15. Neural Style Transfer

Our previous chapter discussed the concept of "deep dreaming" and how we can run a CNN in reverse to generate artistic, and in some cases, hallucinogenic-style images. This deep dreaming algorithm sparked brand new online communities, all generating, sharing, and even competing to create the best deep dreams.

However, the excitement over deep dreaming pales in comparison to *neural style transfer*. The neural style transfer algorithm was first introduced by Gatys et al. in their 2015 paper, *A Neural Algorithm of Artistic Style* [64].

While it's possible to apply *guided deep dreaming* [59] to "guide" your input images to visualize features of a content image, the results were still similar to normal deep dreaming — due to the abundance of dogs, cats, and other animals in the ImageNet dataset (which is what most CNNs were trained on prior to applying deep dreaming), the hype surrounding the algorithm reached an asymptote.

Instead, what users *really* wanted was a method to take:

1. An input image (i.e., a *content image*)
2. A *style image*
3. And apply the *style image* to their *content image*, thereby generating more interesting images and works of art

The work of Gatys et al. made this method possible. Since then, neural style transfer has gone under many iterations and refinements, been built into smartphone apps, and even resulted in style transfer works being sold as works of art.

In this chapter we'll be focusing on the original Gatys et al. implementation — this background will give you a strong foundation to explore other neural style transfer algorithms (and potentially even create your own).

15.1 The Neural Style Transfer Algorithm

The neural style transfer algorithm consists of taking the *style* of one image and then applying it to the *content* of another.

Figure 15.1: **Left:** Our input style image is an autumn mountain range, painted by Bob Ross, the creator of *The Joy of Painting*. **Middle:** An architectural drawing of Frank Lloyd Wright's *Fallingwater* (credit: Larry Hunter [65]). **Right:** The output after after applying the style of the middle image to the input content image.

An example of this process can be seen in Figure 15.1. On the *left* we have our content image — a serene autumn mountain range, painted by critically acclaimed painter and TV host Bob Ross. In the *middle* is our style image, a rendered blueprint poster of Frank Lloyd Wright's famous *Fallingwater* house.

And on the *right* is the output of applying the style of the blueprint image to the content of Bob Ross' mountain range (hence the term *neural style transfer*). Notice how we have *retained* the content of the mountains, river, and trees, but have applied the *style* of the blueprint — it's as if Bob Ross put down his paintbrush and instead roughly sketched the mountain scene instead.

The question is, *how do we define a neural network to perform style transfer?* Interestingly, we don't need to "define" an architecture. We don't need special layers in the network. We don't even need to update the weights of the network!

Instead, consider the core component of neural networks and deep learning: *the loss function.* We define a loss function that will enable us to achieve our end goal and then we optimize over the loss function. Therefore, the question isn't *"what neural network do we use?"* but rather *"what loss function do we use?"*

The answer is a three-component loss function, including:

- Content loss
- Style loss
- Total-variation loss

Each component will be computed individually then combined in a single loss function. By minimizing the "meta" loss function we will in turn be jointly minimizing the content, style, and total-variation losses as well.

15.1.1 Content Loss

As we discussed in the *Starter Bundle*, deep learning algorithms are hierarchical learners. Layers earlier in the network capture local information such as edge-like regions and color blobs. Mid-layers build on the earlier layers to recognize corners. Finally, the highest-level layers in the network are able to capture abstract features, including object parts.

Since the higher-level layers of the network capture the most abstract qualities of the image, a good starting point for our content loss would be to examine the *activations* of these higher-level layers.

According to Gatys et al., to build our loss function we need to:

1. Utilize a pre-trained network, typically on ImageNet or similar dataset
2. Select a higher-level layer of the network to serve as our content loss. This is a hyperparameter we can tune to achieve different output transfers, but for most situations you'll want to use a

higher-level layer of the network.
3. Compute the activation of this particular layer for *both* the content image and style image
4. Take the L2-norm between these activations, respectively

Since the higher-level layers of the network capture more abstract qualities of the input image this loss function ensures that our output generated image will at least look somewhat similar to the content image.

15.1.2 Style Loss

Our content loss used only a *single* layer of a CNN but our style loss will use *multiple* layers. The reason for this is we wish to construct a multi-scale representation of style and texture.

Obtaining this multi-scale representation enables us to capture the style at lower-level layers (edge-like regions) mid-level layers (where corners and other structures start to form), and higher-level layers (which include abstract concepts). This process enables us to capture the texture/style of our style image but *not* the global arrangement of objects in the content image.

Trying to build a loss function that captures the style and texture of multiple layers in a CNN may sound like a complex task, but it's actually fairly straightforward with a bit of statistics. In particular, we'll be computing the *correlations* between the activations of layers in our CNN — we can capture the correlations by computing the *Gram matrix* between the layers activations.

A Gram matrix is the inner product of a set of features maps (if you're familiar with Support Vector Machines you have likely already encountered the Gram matrix or even utilized it without knowing.) During training, our goal will be to minimize the loss between the style of our *output image* and the style of our *style image*, thereby forcing the style of the output image to *correlate* with the style of the style image.

15.1.3 Total-variation Loss

The final loss component is our *total-variation loss*. Unlike the previous two loss function which considered:

1. The output image
2. And either the content or style image

The total-variation loss operates *solely* on the output image (it does not examine either the content image or style image). Total-variation loss was not included in the original Gatys et al. paper but it has been found to generate better, more aesthetically pleasing style transfers by encouraging spatial smoothness across the output image.

15.1.4 Combining the Loss Functions

Our final loss function is the *weighted combination* of all three content loss, style loss, and total-variation loss, respectively. Written in pseudocode (and inspired by Chollet's style transfer example [63]) our loss function would look like:

```
loss = (alpha * D(style(original_image) - style(gen_image))) +
        (beta * D(content(original_image) - content(gen_image))) +
        (gamma * tv(gen_image))
```

The first component is our style loss. We compute the distance `D` between the style feature representations for both the input reference image and output generated image and weight it by some value `alpha`. This component of the loss function ensures that the style of the reference image is mapped to the output image.

The second component handles our content loss. Here we compute the distance between the content feature representations of the original input image and the generated image. We weight this

distance by `beta`. This component ensures that our output image still has similar content to the original input image.

The final component is our total-variation loss which operates over the output image, ensuring spatial continuity. The total-variation loss is weighted by `gamma`.

The final `loss` is the sum of these three weighted loss components. The α, β, and γ values are all hyperparameters we can play with and adjust to generate different style transfers.

15.2 Implementing Neural Style Transfer

Our neural style transfer implementation will consist of two components:

1. A `NeuralStyle` class that encapsulates all logic required to apply neural style transfer to a content image and style image.
2. A Python driver script used to store any parameters, including input image paths, content and style layers, weight values, etc. This script will be responsible for kicking off the neural style transfer process.

15.2.1 Implementing the Style Transfer Driver Script

For the sake of simplicity, we are going to implement the driver script *before* the `NeuralStyle` class. Implementing the driver script script before the `NeuralStyle` class will enable you to see how we supply input image paths, weight values, and layer sets to facilitate the style transfer.

Open up a new file, name it `style_transfer.py`, and insert the following code:

```
# import the necessary packages
from pyimagesearch.nn.conv import NeuralStyle
from keras.applications import VGG19
```

Our first code block handles our imports — we only need two of them here:

1. Our `NeuralStyle` implementation
2. The network we are using to apply neural style transfer. As per the work of Gatys et al., we know that VGG19 tends to produce better style transfers than VGG16.

From there we define our `SETTINGS` dictionary:

```
# initialize the settings dictionary
SETTINGS = {
	# initialize the path to the input (i.e., content) image,
	# style image, and path to the output directory
	"input_path": "inputs/jp.jpg",
	"style_path": "inputs/starry_night.jpg",
	"output_path": "output",

	# define the CNN to be used for style transfer, along with
	# the set of content layer and style layers, respectively
	"net": VGG19,
	"content_layer": "block4_conv2",
	"style_layers": ["block1_conv1", "block2_conv1",
		"block3_conv1", "block4_conv1", "block5_conv1"],

	# store the content, style, and total variation weights,
	# respectively
	"content_weight": 1.0,
	"style_weight": 100.0,
```

```
24         "tv_weight": 10.0,
25
26         # number of iterations
27         "iterations": 50,
28     }
```

This dictionary encapsulates all settings required to run our `NeuralStyle` class. **Lines 9-12** start off by initializing the path to our input image (i.e., the content image), the path to our style image, and then finally the path to our output directory where we'll store the results of applying our neural style transfer algorithm.

Lines 15-18 initialize the CNN we are using for the style transfer (VGG19) along with the `content_layer` and list of `style_layers`.

We then have our content weight, style weight, and total-variation weight on **Lines 22-24**, respectively. **Line 27** controls the number of iterations to apply.

The final code block instantiates our `NeuralStyle` object and starts the transfer process:

```
30 # perform neural style transfer
31 ns = NeuralStyle(SETTINGS)
32 ns.transfer()
```

In the next section we'll execute the `style_transfer.py` script and examine the results of applying neural style transfer.

15.2.2 Implementing the Style Transfer Class

Our implementation of neural style transfer is loosely based on the official implementation François Chollet in the Keras repository [66] and more strongly based on Kevin Zakka's version [67] which wraps the entire algorithm in a single Python class. My main contribution is to provide detailed commentary on the code.

To implement neural style transfer, create a new file named `neuralstyle.py` in the `conv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
...
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- deepergooglenet.py
...
|    |    |    |--- neuralstyle.py
...
|    |    |    |--- shallownet.py
...
```

From there, open up the file and insert the following code:

```
1 # import the necessary packages
2 from keras.applications.vgg16 import preprocess_input
```

```
3  from keras.preprocessing.image import img_to_array
4  from keras.preprocessing.image import load_img
5  from keras import backend as K
6  from scipy.optimize import fmin_l_bfgs_b
7  import numpy as np
8  import cv2
9  import os
```

Our set of imports look fairly standard here, but you'll notice one import we haven't encountered yet: `fmin_l_bfgs_b`. he `fmin_l_bfgs_b` function is an implementation of the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm used to iteratively minimize unconstrained nonlinear optimization problems.

In our case we'll be using L-BFGS to minimize our neural style transfer loss function. L-BFGS is a common optimization function but is outside the scope of this book. If you're interested in learning more about L-BFGS, refer to Liu and Nocedal [68] along with this excellent introduction article from Aria Haghighi [69].

Let's move on to the actual `NeuralStyle` class:

```
11 class NeuralStyle:
12     def __init__(self, settings):
13         # store the settings dictionary
14         self.S = settings
15
16         # grab the dimensions of the input image
17         (w, h) = load_img(self.S["input_path"]).size
18         self.dims = (h, w)
19
20         # load content image and style images, forcing the dimensions
21         # of our input image
22         self.content = self.preprocess(settings["input_path"])
23         self.style = self.preprocess(settings["style_path"])
24         self.content = K.variable(self.content)
25         self.style = K.variable(self.style)
```

On **Line 14** we store our `settings` dictionary which is assumed to be the same `SETTINGS` from our driver script.

Lines 17 and 18 load our input image (i.e., the content image) from disk and grab the spatial dimensions — we'll need these dimensions when preprocessing an image, de-processing an image, and computing our loss.

From there we load our `content` and `style` images from disk and preprocess them (**Lines 22 and 23**), forcing the spatial dimensions of the `content` image on the `style` image to ensure both images are the same size. **Lines 24 and 25** instantiate Keras variables from the input images.

Next, let's prepare our `input` and `output` tensors:

```
27         # allocate memory of our output image, then combine the
28         # content, style, and output into a single tensor so they can
29         # be fed through the network
30         self.output = K.placeholder((1, self.dims[0],
31             self.dims[1], 3))
32         self.input = K.concatenate([self.content, self.style,
33             self.output], axis=0)
```

Lines 30 and 31 initialize our `output` image (i.e., where the final style transfer image will be stored). We assume channels-last ordering here so if you are using channels-first ordering you'll need to modify the code.

The `input` tensor is then defined on **Lines 32 and 33** consisting of all three `content`, `style`, and `output` images concatenated together. Our goal will be to minimize our style loss, content loss, and total-variation loss based on this `input` tensor.

From there we can load our model from disk:

```
35            # load our model from disk
36            print("[INFO] loading network...")
37            self.model = self.S["net"](weights="imagenet",
38                  include_top=False, input_tensor=self.input)
39
40            # build a dictionary that maps the *name* of each layer
41            # inside the network to the actual layer *output*
42            layerMap = {l.name: l.output for l in self.model.layers}
43
44            # extract features from the content layer, then extract the
45            # activations from the style image (index 0) and the output
46            # image (index 2) -- these will serve as our style features
47            # and output features from the *content* layer
48            contentFeatures = layerMap[self.S["content_layer"]]
49            styleFeatures = contentFeatures[0, :, :, :]
50            outputFeatures = contentFeatures[2, :, :, :]
51
52            # compute the feature reconstruction loss, weighting it
53            # appropriately
54            contentLoss = self.featureReconLoss(styleFeatures,
55                  outputFeatures)
56            contentLoss *= self.S["content_weight"]
```

Line 42 builds a dictionary which maps the name of a layer in our CNN to the corresponding output of the layer — we'll need this mapping to compute our loss and build the style transfer. **Lines 48-50** extracts the "features" (i.e., activations) from the *single* content layer.

Given our `styleFeatures` and `outputFeatures` for the content layer we can compute the *feature reconstruction loss*, or more simply, the *style loss* (**Lines 54-56**). We'll be implementing `featureReconloss` later in this section, but for the time being treat it as a blackbox function that computes style loss. Also take note of **Line 56** and how we weight the `contentLoss` accordingly.

Let's move on to computing the style loss:

```
58            # initialize our style loss along with the value used to
59            # weight each style layer (in proportion to the total number
60            # of style layers
61            styleLoss = K.variable(0.0)
62            weight = 1.0 / len(self.S["style_layers"])
63
64            # loop over the style layers
65            for layer in self.S["style_layers"]:
66                  # grab the current style layer and use it to extract the
67                  # style features and output features from the *style
68                  # layer*
69                  styleOutput = layerMap[layer]
```

```
70                  styleFeatures = styleOutput[1, :, :, :]
71                  outputFeatures = styleOutput[2, :, :, :]
72
73                  # compute the style reconstruction loss as we go
74                  T = self.styleReconLoss(styleFeatures, outputFeatures)
75                  styleLoss += (weight * T)
76
77              # finish computing the style loss, compute the total
78              # variational loss, and then compute the total loss that
79              # combines all three
80              styleLoss *= self.S["style_weight"]
81              tvLoss = self.S["tv_weight"] * self.tvLoss(self.output)
82              totalLoss = contentLoss + styleLoss + tvLoss
```

Line 61 initializes our `styleLoss` while **Line 62** initializes the `weight` used to weight our `styleLoss`, proportional to the total number of style layers we are using for the transfer.

Line 65 starts looping over each of the style layers. For each `layer`, we first use the `layerMap` to grab the `styleOutput`. From the `styleOutput` we can extract the `styleFeatures` (index 1) and the `outputFeatures` (index 2).

Based on both the `styleFeatures` and `outputFeatures` we compute the *style reconstruction loss*, or more simply, *the style loss*. We will be implementing the `styleReconLoss` function later in this section, but again, treat it as a blackbox function until we get there.

Line 81 computes our total-variation loss (which operates solely on the `output`) while **Line 82** computes the final `totalLoss`, combining `contentLoss`, `styleLoss`, and `tvLoss` together.

The last step in our initialize is to compute our gradients and initialize the function we'll be apply L-BFGS to:

```
84              # compute the gradients out of the output image with respect
85              # to loss
86              grads = K.gradients(totalLoss, self.output)
87              outputs = [totalLoss]
88              outputs += grads
89
90              # the implementation of L-BFGS we will be using requires that
91              # our loss and gradients be *two separate functions* so here
92              # we create a Keras function that can compute both the loss
93              # and gradients together and then return each separately
94              # using two different class methods
95              self.lossAndGrads = K.function([self.output], outputs)
```

Lines 86-88 compute the gradients of our `output` image with respect to our `totalLoss`.

Line 95 then initializes the `lossAndGrads` function which accepts our `output` tensor and `outputs` gradients as arguments. Due to how the implementation of L-BFGS works, we need two separate functions for (1) the loss and (2) the gradients. This Keras function will return both and enable us to easily exact the loss and/or gradients deepening on which we are interested in.

Our next code block creates two functions used to `preprocess` and `deprocess` images, respectively:

```
97      def preprocess(self, p):
98              # load the input image (while resizing it to the desired
99              # dimensions) and preprocess it
```

```
100             image = load_img(p, target_size=self.dims)
101             image = img_to_array(image)
102             image = np.expand_dims(image, axis=0)
103             image = preprocess_input(image)
104
105             # return the preprocessed image
106             return image
107
108         def deprocess(self, image):
109             # reshape the image, then reverse the zero-centering by
110             # *adding* back in the mean values across the ImageNet
111             # training set
112             image = image.reshape((self.dims[0], self.dims[1], 3))
113             image[:, :, 0] += 103.939
114             image[:, :, 1] += 116.779
115             image[:, :, 2] += 123.680
116
117             # clip any values falling outside the range [0, 255] and
118             # convert the image to an unsigned 8-bit integer
119             image = np.clip(image, 0, 255).astype("uint8")
120
121             # return the deprocessed image
122             return image
```

The `preprocess` function (**Lines 97-106**) loads an input image from disk and preprocesses it so it can be passed through a Keras network pre-trained on the ImageNet dataset. The `deprocess` function on **Lines 108-122** takes the output of our neural stylet transfer algorithm and *de-processes it* so we can save the image to disk.

We're de-processing the image based on the reverse of the pre-processing used for VGG16 and VGG19. If you're *not* using either VGG16 or VGG19 you'll need to modify the `deprocess` function to perform the inverse of the pre-processing required by your network.

The next function `gramMat`, is responsible for computing the Gram matrix:

```
124         def gramMat(self, X):
125             # the gram matrix is the dot product between the input
126             # vectors and their respective transpose
127             features = K.permute_dimensions(X, (2, 0, 1))
128             features = K.batch_flatten(features)
129             features = K.dot(features, K.transpose(features))
130
131             # return the gram matrix
132             return features
```

The Gram matrix is the dot product between the input vectors and their transpose. We'll need the Gram matrix when computing the style loss.

From here we can define `featureReconLoss` which is responsible for computing the *content loss:*

```
134         def featureReconLoss(self, styleFeatures, outputFeatures):
135             # the feature reconstruction loss is the squared error
```

```
136             # between the style features and output output features
137             return K.sum(K.square(outputFeatures - styleFeatures))
```

The content-loss is the L2 norm (sum of squared differences) between the features of our input image and the features of the target, output image. By minimizing the L2 norm of these activations we can force our output image to have similar structural content (but not necessarily similar style) as our input image.

Next, we need to create the `styleReconLoss` function:

```
139     def styleReconLoss(self, styleFeatures, outputFeatures):
140             # compute the style reconstruction loss where A is the gram
141             # matrix for the style image and G is the gram matrix for the
142             # generated image
143             A = self.gramMat(styleFeatures)
144             G = self.gramMat(outputFeatures)
145 
146             # compute the scaling factor of the style loss, then finish
147             # computing the style reconstruction loss
148             scale = 1.0 / float((2 * 3 * self.dims[0] * self.dims[1]) ** 2)
149             loss = scale * K.sum(K.square(G - A))
150 
151             # return the style reconstruction loss
152             return loss
```

To compute the style loss we need to first compute the Gram matrix for both the `styleFeatures` of the input style image and the `outputFeatures` of our output image (**Lines 143 and 144**).

Line 148 computes the scaling factor of the style loss based on the input dimensions of the image. If you're curious how the scaling factors are derived, please refer to the *"Methods"* section of Gatys et al. [64] **Line 149** computes the L2 norm (sum of squared differences) between the Gram matrices and scales appropriately.

We need one final loss function — our total-variation loss:

```
154     def tvLoss(self, X):
155             # the total variational loss encourages spatial smoothness in
156             # the output page -- here we avoid border pixels to avoid
157             # artifacts
158             (h, w) = self.dims
159             A = K.square(X[:, :h - 1, :w - 1, :] - X[:, 1:, :w - 1, :])
160             B = K.square(X[:, :h - 1, :w - 1, :] - X[:, :h - 1, 1:, :])
161             loss = K.sum(K.pow(A + B, 1.25))
162 
163             # return the total variational loss
164             return loss
```

This loss function was not included in the original Gatys et al. paper, but nearly all neural style transfer implementations include this additional loss function as it encourages spatial smoothness throughout the output image). We utilize array slicing here along the borders of the image to avoid border artifacts.

We can now define the `transfer` function which is responsible for jointly minimizing our content loss, style loss, and total variation loss:

```
166     def transfer(self, maxEvals=20):
167         # generate a random noise image that will serve as a
168         # placeholder array, slowly modified as we run L-BFGS to
169         # apply style transfer
170         X = np.random.uniform(0, 255,
171             (1, self.dims[0], self.dims[1], 3)) - 128
172
173         # start looping over the desired number of iterations
174         for i in range(0, self.S["iterations"]):
175             # run L-BFGS over the pixels in our generated image to
176             # minimize the neural style loss
177             print("[INFO] starting iteration {} of {}...".format(
178                 i + 1, self.S["iterations"]))
179             (X, loss, _) = fmin_l_bfgs_b(self.loss, X.flatten(),
180                 fprime=self.grads, maxfun=maxEvals)
181             print("[INFO] end of iteration {}, loss: {:.4e}".format(
182                 i + 1, loss))
183
184             # deprocess the generated image and write it to disk
185             image = self.deprocess(X.copy())
186             p = os.path.sep.join([self.S["output_path"],
187                 "iter_{}.png".format(i)])
188             cv2.imwrite(p, image)
```

Lines 170 and 1791 initializes `X`, a randomly generated placeholder NumPy array with the spatial dimensions of our input content image. The `X` variable will be modified during the training process to take on both the content and style of our respective input images.

On **Line 173** we start looping over the desired number of iterations. We then run L-BFGS over the pixels of generated image to minimize the neural style loss (**Lines 179 and 180**). The function we are minimizing is our `loss`. The gradients of `loss` are specified by `fprime`, our `grads` function. We'll be implementing both `loss` and `grads` in our next code block. Finally, we save the de-processed image at each iteration (**Lines 186-188**).

Our final code block handles defining two helper functions, `loss` and grads':

```
190     def loss(self, X):
191         # extract the loss value
192         X = X.reshape((1, self.dims[0], self.dims[1], 3))
193         lossValue = self.lossAndGrads([X])[0]
194
195         # return the loss
196         return lossValue
197
198     def grads(self, X):
199         # compute the loss and gradients
200         X = X.reshape((1, self.dims[0], self.dims[1], 3))
201         output = self.lossAndGrads([X])
202
203         # extract and return the gradient values
204         return output[1].flatten().astype("float64")
```

As we saw, our `fmin_l_bfgs_b` function requires *two* functions when applying optimization:

1. A loss function

2. A *separate* function that returns the gradients of the loss

Lines 192-196 utilize `lossAndGrads` to grab the `lossValue` and return it (index `0`). **Lines 200-204** then extract the gradients (index `1`) from `lossAndGrads` and return it.

Specifying the indexes here is important as we defined the Keras function, `lossAndGradients` in our constructor to compute both the loss and gradients — specifying the index enables us to extract each, respectively.

15.3 Neural Style Transfer Results

We are now ready to apply neural style transfer to our own input images. Open up `style_transfer.py` and update the `SETTINGS` to include:

```
"input_path": "inputs/jp.jpg",
"style_path": "inputs/mcescher.jpg",
...
"content_weight": 1.0,
"style_weight": 100.0,
"tv_weight": 10.0,
```

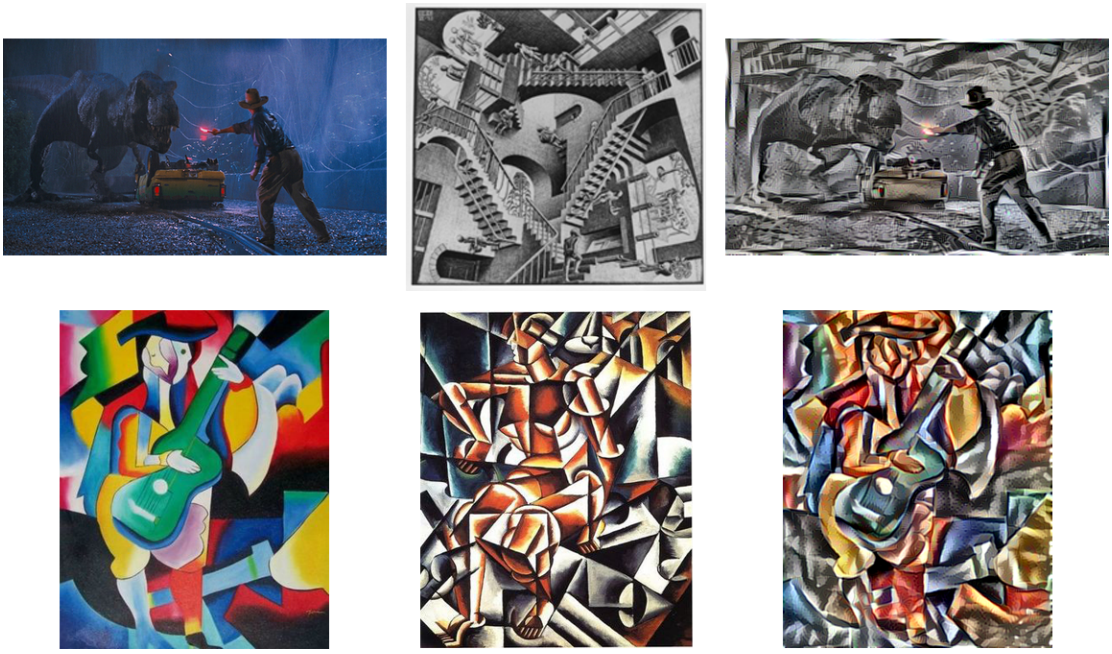

Figure 15.2: **Top:** Applying neural style transfer using an iconic scene from the movie *Jurassic Park* as the input and MC Escher as the style. (Image and still credit to *Jurassic Park* Universal Pictures.) **Bottom:** Pablo Piccaso is used as the as the content image and Popova as the style.

And then execute the `style_transfer.py` script:

```
$ python style_transfer.py
```

The results of applying our neural style transfer algorithm can be seen in Figure 15.2 (*top*). Here we have used an iconic scene from the movie *Jurassic Park* as the input content image. The

style image is MC Escher' `Relativity lattice`. The output image maintains the content of the *Jurassic Park* still but has geometrical shapes similar to MC Escher's work.

Let's try another style transfer, but before we do so, let's update the `SETTINGS` again:

```
"input_path": "inputs/pablo_picasso.jpg",
"style_path": "inputs/popova_air_man_space.jpg",
...
"content_weight": 1.0,
"style_weight": 100.0,
"tv_weight": 100.0,
```

Here we are setting the style weight to be two orders of magnitude larger than the content weight, implying that we are willing to sacrifice a bit of the content for a more artistic style transfer. We will also increase our total variation weight to help ensure spatial smoothness. The result of these updated `SETTINGS` can be seen in Figure 15.2 (*bottom*).

Some of the best style transfers can be generated by doing random walks through the parameter space. This process involves randomly generating values for each of the loss weights. When I perform neural style transfer I typically define a set of parameters to explore, start the experiment, and let the script run overnight. The next morning I go through the results I like the best and then double-down on the parameter ranges that produced the most aesthetically appealing results.

15.4 Summary

In this chapter we discussed the neural style transfer algorithm by Gatys et al. [64], implemented neural style transfer using Python and Keras, and explored varying parameters for content weight, style weight, and total-variation weight.

The neural style transfer algorithm will work best with:

1. Content images that do not require high levels of detail to be either visually appealing or recognizable.
2. Style images that contain a lot of texture. "Flat" style images will not produce aesthetically appealing results.

Finally, be sure to explore your parameter space with applying the neural style transfer algorithm — vary your content weight, style weight, and total-variation weight will give you different results. I suggest creating a Python script that iteratively runs 10-15 experiments with varying parameters so you can get a sense of what parameters (or maybe none at all) will generate aesthetically pleasing results for both your content image and style image.

To help you get started running your own experiments that *automatically* varies parameters, I have included a special script named `generate_examples.py` in the downloads associated with this book. Be sure to refer to this script as a starting point for your own experiments.