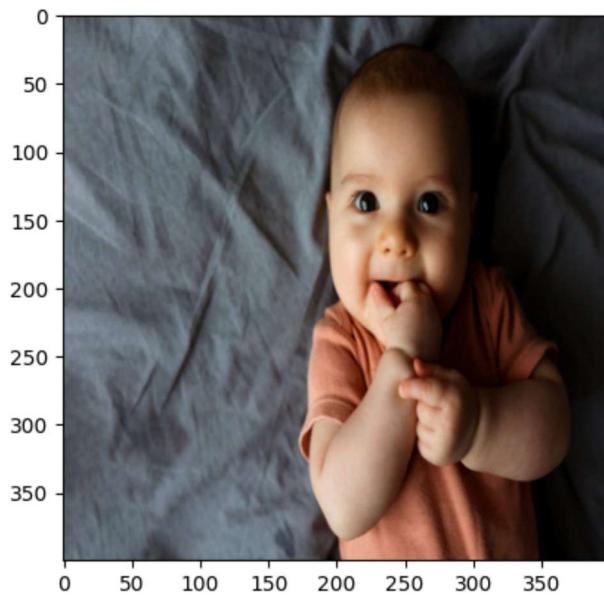
Training the Model

```

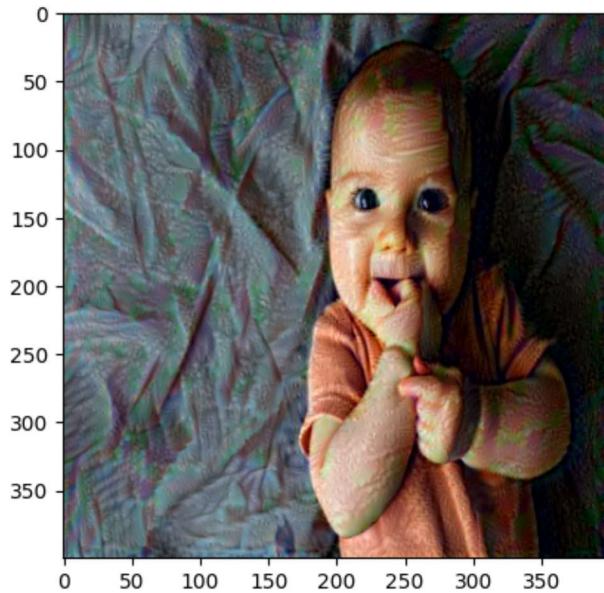
1 # Showing the generated image at some epochs
2 epochs = 20001
3 for i in range(epochs):
4     train_step(generated_image)
5     if i % 500 == 0:
6         print(f"Epoch {i} ")
7         image = tensor_to_image(generated_image)
8         imshow(image)
9         plt.show()

```

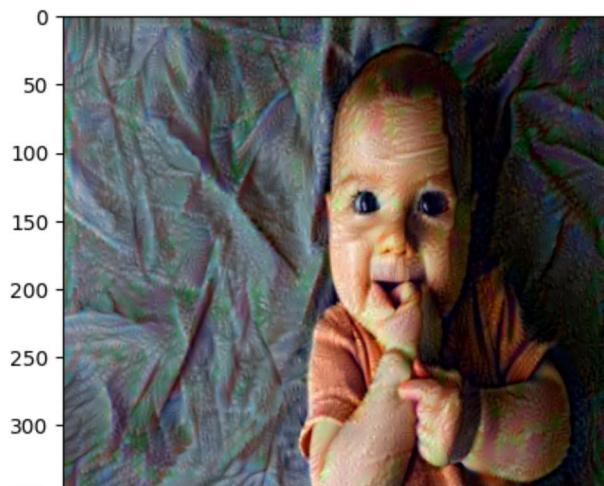
```
<class 'tensorflow.python.framework.ops.SymbolicTensor'> ()  
<class 'tensorflow.python.framework.ops.SymbolicTensor'> (1, 400, 400, 3)  
<class 'tensorflow.python.ops.resource_variable_ops.ResourceVariable'> (1, 400, 400, 3)  
<class 'tensorflow.python.framework.ops.SymbolicTensor'> ()  
<class 'tensorflow.python.framework.ops.SymbolicTensor'> (1, 400, 400, 3)  
<class 'tensorflow.python.ops.resource_variable_ops.ResourceVariable'> (1, 400, 400, 3)  
Epoch 0
```



Epoch 500

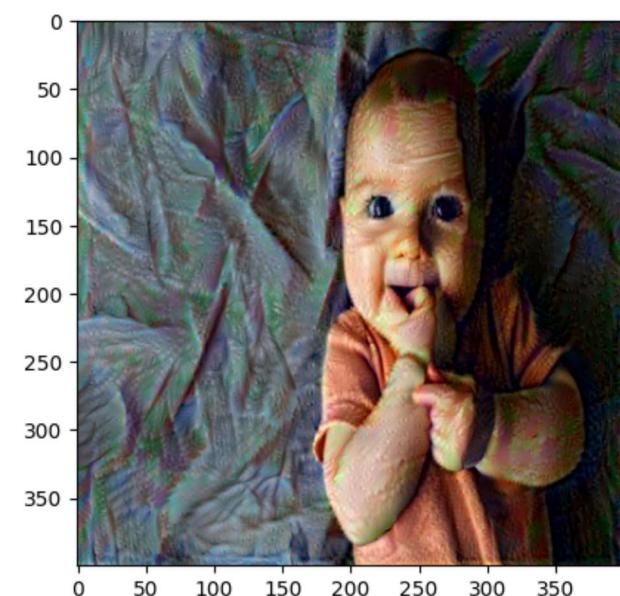


Epoch 1000

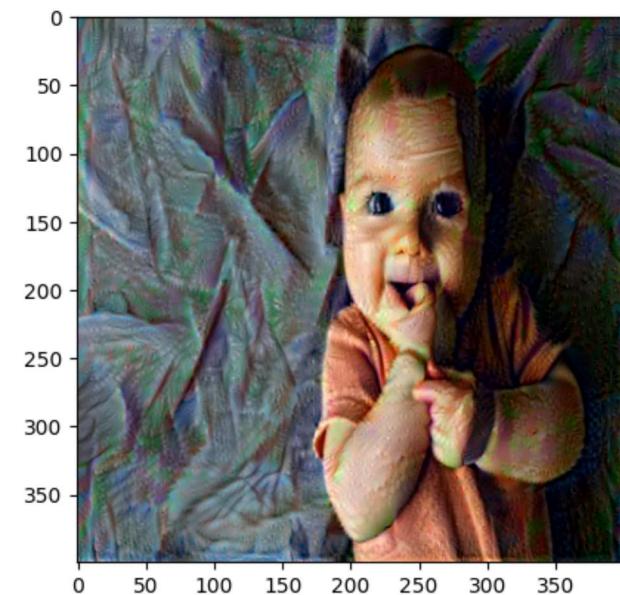




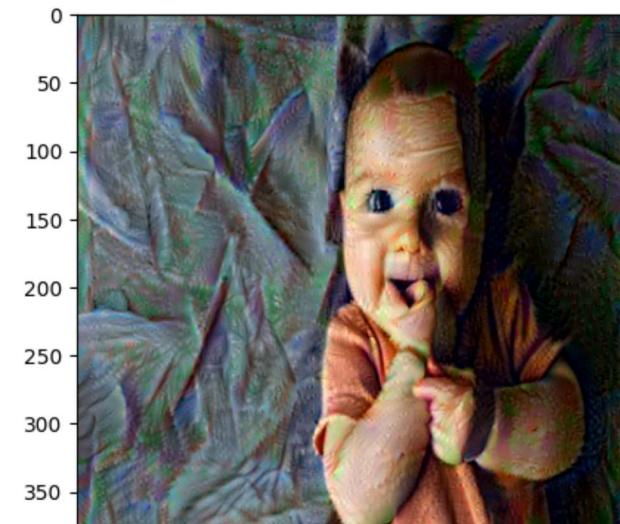
Epoch 1500



Epoch 2000

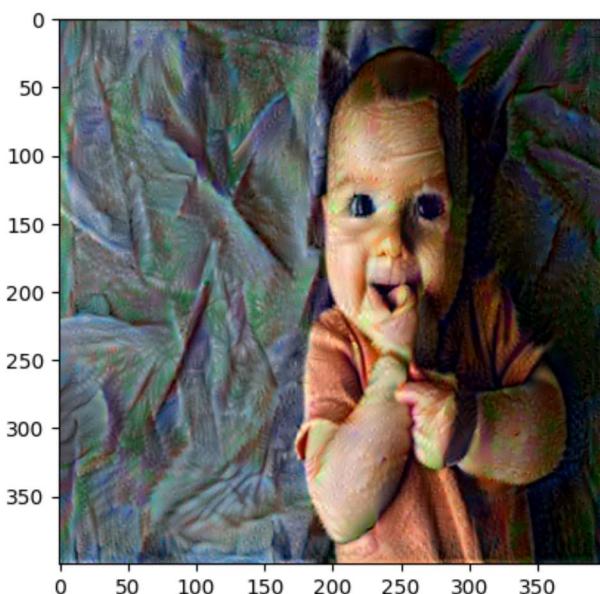


Epoch 2500

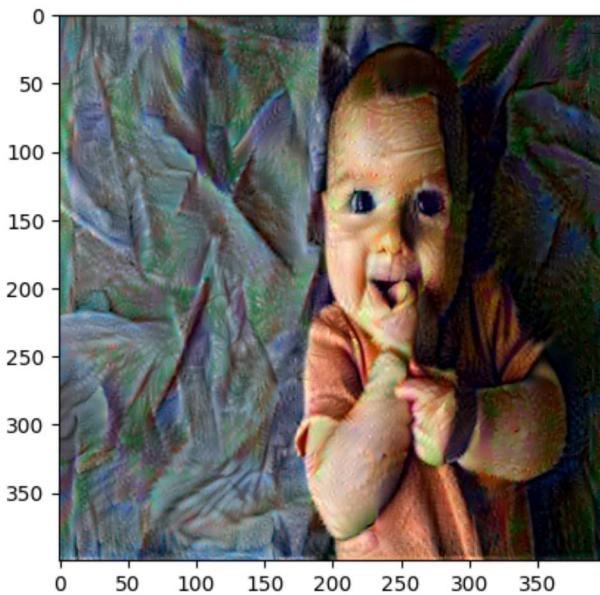




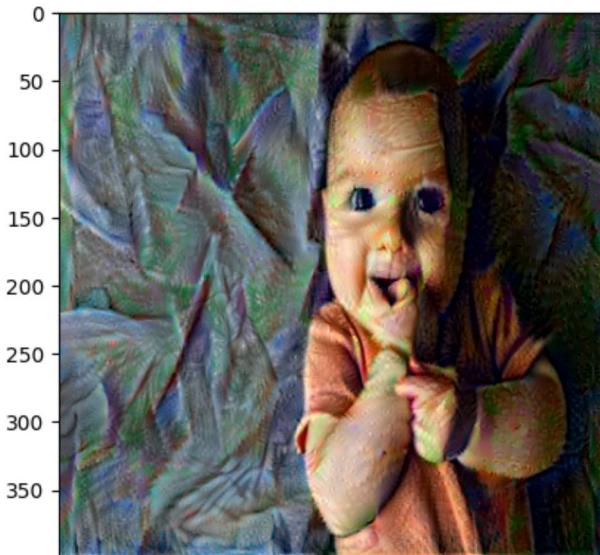
Epoch 3000

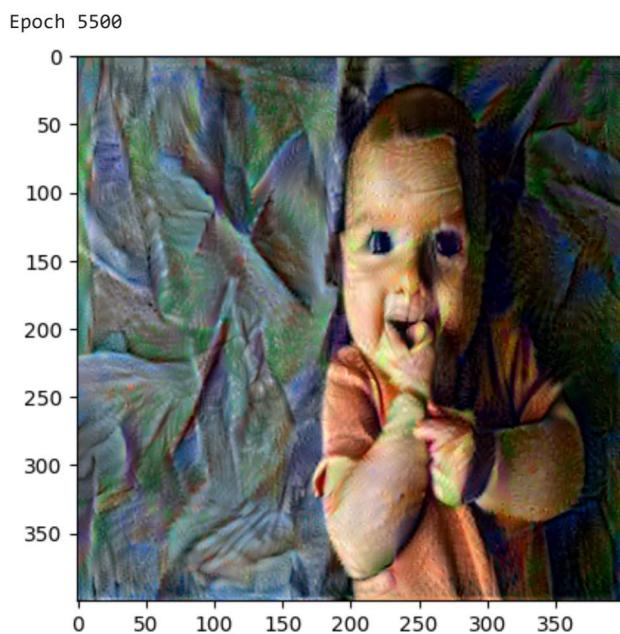
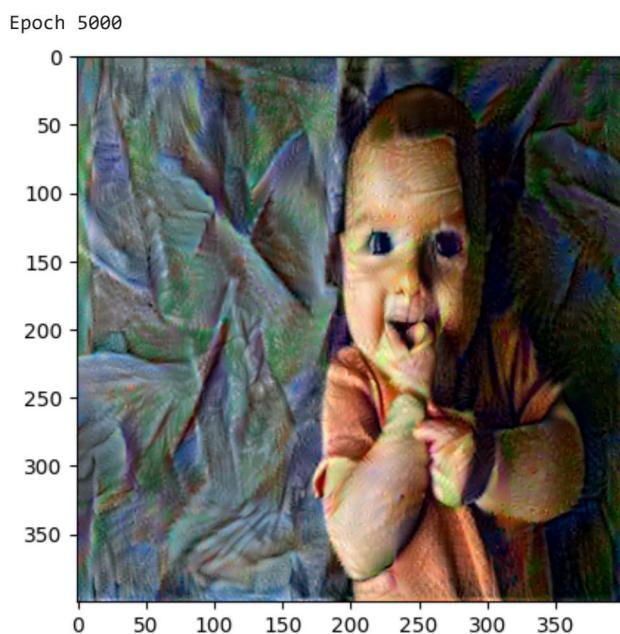
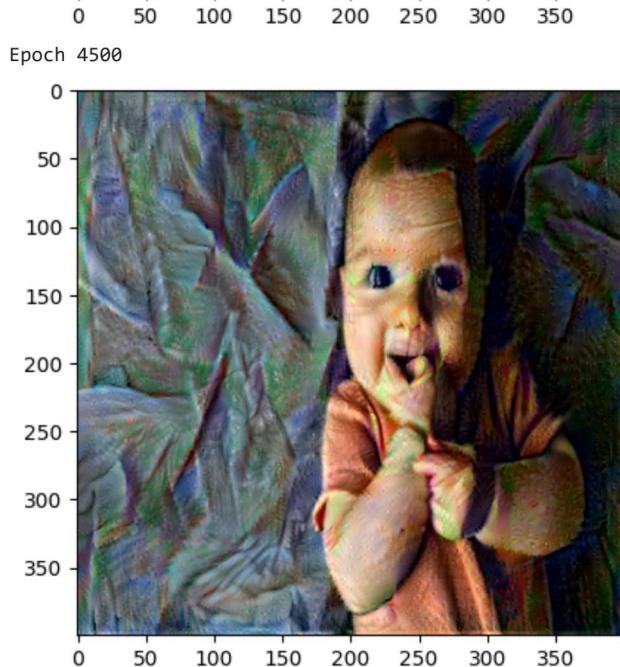


Epoch 3500

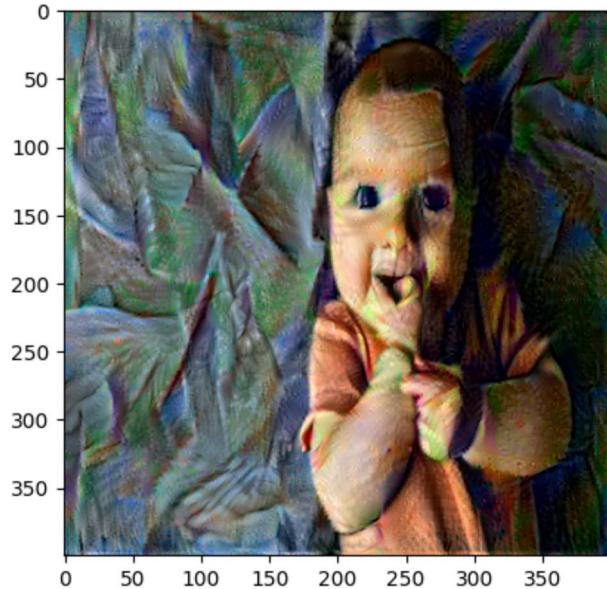


Epoch 4000

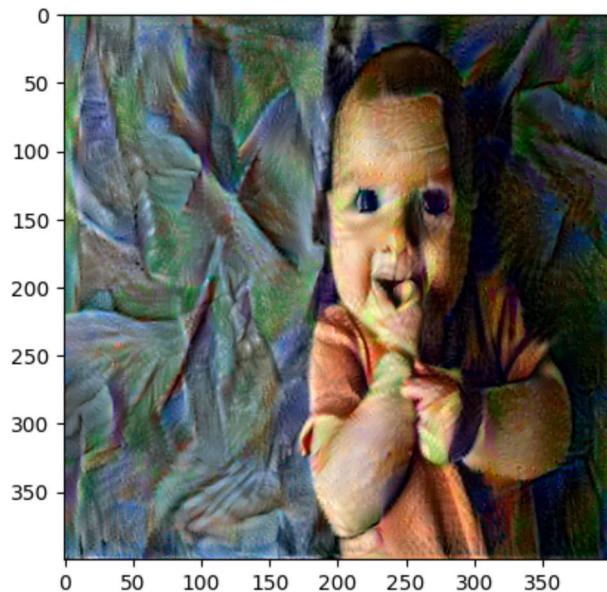




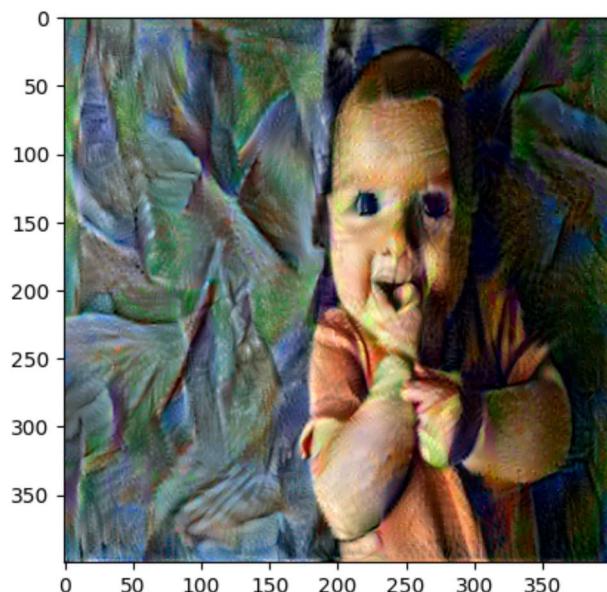
Epoch 6000



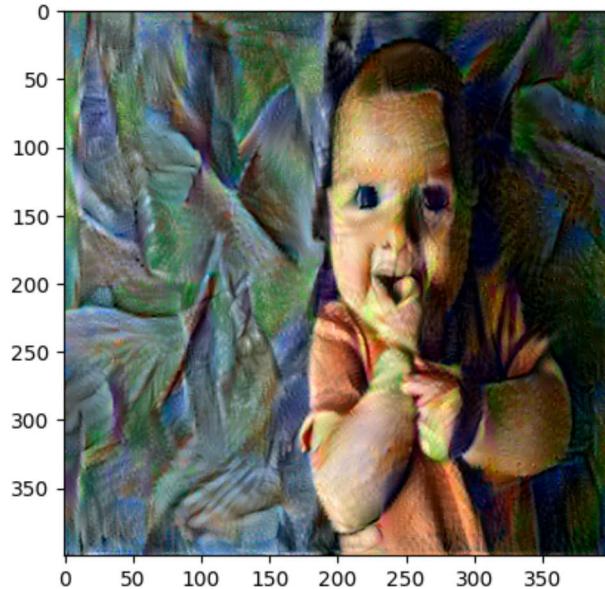
Epoch 6500



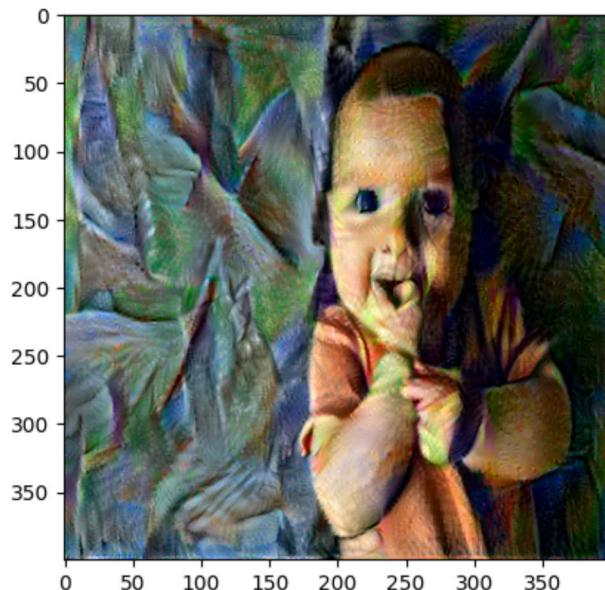
Epoch 7000



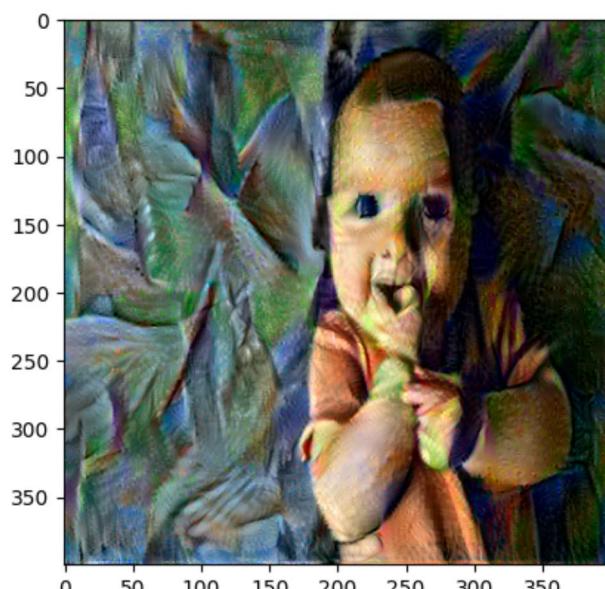
Epoch 7500



Epoch 8000

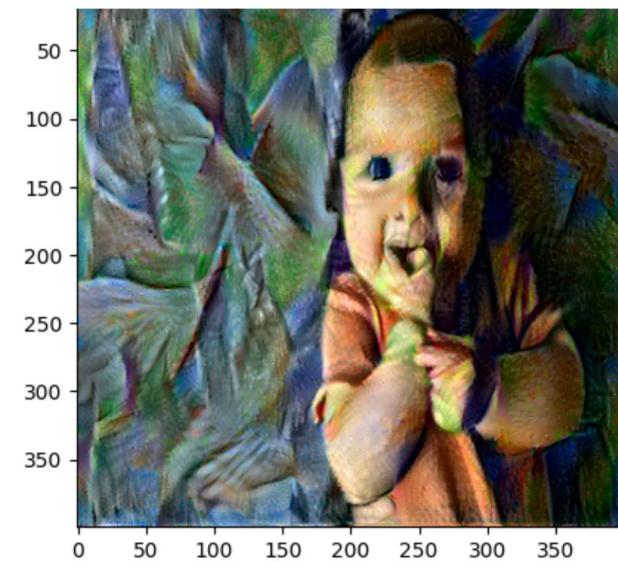


Epoch 8500

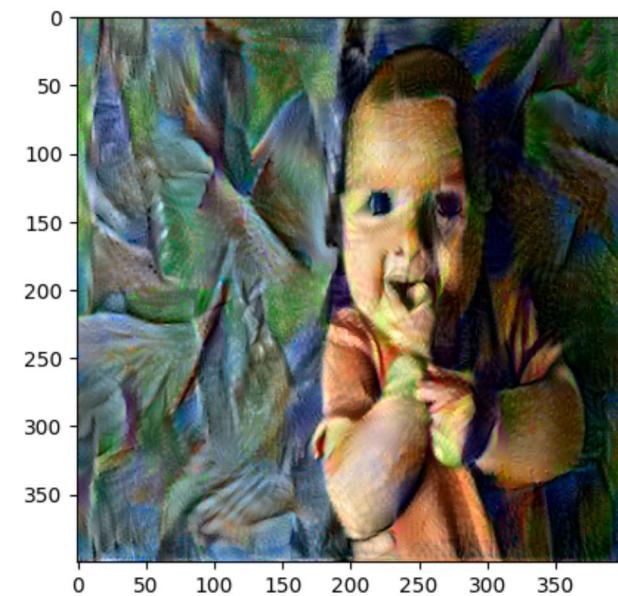


Epoch 9000

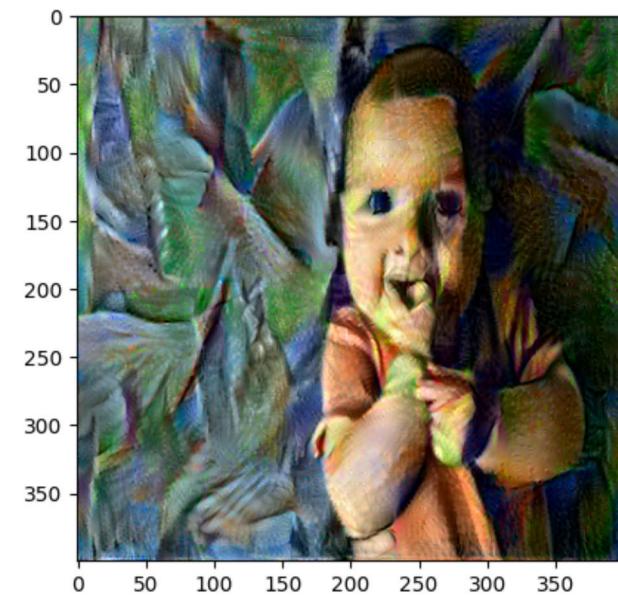




Epoch 9500

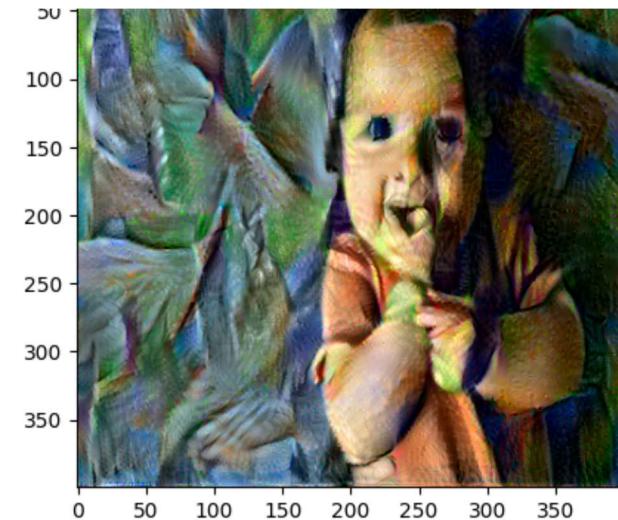


Epoch 10000

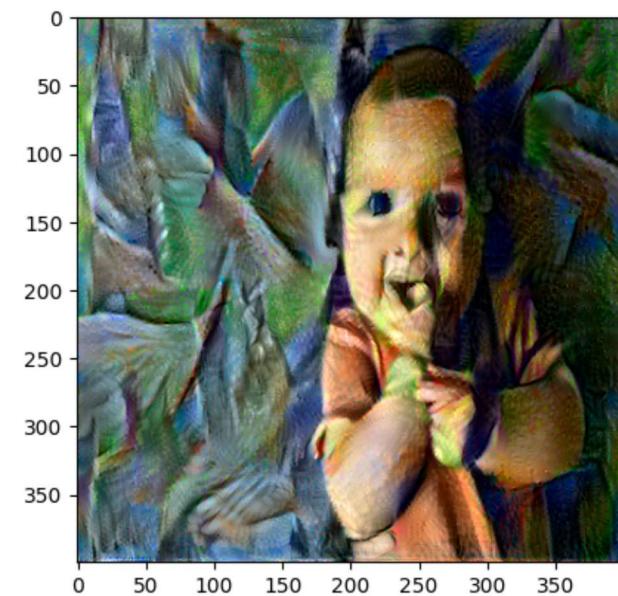


Epoch 10500

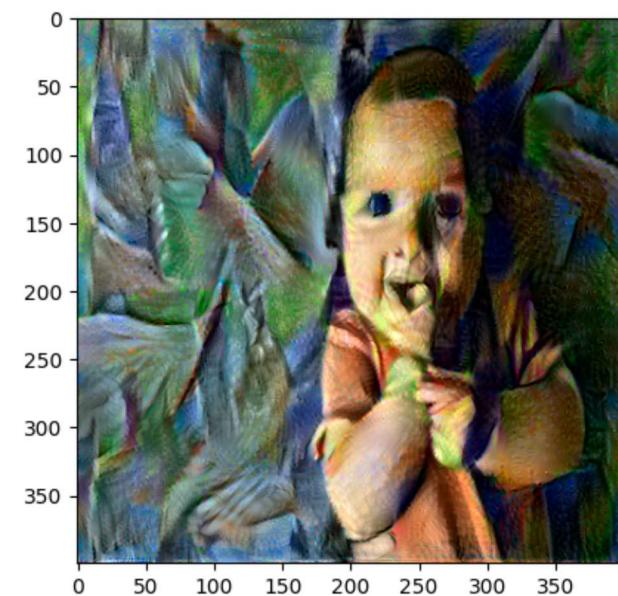




Epoch 11000



Epoch 11500

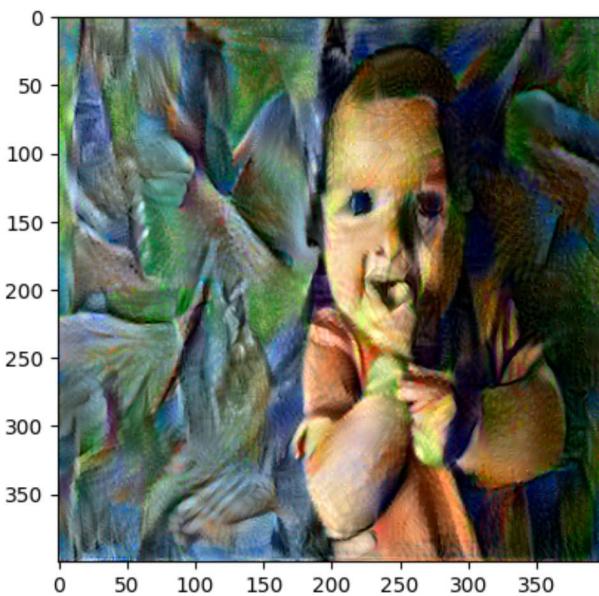


Epoch 12000

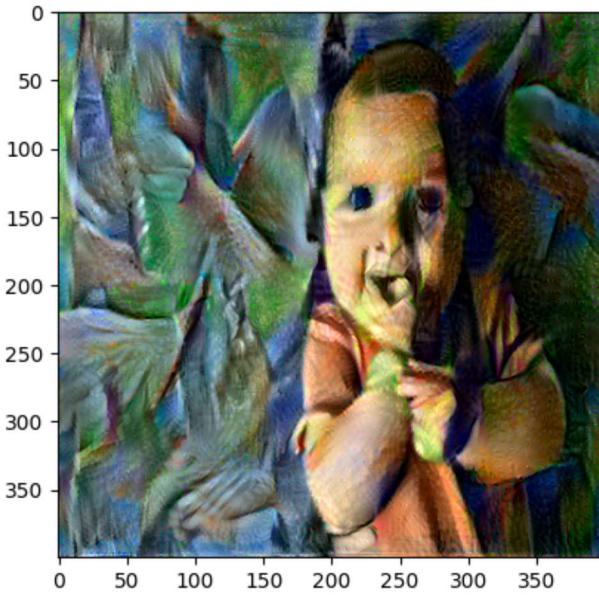




Epoch 12500

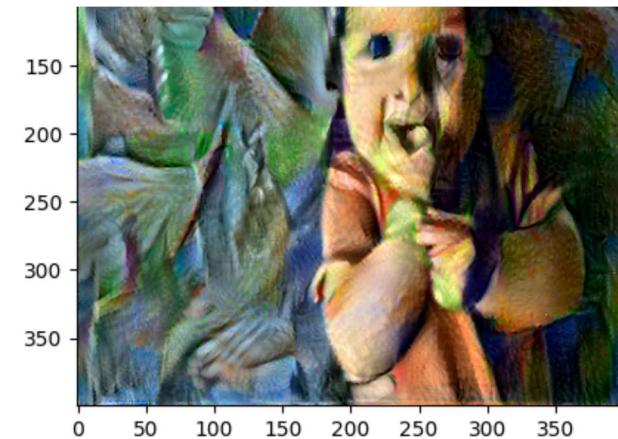


Epoch 13000

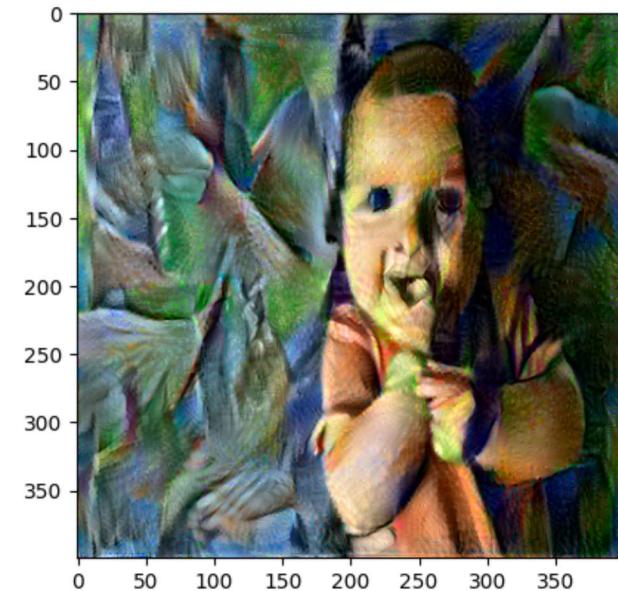


Epoch 13500

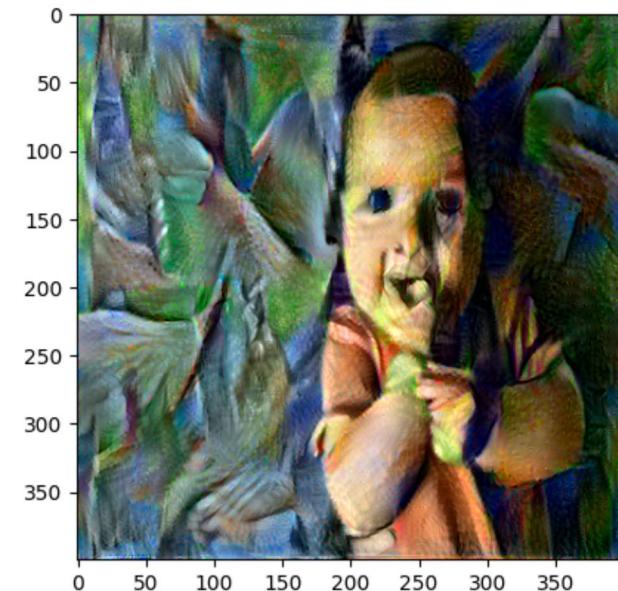




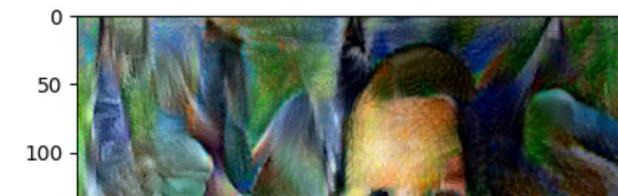
Epoch 14000

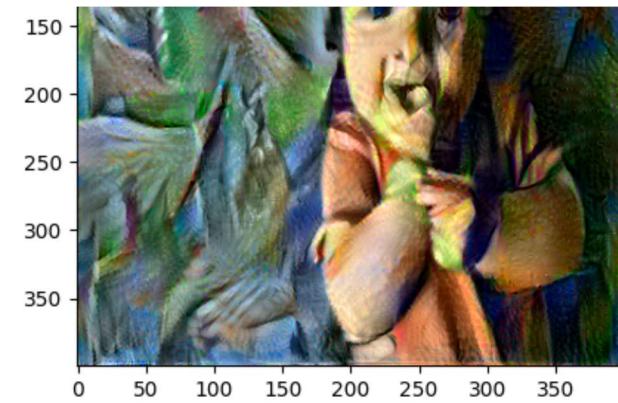


Epoch 14500

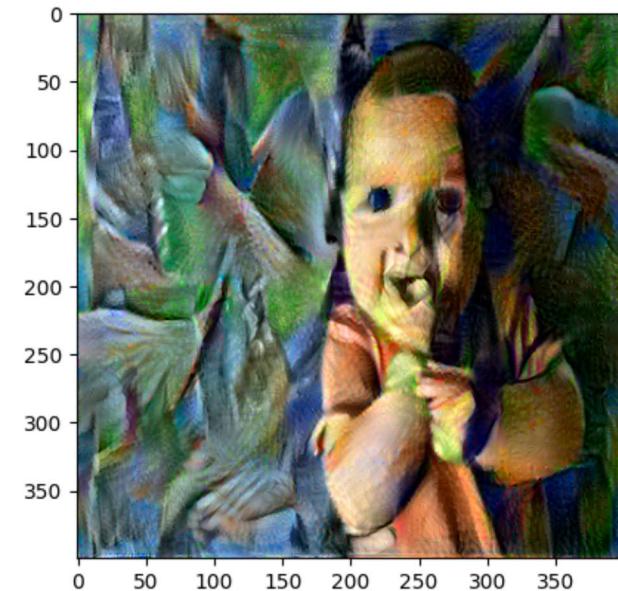


Epoch 15000

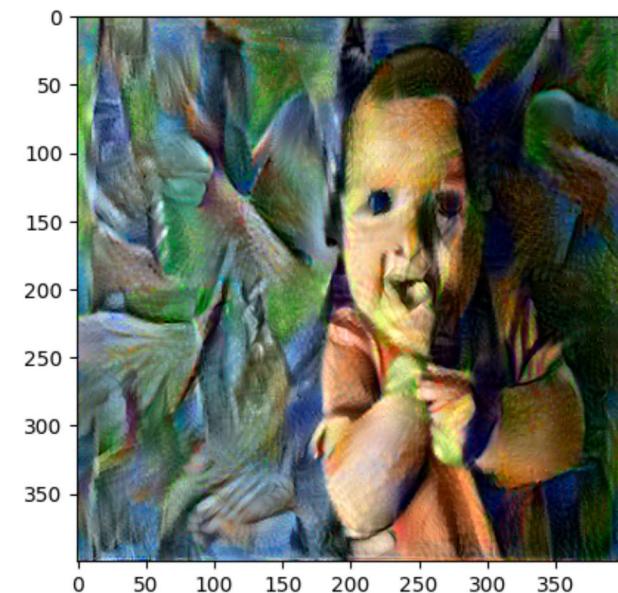




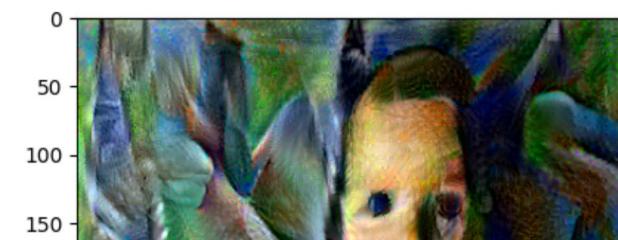
Epoch 15500

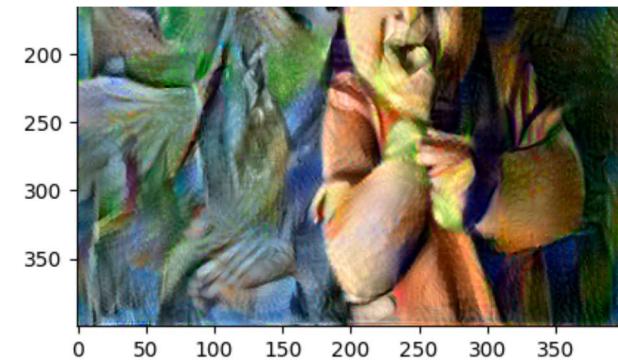


Epoch 16000

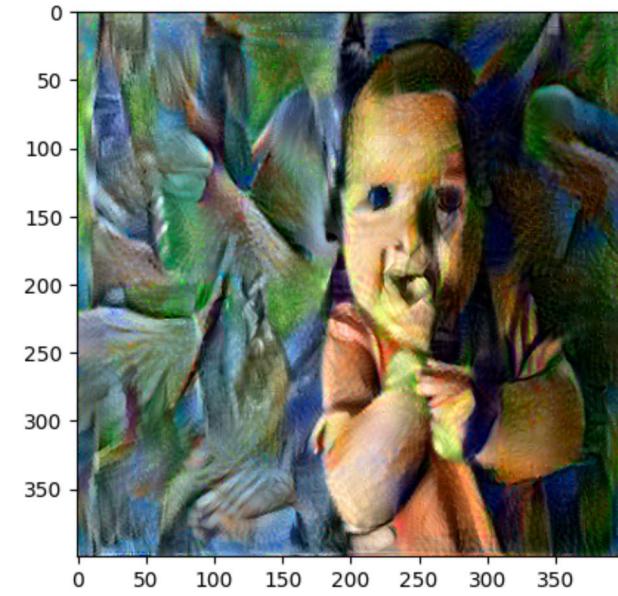


Epoch 16500

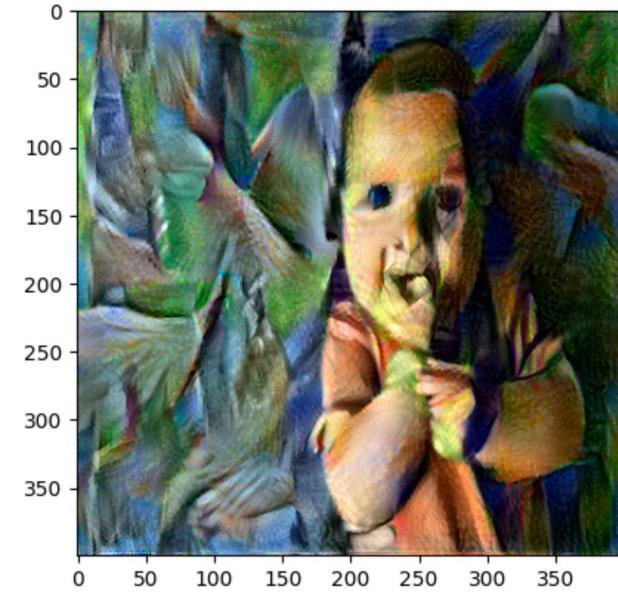




Epoch 17000

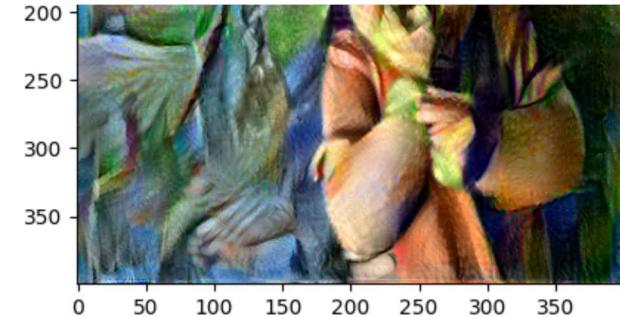


Epoch 17500

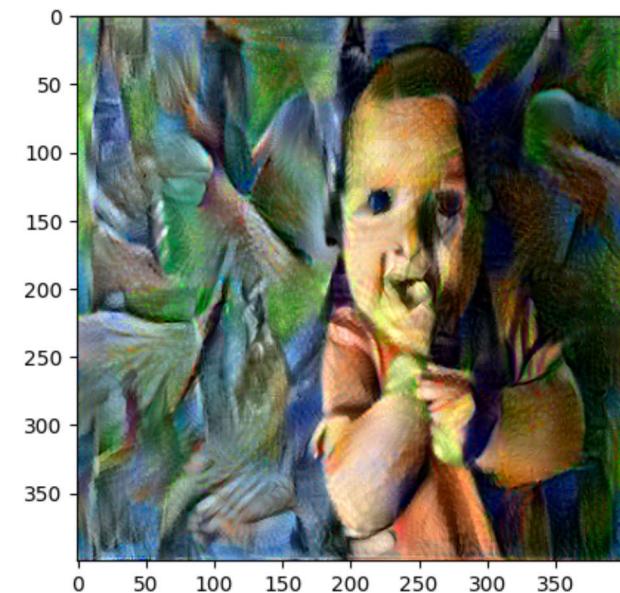


Epoch 18000

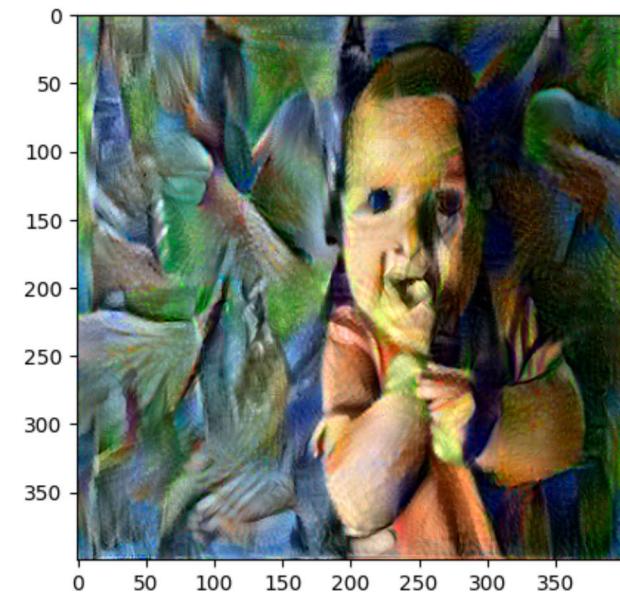




Epoch 18500

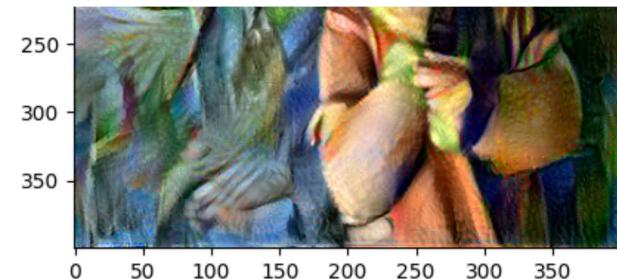


Epoch 19000



Epoch 19500





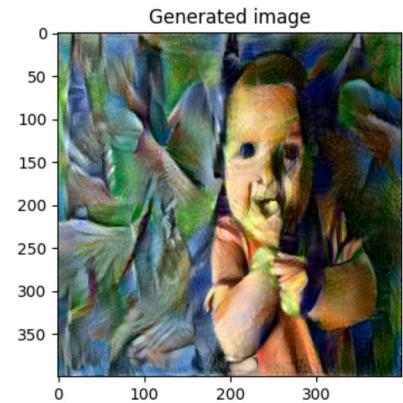
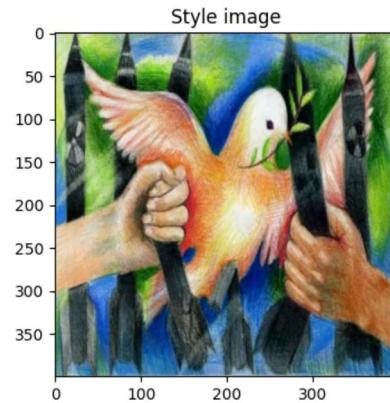
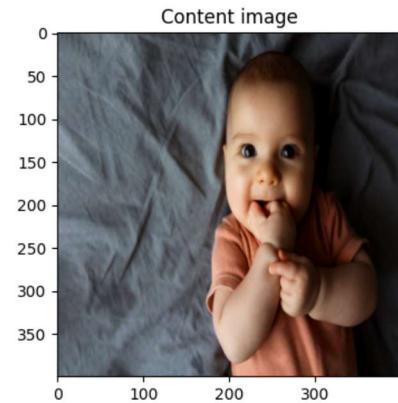
Epoch 20000



```

1 # Show the 3 images in a row
2 fig = plt.figure(figsize=(16, 4))
3 ax = fig.add_subplot(1, 3, 1)
4 imshow(content_image[0])
5 ax.title.set_text('Content image')
6 ax = fig.add_subplot(1, 3, 2)
7 imshow(style_image[0])
8 ax.title.set_text('Style image')
9 ax = fig.add_subplot(1, 3, 3)
10 imshow(generated_image[0])
11 ax.title.set_text('Generated image')
12 plt.show()

```



Quantitative evaluation of style transfer

- Reference- Zhizhong Wang, Lei Zhao, Haibo Chen, Zhiwen Zuo, Ailin Li, Wei Xing, Dongming Lu, **Evaluate and improve the quality of neural style transfer**, *Computer Vision and Image Understanding*, Volume 207, 2021, 103203, ISSN 1077-3142, <https://doi.org/10.1016/j.cviu.2021.103203>.

Content fidelity (CF) -factor to measure the faithfulness to content characteristics in multiple scales.

$$CF\left(\vec{x}, \vec{c}\right) = \frac{1}{N} \sum_{l=1}^N \frac{f_l(\vec{x}) \cdot f_l(\vec{c})}{\|f_l(\vec{x})\| \cdot \|f_l(\vec{c})\|}$$

Global Effects (GE)- When comparing the transferred style with the style image, people often appreciate the global effects first. The global similarity will leave a preliminary impression on humans' visual perception, so it is an important factor for measuring the stylization quality.global effect (GE) factor also includes these two aspects, namely, global colors (GC) and holistic textures (HT).

1