EEC 289A Assignment 2 Report

Chenye Yang, Hanchu Zhou, Haodong Liang, Yibo Ma

1 Introduction

Texture synthesis has been an active research topic in computer vision both as a way to verify texture analysis methods, as well as in its own right. Potential applications of a successful texture synthesis algorithm are broad, including occlusion fill-in, lossy image and video compression, foreground removal, etc.

This project seeks to reproduce and expand upon the work by Efros & Leung (1999) on non-parametric texture synthesis [1]. Originally, this method was groundbreaking for its ability to synthesize new texture fields that maintain the visual characteristics of a given sample texture. Our primary objective is to apply this technique to the textures assigned in the class, exploring both the effectiveness and the limitations of the method in replicating and scaling up these specific patterns.

In addition to replication, this project will extend the methodology to other images and contexts of interest. Particularly, we aim to test the feasibility of applying texture synthesis techniques to generate MNIST digits and a galaxy picture[1], which present a unique challenge in scaling the approach to handle more complex and structured data.

2 Algorithm Overview

This algorithm synthesizes texture by expanding pixel by pixel from an initial seed point. It uses a single pixel, p, as the synthesis unit to capture as much high-frequency information as possible. All previously synthesized pixels within a square window surrounding p (with weights to highlight local structures) serve as the context. To continue the synthesis, probability tables for the distribution of p are needed for all possible contexts. However, while these tables are typically manageable in size for textual data, constructing them explicitly for textures is impractical. An approximation might be achieved using various clustering techniques, but we opt not to construct a model at all. Instead, each new context queries the sample image, and the distribution of p is created as a histogram of all possible values found in the sample image, as demonstrated in Figure 1. This non-parametric sampling method, though simple, is highly effective at capturing statistical processes for which an adequate model has not yet been developed.

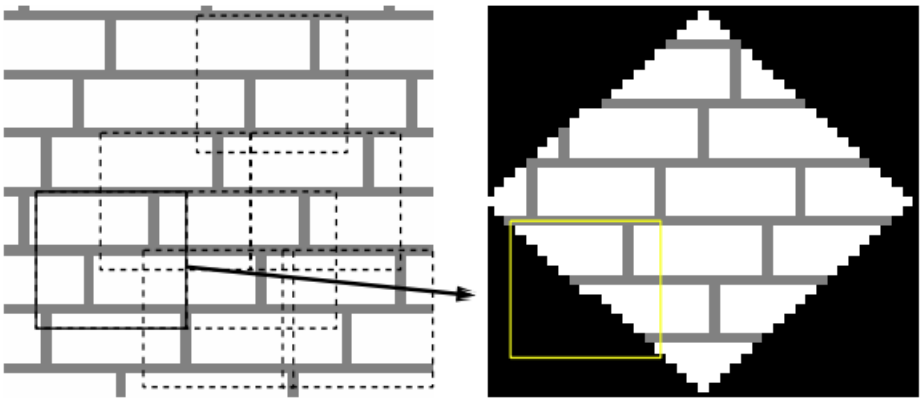


Figure 1: Algorithm Overview

In this work, the texture is modeled as a Markov Random Field (MRF). That is, we assume that the probability distribution of brightness values for a pixel, given the brightness values of its spatial neighborhood, is independent of the rest of the image. The neighborhood of a pixel is modeled as a square window around

[1]Galaxy starry night sky background, free public domain CC0 photo. `https://www.rawpixel.com/image/5924106/photo-image-public-domain-stars-galaxy`

that pixel. The size of the window is a free parameter that specifies how stochastic the user believes this texture to be. More specifically, if the texture is presumed to be mainly regular at high spatial frequencies and mainly stochastic at low spatial frequencies, the size of the window should be on the scale of the biggest regular feature.

2.1 Synthesizing one pixel

Let I be an image that is being synthesized from a texture sample image $I_{\text{smp}} \subset I_{\text{real}}$ where I_{real} is the real infinite texture. Let $p \in I$ be a pixel and let $\omega(p) \subset I$ be a square image patch of width w centered at p. Let $d_{\text{perc}}(\omega_1, \omega_2)$ denote some perceptual distance between two patches. Assume that all pixels in I except for p are known. To synthesize the value of p we first construct an approximation to the conditional probability distribution $P(p \mid \omega(p))$ and then sample from it.

Based on our MRF model, we assume that p is independent of $I \setminus \omega(p)$ given $\omega(p)$. Define a set

$$\Omega(p) = \{\omega' \subset I_{\text{real}} : d_{\text{perc}}(\omega_0, \omega(p)) = 0\}$$

containing all occurrences of $\omega(p)$ in I_{real}, then the conditional PDF of p can be estimated with a histogram of all center pixel values in $\Omega(p)$. Unfortunately, we are only given I_{smp}, a finite sample from I_{real}, which means there might not be any matches for $\omega(p)$ in I_{smp}. Thus, we must use a heuristic which will let us find a plausible $\Omega'(p) \approx \Omega(p)$ to sample from. In our implementation, a variation of the k-Nearest Neighbors technique is used: the closest match $\omega'_{\text{best}} = \arg\min_{\omega} d_{\text{perc}}(\omega(p), \omega) \in I_{\text{smp}}$ is found, and all image patches ω_0 with $d_{\text{perc}}(\omega'_{\text{best}}, \omega') < \epsilon$ are included in Ω', where ϵ is a threshold. The center pixel values of patches in Ω' give us a histogram for p, which can then be sampled, either uniformly or weighted by d_{perc}.

Now it only remains to find a suitable d_{perc}. One choice is a normalized sum of squared differences metric d_{SSD}. However, this metric gives the same weight to any mismatched pixel, whether near the center or at the edge of the window. Since we would like to preserve the local structure of the texture as much as possible, the error for nearby pixels should be greater than for pixels far away. To achieve this effect we set $d_{\text{perc}} = d_{\text{SSD}} \cdot G$ where G is a two-dimensional Gaussian kernel.

2.2 Synthesizing texture

In the previous section, we discussed a method of synthesizing a pixel when its neighborhood pixels are already known. Unfortunately, this method cannot be used for synthesizing the entire texture or even for hole-filling (unless the hole is just one pixel) since for any pixel the values of only some of its neighborhood pixels will be known. The correct solution would be to consider the joint probability of all pixels together, but this is intractable for images of realistic size.

Instead, a Shannon-inspired heuristic is proposed, where the texture is grown in layers outward from a 3-by-3 seed taken randomly from the sample image (in case of hole-filling, the synthesis proceeds from the edges of the hole). Now for any point p to be synthesized only some of the pixel values in $\omega(p)$ are known (i.e., have already been synthesized). Thus, the pixel synthesis algorithm must be modified to handle unknown neighborhood pixel values. This can be easily done by only matching on the known values in $\omega(p)$ and normalizing the error by the total number of known pixels when computing the conditional pdf for p. This heuristic does not guarantee that the pdf for p will stay valid as the rest of $\omega(p)$ is filled in. However, it appears to be a good approximation in practice. One can also treat this as an initialization step for an iterative approach such as Gibbs sampling. However, our trials have shown that Gibbs sampling produced very little improvement for most textures. This lack of improvement indicates that the heuristic indeed provides a good approximation to the desired conditional pdf.

3 Experiments

3.1 Textures

We apply the non-parametric texture synthesis method to the following textures: the textures provided in the class, the MNIST digits, and the galaxy picture we chosen. The textures provided in the class have the size around 530×530, shown in Figure 2.

(a) Texture 2 (b) Texture 4 (c) Texture 9

Figure 2: Texture: provided in the class

For the MNIST digits, we load the MNIST dataset $(28 \times 28$ handwritten digits$)^2$, including 60000 training images and 10000 testing images. We randomly select 20×20 images from the training set as the sample texture of size 560×560, shown in Figure 3.

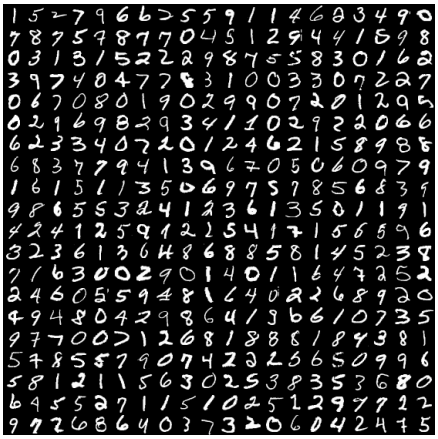


Figure 3: Texture: MNIST digits

For the galaxy picture, we load the galaxy picture (800×585), shown in Figure 4.

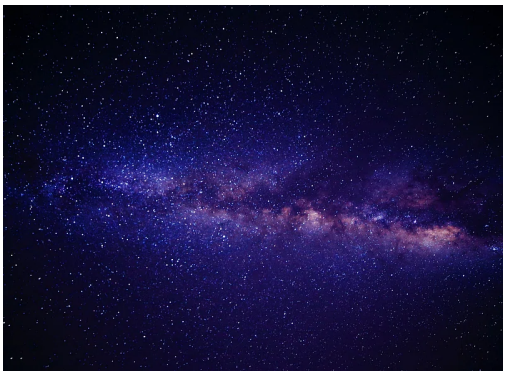


Figure 4: Texture: galaxy picture

[2]Downloaded from `https://git-disl.github.io/GTDLBench/datasets/mnist_datasets/`

3.2 Non-parametric Texture Synthesis

The algorithm proposed by Efros & Leung (1999) [1], Algorithm 1, synthesizes texture by modeling the image as a Markov Random Field (MRF) where each pixel's value is dependent only on the values in its immediate neighborhood. The texture synthesis process grows the new image pixel by pixel from an initial seed based on this model. Here are some important features about the algorithm:

- Non-parametric approach: The algorithm does not build an explicit model but estimates pixel distributions dynamically from the sample image.
- Control over randomness: The size of the neighborhood window can be adjusted to control the randomness of the texture, affecting how structured or stochastic the synthesized texture appears.
- Efficiency: A variation of the k Nearest Neighbors technique is employed to find matching neighborhoods quickly.
- Local structure preservation: Emphasis on maintaining the local structure of the texture to ensure visual continuity and realism in the synthesized image.

Algorithm 1 Texture Synthesis by Non-parametric Sampling

Initialize with a small seed taken randomly from the sample image I_{smp}.
while there are unsynthesized pixels in the image I **do**
    Select an unsynthesized pixel p in I.
    Define the neighborhood $N(p)$ of p as a square window centered at p.
    Find all neighborhoods in I_{smp} that are similar to $N(p)$ using a perceptual distance metric d_{perc}.
    Construct a histogram of pixel values from these similar neighborhoods to approximate the conditional probability distribution $P(p|N(p))$.
    Sample from this distribution to set the value of p.
end while

3.3 Parallelization: Multi-core Processing

To speed up the texture synthesis process, we parallelize the algorithm by dividing the image into blocks and synthesizing each block independently, given that the processing of each pixel (and its surrounding window) is somewhat independent. We use the Python `multiprocessing` library to create a pool of worker processes that can work on different blocks concurrently:

- Split the work: Distribute the processing of each pixel to different CPU cores. This can be done by creating a list of tasks where each task contains the information needed to process a pixel.
- Process pool: Use a multiprocessing pool to process these tasks concurrently.
- Result collection: Collect the results from each task and update the main data structures accordingly.

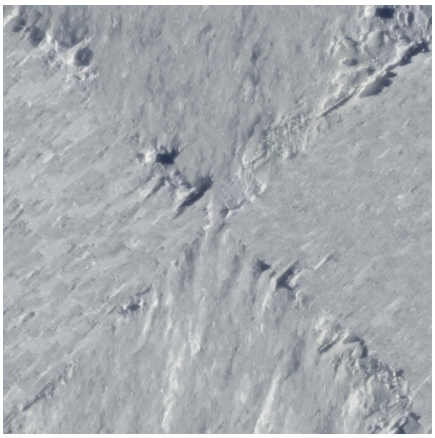
With this parallelization strategy, we can achieve a significant speedup (depending on hardware, around $8\times$) in the texture synthesis process.

4 Results

4.1 Texture Synthesis

In this section, we present the results of applying the non-parametric texture synthesis method to the textures described in the previous section. The patch size is set to 11×11 for all the textures, and the resolution is set to 1000×1000 for all synthesized pictures. The synthesized textures are shown in Figure 5 - 9.

(a) Texture 2



(b) Synthesized Texture 2

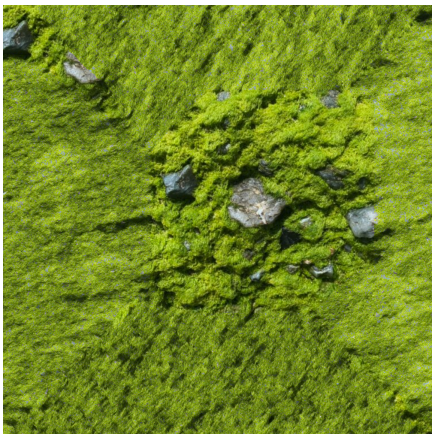
Figure 5: Synthesis of Texture 2

(a) Texture 4

(b) Synthesized Texture 4

Figure 6: Synthesis of Texture 4

(a) Texture 9

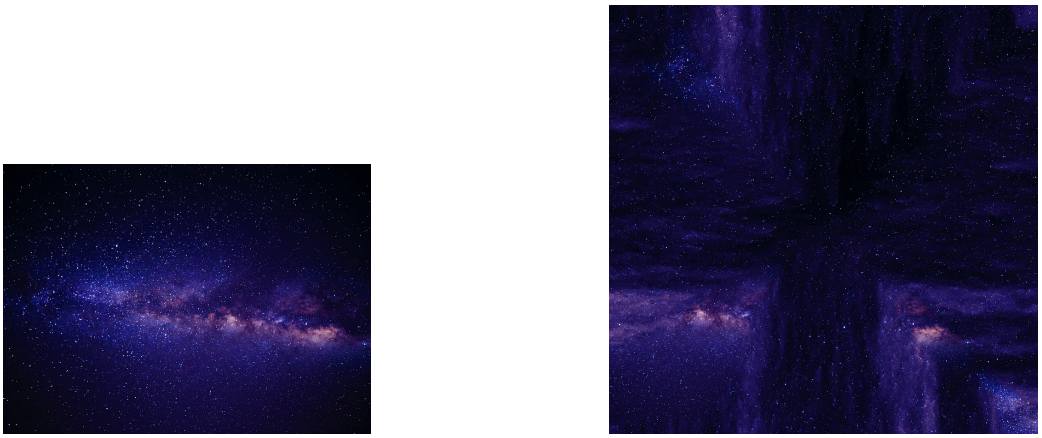


(b) Synthesized Texture 9

Figure 7: Synthesis of Texture 9

(a) MNIST Digits (b) Synthesized MNIST Digits

Figure 8: Synthesis of MNIST Digits

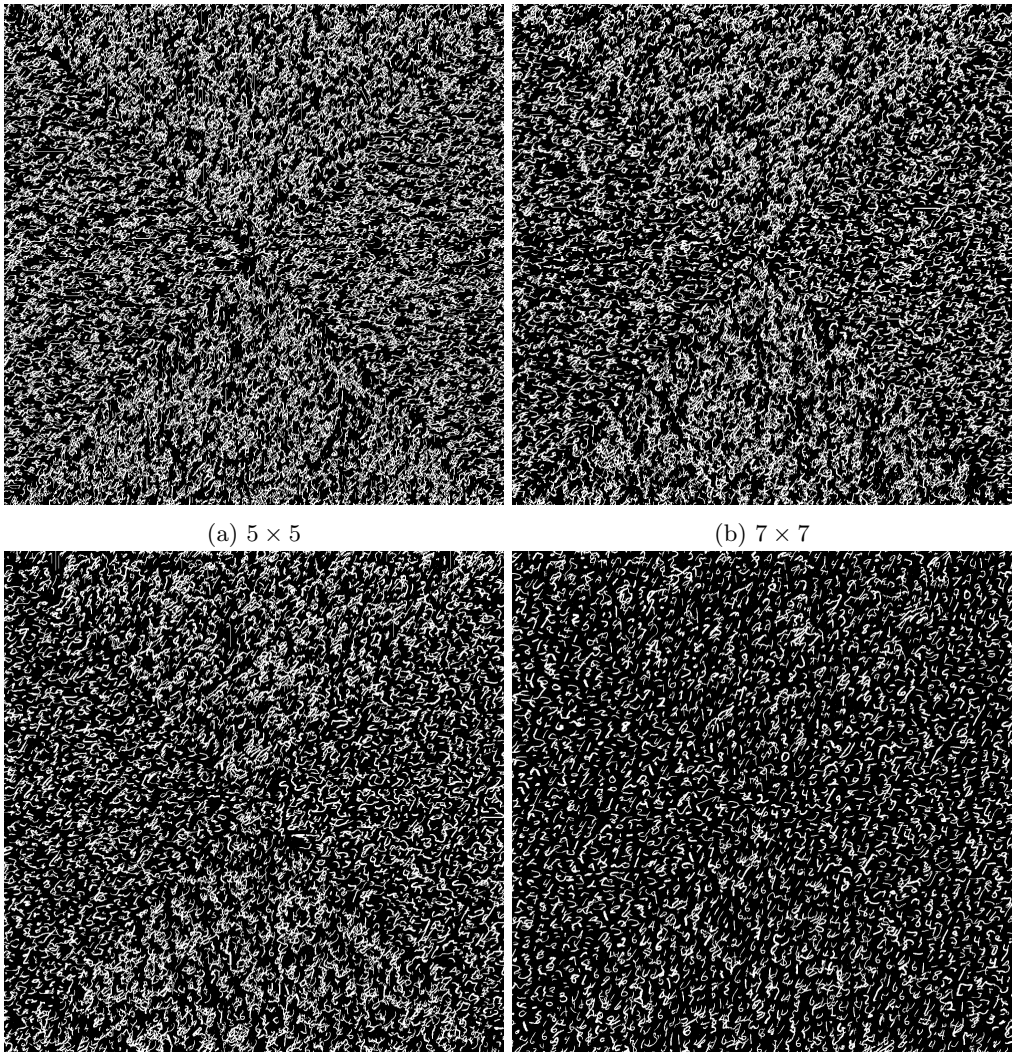


(a) Galaxy Picture (b) Synthesized Galaxy Picture

Figure 9: Synthesis of Galaxy Picture

4.2 Different Neighborhood Patch Sizes

In this section, we present the results of applying the non-parametric texture synthesis method on MNIST with different neighborhood sizes / patch sizes. The results are shown in Figure 10.



(a) 5×5 (b) 7×7

(c) 9×9 (d) 15×15

Figure 10: Different Neighborhood Sizes for MNIST

As we can observe from Figure 10, when we scale the method to synthesize MNIST digits, we need to adjust the patch size to appropriately utilize the neighborhood information. In the above four cases, the patch size of 15×15 produces the best synthesized result.

References

[1] A.A. Efros and T.K. Leung. "Texture synthesis by non-parametric sampling". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1033–1038 vol.2. DOI: 10.1109/ICCV.1999.790383.

Appendix

We referred to the GitHub repo: `https://github.com/goldbema/TextureSynthesis`, modified the code with `multiprocessing` parralelization, and tried `PyTorch` matrix operation speedup.

```
'''
This module is a Python implementation of:

    A. Efros and T. Leung, "Texture Synthesis by Non-parametric Sampling,"
    Proceedings of the Seventh IEEE International Conference on Computer
    Vision, September 1999.

Specifically, this module implements texture synthesis by growing a 3x3 texture patch
pixel-by-pixel. Please see the authors' project page for additional algorithm details:

    https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html

Example:

    Generate a 50x50 texture patch from a texture available at the input path and save it to
    the output path. Also, visualize the synthesis process:

        $ python synthesis.py --sample_path=[input path] --out_path=[output path] --visualize

'''

__author__ = 'Maxwell Goldberg'
__author__ = 'Chenye Yang, Hanchu Zhou, Haodong Liang, Yibo Ma'

import torch
import time
import torch.nn.functional as F
import argparse
import cv2
import numpy as np
from multiprocessing import Pool

EIGHT_CONNECTED_NEIGHBOR_KERNEL = np.array([[1., 1., 1.],
                                            [1., 0., 1.],
                                            [1., 1., 1.]], dtype=np.float64)
SIGMA_COEFF = 6.4       # The denominator for a 2D Gaussian sigma used in the reference implementation.
ERROR_THRESHOLD = 0.1  # The default error threshold for synthesis acceptance in the reference implementation.



# def normalized_ssd(sample, window, mask):
#     if torch.backends.mps.is_available():
#         device = torch.device("mps")
```

```
#     elif torch.cuda.is_available():
#         device = torch.device("cuda")
#     else:
#         device = torch.device("cpu")

#     # Convert inputs to PyTorch tensors and move them to the specified device
#     sample = torch.tensor(sample, dtype=torch.float32).to(device)
#     window = torch.tensor(window, dtype=torch.float32).to(device)
#     mask = torch.tensor(mask, dtype=torch.float32).to(device)

#     # Get dimensions
#     sh, sw = sample.shape
#     wh, ww = window.shape

#     # Compute 2D Gaussian kernel
#     sigma = wh / SIGMA_COEFF  # SIGMA_COEFF should be defined elsewhere in your code
#     kernel = torch.Tensor(cv2.getGaussianKernel(ksize=wh, sigma=sigma)).to(device)
#     kernel_2d = torch.mm(kernel, kernel.t())

#     # Apply the Gaussian kernel to the mask
#     weighted_mask = mask * kernel_2d

#     # Calculate padded size for valid convolution
#     padded_sample = F.pad(sample, (ww//2, ww//2, wh//2, wh//2))

#     # Perform convolution using the flipped window (correlation)
#     window = window.flip([0, 1])
#     ssd_map = F.conv2d(padded_sample[None, None, :, :], window[None, None, :, :], None, stride=1)

#     # Compute sum of squared differences
#     squared_diff = (ssd_map - torch.sum(window)**2)**2

#     # Multiply by the weighted mask and sum over the kernel
#     result_map = F.conv2d(squared_diff, weighted_mask[None, None, :, :], None, stride=1)

#     # Normalize the SSD by the maximum possible contribution
#     total_ssd = torch.sum(weighted_mask)
#     normalized_ssd_map = result_map / total_ssd

#     return normalized_ssd_map[0, 0].cpu().numpy()


# def normalized_ssd(sample, window, mask):
#     if torch.backends.mps.is_available():
#         device = torch.device("mps")
#     elif torch.cuda.is_available():
#         device = torch.device("cuda")
#     else:
#         device = torch.device("cpu")
```

```
93  #     # Ensure all inputs are float32
94  #     sample, window, mask = map(lambda x: torch.tensor(x, dtype=torch.float32
        , device=device), (sample, window, mask))
95
96  #     # Calculate Gaussian kernel in float32, ensure vectors are 1D by
        flattening
97  #     sigma = window.shape[0] / SIGMA_COEFF
98  #     k1 = torch.tensor(cv2.getGaussianKernel(ksize=window.shape[0], sigma=
        sigma).flatten(), dtype=torch.float32, device=device)
99  #     kernel = torch.outer(k1, k1)
100
101 #     # Use unfold to create sliding windows
102 #     unfolded_sample = F.unfold(sample[None, None], kernel_size=window.shape,
         padding=0, stride=1)
103 #     unfolded_sample = unfolded_sample.view(1, window.numel(), sample.shape
        [0]-window.shape[0]+1, sample.shape[1]-window.shape[1]+1)
104
105 #     ssd = (unfolded_sample - window.view(1, -1, 1, 1))**2
106 #     ssd *= kernel.view(1, -1, 1, 1) * mask.view(1, -1, 1, 1)
107 #     ssd = ssd.sum(dim=1)
108
109 #     # Normalize SSD
110 #     total_ssd = torch.sum(mask * kernel)
111 #     normalized_ssd = ssd / total_ssd
112 #     return normalized_ssd.cpu().numpy()
113
114 def normalized_ssd(sample, window, mask):
115     wh, ww = window.shape
116     sh, sw = sample.shape
117
118     # Get sliding window views of the sample, window, and mask.
119     strided_sample = np.lib.stride_tricks.as_strided(sample, shape=((sh-wh+1),
          (sw-ww+1), wh, ww),
120                         strides=(sample.strides[0], sample.strides[1], sample.
                            strides[0], sample.strides[1]))
121     strided_sample = strided_sample.reshape(-1, wh, ww)
122
123     # Note that the window and mask views have the same shape as the strided
         sample, but the kernel is fixed
124     # rather than sliding for each of these components.
125     strided_window = np.lib.stride_tricks.as_strided(window, shape=((sh-wh+1),
          (sw-ww+1), wh, ww),
126                         strides=(0, 0, window.strides[0], window.strides[1]))
127     strided_window = strided_window.reshape(-1, wh, ww)
128
129     strided_mask = np.lib.stride_tricks.as_strided(mask, shape=((sh-wh+1), (sw
        -ww+1), wh, ww),
130                         strides=(0, 0, mask.strides[0], mask.strides[1]))
131     strided_mask = strided_mask.reshape(-1, wh, ww)
132
133     # Form a 2D Gaussian weight matrix from symmetric linearly separable
         Gaussian kernels and generate a
134     # strided view over this matrix.
135     sigma = wh / SIGMA_COEFF
```

```
136     kernel = cv2.getGaussianKernel(ksize=wh, sigma=sigma)
137     kernel_2d = kernel * kernel.T
138 
139     strided_kernel = np.lib.stride_tricks.as_strided(kernel_2d, shape=((sh-wh
            +1), (sw-ww+1), wh, ww),
140                         strides=(0, 0, kernel_2d.strides[0], kernel_2d.strides
                                [1]))
141     strided_kernel = strided_kernel.reshape(-1, wh, ww)
142 
143     # Take the sum of squared differences over all sliding sample windows and
            weight it so that only existing neighbors
144     # contribute to error. Use the Gaussian kernel to weight central values
            more strongly than distant neighbors.
145     squared_differences = ((strided_sample - strided_window)**2) *
            strided_kernel * strided_mask
146     ssd = np.sum(squared_differences, axis=(1,2))
147     ssd = ssd.reshape(sh-wh+1, sw-ww+1)
148 
149     # Normalize the SSD by the maximum possible contribution.
150     total_ssd = np.sum(mask * kernel_2d)
151     normalized_ssd = ssd / total_ssd
152 
153     return normalized_ssd
154 
155 
156 def get_candidate_indices(normalized_ssd, error_threshold=ERROR_THRESHOLD):
157     min_ssd = np.min(normalized_ssd)
158     min_threshold = min_ssd * (1. + error_threshold)
159     indices = np.where(normalized_ssd <= min_threshold)
160     return indices
161 
162 def select_pixel_index(normalized_ssd, indices, method='uniform'):
163     N = indices[0].shape[0]
164 
165     if method == 'uniform':
166         weights = np.ones(N) / float(N)
167     else:
168         weights = normalized_ssd[indices]
169         weights = weights / np.sum(weights)
170 
171     # Select a random pixel index from the index list.
172     selection = np.random.choice(np.arange(N), size=1, p=weights)
173     selected_index = (indices[0][selection], indices[1][selection])
174 
175     return selected_index
176 
177 def get_neighboring_pixel_indices(pixel_mask):
178     # Taking the difference between the dilated mask and the initial mask
179     # gives only the 8-connected neighbors of the mask frontier.
180     kernel = np.ones((3,3))
181     dilated_mask = cv2.dilate(pixel_mask, kernel, iterations=1)
182     neighbors = dilated_mask - pixel_mask
183 
184     # Recover the indices of the mask frontier.
```

```
185     neighbor_indices = np.nonzero(neighbors)
186 
187     return neighbor_indices
188 
189 def permute_neighbors(pixel_mask, neighbors):
190     N = neighbors[0].shape[0]
191 
192     # Generate a permutation of the neigboring indices
193     permuted_indices = np.random.permutation(np.arange(N))
194     permuted_neighbors = (neighbors[0][permuted_indices], neighbors[1][
            permuted_indices])
195 
196     # Use convolution to count the number of existing neighbors for all
            entries in the mask.
197     neighbor_count = cv2.filter2D(pixel_mask, ddepth=-1, kernel=
            EIGHT_CONNECTED_NEIGHBOR_KERNEL, borderType=cv2.BORDER_CONSTANT)
198 
199     # Sort the permuted neighboring indices by quantity of existing neighbors
            descending.
200     permuted_neighbor_counts = neighbor_count[permuted_neighbors]
201 
202     sorted_order = np.argsort(permuted_neighbor_counts)[::-1]
203     permuted_neighbors = (permuted_neighbors[0][sorted_order],
            permuted_neighbors[1][sorted_order])
204 
205     return permuted_neighbors
206 
207 def texture_can_be_synthesized(mask):
208     # The texture can be synthesized while the mask has unfilled entries.
209     mh, mw = mask.shape[:2]
210     num_completed = np.count_nonzero(mask)
211     num_incomplete = (mh * mw) - num_completed
212 
213     return num_incomplete > 0
214 
215 def initialize_texture_synthesis(original_sample, window_size, kernel_size):
216     # Convert original to sample representation.
217     sample = cv2.cvtColor(original_sample, cv2.COLOR_BGR2GRAY)
218 
219     # Convert sample to floating point and normalize to the range [0., 1.]
220     sample = sample.astype(np.float64)
221     sample = sample / 255.
222 
223     # Generate window
224     window = np.zeros(window_size, dtype=np.float64)
225 
226     # Generate output window
227     if original_sample.ndim == 2:
228         result_window = np.zeros_like(window, dtype=np.uint8)
229     else:
230         result_window = np.zeros(window_size + (3,), dtype=np.uint8)
231 
232     # Generate window mask
233     h, w = window.shape
```

```
234     mask = np.zeros((h, w), dtype=np.float64)
235
236     # Initialize window with random seed from sample
237     sh, sw = original_sample.shape[:2]
238     ih = np.random.randint(sh-3+1)
239     iw = np.random.randint(sw-3+1)
240     seed = sample[ih:ih+3, iw:iw+3]
241
242     # Place seed in center of window
243     ph, pw = (h//2)-1, (w//2)-1
244     window[ph:ph+3, pw:pw+3] = seed
245     mask[ph:ph+3, pw:pw+3] = 1
246     result_window[ph:ph+3, pw:pw+3] = original_sample[ih:ih+3, iw:iw+3]
247
248     # Obtain padded versions of window and mask
249     win = kernel_size//2
250     padded_window = cv2.copyMakeBorder(window,
251                                        top=win, bottom=win, left=win, right=
                                               win, borderType=cv2.BORDER_CONSTANT
                                               , value=0.)
252     padded_mask = cv2.copyMakeBorder(mask,
253                                      top=win, bottom=win, left=win, right=win,
                                             borderType=cv2.BORDER_CONSTANT,
                                           value=0.)
254
255     # Obtain views of the padded window and mask
256     window = padded_window[win:-win, win:-win]
257     mask = padded_mask[win:-win, win:-win]
258
259     return sample, window, mask, padded_window, padded_mask, result_window
260
261 def process_pixel(args):
262     sample, window_slice, mask_slice, kernel_size, ch, cw, original_sample =
            args
263     ssd = normalized_ssd(sample, window_slice, mask_slice)
264     indices = get_candidate_indices(ssd)
265     selected_index = select_pixel_index(ssd, indices)
266     selected_index = (selected_index[0] + kernel_size // 2, selected_index[1]
           + kernel_size // 2)
267     return ch, cw, sample[selected_index], original_sample[selected_index[0],
            selected_index[1]]
268
269 def synthesize_texture(original_sample, window_size, kernel_size, visualize):
270     global gif_count
271     (sample, window, mask, padded_window, padded_mask, result_window) =
            initialize_texture_synthesis(original_sample, window_size, kernel_size
            )
272
273     pool = Pool()
274
275     while texture_can_be_synthesized(mask):
276         neighboring_indices = get_neighboring_pixel_indices(mask)
277         neighboring_indices = permute_neighbors(mask, neighboring_indices)
278
```

```
279             tasks = []
280             for ch, cw in zip(neighboring_indices[0], neighboring_indices[1]):
281                 window_slice = padded_window[ch:ch+kernel_size, cw:cw+kernel_size]
282                 mask_slice = padded_mask[ch:ch+kernel_size, cw:cw+kernel_size]
283                 tasks.append((sample, window_slice, mask_slice, kernel_size, ch,
                        cw, original_sample))
284 
285             results = pool.map(process_pixel, tasks)
286 
287             for ch, cw, new_value, result_value in results:
288                 window[ch, cw] = new_value
289                 mask[ch, cw] = 1
290                 result_window[ch, cw] = result_value
291 
292                 if visualize:
293                     cv2.imshow('synthesis window', result_window)
294                     key = cv2.waitKey(1)
295                     if key == 27:
296                         cv2.destroyAllWindows()
297                         pool.close()
298                         pool.join()
299                         return result_window
300 
301     pool.close()
302     pool.join()
303 
304     if visualize:
305         cv2.imshow('synthesis window', result_window)
306         cv2.waitKey(0)
307         cv2.destroyAllWindows()
308 
309     return result_window
310 
311 
312 def validate_args(args):
313     wh, ww = args.window_height, args.window_width
314     if wh < 3 or ww < 3:
315         raise ValueError('window_size must be greater than or equal to (3,3).'
            )
316 
317     if args.kernel_size <= 1:
318         raise ValueError('kernel size must be greater than 1.')
319 
320     if args.kernel_size % 2 == 0:
321         raise ValueError('kernel size must be odd.')
322 
323     if args.kernel_size > min(wh, ww):
324         raise ValueError('kernel size must be less than or equal to the
            smaller window_size dimension.')
325 
326 def parse_args():
327     parser = argparse.ArgumentParser(description='Perform texture synthesis')
328     parser.add_argument('--sample_path', type=str, required=True, help='Path
        to the texture sample')
```

```
329     parser.add_argument('--out_path', type=str, required=False, help='Output path for synthesized texture')
330     parser.add_argument('--window_height', type=int, required=False, default=50, help='Height of the synthesis window')
331     parser.add_argument('--window_width', type=int, required=False, default=50, help='Width of the synthesis window')
332     parser.add_argument('--kernel_size', type=int, required=False, default=11, help='One dimension of the square synthesis kernel')
333     parser.add_argument('--visualize', required=False, action='store_true', help='Visualize the synthesis process')
334     args = parser.parse_args()
335     return args
336
337 def main():
338     args = parse_args()
339
340     sample = cv2.imread(args.sample_path)
341     if sample is None:
342         raise ValueError('Unable to read image from sample_path.')
343
344     validate_args(args)
345
346     start_time = time.time()
347     synthesized_texture = synthesize_texture(original_sample=sample,
348                                              window_size=(args.window_height,
                                                     args.window_width),
349                                              kernel_size=args.kernel_size,
350                                              visualize=args.visualize)
351     print('Synthesis time: {:.2f} seconds'.format(time.time() - start_time))
352
353     if args.out_path is not None:
354         cv2.imwrite(args.out_path, synthesized_texture)
355
356 if __name__ == '__main__':
357     main()
```