CHAPTER 12

Deep Learning for Computer Vision

Deep Learning is not just a keyword abuzz in the industry and academics, it has thrown wide open a whole new field of possibilities. Deep Learning models are being employed in all sorts of use cases and domains, some of which we saw in the previous chapters. Deep neural networks have tremendous potential to learn complex non-linear functions, patterns, and representations. Their power is driving research in multiple fields, including computer vision, audio-visual analysis, chatbots and natural language understanding, to name a few. In this chapter, we touch on some of the advanced areas in the field of computer vision, which have recently come into prominence with the advent of Deep Learning. This includes real-world applications like image categorization and classification and the very popular concept of image artistic style transfer. Computer vision is all about the art and science of making machines understand high-level useful patterns and representations from images and videos so that it would be able to make intelligent decisions similar to what a human would do upon observing its surroundings. Building on core concepts like convolutional neural networks and transfer learning, this chapter provides you with a glimpse into the forefront of Deep Learning research with several real-world case studies from computer vision.

This chapter discusses convolutional neural networks through the task of image classification using publicly available datasets like CIFAR, ImageNet, and MNIST. We will utilize our understanding of CNNs to then take on the task of style transfer and understand how neural networks can be used to understand high-level features. Through this chapter, we cover the following topics in detail:

- Brief overview of convolutional neural networks
- Image classification using CNNs from scratch
- Transfer learning: image classification using pretrained models
- Neural style transfer using CNNs

The code samples, jupyter notebooks, and sample datasets for this chapter are available in the GitHub repository for this book at `https://github.com/dipanjanS/practical-machine-learning-with-python` under the directory/folder for Chapter 12.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are similar to the general neural networks we have discussed over the course of this book. The additional explicit assumption of input being an image (tensor) is what makes CNNs optimized and different than the usual neural networks. This explicit assumption is what allows us to design deep CNNs while keeping the number of trainable parameters in check (in comparison to general neural networks).

D. Sarkar et al., *Practical Machine Learning with Python*, https://doi.org/10.1007/978-1-4842-3207-1_12

We touched upon the concepts of CNNs in Chapter 1 (in the section "Deep Learning") and Chapter 4 (in the section :Feature Engineering on Image Data). However, as a quick refresher, the following are the key concepts worth reiterating:

- **Convolutional Layer**: This is the key differentiating component of a CNN as compared to other neural networks. Convolutional layer or conv layer is a set of learnable filters. These filters help capture spatial features. These are usually small (along the width and height) but cover the full depth (color range) of the image. During the forward pass, we slide the filter across the width and the height of the image while computing the dot product between the filter attributes and the input at any position. The output is a two-dimensional activation map from each filter, which are then stacked to get the final output.
- **Pooling Layer**: These are basically down-sampling layers used to reduce spatial size and number of parameters. These layers also help in controlling overfitting. Pooling layers are inserted in between conv layers. Pooling layers can perform down sampling using functions such as max, average, L2-norm, and so on.
- **Fully Connected Layer**: Also known as FC layer. These are similar to fully connected layers in general neural networks. These have full connections to all neurons in the previous layer. This layer helps perform the tasks of classification.
- **Parameter Sharing**: The unique thing about CNNs apart from the conv layer is parameter sharing. Conv layers use same set of weights across the filters thus reducing the overall number of parameters required.

A typical CNN architecture with all the components is depicted in Figure 12-1, which is a LeNet CNN model (Source: `deeplearning.net`).

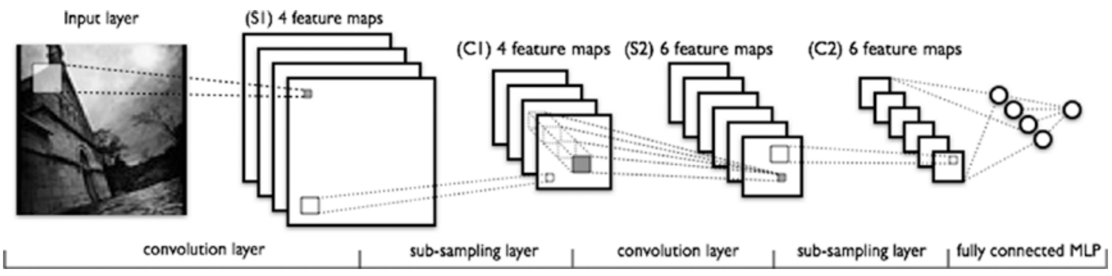


Figure 12-1. *LeNet CNN model (source: deeplearning.net)*

CNNs have been studied in-depth and are being constantly improved and experimented with. For an in-depth understanding of CNNs, refer to courses such as one from Stanford available at `http://cs231n.github.io/convolutional-networks`.

Image Classification with CNNs

Convolutional Neural Networks are prime examples of the potential and power of neural networks to learn detailed feature representations and patterns from images and perform complex tasks, ranging from object recognition to image classification and many more. CNNs have gone through tremendous research and advancements have led to more complex and power architectures, like VGG-16, VGG-19, Inception V3, and many more interesting models.

We begin with a getting some hands-on experience with CNNs by working on an image classification problem. We shared an example of CNN based classification in Chapter 4 through the notebook, `Bonus - Classifying handwritten digits using Deep CNNs.ipynb`, which talks about classifying and predicting human handwritten digits by leveraging CNN based Deep Learning. In case you haven't gone through it, do not worry as we will go through a detailed example here. For our Deep Learning needs, we will be utilizing the `keras` framework with the `tensorflow` backend, similar to what we used in the previous chapters.

Problem Statement

Given a set of images containing real-world objects, it is fairly easy for humans to recognize them. Our task here is to build a multiclass (10 classes or categories) image classifier that can identify the correct class label of a given image. For this task, we will be utilizing the `CIFAR10` dataset.

Dataset

The `CIFAR10` dataset is a collection of tiny labeled images spanning across 10 different classes. The dataset was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton and is available at `https://www.cs.toronto.edu/~kriz/cifar.html` as well as through the `datasets` module in `keras`.

This dataset contains tiny images of size 32 x 32 with 50,000 training and 10,000 test samples. Each image can fall into one and only one of the following classes.

- Automobile
- Airplane
- Bird
- Cat
- Deer
- Dog
- Frog
- Horse
- Ship
- Truck

Each class is mutually exclusive. There is another larger version of the dataset called the `CIFAR100`. For the purpose of this section, we will consider the `CIFAR10` dataset.

We would be accessing the `CIFAR10` dataset through the `keras.datasets` module. Download the required files if they are not already present.

CNN Based Deep Learning Classifier from Scratch

Similar to any Machine Learning algorithm, neural networks also require the input data to be certain shape, size, and type. So, before we reach the modeling step, the first thing is to preprocess the data itself. The following snippet gets the dataset and then performs one hot encoding of the labels. Remember there are 10 classes to work with and hence we are dealing with a multi-class classification problem.

```
In [1]: import keras
   ...: from keras.datasets import cifar10
   ...:
   ...: num_classes = 10
   ...:
   ...: (x_train, y_train), (x_test, y_test) = cifar10.load_data()
   ...:
   ...: # convert class vectors to binary class matrices
   ...: y_train = keras.utils.to_categorical(y_train, num_classes)
   ...: y_test = keras.utils.to_categorical(y_test, num_classes)
```

The dataset, if not already present locally, would be downloaded automatically. The following are the shapes of the objects obtained.

```
In [2]: print('x_train shape:', x_train.shape)
   ...: print(x_train.shape[0], 'train samples')
   ...: print(x_test.shape[0], 'test samples')
   ...:
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
```

Now that we have training and test datasets. The next step is to build the CNN model. Since we have two dimensional images (the third dimension is the channel information), we will be using `Conv2D` layers. As discussed in the previous section, CNNs uses a combination of convolutional layers and pooling layers followed by a fully connected end to identify/classify the data. The model architecture is built as follows.

```
In [3]: model = Sequential()
   ...: model.add(Conv2D(32, kernel_size=(3, 3),
   ...:                         activation='relu',
   ...:                         input_shape=input_shape))
   ...: model.add(Conv2D(64, (3, 3), activation='relu'))
   ...: model.add(MaxPooling2D(pool_size=(2, 2)))
   ...: model.add(Dropout(0.25))
   ...: model.add(Flatten())
   ...: model.add(Dense(128, activation='relu'))
   ...: model.add(Dropout(0.5))
   ...: model.add(Dense(num_classes, activation='softmax'))
```

It starts off with a convolutional layer with a total of 32 3 x 3 filters and activation function as the rectified linear unit (`relu`). The input shape resembles each image size, i.e. 32 x 32 x 3 (color image has three channels—RGB). This is followed by another convolutional layer and a *max-pooling* layer. Finally, we have the fully connected *dense layer*. Since we have 10 classes to choose from, the final output layer has a `softmax` activation.

The next step involves compiling. We use `categorical_crossentropy` as our loss function since we are dealing with multiple classes. Besides this, we use the Adadelta optimizer and then train the classifier on the training data. The following snippet showcases the same.

```
In [4]: model.compile(loss=keras.losses.categorical_crossentropy,
   ...:                 optimizer=keras.optimizers.Adadelta(),
   ...:                 metrics=['accuracy'])
   ...:
   ...: model.fit(x_train, y_train,
   ...:                 batch_size=batch_size,
   ...:                 epochs=epochs,
   ...:                 verbose=1)
Epoch 1/10
50000/50000 [==============================] - 256s - loss: 7.3118 - acc: 0.1798
Epoch 2/10
50000/50000 [==============================] - 250s - loss: 1.7923 - acc: 0.3564
Epoch 3/10
50000/50000 [==============================] - 252s - loss: 1.5781 - acc: 0.4383
 ...
Epoch 9/10
50000/50000 [==============================] - 251s - loss: 1.1019 - acc: 0.6163
Epoch 10/10
50000/50000 [==============================] - 254s - loss: 1.0584 - acc: 0.6284
```

From the preceding output, it is clear that we trained the model for 10 epochs. This takes anywhere between 200-400 secs on a CPU, the performance improves manifold when done using a GPU. We can see the accuracy is around 63% based on the last epoch. We will now evaluate the testing performance, this is checked using the `evaluate` function of the model object. The results are as follows.

```
Test loss: 1.10143025074
Test accuracy: 0.6354
```

Thus, we can see that our very simple CNN based Deep Learning model achieved an accuracy of 63.5%, given the fact that we have built a very simple model and that we haven't done much preprocessing or model tuning. You are encouraged to try different CNN architectures and experiment with hyperparameter tuning to see how the results can be improved.

The initial few conv layers of the model kind of work toward feature extraction while the last couple of layers (fully connected) help in classifying the data. Thus, it would be interesting to see how the image data is manipulated by the `conv-net` we just created. Luckily, `keras` provides hooks to extract information at intermediate steps in the model. They depict how various regions of the image activate the conv layers and how the corresponding feature representations and patterns are extracted.

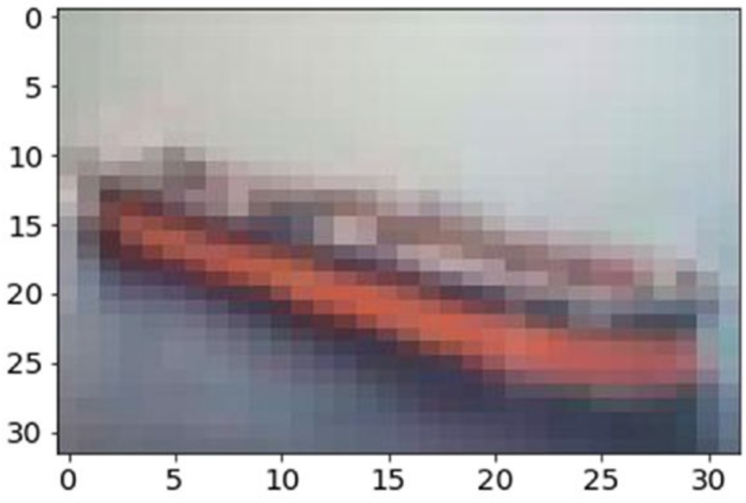


Figure 12-2. *Sample image from the CIFAR10 dataset*

A sample flow of how an image is viewed by the CNN is explained in the notebook `notebook_cnn_cifar10_classifier.ipynb`. It contains rest of the code discussed in this section. Figure 12-2 shows an image from the test dataset. It looks like a ship and the model correctly identifies the same as well as depicted in this snippet.

```
# actual image id
img_idx = 999
# actual image label
In [5]: y_test[img_idx]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.])

# predict label with our model
In [6]: test_image =np.expand_dims(x_test[img_idx], axis=0)
   ...: model.predict_classes(test_image,batch_size=1)
1/1 [==============================] - 0s
Out[16]:
array([8], dtype=int64)
```

You can extract and view the activation maps of the image based on what representations are learned and extracted by the conv layers using the `get_activations(...)` and `display_activations(...)` functions in the notebook. Figure 12-3 shows the activations of initial conv layers of the CNN model we just built.

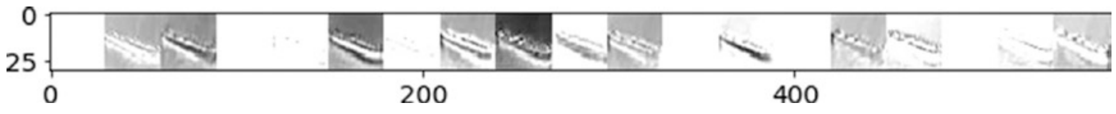


Figure 12-3. *Sample image through a CNN layer*

We also recommend you go through the section "Automated Feature Engineering with Deep Learning" in Chapter 4 to learn more about extracting feature representations from images using convolutional layers.

CNN Based Deep Learning Classifier with Pretrained Models

Building a classifier from scratch has its own set of pros and cons. Yet, a more refined approach is to leverage pre-trained models over large complex datasets. There are many famous CNN architectures like LeNet, ResNet, VGG-16, VGG-19, and so on. These models have deep and complex architectures that have been fine-tuned and trained over diverse, large datasets. Hence, these models have been proven to have amazing performance on complex object recognition tasks.

Since obtaining large labeled datasets and training highly complex and deep neural networks is a time-consuming task (*training a complex CNN like VGG-19 could take a few weeks, even using GPUs*). In practice, we utilize a concept, what is formally termed as *transfer learning*. This concept of transfer learning helps us leverage existing models for our tasks. The core idea is to leverage the *learning,* which the model learned from being trained over a large dataset and then *transfer* this learning by re-using the same model to extract feature representations from new images. There are several strategies of performing transfer learning, some of which are mentioned as follows:

- ***Pre-trained model as feature extractor:*** The pre-trained model is used to extract features for our dataset. We build a fully connected classifier on top of these features. In this case we only need to train the fully connected classifier, which does not take much time.
- ***Fine-tuning pre-trained models:*** It is possible to fine-tune an existing pre-trained model by fixing some of the layers and allowing others to learn/update weights apart from the fully connected layers. Usually it is observed that initial layers capture generic features while the deeper ones become more specific in terms of feature extraction. Thus, depending upon the requirements, we fix certain layers and fine-tune the rest.

In this section, we see an example where we will utilize a pre-trained conv-network as a feature extractor and build fully connected layer based classifier on top of it and train the model. We will not train the feature extraction layers and hence leverage principles of transfer learning by using the pre-trained conv layers for feature extraction.

The VGG-19 model from the Visual Geometry Group of the Oxford University is one state-of-the-art convolutional neural network. This has been shown to perform extremely well on various benchmarks and competitions. VGG19 is a 19-layer conv-net trained on `ImageNet` dataset. `ImageNet` is visual database of hand-annotated images amounting to 10 million spanning across 9,000+ categories. This model has been widely studied and used in tasks such as transfer learning.

■ **Note** More details on this and other research by the VGG group is available at `http://www.robots.ox.ac.uk/~vgg/research/very_deep/`.

This pretrained model is available through the `keras.applications` module. As mentioned, we will utilize VGG-19 to act as feature extractor to help us build a classifier on `CIFAR10` dataset.

Since we would be using VGG-19 for feature extraction, we do not need the top (or fully connected) layers of this model. `keras` makes this as simple as setting a single flag value to `False`. The following snippet loads the `VGG-19` model architecture consisting of the conv layers and leaves out the fully connected layers.

```
In [1]: from keras import applications
   ...:
   ...: vgg_model = applications.VGG19(include_top=False, weights='imagenet')
```

Now that the pre-trained model is available, we will utilize it to extract features from our training dataset. Remember VGG-19 is trained upon `ImageNet` while we would be using `CIFAR10` to build a classifier. Since ImageNet contains over 10 million images spanning across 9,000+ categories, it is safe to assume that `CIFAR10`'s categories would a subset here. Before moving on to feature extraction using the VGG-19 model, it would be a good idea to check out the model's architecture.

```
In [1]: vgg_model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None, None, 3)     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
```

```
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0
_________________________________________________________________
```

From the preceding output, you can see that the architecture is huge with a lot of layers. Figure 12-4 depicts the same in an easier-to-understand visual depicting all the layers. Remember that we do not use the fully connected layers depicted in the extreme right of Figure 12-4. We recommend checking out the paper *Very Deep Convolutional Networks for Large-Scale Image Recognition* by Karen Simonyan and Andrew Zisserman of the Visual Geometry Group, Department of Engineering Science, University of Oxford. The paper is available at https://arxiv.org/abs/1409.1556 and talks in detail about the architecture of these models.

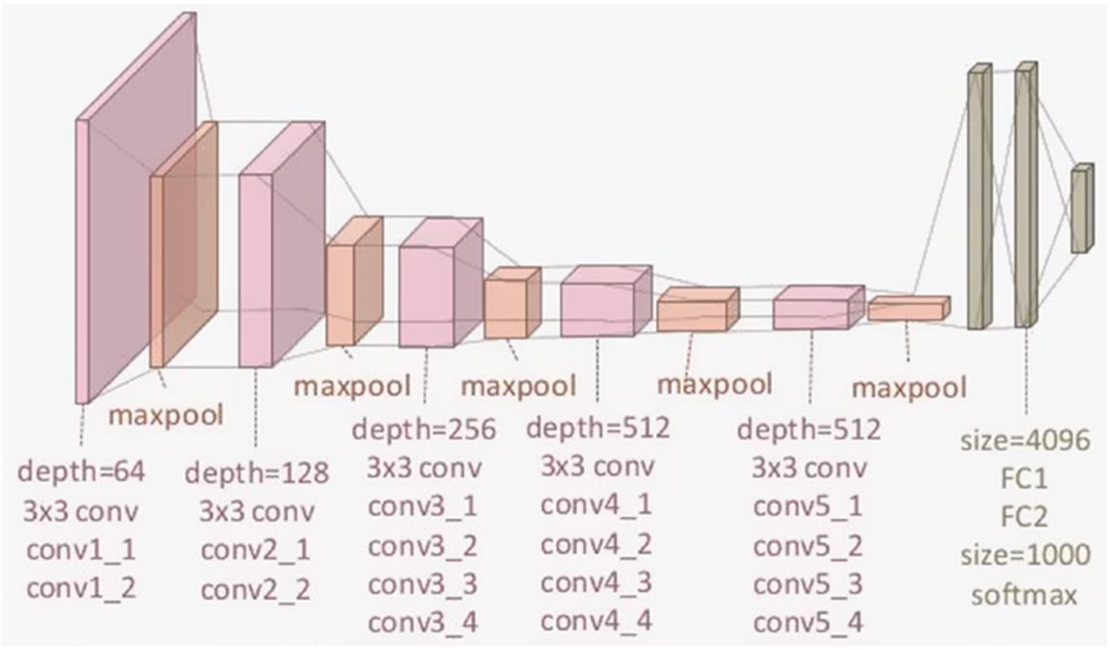


Figure 12-4. *Visual depiction of the VGG-19 architecture*

Loading the `CIFAR10` training and test datasets is the same as discussed in the previous section. We perform similar one hot encoding of the labels as well. Since the VGG19 model has been loaded without the final fully connected layers, the `predict(...)` function of the model helps us get the extracted features on our dataset. The following snippet extracts the features for both training and test datasets.

```
In [2]: bottleneck_features_train = vgg_model.predict(x_train, verbose=1)
   ...: bottleneck_features_test = vgg_model.predict(x_test, verbose=1)
```

These features are widely known as *bottleneck* features due to the fact that there is an overall decrease in the volume of the input data points. It would be worth exploring the model summary to understand how the VGG model transforms the data. The output of this stage (the bottleneck features) is used as input to the classifier we are going to build next. The following snippet builds a simple fully connected two-layer classifier.

```
In [3]: clf_model = Sequential()
   ...: clf_model.add(Flatten(input_shape=bottleneck_features_train.shape[1:]))
   ...: clf_model.add(Dense(512, activation='relu'))
   ...: clf_model.add(Dropout(0.5))
   ...: clf_model.add(Dense(256, activation='relu'))
   ...: clf_model.add(Dropout(0.5))
   ...: clf_model.add(Dense(num_classes, activation='softmax'))
   ...: clf_model.compile(loss=keras.losses.categorical_crossentropy,
                          optimizer=keras.optimizers.Adadelta(),
                          metrics=['accuracy'])
```

The model's input layer matches the dimensions of the *bottleneck* features (for obvious reasons). As with the CNN model we built from scratch, this model also has a dense output layer with `softmax` activation function. Training this model as opposed to a complete VGG19 is fairly simple and fast, as depicted in the following snippet.

```
In [4]: clf_model.fit(bottleneck_features_train, y_train, batch_size=batch_size,
   ...:               epochs=epochs, verbose=1)
Epoch 1/50
50000/50000 [==============================] - 8s - loss: 7.2495 - acc: 0.2799
Epoch 2/50
50000/50000 [==============================] - 7s - loss: 2.2513 - acc: 0.2768
Epoch 3/50
50000/50000 [==============================] - 7s - loss: 1.9096 - acc: 0.3521
 ...
Epoch 48/50
50000/50000 [==============================] - 8s - loss: 0.9368 - acc: 0.6814
Epoch 49/50
50000/50000 [==============================] - 8s - loss: 0.9223 - acc: 0.6832
Epoch 50/50
50000/50000 [==============================] - 8s - loss: 0.9197 - acc: 0.6830
```

We can add hooks to stop the training early based of early stop criteria, etc. But for now, we keep things simple. Complete code for this section is available in the notebook `notebook_pretrained_cnn_cifar10_classifier.ipynb`. Overall, we achieve an accuracy of 68% on the training dataset and around 64% on the test dataset.

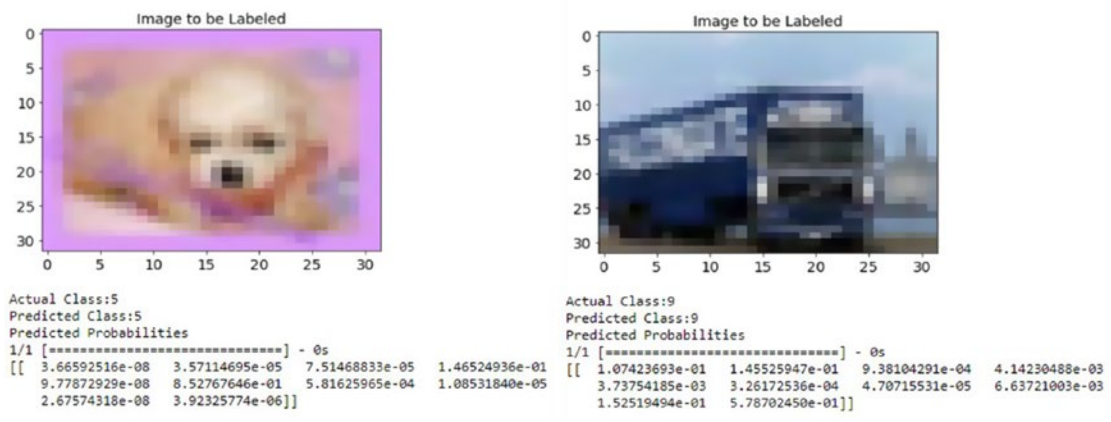
Now you'll see the performance of this classifier built on top a pre-trained model on the test dataset. The following snippet showcases a utility function that takes the index number of an image in the test dataset as input and compares the actual labels and the predicted labels.

```
def predict_label(img_idx,show_proba=True):
    plt.imshow(x_test[img_idx],aspect='auto')
    plt.title("Image to be Labeled")
    plt.show()
    print("Actual Class:{}".format(np.nonzero(y_test[img_idx])[0][0]))

    test_image =np.expand_dims(x_test[img_idx], axis=0)
    bf = vgg_model.predict(test_image,verbose=0)
    pred_label = clf_model.predict_classes(bf,batch_size=1,verbose=0)

    print("Predicted Class:{}".format(pred_label[0]))
    if show_proba:
        print("Predicted Probabilities")
        print(clf_model.predict_proba(bf))
```

The following is the output of the `predict_label(...)` function when tested against a couple of images from the test dataset. As depicted in Figure 12-5, we correctly predicted the images belong to class label 5 (dog) and 9 (truck)!



```
Actual Class:5
Predicted Class:5
Predicted Probabilities
1/1 [==============================] - 0s
[[  3.66592516e-08   3.57114695e-05   7.51468833e-05   1.46524936e-01
    9.77872929e-08   8.52767646e-01   5.81625965e-04   1.08531840e-05
    2.67574318e-08   3.92325774e-06]]
```

```
Actual Class:9
Predicted Class:9
Predicted Probabilities
1/1 [==============================] - 0s
[[  1.07423693e-01   1.45525947e-01   9.38104291e-04   4.14230488e-03
    3.73754185e-03   3.26172536e-04   4.70715531e-05   6.63721003e-03
    1.52519494e-01   5.78702450e-01]]
```

Figure 12-5. *Predicted labels from pre-trained CNN based classifier*

This section demonstrated the power and advantages of *transfer learning*. Instead of spending time reinventing the wheel, with a few lines of code, we were able to leverage state of the art neural network for our classification task.

The concept of transfer learning is what forms the basis of neural *style transfer*, which we will discuss in the next section.

Artistic Style Transfer with CNNs

Paintings (or for that matter any form of art) require special skill which a few have mastered. Paintings present complex interplay of content and style. Photographs on the other hand are a combination of perspectives and light. When the two are combined, the results are spectacular and surprising. One such example is shown in Figure 12-6.

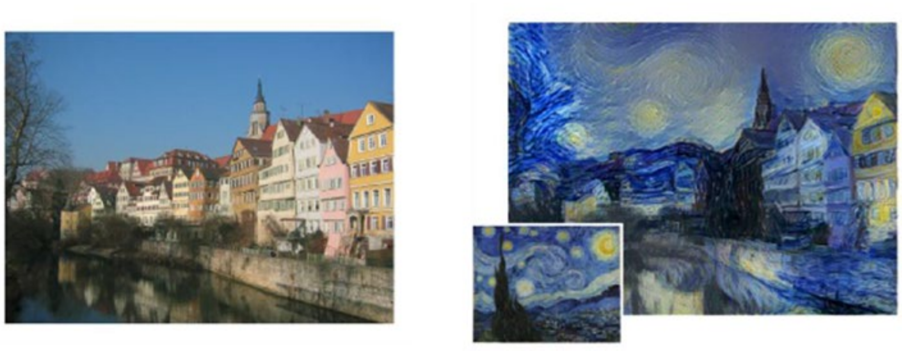


Figure 12-6. *Left Image: The original photograph depicting the Neckarfront in Tubingen, Germany. Right Image: The painting (inset: The Starry Night by Vincent van Gogh) that provided the style for the respective generated image. Source: A Neural Algorithm of Artistic Style, Gatys et al. (arXiv:1508.06576v2)*

The results in Figure 12-6 showcase how a painting's (Van Gogh's The Starry Night) style has been transferred to a photograph of the Neckarfront. At first glance, the process seems to have picked up the content from the photograph, the style, colors, and stroke patterns from the painting and generated the final outcome. The results are amazing, but what is more surprising is, how was it done?

Figure 12-6 showcases a process termed as *artistic style transfer*. The process is an outcome of research by Gatys et al. and is presented in their paper *A Neural Algorithm for Artistic Style*. In this section, we discuss the intricacies of this paper from an implementation point of view and see how we can perform this technique ourselves.

■ **Note** Prisma is an app that transforms photos into works of art using techniques of *artistic style transfer* based on convolution neural networks. More about the app is available at `https://prisma-ai.com/`.

Background

Formally, neural style transfer is the process of applying the "style" of a reference image to a specific target image such that in the process, the original "content" of the target image remains unchanged. Here, style is defined as colors, patterns, and textures present in the reference image, while content is defined as the overall structure and higher-level components of the image.

The main objective here is then, to retain the content of the original target image, while superimposing or adopting the style of the reference image on the target image. To define this concept mathematically, consider three images—the *original content* (denoted as c), the *reference style* (denoted as s), and the *generated image* (denoted as g). Thus, we need a way to measure how different are images *c* and *g* in terms of their content. A function that tends to 0 if *c* and *g* are completely different and grows otherwise. This can be concisely stated in terms of a loss function as:

$$L_{content} = distance(c, g)$$

Where *distance* is a norm function like *L2*. On the same lines, we can define another function that captures how different images *s* and *g* are in terms of their style. In other words, this can be stated as follows:

$$L_{style} = distance(s, g)$$

Thus, for the overall process of neural style transfer, we have an overall loss function, which can be defined as a combination of content and style loss functions.

$$L_{style-transfer} = argmin_g(\alpha L_{content}(c, g) + \beta L_{style}(s, g))$$

Where α and β are weights used to control the impact of content and style components on the overall loss. The loss function we will try to minimize consists of three parts namely, the *content loss*, the *style loss,* and the *total variation loss*, which we will be talking about later.

The beauty of Deep Learning is that by leveraging architectures like deep convolutional neural networks (CNNs), we can mathematically define the above-mentioned style and content functions. We will be using principles of transfer learning in building our system for neural style transfer. We introduced the concept of transfer learning using a pre-trained deep CNN model like VGG-19. We will be leveraging the same pre-trained model for the task of neural style transfer. The main steps are outlined as follows.

- Leverage VGG-19 to help compute layer activations for the style, content, and generated image.
- Use these activations to define specific loss functions mentioned earlier.
- Finally, use gradient descent to minimize the overall loss.

We recommend you follow along this section with the notebook titled `Neural Style Transfer.ipynb`, which contains step-by-step details of the style transfer process. We would also like to give a special mention and thanks to François Chollet as well as Harish Narayanan for providing some excellent resources on style transfer. Details on the same will be mentioned later. We also recommend you check out the following papers (detailed links shared later on).

- *A Neural Algorithm of Artistic Style* by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge
- *Perceptual Losses for Real-Time Style Transfer and Super-Resolution* by Justin Johnson, Alexandre Alahi, and Li Fei-Fei

Preprocessing

The first and foremost step toward implementation of such a network is to preprocess the data or images in this case. The following are quick utilities to preprocess images for size and channel adjustments.

```
import numpy as np
from keras.applications import vgg19
from keras.preprocessing.image import load_img, img_to_array

def preprocess_image(image_path, height=None, width=None):
    height = 400 if not height else height
    width = width if width else int(width * height / height)
    img = load_img(image_path, target_size=(height, width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return img

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

As we would be writing custom loss functions and manipulation routines, we would need to define certain placeholders. keras is a high-level library that utilizes tensor manipulation backends (like tensorflow, theano, and CNTK) to perform the heavy lifting. Thus, these placeholders provide high-level abstractions to work with the underlying *tensor* object. The following snippet prepares placeholders for style, content, and generated images along with the input tensor for the neural network.

```
In [1]: # This is the path to the image you want to transform.
   ...: TARGET_IMG = 'data/city_road.jpg'
   ...: # This is the path to the style image.
   ...: REFERENCE_STYLE_IMG = 'data/style2.png'
   ...:
   ...: width, height = load_img(TARGET_IMG).size
   ...: img_height = 320
   ...: img_width = int(width * img_height / height)
   ...:
   ...:
   ...: target_image = K.constant(preprocess_image(TARGET_IMG,
   ...:                                             height=img_height,
   ...:                                             width=img_width))
   ...: style_image = K.constant(preprocess_image(REFERENCE_STYLE_IMG,
   ...:                                             height=img_height,
   ...:                                             width=img_width))
   ...:
   ...: # Placeholder for our generated image
   ...: generated_image = K.placeholder((1, img_height, img_width, 3))
   ...:
   ...: # Combine the 3 images into a single batch
   ...: input_tensor = K.concatenate([target_image,
   ...:                               style_image,
   ...:                               generated_image], axis=0)
```

We will load the pre-trained VGG-19 model as we did in the previous section, i.e., without the top fully connected layers. The only difference here is that we would be providing the model constructor, the size dimensions of the input tensor. The following snippet fetches the pretrained model.

```
In [2]: model = vgg19.VGG19(input_tensor=input_tensor,
   ...:                     weights='imagenet',
   ...:                     include_top=False)
```

You may use the summary() function to understand the architecture of the pre-trained model.

Loss Functions

As discussed in the background subsection, the problem of neural style transfer revolves around loss functions of content and style. In this subsection, we will discuss and define the required loss functions.

Content Loss

In any CNN-based model, activations from top layers contain more global and abstract information (high-level structures like a face) and bottom layers will contain local information (low-level structures like eyes, nose, edges, and corners) about the image. We would want to leverage the top layers of a CNN for capturing the right representations for the content of an image.

Hence, for the content loss, considering we will be using the pretrained VGG-19 CNN, we can define our loss function as the L2 norm (scaled and squared Euclidean distance) between the activations of a top layer (giving feature representations) computed over the target image and the activations of the same layer computed over the generated image. Assuming we usually get feature representations relevant to the content of images from the top layers of a CNN, the generated image is expected to look similar to the base target image. The following snippet showcases the function to compute the content loss.

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

Style Loss

The original paper on neural style transfer, A Neural Algorithm of Artistic Style by Gatys et al., leverages multiple convolutional layers in the CNN (instead of one) to extract meaningful patterns and representations capturing information pertaining to appearance or style from the reference style image across all spatial scales irrespective of the image content.

Staying true to the original paper, we will be leveraging the *Gram matrix* and computing the same over the feature representations generated by the convolution layers. The Gram matrix computes the inner product between the feature maps produced in any given conv layer. The inner products terms are proportional to the covariances of corresponding feature sets and hence captures patterns of correlations between the features of a layer that tend to activate together. These feature correlations help capture relevant aggregate statistics of the patterns of a particular spatial scale, which correspond to the style, texture, and appearance and not the components and objects present in an image.

The style loss is thus defined as the scaled and squared *Frobenius* norm of the difference between the Gram matrices of the reference style and the generated images. Minimizing this loss helps ensure that the textures found at different spatial scales in the reference style image will be similar in generated image.

The following snippet thus defines a style loss function based on Gram matrix calculation.

```
def style_loss(style, combination, height, width):

    def build_gram_matrix(x):
        features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
        gram_matrix = K.dot(features, K.transpose(features))
        return gram_matrix

    S = build_gram_matrix(style)
    C = build_gram_matrix(combination)
    channels = 3
    size = height * width
    return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

Total Variation Loss

It was observed that optimization to reduce only the style and content losses led to highly pixelated and noisy outputs. To cover the same, *total variation loss* was introduced.

The total variation loss is analogous to *regularization* loss. This is introduced for ensuring spatial continuity and smoothness in the generated image to avoid noisy and overly pixelated results. The same is defined in the function as follows.

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

Overall Loss Function

Having defined the components of the overall loss function for neural style transfer, the next step is to piece together these building blocks. Since content and style information is captured by the CNNs at different depths in the network, we need to apply and calculate loss at appropriate layers for each type of loss. Utilizing insights and research by Gatys et al. and Johnson et al. in their respective papers, we define the following utility to identify the content and style layers from the VGG-19 model. Even though Johnson et al. leverages the VGG-16 model for faster and better performance, we constrain ourselves to the VGG-19 model for ease of understanding and consistency across runs.

```
# define function to set layers based on source paper followed
def set_cnn_layers(source='gatys'):
    if source == 'gatys':
        # config from Gatys et al.
        content_layer = 'block5_conv2'
        style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1',
                        'block4_conv1', 'block5_conv1']
    elif source == 'johnson':
        # config from Johnson et al.
        content_layer = 'block2_conv2'
        style_layers = ['block1_conv2', 'block2_conv2', 'block3_conv3',
                        'block4_conv3', 'block5_conv3']
    else:
        # use Gatys config as the default anyway
        content_layer = 'block5_conv2'
        style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1',
                        'block4_conv1', 'block5_conv1']
    return content_layer, style_layers
```

The following snippet then applies the overall loss function based on the layers selected from the `set_cnn_layers()` function for content and style.

```
In [2]: # weights for the weighted average loss function
   ...: content_weight = 0.025
   ...: style_weight = 1.0
   ...: total_variation_weight = 1e-4
   ...:
   ...: # set the source research paper followed and set the content and style layers
   ...: source_paper = 'gatys'
   ...: content_layer, style_layers = set_cnn_layers(source=source_paper)
   ...:
   ...: ## build the weighted loss function
   ...:
   ...: # initialize total loss
   ...: loss = K.variable(0.)
   ...:
   ...: # add content loss
   ...: layer_features = layers[content_layer]
   ...: target_image_features = layer_features[0, :, :, :]
   ...: combination_features = layer_features[2, :, :, :]
   ...: loss += content_weight * content_loss(target_image_features,
   ...:                                        combination_features)
   ...:
   ...: # add style loss
   ...: for layer_name in style_layers:
   ...: layer_features = layers[layer_name]
   ...: style_reference_features = layer_features[1, :, :, :]
   ...: combination_features = layer_features[2, :, :, :]
   ...: sl = style_loss(style_reference_features, combination_features,
   ...:                 height=img_height, width=img_width)
   ...: loss += (style_weight / len(style_layers)) * sl
   ...:
   ...: # add total variation loss
   ...: loss += total_variation_weight * total_variation_loss(generated_image)
```

Custom Optimizer

The objective is to iteratively minimize the overall loss with the help of an optimization algorithm. In the paper by Gatys et al., optimization was done using the L-BFGS algorithm, which is an optimization algorithm based on *quasi-Newton* methods, which is popularly used for solving non-linear optimization problems and parameter estimation. This method usually converges faster than *standard gradient descent*. SciPy has an implementation available in scipy.optimize.fmin_l_bfgs_b(). However, limitations include the function being applicable only to flat 1D vectors, unlike 3D image matrices which we are dealing with, and the fact that value of loss function and gradients need to be passed as two separate functions.

We build an Evaluator class based on patterns followed by keras creator François Chollet to compute both loss and gradients values in one pass instead of independent and separate computations. This will return the loss value when called the first time and will cache the gradients for the next call. Thus, it would be more efficient than computing both independently. The following snippet defines the Evaluator class.

```
class Evaluator(object):

    def __init__(self, height=None, width=None):
        self.loss_value = None
        self.grads_values = None
        self.height = height
        self.width = width
    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, self.height, self.width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value

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

The loss and gradients are retrieved as follows. The snippet also creates an object of the Evaluator class.

```
In [3]: # Get the gradients of the generated image wrt the loss
   ...: grads = K.gradients(loss, generated_image)[0]
   ...:
   ...: # Function to fetch the values of the current loss and the current gradients
   ...: fetch_loss_and_grads = K.function([generated_image], [loss, grads])
   ...:
   ...: # evaluator object
   ...: evaluator = Evaluator(height=img_height, width=img_width)
```

Style Transfer in Action

The final piece of the puzzle is to use all the building blocks and see the style transfer in action. The art/style and content images are available *data* directory for reference. The following snippet outlines how loss and gradients are evaluated. We also write back outputs after regular intervals (5, 10, and so on iterations) to later understand how the process of neural style transfer transforms the images in consideration.

```
In [4]: result_prefix = 'style_transfer_result_'+TARGET_IMG.split('.')[0]
   ...: result_prefix = result_prefix+'_'+source_paper
   ...: iterations = 20
   ...:
   ...: # Run scipy-based optimization (L-BFGS) over the pixels of the generated image
   ...: # so as to minimize the neural style loss.
   ...: # This is our initial state: the target image.
   ...: # Note that `scipy.optimize.fmin_l_bfgs_b` can only process flat vectors.
   ...: x = preprocess_image(TARGET_IMG, height=img_height, width=img_width)
   ...: x = x.flatten()
   ...:
   ...: for i in range(iterations):
   ...:         print('Start of iteration', (i+1))
   ...:         start_time = time.time()
   ...:         x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x,
   ...:                                 fprime=evaluator.grads, maxfun=20)
   ...:         print('Current loss value:', min_val)
   ...:         if (i+1) % 5 == 0 or i == 0:
   ...:                 # Save current generated image only every 5 iterations
   ...:                 img = x.copy().reshape((img_height, img_width, 3))
   ...:                 img = deprocess_image(img)
   ...:                 fname = result_prefix + '_at_iteration_%d.png' %(i+1)
   ...:                 imsave(fname, img)
   ...:                 print('Image saved as', fname)
   ...:         end_time = time.time()
   ...:         print('Iteration %d completed in %ds' % (i+1, end_time - start_time))
```

It must be pretty evident by now that neural style transfer is a computationally expensive task. For the set of images in consideration, each iteration took between 500-1000 seconds on a Intel i5 CPU with 8GB RAM. On an average, each iteration takes around 500 seconds but if you run multiple networks together, each iteration takes up to 1,000 seconds. You may observe speedups if the same is done using GPUs. The following is the output of some of the iterations. We print the loss and time taken for each iteration and save the image after five iterations.

```
Start of iteration 1
Current loss value: 2.4219e+09
Image saved as style_transfer_result_city_road_gatys_at_iteration_1.png
Iteration 1 completed in 506s
Start of iteration 2
Current loss value: 9.58614e+08
Iteration 2 completed in 542s
Start of iteration 3
Current loss value: 6.3843e+08
Iteration 3 completed in 854s
Start of iteration 4
Current loss value: 4.91831e+08
Iteration 4 completed in 727s
Start of iteration 5
Current loss value: 4.03013e+08
Image saved as style_transfer_result_city_road_gatys_at_iteration_5.png
Iteration 5 completed in 878s
 ...
Start of iteration 19
Current loss value: 1.62501e+08
Iteration 19 completed in 836s
Start of iteration 20
Current loss value: 1.5698e+08
Image saved as style_transfer_result_city_road_gatys_at_iteration_20.png
Iteration 20 completed in 838s
```

Now you'll learn how the neural style transfer has worked out for the images in consideration. Remember that we performed checkpoint outputs after certain iterations for every pair of style and content images.

■ **Note** The style we use for our first image, depicted in Figure 12-7, is named **Edtaonisl.** This is a 1913 master piece by Francis Picabia. Through this oil painting Francis Picabia pioneered a new visual language. More details about this painting are available at `http://www.artic.edu/aic/collections/artwork/80062`.

We utilize `matplotlib` and `skimage` libraries to load and understand the style transfer magic! The following snippet loads the city road image as our content and *Edtaonisl* painting as our style image.

```
In [5]: from skimage import io
   ...: from glob import glob
   ...: from matplotlib import pyplot as plt
   ...:
   ...: cr_content_image = io.imread('results/city road/city_road.jpg')
   ...: cr_style_image = io.imread('results/city road/style2.png')
   ...:
   ...:
   ...: fig = plt.figure(figsize = (12, 4))
   ...: ax1 = fig.add_subplot(1,2, 1)
   ...: ax1.imshow(cr_content_image)
   ...: t1 = ax1.set_title('City Road Image')
   ...: ax2 = fig.add_subplot(1,2, 2)
   ...: ax2.imshow(cr_style_image)
   ...: t2 = ax2.set_title('Edtaonisl Style')
```

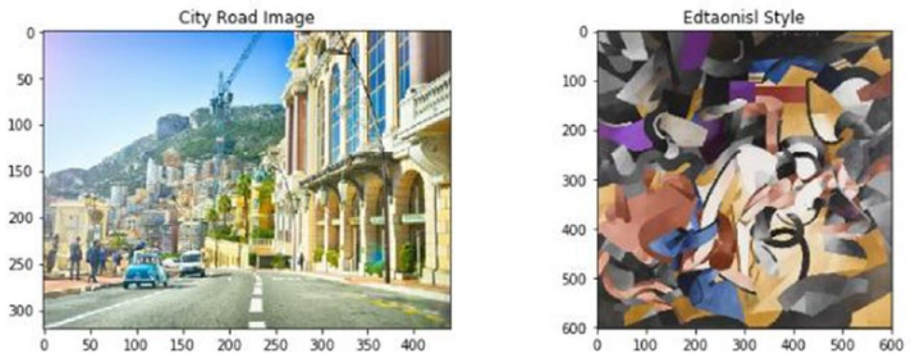


Figure 12-7. *The City Road image as content and the Edtaonisl painting as style image for neural style transfer*

The following snippet loads the generated images (style transferred images) as observed after the first, tenth, and twentieth iteration.

```
In [6]: fig = plt.figure(figsize = (20, 5))
   ...: ax1 = fig.add_subplot(1,3, 1)
   ...: ax1.imshow(cr_iter1)
   ...: t1 = ax1.set_title('Iteration 1')
   ...: ax2 = fig.add_subplot(1,3, 2)
   ...: ax2.imshow(cr_iter10)
```

```
    ...: t2 = ax2.set_title('Iteration 10')
    ...: ax3 = fig.add_subplot(1,3, 3)
    ...: ax3.imshow(cr_iter20)
    ...: t3 = ax3.set_title('Iteration 20')
    ...: t = fig.suptitle('City Road Image after Style Transfer')
```

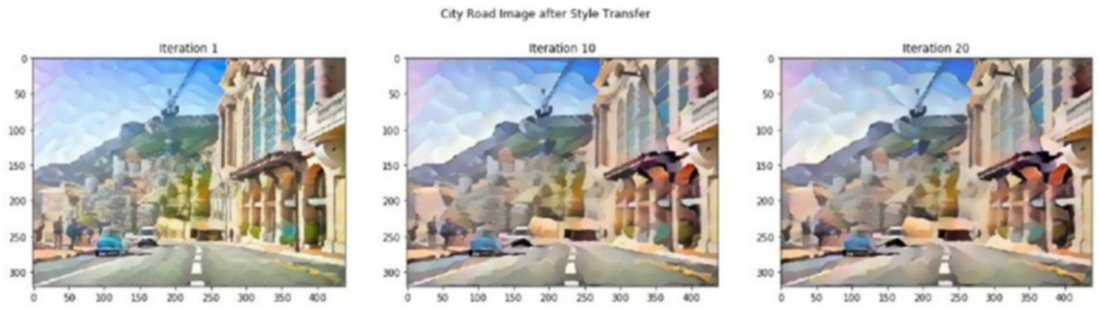


Figure 12-8. *The City Road image style transfer at the first, tenth, and twentieth iteration*

The results depicted in Figure 12-8 sure seem pleasant and amazing. It is quite apparent how the generated image in the initial iterations resembles the structure of the content and, as the iterations progress, the style starts influencing the texture, color, strokes, and so on, more and more.

■ **Note** The style used in our next example depicted in Figure 12-9 is the famous painting named *The Great Wave* by Katsushika Hokusai. The artwork was completed in 1830-32. It is amazing to see the styles of such talented artists being transferred to everyday photographs. More on this artwork is available at `http://www.metmuseum.org/art/collection/search/36491`.

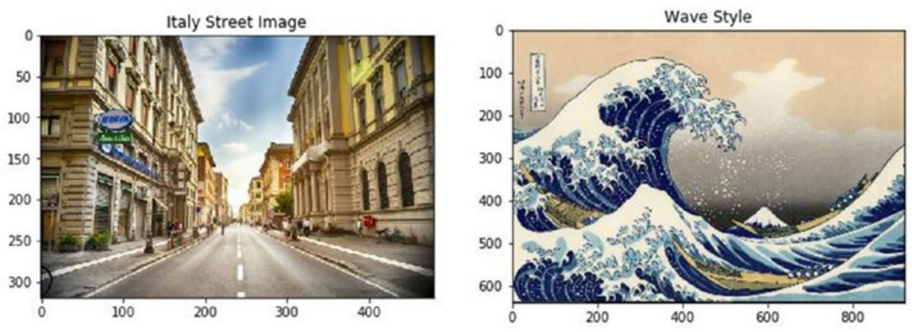


Figure 12-9. *The Italy Street image as content and Wave Style painting as the style image for neural style transfer*

We experimented with a few more sets of images and the results truly were surprising and pleasant to look at. The output from neural style transfer for an image depicting an Italian street (see Figure 12-9) is shown in Figure 12-10 at different iterations.

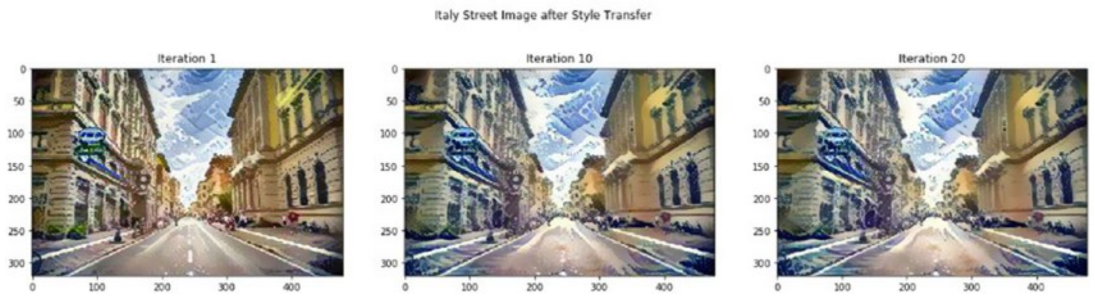


Figure 12-10. *Italian street image style transfer at the first, tenth and twentieth iteration*

The results depicted in Figure 12-10 are definitely a pleasure to look at and give the feeling of an entire city underwater! We encourage you to use images of your own with this same framework. Also feel free to experiment with leveraging different convolution layers for the style and content feature representations as mentioned in Gatys et al. and Johnson et al.

■ **Note** The concept and details of neural style transfer were introduced and explained by Gatys et al. and Johnson et al. in their respective papers available at `https://arxiv.org/abs/1508.06576` and `https://arxiv.org/abs/1603.08155`. You can also check out the book *Deep Learning with Python* by François Chollet as well as Harish Narayanan's excellent blog for a detailed step-by-step guide on neural style transfer: `https://harishnarayanan.org/writing/artistic-style-transfer/`.

Summary

This chapter presented topics from the very forefront of the Machine Learning landscape. Through this chapter we utilized our learnings about Machine Learning in general and Deep Learning in particular to understand the concepts of image classification, transfer learning, and style transfer. The chapter started off with a quick brush up of concepts related to Convolutional Neural Networks and how they are optimized architectures to handle image related data. We then worked towards developing image classifiers. The first classifier was developed from scratch and with the help of `keras` we were able to achieve decent results. The second classifier utilized a pre-trained VGG-19 deep CNN model as an image feature extractor. The pre-trained model based classifier helped us understand the concept of transfer learning and how it is beneficial. The closing section of the chapter introduced the advanced topic of Neural Style Transfer, the main highlight of this chapter. Style transfer is the process of applying the style of a reference image to a specific target image such that in the process, the original content of the target image remains unchanged. This process utilizes the potential of CNNs to understand image features at different granularities along with transfer learning. Based on our understanding of these concepts and the research work by Gatys et al. and Johnson et al., we provided a step-by-step guide to implement a system of neural style transfer. We concluded the section by presenting some amazing results from the process of neural style transfer.

Deep Learning is opening new doors every day. Its application to different domains and problems is showcasing its potential to solve problems previously unknown. Machine Learning is an ever evolving and a very involved field. Through this book, we traveled from the basics of Machine Learning frameworks, Python ecosystem to different algorithms and concepts. We then covered multiple use cases across chapters showcasing different scenarios and ways a problem can be solved using the tools from the Machine Learning toolbox. The universe of Machine Learning is expanding at breakneck speeds; our attempt here was to get you started on the right track, on this wonderful journey.