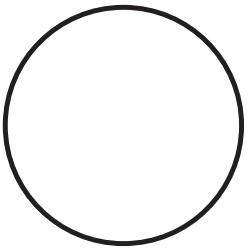
3

PROCESSING IMAGES WITH CONVOLUTIONAL NEURAL NETWORKS



In this chapter, you'll dive into convolutional neural networks, the primary tool for using deep learning for image analysis. You'll then apply this knowledge to classifying blood smears used to identify malaria and to measuring the position of microscopic particles from their images.

You'll begin by exploring how convolutions provide powerful tools to extract information from images. You'll learn to implement convolutional layers—the fundamental building blocks of convolutional neural networks. Then you'll discover how downsampling and upsampling layers are used to modify the spatial resolution of the feature maps generated by convolutional layers in advanced convolutional architectures. You'll also learn how to use heatmaps to better understand the workings of convolutional neural networks, showcasing the features they learn in an accessible and insightful way.

This chapter ends with two projects that demonstrate the creative potential of convolutional neural networks by generating artistic and visually stimulating outputs. The DeepDreams project demonstrates how to use

convolutional neural networks to transform images into dreamlike scenes, showcasing the network's ability to amplify patterns in a visually intriguing way. Finally, the style transfer project explores how you can apply convolutional neural networks to merge the style of one image with the content of another, leading to captivating results.

Understanding Convolutions

A *convolution* is a blending process that combines two sets of data to produce a new set. Imagine you have a sequence of numbers (call this your main data) and a smaller set of numbers (think of this as a filter). You apply this filter to your main data by sliding it across, step-by-step.

At each step, you multiply the numbers in the filter with the numbers they cover in the main data, and then add these up to get a single number. This process is repeated across the entire main data sequence. The resulting series of numbers is your output—a transformed version of the original data that can highlight some of its properties.

This method is widely used in signal and image processing, where it helps in tasks like sharpening or blurring signals and images, as you'll see shortly.

Convolving 1D Data

To understand how a 1D convolution works, consider a sequence of values in a 1D signal. You may want to compute its *moving average*, which is derived at each point from its adjacent values in the sequence. You can implement this with a 1D convolution, as shown in Listing 3-1.

```
import numpy as np

signal = np.array([0, 2, 0, 2, 0, 2, 0, 2, 0])

filter1d = np.ones(2) / 2

❶ conv1d_length = signal.shape[0] - filter1d.shape[0] + 1
conv1d = np.zeros((conv1d_length,))
for i in range(conv1d_length):
  ❷ conv1d[i] = np.sum(signal[i:i + filter1d.shape[0]] * filter1d)
```

Listing 3-1: Implementing a moving average with a 1D convolution

This script calculates the moving average of a 1D signal by using a 1D convolution. It begins by defining the `signal` array with nine data points, representing the original 1D signal. Next, it creates the `filter1d` filter of length 2 with values `[0.5, 0.5]`, indicating that each element within a two-point window of the signal will contribute equally to the average.

The convolution process starts by first determining the length of the output array ❶, which is calculated as the length of the signal minus the length of the filter plus 1. This calculation is essential to ensure that the convolution process covers the entire signal without exceeding its bounds. The reason for adding 1 is to account for the number of *valid complete placements* of the filter over the signal. The new `conv1d` array of this length is initialized to store the results. The script then performs the convolution operation by iteratively sliding the filter along the signal. At each step ❷, the script multiplies corresponding elements of the signal and the filter, sums them up, and stores the result in `conv1d`.

You can use `print(conv1d)` to print the resulting array, which represents the convolved signal:

```
[1 1 1 1 1 1 1 1]
```

This is a smoothed version of the original signal; while the original signal alternated 0s and 2s, the resulting signal is always 1.

Figure 3-1 illustrates this process. The signal on the left of the figure is convolved with a rectangular filter `[0.5, 0.5]` obtaining the averaged signal, corresponding to that calculated in Listing 3-1.

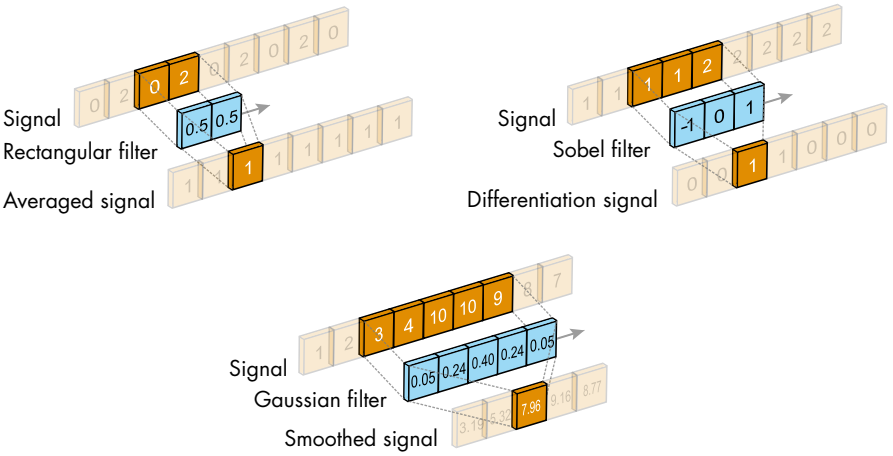


Figure 3-1: 1D convolutions of a signal with various filters

The other two panels of Figure 3-1 show the effect of convolving a signal with different filters. On the right, a signal is convolved with a *Sobel filter* of length 3 performing a differentiation operation often used to detect steps. Finally, at the bottom, another signal is convolved with a *Gaussian filter* of length 5 with unitary standard deviation obtaining another smoothed version of the signal.

EXERCISES

3-1: Detect the presence of steps in a signal by using a Sobel filter. For example, use the Sobel filter of length 3 `[-1, 0, 1]` with various signals (for your reference, in 1D, the Sobel filter is the same as the *Prewitt filter*). With a constant signal, you should get a null convolution. On the other hand, with a signal featuring an abrupt change (such as a step function), you should get a convolution with a spike near the change point.

3-2: Smooth a signal by convolving it with a Gaussian filter. For example, you can use a Gaussian filter of length 5 `[0.0545, 0.2442, 0.4026, 0.2442, 0.0545]`.

3-3: Detect a specific pattern in a signal by convolving it with a filter matching the pattern. The convolution will have a high value at the locations where the pattern is present in the signal.

3-4: Sharpen a signal by first smoothing it with a Gaussian filter and then subtracting this smoothed version of the signal from the original one.

Convolving 2D Data

To extend the concept of convolutions to higher dimensions, you'll now implement a 2D convolution using Listing 3-2.

```
❶ image = np.array([
      [1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
      [1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
      [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
      [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
  ])

❷ filter2d = np.ones((2, 2)) / 4

  conv2d_height = image.shape[0] - filter2d.shape[0] + 1
  conv2d_width = image.shape[1] - filter2d.shape[1] + 1
  conv2d = np.zeros((conv2d_height, conv2d_width))
  for i in range(conv2d_height):
      for j in range(conv2d_width):
          conv2d[i, j] = np.sum(
              image[i:i + filter2d.shape[0], j:j + filter2d.shape[1]] * filter2d
          )
```

Listing 3-2: Implementing a 2D convolution

This script convolves a 4×10–pixel (height by width) image ❶ with a 2×2–pixel filter ❷, using a procedure similar to that used for the 1D convolution. The 2D convolution process uses two nested `for` loops, which iterate over each pixel position in the image where the filter can be applied. At each position, the filter overlaps with a part of the image, and an element-wise

multiplication followed by a sum is computed. This sum represents the convolved value at that specific location in the output array. This procedure is repeated across the entire image.

If you print the resulting convolution with `print(conv2d)`, you get a smoothed version of the original image:

```
[[1.  0.5 0.  0.5 1.  0.5 0.  0.5 1. ]
 [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]
 [0.  0.5 1.  0.5 0.  0.5 1.  0.5 0. ]]
```

You can see that this output is a smoothed version of the original image from the gradual transitions in pixel values: Instead of abrupt shifts from 1 to 0 as in the original image, there is now an intermediate value of 0.5. This output is also an image with smaller dimensions (3×9 pixels) than the original image, corresponding to the valid *complete placements* of the filter over the image, which are areas where the filter fully fits within the original image boundaries during the convolution process.

This process is illustrated in Figure 3-2.

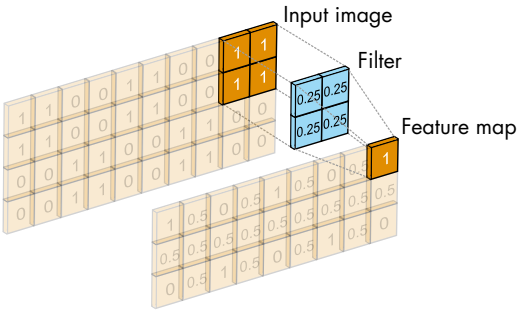


Figure 3-2: A 2D convolution of an image with a filter

The input image at the top is convolved with the filter, generating the convolved image at the bottom, which is often referred to as a *feature map*.

EXERCISES

3-5: Detect the edges in an image by convolving it with a 2D Prewitt filter. For example, use a filter with values `[[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]` to detect vertical edges, and a filter with values `[[1, 1, 1], [0, 0, 0], [-1, -1, -1]]` to detect horizontal edges. Alternatively, you could use 2D Sobel filters with values `[[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]` and `[-1, -2, -1], [0, 0, 0], [1, 2, 1]` for the same purposes.

3-6: Blur an image by convolving it with a 2D Gaussian filter. For example, use a Gaussian filter of size 5×5 with a standard deviation of 1.

3-7: Detect a specific pattern in an image by convolving it with a filter matching the pattern.

If the input image contains multiple color channels, such as in RGB (red, green, and blue) images, the convolution combines all the color channels, as shown in Figure 3-3.

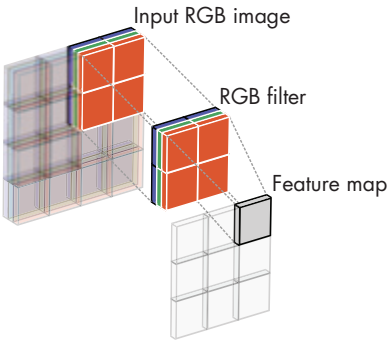


Figure 3-3: A 2D convolution of an RGB image with a filter

For a multichannel image, the image data is represented as a 3D array (height by width by channels), as shown on the left. The convolution operation uses a filter that extends through all the color channels. This filter performs the convolution across the height and width of the image, while simultaneously aggregating information from all the color channels. By combining all color channels, this convolution effectively integrates the spatial and color information, leading to a more comprehensive analysis of the image's features.

Using Convolutions in a Neural Network

Now that you've had a glimpse of the power of convolutions in analyzing signals and images, you're ready to integrate them within neural networks.

Convolutional layers consist of neural network layers containing multiple convolutions. They enable the construction of sophisticated neural network architectures to analyze signals and images. The versatility of convolutions in signal and image processing stems from their ability to perform varied operations with different filters, which is especially crucial in deep learning.

Filters are typically learned during training to achieve specific objectives, allowing for the extraction of significant features from signals and images. This adaptability through training explains why convolutions are a fundamental element in neural network architectures.

Implementing Neural Networks in PyTorch

In this section, you'll use PyTorch to learn some of the basic building blocks of neural network architectures. You'll see convolutional, activation, pooling, downsampling, and dense layers, as well as how to sequentially stack them to create a deep neural network.

Start by creating a sample image representing a checkerboard, using Listing 3-3.

```
import torch

H, W, S = 12, 16, 4  # Height, width, square size
❶ image = torch.zeros(1, H, W)
for idx in range(0, H, S):
    for idy in range(0, W, S):
      ❷ image[0, idx:idx + S, idy:idy + S] = (-1) ** (idx // S + idy // S)
```

Listing 3-3: Creating a sample image

The image is stored in the `image` PyTorch tensor (a multidimensional array used by PyTorch to store and process data efficiently for deep learning models, which allows for optimized computations on GPUs). This tensor has three dimensions, corresponding to the number of color channels (`1`), the height (`H`), and the width (`W`) ❶. The image is a checkerboard with squares of size `S = 4` taking as values either 1 or −1 ❷. This image has a single color channel, so you can think of it as grayscale.

You can now write the `plot_image()` function shown in Listing 3-4.

```
import matplotlib.pyplot as plt

def plot_image(image):
    """Render an image."""
  ❶ plt.imshow(image, cmap="gray", aspect="equal", vmin=-2, vmax=2,
               extent=[0, image.shape[1], 0, image.shape[0]])
    plt.colorbar()
  ❷ plt.xticks(range(0, image.shape[1] + 1))
  ❸ plt.yticks(range(0, image.shape[0] + 1))
  ❹ plt.grid(color="red", linewidth=1)
    plt.show()
```

Listing 3-4: The function to render an image

This function renders the image via the Matplotlib `imshow()` function with a grid highlighting its pixels ❹, ensuring that the grid lines are at the beginning and end of each pixel ❷ ❸. Even though the values of the image range from −1 to 1, the color bar limits are set from −2 to 2 ❶ to make this image directly comparable with the subsequent ones.

Now, use this function to plot the image you've created:

```
plot_image(image.squeeze())
```

When passing `image` to the `plot_image()` function, you need to eliminate the extra dimensions that are of size 1, which is done using the `squeeze()` method.

Figure 3-4 shows the rendered image.

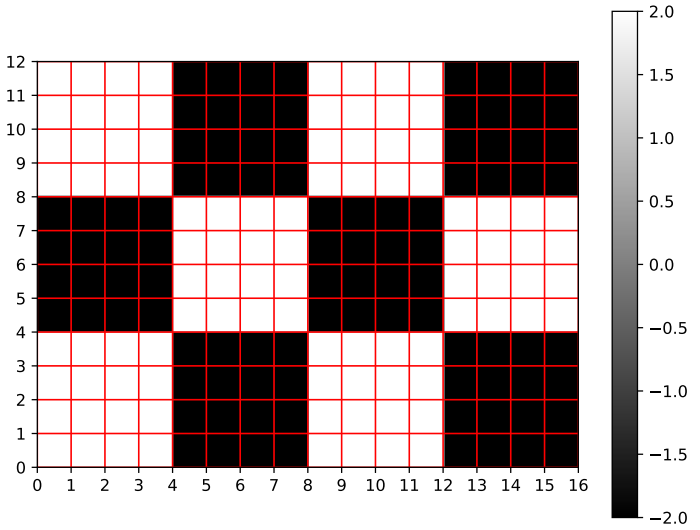


Figure 3-4: The sample image representing a checkerboard

The image is a checkerboard with three rows and four columns of 4×4-pixel squares with values −1 (dark gray) and 1 (light gray). In total, the image has 12×16 pixels (height by width).

Defining Convolutional Layers

You'll now define a convolutional layer, set the values of its filters, and apply it to the checkerboard image to see how it transforms, as shown in Listing 3-5.

```
import torch.nn as nn

conv = nn.Conv2d(in_channels=1, out_channels=2, kernel_size=(1, 3), bias=False)
❶ filters = torch.zeros(conv.out_channels, conv.in_channels, *conv.kernel_size)
❷ filters[0, 0, :, :] = torch.tensor([[1, 1, 1],]) / 3
❸ filters[1, 0, :, :] = torch.tensor([[-1, 0, 1],])
❹ conv.weight = nn.Parameter(filters)

❺ features_conv = conv(image.unsqueeze(0))
```

Listing 3-5: Implementing a convolutional layer in PyTorch

This code creates the `conv` convolutional layer with a single color channel (`in_channels=1`) and two filters (`out_channels=2`) with size `kernel_size=(1, 3)` (the terms *kernel* and *filter* are frequently used interchangeably), while setting the bias to 0 (`bias=False`). Similar to the bias in neurons discussed in Chapters 1 and 2, the bias in convolutional layers adds a constant offset to

the output. In the rest of this section, you'll set the bias to 0 to focus on the convolutional operation.

By default, the filters of a convolutional layer are initialized to random numbers. In Listing 3-5, however, you set the filters by overwriting the randomly initialized ones. You first initialize a `filters` tensor with 0s, conforming to the shape required by the convolution layer ❶. The first index in `filters` corresponds to the filter number, and the second index to the channel number. Then you modify this tensor to define two specific kernels: the first, which is an averaging filter ❷, and the second, which is an edge-detection filter ❸. Finally, you set these custom kernels as weights in the convolutional layer ❹.

You can think of the output, `features_conv`, as a new image with two pseudo–color channels created by the kernels. Importantly, you need to batch the input image before passing it through the convolutional layer, which is a requirement of PyTorch (and most other deep learning frameworks). This involves adding an additional dimension to represent the batch size by using the `unsqueeze(0)` method ❺. To batch the image, in this case, you transform `image`, which is a 3D tensor of shape `[1, 12, 16]` (channels, height, width), into a 4D tensor of shape `[1, 1, 12, 16]`, where the first dimension represents the batch.

To visualize the output feature maps, implement the `plot_channels()` function in Listing 3-6.

```
def plot_channels(channels, figsize=(15, 5)):
    """Render multiple channels."""
    fig, axs = plt.subplots(1, channels.shape[0], figsize=figsize)
    for channel, ax, i in zip(channels, axs, range(channels.shape[0])):
     ❶ im = ax.imshow(channel, cmap="gray", aspect="equal", vmin=-2, vmax=2,
                      extent=[0, channel.shape[1], 0, channel.shape[0]])
       plt.colorbar(im)
       ax.set_title(f"Channel {i}", fontsize=24)
       ax.set_xticks(range(0, channel.shape[1] + 1))
       ax.set_yticks(range(0, channel.shape[0] + 1))
     ❷ ax.grid(color="red", linewidth=1)
    plt.show()
```

Listing 3-6: The function to render the multiple channels of an image

For each channel of an image, this function renders the channel ❶ and overlays a grid to highlight the pixels ❷.

Now you can use this function to render the feature maps obtained by the convolution:

```
plot_channels(features_conv[0].detach())
```

When passing the feature maps to the `plot_channels()` function, you need to first extract the first (and, in this case, only) image of the batch and then use the `detach()` method to tell PyTorch that you don't require gradient calculation for the image (typically required for the backpropagation).

You should get Figure 3-5.

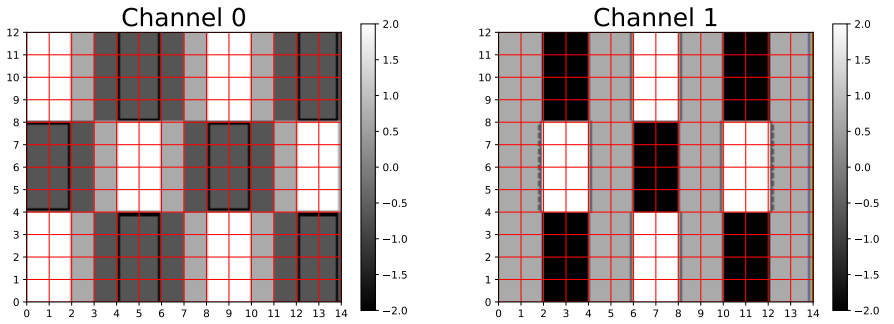


Figure 3-5: The feature maps obtained by convolutions with the filters

You can see two feature maps (channels), corresponding to the two filters in the convolutional layer you've defined in Listing 3-5. The first channel (Channel 0) corresponds to the feature map obtained by applying the averaging filter, which acts only along the horizontal direction. By averaging the intensity values, this filter produces a smoothing effect, reducing the distinction between adjacent horizontal regions.

The second channel (Channel 1) represents the feature map produced by the edge-detection filter. This filter highlights the horizontal transitions in intensity by accentuating edges perpendicular to the horizontal axis, as shown by the pronounced contrast between adjacent areas in the horizontal direction.

EXERCISES

3-8: Revisit the example in this section, using various filters. For example, use a vertical edge detector and a 3×3 Gaussian filter.

3-9: Until now, you've used grayscale images, but usually images have multiple colors. Revisit the example in this section, using an RGB image as input.

Adding ReLU Activation

You can now add a ReLU activation to the output of the convolutional layer:

```
relu = nn.ReLU()
model_relu = nn.Sequential(conv, relu)

features_relu = model_relu(image.unsqueeze(0))

plot_channels(features_relu[0].detach())
```

After creating a ReLU activation (`relu`), this code combines the convolutional layer and the ReLU activation via `nn.Sequential()`. The resulting output is rendered in Figure 3-6.

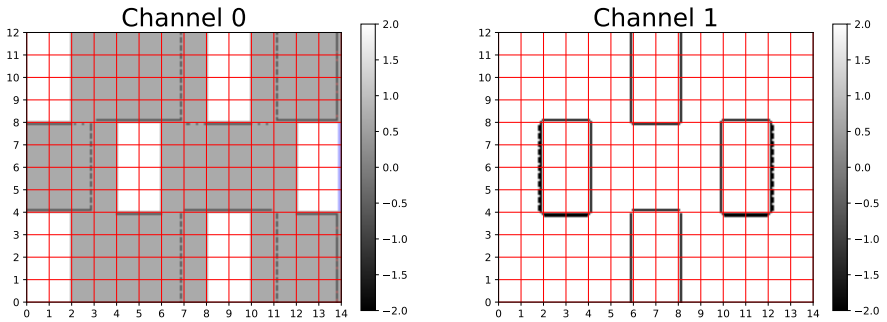


Figure 3-6: The image after convolution and ReLU activation

Comparing Figure 3-6 with Figure 3-5, you can see that all negative values have been converted to 0s.

EXERCISE

3-10: While the ReLU activation function is commonly used for its simplicity and efficiency, there are several other options that can impact the network's performance, as you saw in Chapter 1. For instance, the sigmoid function, which maps input values to a range of 0 to 1, can be useful for binary classification tasks. Another option is the hyperbolic tangent function, which scales the inputs to a range of –1 to 1, offering a centered scaling. A third option is the leaky ReLU, a variation of ReLU that allows a small gradient when the unit is inactive and can help with the *dying ReLU problem* but can hinder training if many of the layer's outputs are negative and converted to 0. Modify the activation function in the given code to use these alternatives and observe how they affect the output feature maps.

Adding Pooling Layers

Pooling layers (also known as *downsampling layers*) reduce the spatial resolution of feature maps, downsampling the data. For example, you might add a pooling layer to the convolutional layer to reduce the computational load by decreasing the number of parameters and operations needed in the network. Furthermore, this approach helps in detecting larger-scale features by summarizing the presence of features in larger patches of the input image. You can implement a pooling layer with the following code:

```
pool = nn.MaxPool2d(kernel_size=(2, 1), stride=(2, 1))
model_pool = nn.Sequential(conv, pool)

features_pool = model_pool(image.unsqueeze(0))

plot_channels(features_pool[0].detach())
```

This code creates a *max pooling layer* (`pool`) with `kernel_size=(2, 1)`, which means it extracts the maximum value over a window of 2 pixels in the vertical

direction (height) and 1 pixel in the horizontal direction (width). The maximum extraction is executed with `stride=(2, 1)`, which means that the window sequentially slides over the image with two steps in the vertical direction and one step in the horizontal one, effectively reducing the height by half and keeping the width the same. The code then combines the convolutional layer with the max pooling layer and plots the resulting feature maps, which are shown in Figure 3-7.

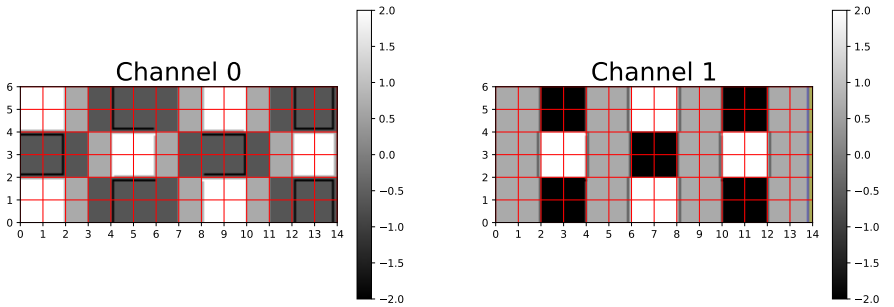


Figure 3-7: The image after convolution and pooling

By comparing Figure 3-7 with Figure 3-5, you'll notice that the pooled feature maps have half the height of those at the output of the convolutional layer. This reduction in dimensionality lessens the computational burden and helps in summarizing the information contained in larger patterns in the image.

Apart from max pooling, which selects the maximum value from the feature map within the pooling window, many alternative operations can be used by a pooling layer to downsample the feature maps. For example, *average pooling* computes the average value within the window, effectively smoothing the features; L_2*-norm pooling* calculates the square root of the sum of the squares of the pixel values, preserving the magnitude of large features; and *min pooling* selects the minimum value, useful when the absence of features is critical, as in background suppression or noise reduction.

EXERCISE

3-11: Revisit the given code to implement these alternative pooling operations; you can also change the `kernel_size` and `stride`. Observe how the output feature maps change.

Using Upsampling Layers

Upsampling layers (also known as *unpooling layers*) perform the inverse operation of pooling layers, increasing the spatial resolution of the feature maps. Understanding and utilizing both pooling and upsampling layers is crucial, as they are complementary techniques for manipulating feature

maps: While pooling layers reduce dimensionality to improve computational efficiency and feature-detection robustness, upsampling layers restore or enhance the resolution, which is especially important for tasks like image segmentation or generating high-resolution outputs from lower-resolution inputs. You can implement an upsampling layer with the following code:

```
upsample = nn.Upsample(scale_factor=(2, 1))
model_upsample = nn.Sequential(conv, upsample)

features_upsample = model_upsample(image.unsqueeze(0))

plot_channels(features_upsample[0].detach(), figsize=(15, 8))
```

This code defines an upsampling layer (`upsample`) with `scale_factor=(2, 1)`, which replaces each pixel by two vertically stacked pixels with the same value, thereby doubling the height of the feature maps while maintaining their original width. This upsampling layer is then combined with the convolutional layer and used to generate the output feature maps, which are rendered in Figure 3-8.

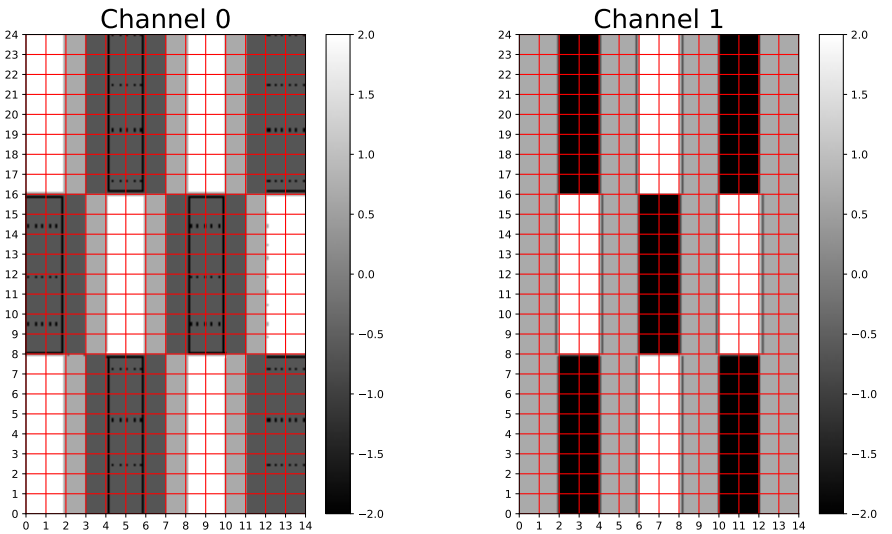


Figure 3-8: The image after convolution and upsampling

By comparing Figure 3-8 with Figure 3-5, you can verify that the upsampling has doubled the height of the feature maps, repeating each pixel twice along the vertical direction.

The upsampling method you've seen in this section uses a straightforward duplication approach that replicates pixels to enlarge the feature map. However, alternative upsampling techniques can offer different benefits. For example, *nearest neighbor upsampling* replicates the nearest pixel, potentially preserving sharper edges; *bilinear upsampling* uses linear interpolation,

leading to smoother transitions; and *trilinear upsampling*, suitable for 3D data, interpolates in three dimensions.

EXERCISES

3-12: Experiment with different upsampling methods by modifying the upsampling layer in the given code. Observe and compare the effects on the spatial resolution and the overall appearance of the upsampled feature maps.

3-13: Perform subsequent pooling and upsampling operations on an image. What do you observe? How does the result depend on the order of the two operations? Try different combinations of pooling and upsampling layers.

Transforming Images

Now that you've learned about convolutional, activation, pooling, and upsampling layers, you can combine them to construct complex convolutional architectures that transform images. This means applying operations to modify or analyze input images, crucial for tasks like image classification or object detection. The images being transformed will vary based on the application, such as facial recognition or medical imaging.

Listing 3-7 shows an example of a convolutional neural network for transforming images.

```
model_trans = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
)
❶ image_trans = model_trans(image.unsqueeze(0))[0]
```

Listing 3-7: Implementing a convolutional neural network to transform an image

This code defines a neural network model named `model_trans`, which consists of two convolutional layers, each followed by a ReLU activation function and a max pooling layer.

The first convolutional layer has an input channel size of 1 (suitable for grayscale images) and an output channel size of 16, with a kernel size of 3 in both directions (`kernel_size=3` is equivalent to `kernel_size=(3, 3)`). This is immediately followed by a ReLU activation function, which introduces nonlinearity in the network. Next, a max pooling layer with a kernel size of 2 and a stride of 2 in both directions reduces the spatial dimensions of the feature maps.

A similar sequence of convolutional layer, ReLU activation, and max pooling is repeated, with the second convolutional layer further increasing the number of output channels to 32.

When you use this network to process the usual input image, `image`, the result is a transformed image, `image_trans` ❶. Note that the first (and only, in this case) image is extracted from the batch. You can check the dimensions of the input and output with

```
print(f"Input image with {image.shape}")
print(f"Output image with {image_trans.shape}")
```

which prints:

```
Input image with torch.Size([1, 12, 16])
Output image with torch.Size([32, 1, 2])
```

The first indices indicate that the input image has one single color channel (it's grayscale), while the output image has 32 features. The second and third indices indicate that the input image has 12×16 pixels, but each of the output feature maps has only 1×2 pixels. Consequently, although the image has diminished in spatial resolution, it has simultaneously gained in the richness of its feature representation.

EXERCISES

3-14: Use Gaussian filters to get a smoothed, downsampled version of an image. Apply this to the MNIST digits.

3-15: Implement a code that enlarges an image through a series of upsampling and convolutional layers. Use Gaussian filters to smooth the upsampled image. Apply this to the MNIST digits.

3-16: Combine the reduction and enlargement of an image into a telescopic convolutional architecture. Apply this to the MNIST digits.

Using Dense Layers to Classify Images

You can also use convolutional neural networks for the classification of images. In this case, you typically need to flatten the feature maps obtained from the convolutional layers (known as the *convolutional base*) and couple them with a dense output layer (a *dense top*). For example, you can do this by expanding Listing 3-7 as shown in Listing 3-8.

```
model_clas = nn.Sequential(
    --snip--
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(in_features=32 * 1 * 2, out_features=2),
```

```
    nn.Softmax(dim=1),
)
classification = model_clas(image.unsqueeze(0))
```

Listing 3-8: Implementing a convolutional neural network with a dense top to classify images (by modifying Listing 3-7)

This code adds a dense top to the convolutional base, transitioning from feature extraction to classification. The addition of a flattening layer reshapes the multidimensional output of the convolutional layers into a 1D tensor, which is a necessary step before feeding the data into fully connected layers. The flattening is followed by a dense layer with two output features, corresponding to a binary classification. Finally, a softmax activation is applied, obtaining some values that are often interpreted as the relative likelihoods of the two possible classes, though they aren't probabilities in the traditional statistical sense.

If you use this network to process the usual image, `image`, it results in a classification vector, `classification`. You can check the dimensions of the input and output with

```
print(f"Input image with {image.shape}")
print(f"Output classification with {classification.shape}")
```

which prints:

```
Input image with torch.Size([1, 12, 16])
Output classification with torch.Size([2])
```

The output classification has two values corresponding to two classes.

NOTE *Code Example 3-1, "Implementing Neural Networks in PyTorch," is available at* https://github.com/DeepTrackAI/DeepLearningCrashCourse. *Navigate to the* Ch03_CNN *folder and then* ec03_1_cnn. *The* cnn.ipynb *notebook provides a set of code examples to implement neural networks with PyTorch.*

EXERCISE

3-17: Use Prewitt or Sobel filters in the convolutional layers to construct a classifier capable of distinguishing between images predominantly featuring horizontal or vertical lines.

Project 3A: Classifying Malaria-Infected Blood Smears

Malaria is a blood disease transmitted by mosquitoes. It's commonly diagnosed by visually examining blood smears. The use of neural networks to automatically screen samples can help improve response time, decrease the workload of experts, and ensure reproducible results. In this project, you'll train neural networks to identify malaria-infected blood cells.

Loading the Malaria Dataset

The malaria dataset was published by Sivaramakrishnan Rajaraman and coworkers in 2018 in *PeerJ* (volume 6, article number e4568) and is publicly available. The dataset consists of 27,558 cell images with equal instances of uninfected and parasitized cells. You can download and extract the images with Listing 3-9.

```
import os
from torchvision.datasets.utils import _extract_zip, download_url

dataset_path = os.path.join(".", "blood_smears_dataset")
if not os.path.exists(dataset_path):
    url = "https://data.lhncbc.nlm.nih.gov/public/Malaria/cell_images.zip"
    download_url(url, ".")
    _extract_zip("cell_images.zip", dataset_path, None)
    os.remove("cell_images.zip")
```

Listing 3-9: Downloading and extracting the malaria dataset

This code downloads the zipped images, unzips them in the *blood_smears_dataset* folder, and finally removes the ZIP file.

Visualizing Blood Smears

You can use the `ImageFolder` class to load the images into a dataset, as shown in Listing 3-10.

```
from torchvision.datasets import ImageFolder

❶ base_dir = os.path.join(dataset_path, "cell_images")
dataset = ImageFolder(base_dir)
```

Listing 3-10: Loading the images into a dataset

This code populates the `dataset` with images. The base directory ❶ comprises two distinct folders, *Parasitized* and *Uninfected*, enabling you to categorize images under these specific labels. Consequently, the images are classified into two separate classes: parasitized cell images, assigned the value `0`, and uninfected cell images, assigned the value `1`, in alignment with their alphabetical ordering.

You can now have a look at a few images to gauge the best strategy to analyze the data. Write the `plot_blood_smears()` function shown in Listing 3-11.

```
import matplotlib.pyplot as plt
import numpy as np

def plot_blood_smears(dataset, parasitized):
    """Plot blood smears."""
    fig, axs = plt.subplots(3, 6, figsize=(16, 8))
    for ax in axs.ravel():
        image, label = dataset[np.random.randint(0, len(dataset))]
```

```
        ax.imshow(image)
        ax.set_title(f"Parasitized ({label})" if label == parasitized
                     else f"Uninfected ({label})", fontsize=16)
    plt.tight_layout()
    plt.show()
```

Listing 3-11: The function to plot blood smears

Then use this function to plot examples of the images:

```
plot_blood_smears(dataset, parasitized=0)
```

This generates an image similar to that shown in Figure 3-9. The `parasitized` parameter indicates that the infected cells are labeled with 0s.

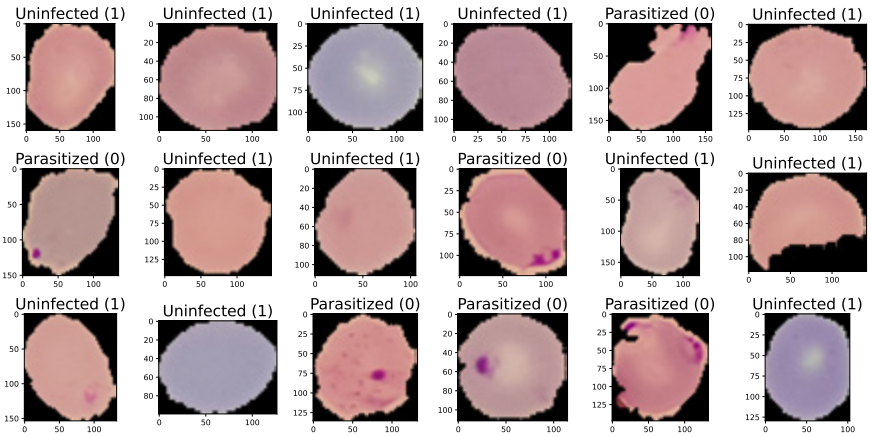


Figure 3-9: Samples of uninfected and parasitized blood-smear images

Notice that the infected cells present some spots due to the presence of the malaria plasmodium, which permits you to identify them. Also notice that the number of pixels in the images (on the order of tens of thousands of pixels) is quite large, so you'll likely need to downsample them significantly in order to be able to process them with a relatively small neural network.

Furthermore, the dimensions of the images aren't consistent; this is a problem if you want to use dense layers as part of the neural network architecture (either in a dense neural network or as the dense top of a convolutional architecture), because they require inputs of fixed length. Finally, in the medical literature, typically 1s signify the presence of a condition and 0s its absence, which is the opposite of the ground truth used in the dataset, where malaria-infected cells are denoted as 0s and uninfected cells with 1s. You'll take care of all these issues in the next section.

Transforming Images and Their Ground Truth

To take care of the identified issues, you'll need to transform the images and their labels. This can be done when creating `dataset`.

First, define a transformation to resize and normalize the images, as shown in Listing 3-12.

```
from torchvision.transforms import Compose, Resize, ToTensor

image_trans = Compose([Resize((28, 28)), ToTensor()])
```

Listing 3-12: Transformation for the images

This transformation will resize the image to 28×28 pixels and convert it to a PyTorch tensor (note that `ToTensor()` also normalizes the image values from 0 to 1).

You then need to create a transformation for the labels, as shown in Listing 3-13.

```
import torch

def label_trans(label):
    """Transform label."""
 ❶ return torch.tensor(1 - label).float().unsqueeze(-1)
```

Listing 3-13: Transformation for the labels

This defines a transformation to make the target label equal to 0 for the uninfected cells and to 1 for the parasitized ones ❶. Next, the code converts the label into a PyTorch tensor with `tensor()` and into a floating-point type with `float()`, enhancing the label's compatibility with PyTorch computational requirements. Finally, the code adds a new dimension to the tensor by using `unsqueeze(-1)`, preparing it for batch processing in neural network models.

Next, update Listing 3-10 to make these transformations, as shown in Listing 3-14.

```
--snip--
dataset = ImageFolder(base_dir, transform=image_trans,
                      target_transform=label_trans)
```

Listing 3-14: Loading the images into a dataset while transforming images and labels (by modifying Listing 3-10)

This code assigns `image_trans` (Listing 3-12) to transform the input images and `label_trans` (Listing 3-13) to transform the labels.

Now you're ready to plot the transformed images and relative labels. You can't use the `plot_blood_smears()` function directly just yet because the images returned by `dataset` are PyTorch tensors and not NumPy arrays, but to address this, you can update the function as shown in Listing 3-15.

```
def plot_blood_smears(dataset, parasitized):
    """Plot blood smears for NumPy arrays and PyTorch tensors."""
    --snip--
    for ax in axs.ravel():
        image, label = dataset[randint(0, len(dataset))]
```

```
    if isinstance(image, torch.Tensor):
        image, label = image.numpy().transpose(1, 2, 0), label.numpy()
    ax.imshow(image)
    --snip--
```

Listing 3-15: The function to plot blood smears as either NumPy arrays or PyTorch tensors (by modifying Listing 3-11)

This revised function checks whether the image is a PyTorch tensor. If so, the function transforms the tensor into a NumPy array and transposes its dimensions so that the colors are the third dimension, as is normal for RGB images, using the `transpose(1, 2, 0)` method. The updated function also transforms the label from a PyTorch tensor to a NumPy array with the `numpy()` method. You can now use this function to plot the transformed images:

```
plot_blood_smears(dataset, parasitized=1)
```

The result should be similar to Figure 3-10. The `parasitized` parameter now indicates that the infected cells are those labeled with 1.

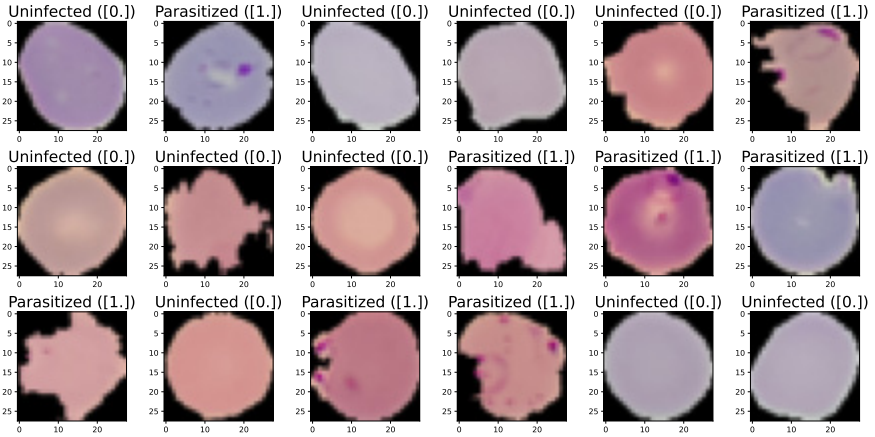


Figure 3-10: Samples of uninfected and parasitized blood-smear images after transformation

Now all the images are the same size, and the parasitized cell images are labeled as 1s and the uninfected cell images as 0s. Furthermore, you can see that the labels are NumPy arrays.

Splitting the Dataset and Defining the Data Loaders

You can now split the dataset into training (80 percent) and test (20 percent) sets with Listing 3-16.

```
train, test = torch.utils.data.random_split(dataset, [0.8, 0.2])
```

Listing 3-16: Splitting the dataset into training and test sets

Finally, define the data loaders for both sets, using Listing 3-17.

```
train_loader = torch.utils.data.DataLoader(train, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test, batch_size=256, shuffle=False)
```

Listing 3-17: Defining the data loaders

This code defines the two data loaders. The batch size for the training data loader is set to 32, while a larger batch size of 256 is used for the test set. Another difference between the data loaders stems from the fact that it's best to shuffle the training data to improve training performance, while this isn't necessary for the test data.

Classifying with Dense Neural Networks

You can now implement a dense neural network to detect the presence of malaria infection, using Listing 3-18.

```
import deeplay as dl

dnn = dl.MultiLayerPerceptron(
    in_features=28 * 28 * 3, hidden_features=[128, 128], out_features=1,
    out_activation=torch.nn.Sigmoid,
)
```

Listing 3-18: Implementing the dense neural network

This code implements a dense neural network with one input for each pixel and color channel of the input image, two layers with 128 neurons, and a single sigmoidal output. You can use `print(dnn)` to check its detailed architecture.

Next, compile this neural network, assigning it a loss and an optimizer, as shown in Listing 3-19.

```
dnn_classifier = dl.BinaryClassifier(
    model=dnn, optimizer=dl.RMSprop(lr=0.001),
).create()
```

Listing 3-19: Compiling the dense neural network

This code compiles the dense neural network as a binary classifier, using `RMSprop` as an optimizer. You can use `print(dnn_classifier)` to print out the compiled network. This printout also contains information about the loss function and the tracked metrics.

The default loss of `dl.BinaryClassifier` is a *binary cross-entropy loss*, which is particularly well suited for scenarios where the output can be interpreted as a probability (that is, assuming values from 0 to 1 that sum up to 1). Its mathematical formulation is $L_{\mathrm{BCE}} = -\left[y \log(p) + (1 - y) \log(1 - p)\right]$, where y

is the true label (0 or 1), and p is the predicted probability of the class with label 1. This loss function penalizes the predictions based on the divergence between the predicted probability and the actual label.

Furthermore, `dl.BinaryClassifier` uses *binary accuracy* as its default metric. This metric calculates the proportion of correct predictions by summing the true positives (TP) and true negatives (TN) and dividing by the total number of predictions. The formula is

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

where FP indicates false positives, and FN indicates false negatives.

Training

Next, train the dense neural network:

```
dnn_trainer = dl.Trainer(max_epochs=5, accelerator="auto")
dnn_trainer.fit(dnn_classifier, train_loader)
```

This code creates a trainer for the dense neural network with five epochs. This trainer uses error backpropagation to train the network via the `fit()` method.

Testing

Now let's test the trained dense neural network:

```
dnn_trainer.test(dnn_classifier, test_loader)
```

This code prints the final binary cross-entropy loss, which should be around 0.60, and the final binary accuracy, which should be around 0.68. This performance is better than chance (which would result in a binary accuracy of 0.50), but not very impressive. You'll greatly improve the performance in the next sections by using convolutional architectures.

Plotting the ROC Curve

The *ROC curve* is a way to evaluate the performance of a binary classifier. A binary classifier outputs a continuous value from 0 to 1, which is then converted to a binary classification by setting a *threshold* (or *cutoff*). This allows you to tune the behavior of the classifier. For example, it may be more acceptable on medical grounds to incorrectly classify an uninfected cell as parasitized than the other way around—in this case, it would be desirable to choose a lower threshold.

The ROC curve shows the relationship between the *true-positive rate (TPR)* and the *false-positive rate (FPR)* as the threshold is changed. Also known as *sensitivity*, or *recall*, the TPR is the proportion of positive samples that are

correctly classified. The FPR, or *fallout*, is the proportion of negative samples that are incorrectly classified. The ROC curve is constructed by plotting TPR against FPR as the threshold is changed from 0 to 1. The *AUROC*, or area under the ROC curve, is a measure of classifier quality, with 0.5 being equivalent to a random classifier and 1 being a perfect classifier.

NOTE *In case you're curious, the* receiver operating characteristic (ROC) *was developed during World War II for the analysis of radar signals. It was initially used to distinguish between signals (such as enemy aircraft) and noise (like birds or clouds) in radar technology. The ROC curve was later adopted in various fields, particularly in medicine and machine learning, to evaluate the performance of diagnostic tests and classification models by plotting the TPR against the FPR at various threshold settings.*

You can compute and plot the ROC curve with the `plot_roc()` function shown in Listing 3-20.

```
import torchmetrics as tm

def plot_roc(classifier, loader):
    """Plot ROC curve."""
 ❶ roc = tm.ROC(task="binary")
    for image, label in loader:
     ❷ roc.update(classifier(image), label.long())

 ❸ fig, ax = roc.plot(score=True)
    ax.grid(False)
    ax.axis("square")
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.legend(loc="center right")
    plt.show()
```

Listing 3-20: The function to calculate and plot the ROC curve of a classifier

This function uses the `ROC` class from TorchMetrics ❶ to calculate and plot the ROC curve. The labels are converted to the long format ❷, and the plot will include the AUROC value ❸.

You can then use this function to plot the ROC curve for the trained `dnn_classifier`:

```
plot_roc(dnn_classifier, test_loader)
```

The result is shown in Figure 3-11.

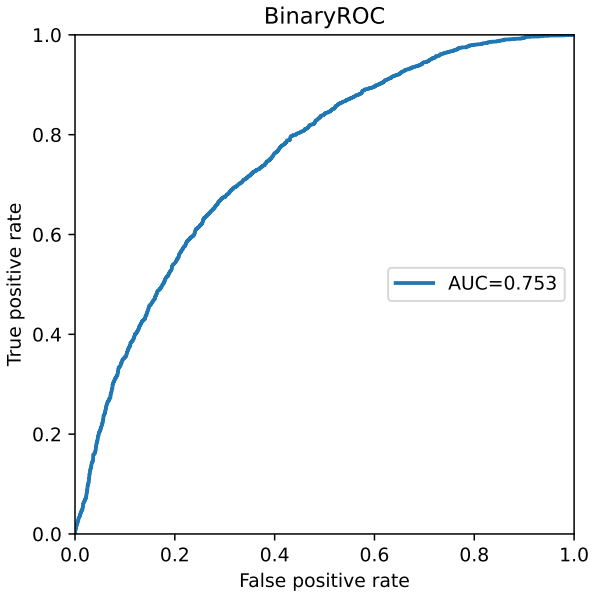


Figure 3-11: The ROC curve for the classifier, using the dense neural network

You should get an AUROC of around 0.75, indicating a decent but not exceptional ability to distinguish between the uninfected and parasitized cells. For example, Figure 3-11 shows that you can correctly identify 90 percent of parasitized cells, as long as you are willing to accept mislabeling about 60 percent of the uninfected ones as false positives—probably not as good as you'd like for medical tests.

Classifying with Convolutional Neural Networks

You can improve the performance of the classifier by using a convolutional neural network with a dense top, as implemented by Listing 3-21.

```
conv_base = dl.ConvolutionalNeuralNetwork(
    in_channels=3, hidden_channels=[16, 16, 32], out_channels=32,
)
❶ conv_base.blocks[2].pool.configure(torch.nn.MaxPool2d, kernel_size=2)

connector = dl.Layer(torch.nn.AdaptiveAvgPool2d, output_size=1)

dense_top = dl.MultiLayerPerceptron(
    in_features=32, hidden_features=[], out_features=1,
    out_activation=torch.nn.Sigmoid,
)

cnn = dl.Sequential(conv_base, connector, dense_top)
```

Listing 3-21: Implementing a convolutional neural network with a dense top

This code creates a convolutional base (`conv_base`) with four layers of 16, 16, 32, and 32 filters. A max pooling layer is added between the second and third layers, with a kernel size of 2×2 (and implicitly a stride of 2 in both directions) ❶. The `connector` layer is created by using an average pooling layer that downsamples each one of the output filters to a single number. The code creates a dense top (`dense_top`) with 32 neurons and a single sigmoidal output. Finally, all these components are combined into a single convolutional neural network with a dense top (`cnn`).

You can visualize the overall architecture via `print(cnn)`. In this architecture, the convolutional layers extract important features from the images, which can then be processed more effectively by the dense top.

Next, compile this neural network with Listing 3-22.

```
cnn_classifier = dl.BinaryClassifier(
    model=cnn, optimizer=dl.RMSprop(lr=0.001),
).create()
```

Listing 3-22: Compiling the convolutional neural network with a dense top

You are now ready to train your dense neural network.

Training

Train the convolutional neural network:

```
cnn_trainer = dl.Trainer(max_epochs=5, accelerator="auto")
cnn_trainer.fit(cnn_classifier, train_loader)
```

This code creates a trainer for the convolutional neural network, then calls the `fit()` method of the trainer to train the network.

Testing

Let's test the convolutional neural network:

```
cnn_trainer.test(cnn_classifier, test_loader)
```

This code prints the final binary cross-entropy loss, which should be around 0.12, and the final binary accuracy, which should be around 0.95. These results are much better than those with the dense neural network and, in absolute terms, are pretty good values for a binary classifier.

Plotting the ROC Curve

Now compute and display the ROC curve of this improved classifier:

```
plot_roc(cnn_classifier, test_loader)
```

You should see a ROC curve similar to that shown in Figure 3-12.

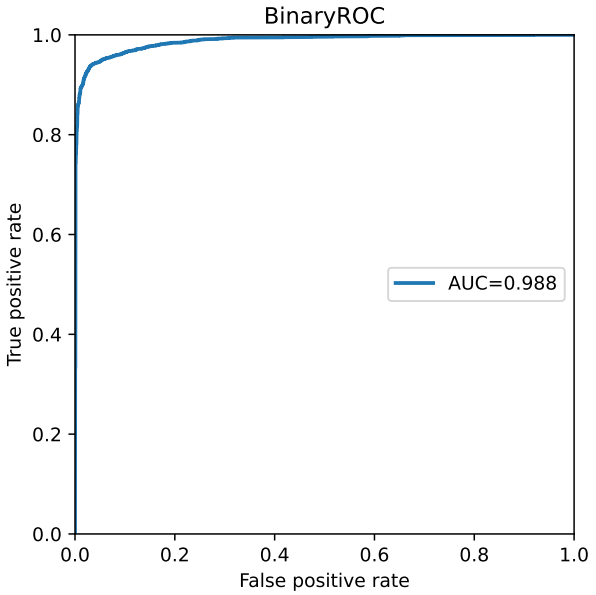


Figure 3-12: The ROC curve for the classifier, using the convolutional neural network with a dense top

This curve has an AUROC of about 0.99, which means that this is an excellent model capable of separating with high confidence the positive and negative classes.

Checking the Values of the Filters

Now that you've successfully trained a convolutional neural network to identify the malaria-infected cells, you might be curious to check the values of the filters that have been optimized in the training process. You can extract a filter as follows:

```
filter = cnn_classifier.model[0].blocks[0].layer.weight[15]
```

This accesses the model's first module, selects the initial block corresponding to the first convolutional layer, and selects the 16th filter. Then you can use `print(filter)` to print the filter values:

```
tensor([[[ 0.1500, -0.2196,  0.0505],
         [-0.0819,  0.0980,  0.0494],
         [ 0.0911, -0.0277, -0.1928]],

        [[-0.2429,  0.0118, -0.1666],
         [-0.0016,  0.2690,  0.1233],
         [ 0.1608,  0.2330, -0.1441]],
```

```
        [[-0.1029,  0.0322, -0.0945],
         [-0.0732,  0.1109,  0.1606],
         [-0.0897,  0.0135, -0.2524]]], grad_fn=<SelectBackward0>)
```

This is a PyTorch tensor with dimensions 3 (number of color channels) by 3 (filter height) by 3 (filter width), just as you saw in Figure 3-3. Note that this tensor isn't detached from the computational graph, as indicated by `grad_fn=<SelectBackward0>`.

Printing out the values of the filters alone may not provide much clarity, as interpreting their specific functions can be challenging. Instead, it's more insightful to observe the activations associated with each filter when an image is processed through the neural network, which you'll do next.

Visualizing Activations of Convolutional Layers

Start by selecting an image from the dataset, using Listing 3-23.

```
from PIL import Image

im_ind = 0
image_filename = dataset.samples[im_ind][0]
image_hr = Image.open(image_filename)
image = image_trans(image_hr)
```

Listing 3-23: Loading an image from the dataset

This code retrieves the original image with index `im_ind` and applies the same transformation used in preprocessing (corresponding to Listing 3-12).

You can then verify whether this is an image of parasitized cells with `print(label_trans(dataset.targets[im_ind]))`, which will print `tensor([0.])` if the cells in the image are uninfected, or `tensor([1.])` if they are parasitized.

To gain insights into the inner workings of a neural network, you can attach hooks to a specific layer of the network. *Hooks* are functions that can be set to execute at certain points during the forward or backward pass of the network. For example, you can use Listing 3-24 to access the activations in the forward pass of the neural network.

```
def hook_func(layer, input, output):
    """Hook for activations."""
 ❶ activations = output.detach().clone()
 ❷ print(f"Activations size: {activations.size()}")

layer = cnn_classifier.model[0].blocks[0].layer
❸ layer.register_forward_hook(hook_func)

❹ pred = cnn_classifier.model(image.unsqueeze(0))
```

Listing 3-24: Adding a hook to access the activations in the forward pass

This code defines the `hook_func()` function. Such a hook function takes three parameters: the layer itself, the input to the layer, and the output from the layer. The code then selects a layer to which it registers the hook ❸. When the network processes data, the hook is triggered during the forward pass ❹. This allows you to execute a custom function at that point, enabling you to observe or manipulate the activations. In this case, the hook will first detach from the current computation graph and clone the activations ❶ and then print the activation size ❷.

NOTE *Beyond visualizing activations, hooks can help significantly in debugging and analyzing neural networks.*

After utilizing the hook for its intended purpose, it's good practice to remove it from the network. This step is important because hooks, if left attached, can continue to execute on every forward pass, potentially leading to unintended side effects or performance issues. You can do this by modifying Listing 3-24 as shown in Listing 3-25.

```
--snip--
handle_hook = layer.register_forward_hook(hook_func)

try:
    pred = cnn_classifier.model(image.unsqueeze(0))
except Exception as e:
    print(f"An error occurred during model prediction: {e}")
finally:
 ❶ handle_hook.remove()
```

Listing 3-25: Adding and removing a hook in the forward pass (by modifying Listing 3-24)

This code saves the `handle_hook` reference to the hook handle returned when the code originally registers the hook. After the forward pass, the code uses this handle to remove the hook ❶. Furthermore, the evaluation of the neural network is enclosed inside a try-except construct to ensure that the hook removal is always executed, even if some errors occur during evaluation.

Next, write the `plot_activations()` function to visualize the activations, as shown in Listing 3-26.

```
def plot_activations(activations, cols=8):
    """Visualize activations."""
    rows = -(activations.shape[0] // -cols)

    fig, axs = plt.subplots(rows, cols, figsize=(2 * cols, 2 * rows))
    for i, ax in enumerate(axs.ravel()):
        ax.axis("off")
        if i < activations.shape[0]:
            ax.imshow(activations[i].numpy())
            ax.set_title(i, fontsize=16)
    plt.show()
```

Listing 3-26: The function to visualize the activations

You can now use this function within the hook, updating Listing 3-25 as shown in Listing 3-27.

```
def hook_func(layer, input, output):
    """Hook to plot activations."""
    activations = output.detach().clone()
    plot_activations(activations[0])
--snip--
```

Listing 3-27: Visualizing activations in the hook in the forward pass (by modifying Listing 3-25)

When passing `activations` to the `plot_activations()` function, this code uses `[0]` to select the first element of the `activations` tensor, representing the activations for the first image in the batch processed by the neural network.

Finally, make the last modification to Listing 3-27 to plot the activations of all convolutional layers, as shown in Listing 3-28.

```
--snip--
for block in cnn_classifier.model[0].blocks:
    layer = block.layer
    --snip--
```

Listing 3-28: Plotting the activations of all convolutional layers (by modifying Listing 3-27)

This final version of the code uses a `for` loop to plot the activations for each convolutional layer.

Figure 3-13 shows the activations of the first layer.

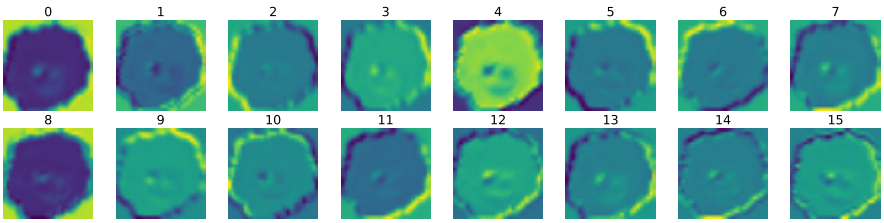


Figure 3-13: Activations of the first convolutional layer

These activations capture basic features, such as edges and simple textures, as is typical for the activations of the first layer of a convolutional neural network.

Figure 3-14 shows the fourth, and final, convolutional layer's activations.

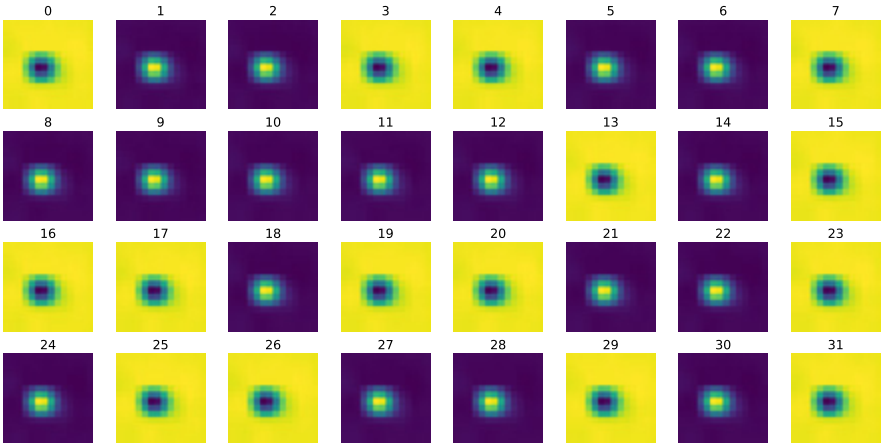


Figure 3-14: Activations of the fourth (last) convolutional layer

This last layer typically captures high-level content, representing full objects or specific parts that are relevant for making predictions. In fact, you can clearly see that the activations highlight the presence of a malaria plasmodium within the image, which then permits the dense top to classify the imaged cells as parasitized. You can also see that all the activations appear quite similar, suggesting that the convolutional network could likely be reduced in size—by decreasing the number of layers and features—without sacrificing performance.

By examining the progression from simple to complex feature detection across the layers, you gain valuable insights into how neural networks build a hierarchy of features, starting from basic edges and textures to more-complex patterns. By recognizing which types of features are extracted at each layer, you can make more-informed decisions about network architecture, such as the number of layers or their types, to better capture the relevant features for your specific task. These insights also aid in diagnosing and improving network performance, as you can pinpoint where the network might be failing to capture essential features or where it's focusing on irrelevant details.

Visualizing Heatmaps

Heatmaps give you a visual representation of which parts of the inputs are firing up your convolutional neural network, particularly in the later layers. They're like X-rays that show you what the model considers important in an image. In this section, you'll use an approach known as *Grad-CAM*, which stands for *gradient-weighted class activation mapping*, to generate these heatmaps.

First, you need to store the activations and the gradients corresponding to the layer for which you want to see the heatmap. You can do this by using hooks for both the forward and backward passes of the neural network, as shown in Listing 3-29.

```
hookdata = {}

def fwd_hook_func(layer, input, output):
    """Forward hook function."""
 ❶ hookdata["activations"] = output.detach().clone()

def bwd_hook_func(layer, grad_input, grad_output):
    """Backward hook function."""
 ❷ hookdata["gradients"] = grad_output[0].detach().clone()

layer = cnn_classifier.model[0].blocks[3].layer
handle_fwd_hook = layer.register_forward_hook(fwd_hook_func)
handle_bwd_hook = layer.register_full_backward_hook(bwd_hook_func)

try:
    pred = cnn_classifier.model(image.unsqueeze(0))
 ❸ pred.sum().backward()
except Exception as e:
    print(f"An error occurred during model prediction: {e}")
finally:
    handle_fwd_hook.remove()
    handle_bwd_hook.remove()
```

Listing 3-29: Implementing hooks for forward and backward passes

This code stores the activations and gradients from the last convolutional layer of the neural network. You initialize the `hookdata` dictionary to store the activations ❶ and gradients ❷ captured by the hooks so that you'll also be able to use them outside the hook functions themselves.

The `fwd_hook_func()` function is called during the forward pass to store the activations ❶. These activations are detached from the current computation graph and cloned to ensure that they are preserved as they are at the time of the forward pass.

The `bwd_hook_func()` function is another hook function called during the backward pass. This function captures the gradients flowing back through the layer immediately after the loss is calculated and the backward pass is initiated. The first gradient tensor `grad_output[0]` is detached and cloned to the dictionary ❷.

The two hooks are registered to the final convolutional layer, saving the corresponding `handle_fwd_hook` and `handle_bwd_hook` handles. The prediction obtained by the forward pass is used to call the `backward()` method ❸, initiating the backward pass and triggering the backward hook. Here, the code uses `pred.sum().backward()` instead of `pred.backward()` to combine all outputs into a single scalar before computing gradients, which allows PyTorch to perform the backward pass without specifying a custom gradient vector. After the forward and backward passes are complete, both the forward and backward hooks are removed.

Now you can combine activations and gradients to calculate the heatmap, as shown in Listing 3-30.

```
from torch.nn.functional import relu

activations = hookdata["activations"][0]
gradients = hookdata["gradients"][0]

pooled_gradients = gradients.mean(dim=[1, 2], keepdim=True)
❶ heatmap = relu((pooled_gradients * activations).sum(0)).detach().numpy()
```

Listing 3-30: Calculating the heatmap

This code pools the gradients via the `mean()` function, which averages them across the spatial dimensions (`dim=[1, 2]`). The use of `keepdim=True` results in a pooled gradient with the same number of channels as the original output, but with a spatial dimension of 1×1.

The heatmap is then calculated by multiplying these pooled gradients with the activations (`pooled_gradients * activations`) ❶. This operation is intended to weigh the activations by how much each channel contributed to the increase in the output. The `sum(0)` function aggregates these weighted activations across all channels, resulting in a single 2D heatmap.

Finally, the `relu()` function is applied to the aggregated heatmap to zero out any negative values, because you're interested in only the features that have a positive influence on the target class. Negative values would indicate pixels that decrease the output for the target class, which isn't useful for the heatmap. The heatmap is then detached from the computation graph and converted into a NumPy array for visualization purposes.

Visualize the heatmap with Listing 3-31.

```
from numpy import array
from skimage.exposure import rescale_intensity
from skimage.transform import resize

rescaled_image = rescale_intensity(array(image_hr), out_range=(0, 1))
resized_heatmap = resize(heatmap, rescaled_image.shape, order=2)
rescaled_heatmap = rescale_intensity(resized_heatmap, out_range=(0.25, 1))

plt.figure(figsize=(12, 5))

plt.subplot(1, 3, 1)
❶ plt.imshow(rescaled_image, interpolation="bilinear")
plt.title("Original image", fontsize=16)
plt.axis("off")

plt.subplot(1, 3, 2)
❷ plt.imshow(rescaled_heatmap.mean(axis=-1), interpolation="bilinear")
plt.title("Heatmap with Grad-CAM", fontsize=16)
plt.axis("off")
```

```
plt.subplot(1, 3, 3)
❸ plt.imshow(rescaled_image * rescaled_heatmap)
plt.title("Overlay", fontsize=16)
plt.axis("off")

plt.show()
```

Listing 3-31: Plotting a heatmap

This code rescales the image's intensity as well as the heatmap's size and intensity to match them. Then the code plots the image ❶, the heatmap ❷, and their overlay ❸. Figure 3-15 shows the resulting plots.

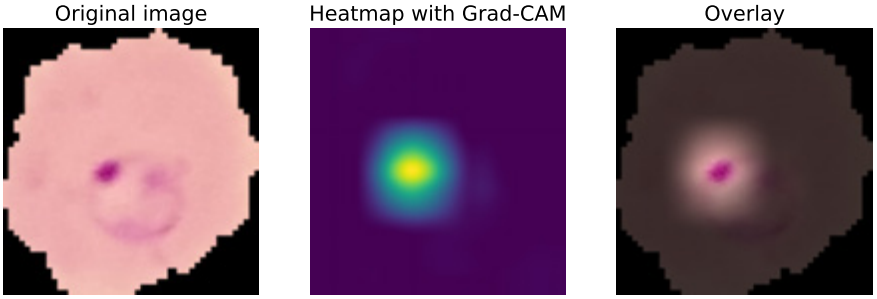


Figure 3-15: Input image (left), heatmap obtained with Grad-CAM (center), and overlay (right)

You can see how the neural network identifies the plasmodium present in the parasitized cell and then uses this as key information to classify the cell image. The original image on the left presents the cell as viewed under a microscope, with the features of interest being relatively indistinct to the unaided eye. The Grad-CAM image in the middle provides a heatmap that highlights the specific regions the neural network is focusing on, with warmer colors indicating areas of higher importance for the model's predictions. Here, the bright spot pinpoints the location of the plasmodium. The overlay image on the right combines these two, superimposing the heatmap onto the original image, thereby illustrating exactly where the plasmodium is situated within the cell.

And there you have it: a window into your model's mind. By analyzing these heatmaps, you can get a sense of the features your model is focusing on to make its decisions. This can be incredibly useful for debugging and understanding your neural network's behavior. For instance, if the heatmap highlights an area devoid of relevant features, like the corner of an image where only background is present rather than a plasmodium, this could indicate that the model is learning to focus on noise or artifacts rather than the meaningful patterns necessary for accurate predictions. Such a discrepancy would suggest that further data preprocessing is needed to remove noise, or that the model's architecture or training process requires adjustments to ensure that it learns to prioritize biologically relevant signals.

NOTE *Code Example 3-A, "Classifying Blood Smears with a Convolutional Neural Network," is available at* https://github.com/DeepTrackAI/DeepLearningCrashCourse. *Navigate to the* Ch03_CNN *folder and then* ec03_A_blood_smears. *The* blood_smears.ipynb *notebook provides the complete code example that loads the malaria dataset, trains a convolutional neural network with a dense top to classify the images of cells with and without malaria, analyzes where the trained network fails, and shows the network activations and heatmaps.*

Project 3B: Localizing Microscopic Particles

Determining the position of particles within an image is a fundamental task for microscopy. In this project, you'll build a neural network to determine the position of an optically trapped microparticle in a video.

Loading the Videos

Start by downloading the particle videos, using Listing 3-32.

```
import os

if not os.path.exists("particle_dataset"):
    os.system("git clone https://github.com/DeepTrackAI/particle_dataset")
```

Listing 3-32: Downloading the videos of an optically trapped particle

This code downloads the particle dataset into *particle_dataset*. This folder contains two videos with an optically trapped particle. One video is acquired with very low noise (*low_noise.avi*), and the other with very high noise (*high_noise.avi*). In both cases, an optically trapped microscopic particle jiggles around the center of the frame because of Brownian motion. This dataset was published in 2019 by Saga Helgadottir and co-workers in *Optica* (volume 6, pages 506–513).

Next, implement the `load_video()` function in Listing 3-33.

```
import cv2
import numpy as np

def load_video(path, frames_to_load, image_size):
    """Load video."""
    video = cv2.VideoCapture(path)

    data = []
    for _ in range(frames_to_load):
        _, frame = video.read()
        frame = cv2.normalize(frame, None, 0, 255, cv2.NORM_MINMAX)
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) / 255
        frame = cv2.resize(frame, (image_size, image_size))
```

```
            data.append(frame)

    return np.array(data)
```

Listing 3-33: The function to load a video

This function takes as input the path to the video file, loads the video, and returns a NumPy array containing the extracted frames. After loading the video, the function returns `frames_to_load` frames. The code extracts each `frame`, normalizes its pixel values to the range 0 to 255, converts it to grayscale and normalizes the pixel values in the range 0 to 1, resizes it to `image_size` by `image_size` pixels, and stores the frame in the `data` list. Finally, the function returns the list of frames as a NumPy array.

Now use this function to load the first 100 frames for each video with Listing 3-34.

```
image_size = 51
video_low_noise = \
    load_video(os.path.join("particle_dataset", "low_noise.avi"),
               frames_to_load=100, image_size=image_size)
video_high_noise = \
    load_video(os.path.join("particle_dataset", "high_noise.avi"),
               frames_to_load=100, image_size=image_size)
```

Listing 3-34: Loading the first 100 frames of each video

Let's have a look at some of the frames, using Listing 3-35.

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 6, figsize=(24, 8))
for i in range(6):
    axs[0, i].imshow(video_low_noise[i], cmap="gray", vmin=0, vmax=1)
    axs[0, i].text(0, 5, f"Frame {i}", color="white", fontsize=24)
    axs[0, i].axis("off")

    axs[1, i].imshow(video_high_noise[i], cmap="gray", vmin=0, vmax=1)
    axs[1, i].text(0, 5, f"Frame {i}", color="white", fontsize=24)
    axs[1, i].axis("off")
plt.subplots_adjust(wspace=0.1, hspace=0.1)
plt.show()
```

Listing 3-35: Plotting the first six frames of each video

This code plots the first six frames of each video, as shown in Figure 3-16.

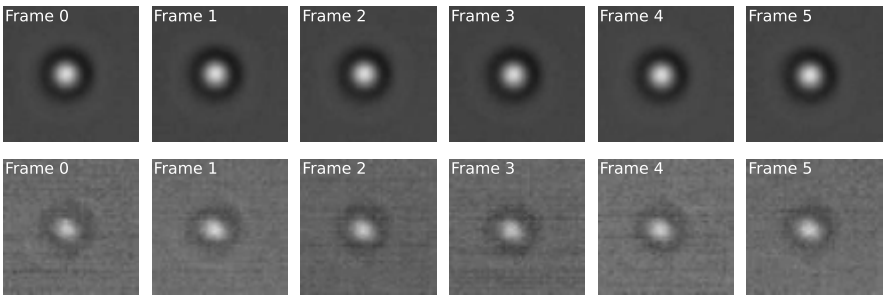


Figure 3-16: The first six frames of the low-noise (top) and high-noise (bottom) videos

In the top row, the frames are captured with high-quality illumination, so it's easy to identify the particle and locate its center. The particle is always close to the center of the frame because of the optical trap, but not exactly in the same position because it's continually shaken by the Brownian motion. In the bottom row, the same particle is captured with poor illumination. Now, it isn't so easy anymore to accurately locate the particle position, as you'll experience when trying to manually annotate this data in the next section.

Manually Annotating the Videos

To train a neural network, you need to know the ground truth. In this case, for each video frame, you need the corresponding particle position. A common way to get particle positions that can be used as a ground truth is to manually annotate some of the data and then use it for training and testing.

You can use the `ManualAnnotation` class shown in Listing 3-36 to do this.

```
from matplotlib.widgets import Cursor

class ManualAnnotation:
    """Graphical interface for manual annotation."""

    def __init__(self, images):
        """Initialize manual annotation."""
        self.images, self.positions, self.i = images, [], 0
        self.fig, self.ax = plt.subplots(1, 1, figsize=(5, 5))
        self.fig.canvas.header_visible = False
        self.fig.canvas.footer_visible = False

    def start(self):
        """Start manual annotation."""
        self.im = self.ax.imshow(self.images[self.i], cmap="gray",
                                 vmin=0, vmax=1)
        self.text = self.ax.text(3, 5,
                                 f"Frame {self.i + 1} of {len(self.images)}",
                                 color="white", fontsize=12)
        self.ax.axis("off")
```

```
            self.cursor = Cursor(self.ax, useblit=True, color="red", linewidth=1)
            self.cid = self.fig.canvas.mpl_connect("button_press_event",
                                                   self.onclick)
            self.next_image()
            plt.show()

        def next_image(self):
            """Get next image."""
            self.im.set_data(self.images[self.i])
            self.text.set_text(f"Frame {self.i + 1} of {len(self.images)}")
            self.fig.canvas.draw_idle()

        def onclick(self, event):
            """Save position on click."""
            self.positions.append([event.xdata, event.ydata])
            if self.i < len(self.images) - 1:
                self.i += 1
                self.next_image()
            else:
                self.fig.canvas.mpl_disconnect(self.cid)
                plt.close()
                return
```

Listing 3-36: The class to manually annotate the video frames with the particle position

This class creates an object that uses the Matplotlib library to create a simple graphical user interface (GUI) that allows you to record the position of the particle center by clicking it. The `__init__()` method initializes the GUI, setting up the figure and axes and preparing the image sequence for display. The `start()` method is where the actual GUI is displayed; it shows the first image and sets up the cursor for annotation. The `next_image()` method updates the display with the next frame in the sequence. The `onclick()` method captures the user's clicks, recording the position of the particle in each frame. This method also handles the progression to the next frame and concludes the annotation process when all frames have been processed.

You can use this class to annotate a subset of images, combining both low-noise and high-noise ones, as shown in Listing 3-37.

```
❶ %matplotlib ipympl

number_of_images_to_annotate = 100

dataset = np.concatenate([video_low_noise, video_high_noise], axis=0)
np.random.shuffle(dataset)
images_to_annotate = np.random.choice(
    np.arange(dataset.shape[0]), number_of_images_to_annotate, replace=False,
)

manual_annotation = ManualAnnotation(dataset[images_to_annotate])
```

```
manual_annotation.start()
annotated_images = manual_annotation.images
manual_positions = manual_annotation.positions

❷ %matplotlib inline
```

Listing 3-37: Manually annotating some video frames with the particle positions

This demonstrates the process of using the `ManualAnnotation` class for annotating a series of images with particle positions. Initially, the script sets up an interactive Matplotlib environment suitable for notebooks ❶. The script prepares a dataset for annotation by combining low-noise and high-noise video datasets and shuffling them to ensure variability. Then the script randomly selects a subset of `number_of_images_to_annotate` images from this combined dataset for annotation.

Subsequently, the script instantiates the `ManualAnnotation` object with the selected images and starts the annotation process. This opens a GUI for the user to manually annotate the particle positions in each image, shown in Figure 3-17.

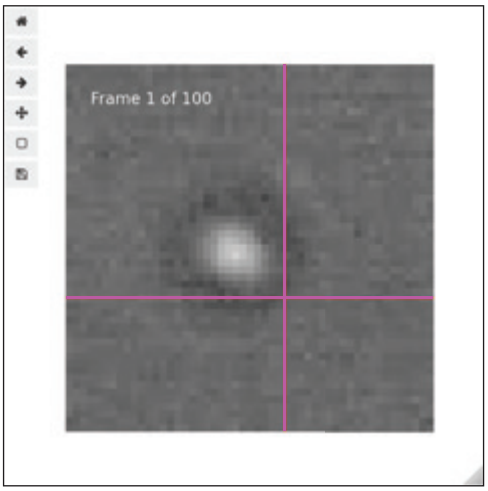


Figure 3-17: The GUI for manually annotating the video frames with the particle positions

After completing the annotation, the script retrieves the annotated images (`annotated_images`) and the corresponding manually recorded positions (`manual_positions`). Finally, the script switches back to the normal Matplotlib backend ❷, concluding the interactive session and returning to the standard plotting environment.

Save the annotated images and relative positions into files, using Listing 3-38.

```
if not os.path.exists("annotated_images.npy"):
    np.save("annotated_images.npy", np.array(annotated_images))
if not os.path.exists("manual_positions.npy"):
    np.save("manual_positions.npy", np.array(manual_positions))
```

Listing 3-38: Saving the annotated images and relative positions

This script saves the data as NumPy arrays into the *annotated_images.npy* and *manual_positions.npy* files, but only if these files don't already exist.

Now you need to define the data loaders to use the data in the training. To do this, you first need to define a custom dataset, as shown in Listing 3-39.

```
import torch

class AnnotatedDataset(torch.utils.data.Dataset):
    """Manually annotated dataset."""

    def __init__(self, file_images, file_positions):
        """Load annotated images and manual positions."""
        self.images = np.load(file_images)
        self.positions = np.load(file_positions)

    def __len__(self):
        """Return number of images."""
        return self.images.shape[0]

    def __getitem__(self, idx):
        """Get next image and annotated position."""
     ❶ im = torch.tensor(self.images[idx, np.newaxis, :, :]).float()
     ❷ pos = torch.tensor(self.positions[idx] / im.shape[-1] - 0.5).float()
        return [im, pos]
```

Listing 3-39: Implementing a custom dataset for the manually annotated particle data

This class inherits from the PyTorch `Dataset` class. It contains the particles' images and positions loaded from the files saved during the annotation process. When a new item is requested, the `__getitem__()` method of this class converts the image into a PyTorch tensor with an added new axis for the batch dimension and data type set to `float` for computational efficiency ❶. The `__getitem__()` method then normalizes the position by dividing it by the image width (`im.shape[-1]`), offsets the position by 0.5 to refer it to the center of the image (and not to its lower-left corner), and converts the position into a float PyTorch tensor ❷. Finally, the class combines the processed image tensor and the position tensor into a list to be returned.

Now that you've defined a custom dataset, create an instance of it

```
ann_dataset = AnnotatedDataset(file_images="annotated_images.npy",
                               file_positions="manual_positions.npy")
```

and use it to create the data loaders:

```
import deeplay as dl

train_ann_dataset, test_ann_dataset = \
    torch.utils.data.random_split(ann_dataset, [0.8, 0.2])

train_ann_dataloader = dl.DataLoader(train_ann_dataset, batch_size=1)
test_ann_dataloader = dl.DataLoader(test_ann_dataset, batch_size=1)
```

This code splits the annotated data into training and test sets so that 80 percent of the data is in the training set. Then the code creates a training data loader and a testing data loader.

Implementing a Convolutional Neural Network

Now you're ready to implement the neural network for classification with Listing 3-40.

```
cnn = dl.Sequential(
    dl.ConvolutionalNeuralNetwork(
        in_channels=1, hidden_channels=[16, 32], out_channels=64,
        pool=torch.nn.MaxPool2d(kernel_size=2), out_activation=torch.nn.ReLU,
    ),
    dl.Layer(torch.nn.MaxPool2d, kernel_size=2),
    dl.Layer(torch.nn.Flatten),
    dl.MultiLayerPerceptron(
        in_features=6 * 6 * 64, hidden_features=[32, 32], out_features=2,
        out_activation=torch.nn.Identity,
    ),
)
```

Listing 3-40: Implementing the convolutional neural network with a dense top

This code implements a convolutional neural network with a dense top that accepts as input an image of the particle and returns as output its predicted x- and y-coordinates. Before proceeding further, print out this neural network with `print(cnn)` and explore its details; it's indeed good practice to verify in the printout that the various layers and their properties match the way you set them up.

Next, compile the neural network with Listing 3-41.

```
from torchmetrics import MeanAbsoluteError as MAE

❶ cnn_regressor_template = dl.Regressor(
      model=cnn, loss=torch.nn.MSELoss(), optimizer=dl.Adam(), metrics=[MAE()],
  )
❷ cnn_ann_regressor = cnn_regressor_template.create()
```

Listing 3-41: Compiling the neural network

This code creates a template to use the neural network as a regressor ❶, setting up its loss function (`torch.nn.MSELoss()`), its optimizer (`dl.Adam()`), and a metric to be tracked (`MAE()`). Then the code creates a concrete instance of this template ❷. You can check the structure of this regressor with `print(cnn_ann_regressor)` and verify that its architecture and properties are as set up.

Training with Annotated Data

To train the neural network with the manually annotated data, use

```
cnn_ann_trainer = dl.Trainer(max_epochs=50, accelerator="auto")
cnn_ann_trainer.fit(cnn_ann_regressor, train_ann_dataloader)
```

which trains the neural network for 50 epochs with the manually annotated data.

Testing the Trained Neural Network

Next, evaluate the trained neural network performance:

```
test_ann_results = cnn_ann_trainer.test(cnn_ann_regressor, test_ann_dataloader)
MAE_ann = test_ann_results[0]["testMeanAbsoluteError_epoch"] * image_size
print(f"Mean pixel error (MAE): {MAE_ann:.3f} pixels")
```

This code tests the neural network against the test annotated data, then retrieves the MAE from the tracked metrics. This error is transformed from normalized units into pixels by multiplying it by `image_size`, and the result is printed. Depending on the consistency of your annotations, you might observe mean pixel errors varying from 0.2 to 2 pixels, indicating a substantial margin of error.

NOTE *The observed error in this context relates to the* precision *of your manual annotations.* Precision *here refers to the consistency or repeatability of your annotations. Higher error values indicate lower precision, suggesting that repeating annotations on the same image may yield significantly different positions. This concept of precision is distinct from the accuracy of the manual annotation process.* Accuracy *concerns how closely your annotations align with the actual ground-truth values. However, without access to these ground-truth values, it isn't possible to directly assess the accuracy of your manual annotations. Therefore, while you can gauge the precision of your work, assessing its accuracy remains a challenge in the absence of a known standard or reference.*

Repeating the annotation process with your family and friends could be a fascinating experiment to explore the variability and reliability of manual annotations. By involving multiple annotators, you can assess how different individuals perceive and mark the same set of images. This approach offers several benefits and insights, such as understanding the range of precision among different people, identifying potential biases in annotation, and determining whether certain images are consistently difficult to annotate.

Visualizing the Predictions

Finally, you can visualize the predictions of the trained network and compare them with your manual annotations, using Listing 3-42.

```
❶ indices = np.random.choice(np.arange(len(test_ann_dataset)), 6, replace=False)
❷ images = [test_ann_dataset[index][0] for index in indices]
❸ annotations = [test_ann_dataset[index][1] for index in indices]
❹ predictions = cnn_ann_regressor(torch.stack(images))

fig, axs = plt.subplots(1, 6, figsize=(25, 8))
for ax, im, ann, pred in zip(axs, images, annotations, predictions):
    ax.imshow(im.numpy().squeeze(), cmap="gray")

    ann = ann * image_size + image_size / 2
    ax.scatter(ann[0], ann[1], marker="+", c="g", s=500, linewidth=6,
               label="Annotation")

    pred = pred.detach().numpy() * image_size + image_size / 2
    ax.scatter(pred[0], pred[1], marker="x", c="r", s=500, linewidth=4,
               label="Prediction")

    ax.set_axis_off()
ax.legend(loc=(0.5, 0.8), framealpha=1, fontsize=24)
plt.show()
```

Listing 3-42: Comparing the predictions with the manual annotations

This code chooses some random frame indices ❶. Then it extracts the relative images ❷ and annotations ❸, and predicts the positions by using the neural network trained with the manual annotations ❹. The code then generates images that should look similar to those in Figure 3-18.

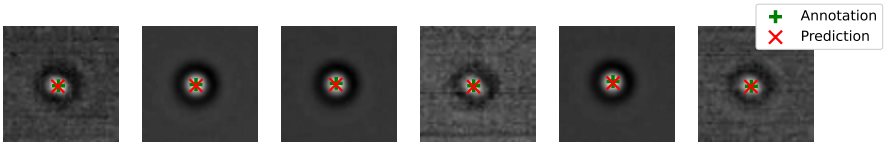


Figure 3-18: Particle images with annotated and predicted positions

The network has, in fact, learned to predict particle positions that are close to the manual annotations used for its training, as you can verify by observing that the annotation and prediction markers match.

Simulating the Training Data

You can enhance your neural network's performance by using simulated data rather than relying on manually annotated data. Manual annotations can be inconsistent because of differences in the way different people interpret data (*interobserver variability*) and because the same person may interpret data differently over time (*intraobserver variability*). Additionally, manually labeling data is often slow, effort intensive, and difficult to scale.

In contrast, using simulated datasets for training removes the inconsistencies associated with human annotations and enables the generation of substantially larger datasets. Such large datasets are vital for effectively training deep learning models.

To create simulated datasets that closely mimic experimental conditions, you need to carefully design them to incorporate a range of variable properties and realistic noise. This process requires simulating the various scenarios, conditions, and outliers that the neural network might encounter in real-world applications. You need to integrate factors such as differing lighting conditions and sensor noise to ensure that the dataset is comprehensive and challenging enough, replicating the complexity and unpredictability of real-world data as closely as possible.

Creating Particle Images

You'll use the DeepTrack2 library to generate simulated particle images, which will then be used to train your model. As you'll see, this method streamlines the training process while also potentially increasing the robustness and accuracy of the neural network.

Define the particle via the `MieSphere` class, as shown in Listing 3-43.

```
import deeptrack as dt

particle = dt.scatterers.MieSphere(
    position=(25, 25), z=0, radius=500e-9, refractive_index=1.37,
    position_unit="pixel",
)
```

Listing 3-43: Defining a microscopic particle with its physical parameters

This code defines the position of the particle in the image along the x- and y-axes, its z-coordinate, its radius in meters, its refractive index, and the units of the positions.

Next, you need to specify the optical device to image the particle. You'll use a bright-field microscope, implemented by the `Brightfield` class, as shown in Listing 3-44.

```
brightfield_microscope = dt.optics.Brightfield(
    wavelength=630e-9, NA=0.8, resolution=1e-6, magnification=15,
    refractive_index_medium=1.33, output_region=(0, 0, image_size, image_size),
)
```

Listing 3-44: Defining the optical device

This code defines the wavelength of the illuminating light in meters, the numerical aperture of the objective, the effective camera pixel size in meters, the magnification of the optical device, the refractive index of the medium where the particle is immersed, and the size of the camera sensor in pixels. These optical device parameters are defined such that they (loosely) match the experimental conditions in which the videos were acquired.

To create images of the particles, you need to combine `brightfield_microscope` with `particle`:

```
imaged_particle = brightfield_microscope(particle)
```

This code passes the particle to the bright-field microscope, creating the new `imaged_particle` pipeline that you'll use to create images of the particle. A *pipeline* is a series of steps or processes that allows for the systematic transformation, analysis, and interpretation of data.

Plot the simulated particle image with the `plot_simulated_particles()` function in Listing 3-45.

```
def plot_simulated_particles(image_pipeline):
    """Plot simulated particles."""
    fig, axs = plt.subplots(1, 6, figsize=(25, 8))
    for i, ax in enumerate(axs.flatten()):
        image = image_pipeline.update().resolve()
        ax.imshow(np.squeeze(image), cmap="gray")
        ax.set_xticks([])
        ax.set_yticks([])
    plt.show()
```

Listing 3-45: The function to plot the images of simulated particles

This function calls the `update()` method on the `image_pipeline` to get a new instance of the particle, and the `resolve()` method to resolve the simulation pipeline and to return the image of the particle. Then the function squeezes the resulting `image` to plot it.

You can plot multiple particle images obtained with this pipeline, using Listing 3-46.

```
plot_simulated_particles(imaged_particle)
```

Listing 3-46: Rendering images of the particle

Figure 3-19 shows the resulting examples of simulated particles.

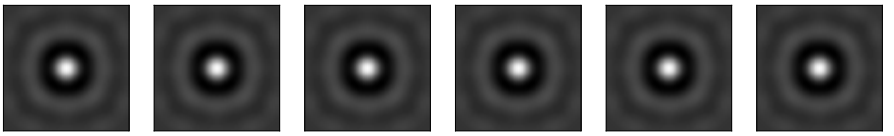


Figure 3-19: The simulated particle images

At the moment, the simulation pipeline is completely deterministic so that the particle will always have the same properties and be at the same position. This is why the six images are exactly the same. In the following section, you'll see how to make images of particles with a range of random properties.

Varying Particle Properties

To simulate images representative of the experimental dataset, you need to vary the particle's position, radius, and refractive index. You can do this by defining the particle's properties as functions that return a random value within a given range, updating Listing 3-43 as shown in Listing 3-47.

```
particle = dt.scatterers.MieSphere(
    position=lambda: np.random.uniform(image_size / 2 - 5,
                                       image_size / 2 + 5, 2),
    z=lambda: np.random.uniform(-1, -1),
    radius=lambda: np.random.uniform(500, 600) * 1e-9,
    refractive_index=lambda: np.random.uniform(1.37, 1.42),
    position_unit="pixel",
)
imaged_particle = brightfield_microscope(particle)
```

Listing 3-47: Simulating a particle with variable properties (by modifying Listing 3-43)

This code defines the particle's properties as lambda functions that return a random value within a given range. *Lambda functions* are small anonymous functions in programming that have a simple expression. The ranges are chosen to match the experimental conditions.

The position of the particle is a lambda function that returns two random values defined in the range from `image_size / 2 - 5` to `image_size / 2 + 5`, its z-coordinate is a lambda function that returns a random value drawn from a uniform distribution defined in the range from −1 to 1, its radius is a lambda function that returns a random value in the range from 500 nm to 600 nm, and its refractive index is a lambda function that returns a random value defined in the range (1.37, 1.42). The code then resets `imaged_particle` to use the new particle.

When you plot examples of the resulting particle images with Listing 3-46, you should get particle images like those shown in Figure 3-20.

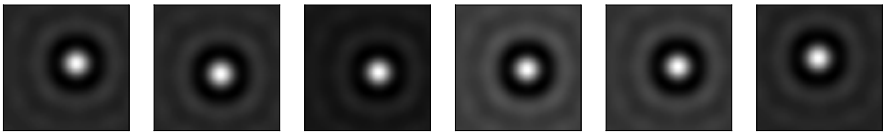


Figure 3-20: The simulated particle images with random parameters

These particle images are different each time `imaged_particle` is updated.

Generating Noisy Images

Although now the simulated images reproduce the variability encountered in experiments, they are still lacking the noise always present in experimental images. You can generate noisy images by adding a Poisson noise source to the simulation pipeline, as shown in Listing 3-48.

```
noise = dt.Poisson(
    min_snr=5, max_snr=20, background=1,
 ❶ snr=lambda min_snr, max_snr: np.random.uniform(min_snr, max_snr),
)
noisy_imaged_particle = imaged_particle >> noise

plot_simulated_particles(noisy_imaged_particle)
```

Listing 3-48: Adding noise to the simulation pipeline

This code adds noise to the simulated images of the particles, defining the minimum signal-to-noise ratio (SNR) as 5 and the maximum SNR as 20. Then a lambda function is defined to determine the SNR as a random value between the minimum and maximum SNR ❶, which is also a good example of a lambda function with two inputs. The code also defines the background intensity as 1, which is used to calculate the signal of the image. Finally, the code adds the noise source to the simulation pipeline by using the `>>` operator and plots some examples of the resulting images, shown in Figure 3-21.

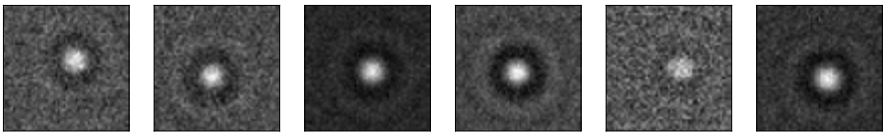


Figure 3-21: The simulated particle images with noise

Finally, to ensure that the neural network inputs are consistently scaled, facilitating more-stable and efficient learning, you should normalize the images to the range of 0 to 1 with Listing 3-49.

```
normalization = dt.NormalizeMinMax(lambda: np.random.uniform(0.0, 0.2),
                                   lambda: np.random.uniform(0.8, 1.0))
image_pipeline = noisy_imaged_particle >> normalization
```

Listing 3-49: Normalizing the simulated images

This code defines the normalization function, setting the minimum and maximum values of the normalization to random values in the range (0, 0.2) and (0.8, 1), respectively. Then the normalization function is added to the simulation pipeline via the >> operator.

Getting Particle Positions

Now you need to extract the particle's position from the simulated image to use as the ground truth when training the neural network:

```
pipeline = image_pipeline & particle.position
```

This code uses the `&` operator to create a pipeline to get both the simulated image and the particle position, which you can get as follows:

```
image, position = pipeline.update().resolve()
```

Then update the `plot_simulated_particles()` function to also plot the positions of the particles, as shown in Listing 3-50.

```
def plot_simulated_particles_with_positions(pipeline):
    """Plot simulated particles with positions."""
    --snip--
    for i, ax in enumerate(axes.flatten()):
        image, position = pipeline.update().resolve()
        ax.imshow(np.squeeze(image), cmap="gray")
        ax.scatter(position[1], position[0], s=500, facecolors="none",
                   edgecolor="g", linewidth=6)
        --snip--
```

Listing 3-50: The function to plot the images of simulated particles with their position (by modifying Listing 3-45)

Use this function to plot the simulated particles with their ground-truth positions:

```
plot_simulated_particles_with_positions(pipeline)
```

The resulting images should look like those in Figure 3-22.

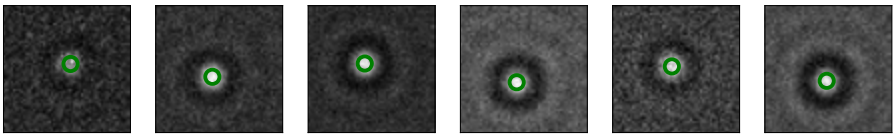


Figure 3-22: The simulated particles with ground-truth positions

As expected, the ground-truth positions (plotted as circles) are perfectly at the center of the simulated images of particles, providing the data you need to train the neural network.

Preprocessing the Simulated Data

Modify the `AnnotatedDataset` class to work with the simulation pipeline instead of the annotated data, creating the `SimulatedDataset` class in Listing 3-51.

```
class SimulatedDataset(torch.utils.data.Dataset):
    """Dataset with simulated particles."""

    def __init__(self, pipeline, data_size):
        """Initialize simulated dataset."""
        images, positions = [], []
        for _ in range(data_size):
         ❶ image, position = pipeline.update().resolve()
         ❷ images.append(image), positions.append(position[[1, 0]])
        self.images, self.positions = np.array(images), np.array(positions)

    --snip--

    def __getitem__(self, idx):
        """Get next image and annotated position."""
     ❸ im = torch.tensor(self.images[idx]).float().permute(2, 0, 1)
     ❹ pos = torch.tensor(self.positions[idx] / im.shape[-1] - 0.5).float()
        return [im, pos]
```

Listing 3-51: Implementing a custom dataset for the simulated data (by modifying Listing 3-39)

In its initialization, this updated class creates the required number of simulations ❶, from which it saves as class attributes the images themselves and the relative ground-truth positions ❷. When getting an item, the `__getitem__()` method returns the image ❸ and the relative position ❹ already prepared as float PyTorch tensors.

Use this class to create the training and test data loaders, as shown in Listing 3-52.

```
train_sim_dataloader = dl.DataLoader(
    SimulatedDataset(pipeline=pipeline, data_size=10_000), batch_size=32,
)
test_sim_dataloader = dl.DataLoader(
    SimulatedDataset(pipeline=pipeline, data_size=100), batch_size=32,
)
```

Listing 3-52: Creating the data loaders for the simulated data

This code creates a training data loader with a dataset of 10,000 images, many more than what you could manually annotate in a reasonable amount of time. For the test dataset, you can still use just 100 images.

Training with Simulated Data

Now you're all set to train the neural network with simulated data:

```
cnn_sim_regressor = cnn_regressor_template.create()
cnn_sim_trainer = dl.Trainer(max_epochs=50, accelerator="auto")
cnn_sim_trainer.fit(cnn_sim_regressor, train_sim_dataloader)
```

This code creates a new instance of `cnn_regressor_template` and a new trainer, and proceeds to train this second neural network with the simulated dataset.

Testing the Trained Neural Network

Test the trained neural network:

```
test_sim_results = cnn_sim_trainer.test(cnn_sim_regressor, test_sim_dataloader)
MAE_sim = test_sim_results[0]["testMeanAbsoluteError_epoch"] * image_size
print(f"Mean pixel error (MAE): {MAE_sim:.3f} pixels")
```

This code computes and prints the MAE of the neural network trained with simulated data, expected to be below 0.1 pixels. This represents a significant improvement in the neural network's precision compared to the model trained on manually annotated data. The improvement primarily stems from the systematic nature of the simulated data, which eliminates human-induced inconsistencies, as well as from the larger size of the dataset, which is crucial for deep learning models.

As an interesting exercise, you could analyze the impact of varying the size of the training dataset on the model's performance, gaining insights into the scalability and adaptability of the neural network to different data volumes.

Plotting the Predictions vs. the Ground Truth

To further evaluate the network's performance, you can plot the predicted particle positions versus the true particle positions, using Listing 3-53.

```
preds, gts = [], []
for image, position in iter(test_sim_dataloader):
 ❶ preds.append(cnn_sim_regressor(image))
 ❷ gts.append(position)
preds = torch.cat(preds, dim=0).detach().numpy()
gts = torch.cat(gts, dim=0).numpy()

fig, axs = plt.subplots(1, 2)
for i, ax, coordinate in zip([0, 1], axs, ["x", "y"]):
    gt, pred = gts[:][:, i], preds[:][:, i]
    ax.scatter(gt, pred, alpha=0.2)
    ax.plot([np.min(gt), np.max(gt)], [np.min(pred), np.max(pred)], c="k")
    ax.set_title(f"{coordinate}-coordinates")
    ax.set_xlabel("Prediction")
    ax.set_ylabel("Ground truth")
    ax.set_aspect("equal")
```

```
    ax.set_xlim([-0.07, 0.07])
    ax.set_ylim([-0.07, 0.07])
    ax.label_outer()
plt.show()
```

Listing 3-53: Plotting the predictions versus the ground truth

In the `for` loop, this code iterates over each item in the test data loader. For each item, the neural network predicts the particle position and appends it to the `preds` list ❶. Correspondingly, the actual ground-truth position from the data item is appended to the `gts` list ❷. After completing the loop, the lists of predictions and ground truths are concatenated into tensors, and then converted into NumPy arrays for ease of handling and visualization. Finally, they are plotted in the predictions-versus-ground-truth plots shown in Figure 3-23.

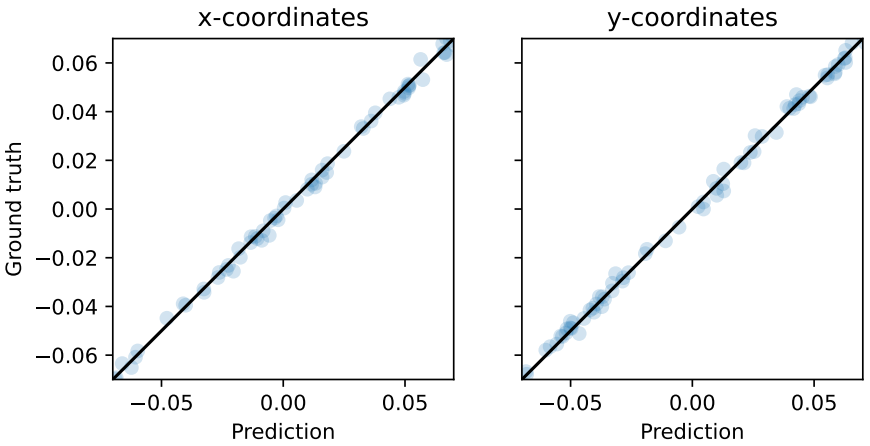


Figure 3-23: Ground-truth versus predicted positions

The predicted particle positions are in good agreement with the true particle positions, demonstrating the high quality of the training.

Comparing with the Annotated Data

Let's compare the positions obtained by the neural network trained on simulated data with your manual annotations:

```
test_ann_results_with_cnn_sim = \
    cnn_sim_trainer.test(cnn_sim_regressor, test_ann_dataloader)
MAE_ann_with_cnn_sim = (test_ann_results_with_cnn_sim[0]
                        ["testMeanAbsoluteError_epoch"] * image_size)
print(f"Mean pixel error (MAE): {MAE_ann_with_cnn_sim:.3f} pixels")
```

In this case, the MAE obtained from this test will strongly depend on your manual annotations.

As a last check, you can plot both the manual annotations and the network predictions on top of the experimental images, using Listing 3-54, which is a modification of Listing 3-42.

```
--snip--
predictions = cnn_sim_regressor(torch.stack(images))
--snip--
```

Listing 3-54: Plotting the predictions of the simulated data–trained neural network in comparison with the manual annotations (by modifying Listing 3-42)

Figure 3-24 shows the resulting images.

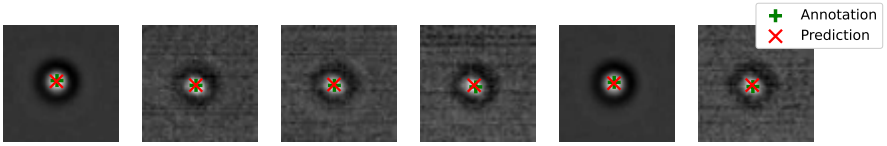


Figure 3-24: The manual annotations and predictions of the neural network trained on simulated data

The neural network predictions and the manual annotations are quite close—at least in this example, as this will strongly depend on the quality of your manual annotations.

NOTE

Code Example 3-B, "Localizing Microscopic Particles," is available at https://github.com/DeepTrackAI/DeepLearningCrashCourse. *Navigate to the* Ch03_CNN *folder and then* ec03_B_particle_localization. *The* particle_localization.ipynb *notebook provides a complete code example that trains a convolutional neural network with a dense top, using annotated and simulated data, and then applies the trained neural network to experimental videos of an optically trapped particle.*

Project 3C: Creating DeepDreams

In this project, you'll explore *DeepDreams*—a concept conceived by Google engineer Alexander Mordvintsev. The fundamental question at the heart of DeepDreams is: How can an image be transformed to maximally activate a specific layer in a trained convolutional neural network?

This approach essentially flips the neural network training process on its head. Instead of tuning the weights of the network layers for optimal performance, you'll use backpropagation and gradient ascent to morph the input image into a form that highly activates a target layer. This method gives you a unique look into what a particular layer in a neural network is most responsive to, or "sees."

The DeepDreams algorithm has been aptly named for its ability to produce dreamlike, almost hallucinogenic imagery through a process of deliberate overprocessing. Interestingly, this process was named *Inceptionism*, which is a reference to InceptionNet, the first convolutional neural network used to generate DeepDreams, and the movie *Inception*.

For this project, you'll implement the DeepDreams algorithm and apply it to transform an image. Prior to starting this project, it's essential that you

are already acquainted with the concepts introduced in Project 3A, particularly those relating to hooks and layer activations.

Loading an Image

You'll start by loading the image in the *neuraltissue_with_colorlabels.png* file. This image, sourced from the Drosophila ssTEM dataset at *https://figshare.com/articles/dataset/Segmented_anisotropic_ssTEM_dataset_of_neural_tissue/856713*, features a cross-section of neural tissue from the ventral nerve cord of a fruit fly larva (scientifically known as *Drosophila melanogaster*). Load the image with Listing 3-55.

```
from PIL import Image

image_file = "neuraltissue_with_colorlabels.png"
im = Image.open(image_file).convert("RGB").resize((256, 256))
```

Listing 3-55: Loading the image

This code uses Pillow, the common Python package for image processing, to load the image, convert it to RGB, and resize it from the original 1024×1024 pixels to 256×256 pixels for computational convenience.

Once the image is loaded, plot it by using Listing 3-56.

```
import matplotlib.pyplot as plt

plt.imshow(im)
plt.axis("off")
plt.show()
```

Listing 3-56: Plotting the image

Figure 3-25 shows the resulting image.

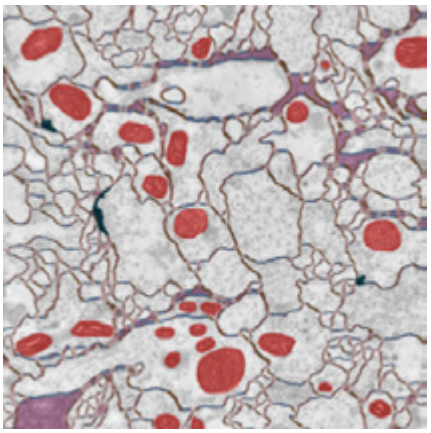


Figure 3-25: A transmission electron microscopy image of the neural tissue of a fruit fly larva

In the image, various neural structures are labeled with different shades to facilitate identification. In particular, the roundish shapes are the neuron mitochondria. Each pixel corresponds to a square with a side of length 18.4 nm (in the original image, before downsampling, it's 4.6 nm).

Loading a Pretrained Neural Network

To create DeepDreams, you need to use a pretrained network. So, you'll import the VGG16 model, a pretrained neural network known for its proficiency in image-recognition tasks, using Listing 3-57.

```
from torchvision.models import vgg16, VGG16_Weights

model = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
model.eval()
model.requires_grad_(False)
```

Listing 3-57: Loading the VGG16 pretrained neural network

This code loads the VGG16 model with weights trained on the ImageNet dataset. This large-scale, extensively annotated image database, designed for use in research on visual object recognition software, contains millions of images categorized according to a hierarchy of descriptive labels. The code then sets the model to evaluation mode with the `eval()` method and freezes all weights to prevent further changes with the `requires_grad_(False)` method.

When you print the VGG16 model with `print(model)`, you get Listing 3-58.

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ...
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ...
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ...
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
```

```
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ...
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ...
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Listing 3-58: Printout of the VGG16 model

The VGG16 model is structured hierarchically, starting with a `features` module composed of a series of convolutional (`Conv2d`) and ReLU (`ReLU`) layers. These layers progressively increase the number of feature maps, starting from 64 channels and going up to 512, with each convolutional layer using 3×3 kernel sizes and 1×1 strides, often followed by ReLU for nonlinear activation. This module also includes multiple max pooling layers (`MaxPool2d`) with a 2×2 kernel and a stride of 2, reducing the spatial dimensions after certain stages.

The `avgpool` module, which is an adaptive average pooling layer, follows the `features` module, resizing the output to a fixed size of 7×7, independent of the size of the input image—hence its adaptive nature.

Finally, the `classifier` module comprises three fully connected layers (`Linear`), each with 4,096 neurons (except the last one, which has 1,000 neurons corresponding to the classes of the ImageNet dataset), interspersed with ReLU activations and dropout layers to prevent overfitting.

When using the VGG16 model, it's crucial to normalize the input images by using the mean and standard deviation values specific to the ImageNet dataset's color channels. This ensures that the input data aligns with the data distribution the model was originally trained on. You can do this with Listing 3-59.

```
import numpy as np
import torch

mean = np.array([0.485, 0.456, 0.406], dtype=np.float32)
std = np.array([0.229, 0.224, 0.225], dtype=np.float32)
```

```
low = torch.tensor((- mean / std).reshape(1, -1, 1, 1))
high = torch.tensor(((1 - mean) / std).reshape(1, -1, 1, 1))
```

Listing 3-59: Defining the normalization parameters for VGG16

This code defines the normalization parameters. The `mean` array contains the mean values for the red, green, and blue channels, while the `std` array holds the standard deviations for these channels.

Implementing the DeepDreams Algorithm

Now that you've loaded the image you want to transform into a DeepDream and the neural network you want to use to do so, you can proceed to implement the DeepDreams algorithm.

Before actually implementing the algorithm, recall how a neural network works: When an image is fed into the network, the activations of the different layers of the neural network depend on which features they've been trained to detect. Early layers might detect simple features like edges and textures, while deeper layers recognize more-complex features like shapes or specific objects.

The DeepDreams algorithm reverses this process. Instead of using the network to detect features, the algorithm modifies the original image to amplify the features that the network detects. This is done by applying gradients to the image that transform it to increase the activation of certain layers. The core technique is *gradient ascent* in the input space. This technique involves asking the network's layers not just to detect features in an image but to enhance the features they detect. For example, if a cloud looks slightly like a bird, the DeepDreams algorithm will adjust the image to make it look even more like a bird.

The DeepDreams process is iterative. An image is put through the network, tweaked slightly to enhance its features, and then put through the network again, in a loop. With each pass through the network, the image is further modified to enhance its features—creating familiar shapes and patterns where none existed, often resulting in surreal, dreamlike images that have swirling patterns or fantastical creatures.

Using Gradient Ascent

First, you need to convert the image into a PyTorch tensor. You can do this with the `image_to_tensor()` function shown in Listing 3-60.

```
import torchvision.transforms as tt

def image_to_tensor(im, mean, std):
    """Convert image to tensor."""
    normalize = tt.Compose([tt.ToTensor(), tt.Normalize(mean, std)])
    return normalize(im).unsqueeze(0).requires_grad_(True)
```

Listing 3-60: The function to convert an image to a tensor

This function defines the `normalize` transformation sequence, which first converts the image into a tensor and then normalizes it via the provided

`mean` and `std` values. The normalized image tensor is then expanded by one dimension via `unsqueeze(0)` to make it into a batch that can be passed to VGG16. Finally, gradient calculation is enabled for the image tensor via `requires_grad_(True)`, which is essential to transform the image through gradient ascent.

Now use this function to convert the image into a PyTorch tensor:

```
im_tensor = image_to_tensor(im, mean, std)
```

Next, write the core of the DeepDreams algorithm, shown in Listing 3-61.

```
❶ layer = model.features[1]
iter_num = 100
eta = 0.1

hookdata = {}

def hook_func(layer, input, output):
    """Hook for activations."""
    hookdata["activations"] = output

for _ in range(iter_num):
    handle = layer.register_forward_hook(hook_func)
    try:
     ❷ _ = model(im_tensor)
    except Exception as e:
        print(f"An error occurred during model prediction: {e}")
    finally:
        handle.remove()

 ❸ loss = hookdata["activations"].mean()
 ❹ loss.backward()

   grad_mean = torch.mean(im_tensor.grad.data)
   grad_std = torch.std(im_tensor.grad.data)
 ❺ normalized_grad = (im_tensor.grad.data - grad_mean) / (grad_std + 1e-8)

 ❻ im_tensor.data = im_tensor.data + eta * normalized_grad

 ❼ im_tensor.grad.zero_()

 ❽ im_tensor.data.clamp_(low, high)
```

Listing 3-61: Performing the DeepDreams gradient ascent

This script calculates the DeepDreams version of the image that maximizes the activation of the first ReLU layer ❶ (the second layer of the `features` module of `model`).

The iterative process transforming the image by gradient ascent performs the following steps:

1. Applies a forward hook (defined by the `hook_func()` function) to capture layer activations ❷. No output is needed, as you need just the activations calculated in the forward pass and saved by the hook in `hookdata`.
2. Calculates the mean activation of the layer to be used as loss ❸.
3. Propagates the loss backward to calculate the gradients in the image ❹. No gradients are calculated for the VGG16 model, as it's set in evaluation mode and its weights are frozen.
4. Normalizes the gradient of the image ❺.
5. Updates the image based on the calculated gradients performing the gradient ascent ❻.
6. Clears the gradient in preparation of the next iteration ❼.
7. Clamps the image data to the bounds of the normalized image values to ensure that the image's pixel values are valid ❽.

NOTE *In implementing algorithms like DeepDreams, choosing the right number of iterations (`iter_num`) and step size (`eta`, which is quite similar to the learning rate that you've seen in the backpropagation algorithm) is crucial and often involves a bit of trial and error. Typically, `iter_num` ranges from 10 to 100, affecting the depth of the image transformation (more iterations lead to more-complex changes but could introduce noise). The value of `eta` is usually set from about 0.01 to 0.1 and dictates the magnitude of each image update. A larger step size makes more drastic changes per iteration, while a smaller step size results in subtler, more gradual alterations. The key is to balance these parameters to achieve meaningful transformations without excessively distorting the original image.*

Next, you need convert the image tensor back to the image format. You can do this with the `tensor_to_image()` function shown in Listing 3-62.

```
def tensor_to_image(image, mean, std):
    """Convert tensor to image."""
 ❶ denormalize = tt.Normalize(mean=- mean / std, std=1 / std)
    im_array = denormalize(image.data.clone().detach().squeeze()).numpy()
 ❷ im_array = np.clip(im_array.transpose(1, 2, 0) * 255, 0, 255)
 ❸ im_array = im_array.astype(np.uint8)
 ❹ return Image.fromarray(im_array, "RGB")
```

Listing 3-62: The function to convert a tensor back to an image

This function first defines a denormalization operation ❶, which reverses the effect of the previous normalization by using negative mean values and reciprocal standard deviation values. Next, the function applies the denormalization to the `image` tensor after cloning it, detaching it from the current computation graph, and squeezing it to remove the batch dimension. Finally, it converts `image` to the `im_array` NumPy array. This array is transposed to adjust the channel order from CHW (channels, height, width) to HWC ❷, scaled back to the original 0-to-255 range, clipped to ensure that

the values stay within this range, and converted to an unsigned 8-bit integer format ❸. The final step is to create and return a Pillow `Image` from this array, specifying `RGB` as the color mode ❹.

Now use this function to convert the tensor back to the image format:

```
im_deepdream = tensor_to_image(im_tensor, mean, std)
```

Finally, you can render your first DeepDream, using Listing 3-63.

```
plt.imshow(im_deepdream)
plt.title("DeepDream for Layer 1")
plt.axis("off")
plt.show()
```

Listing 3-63: Plotting a DeepDream

You should get something similar to Figure 3-26.

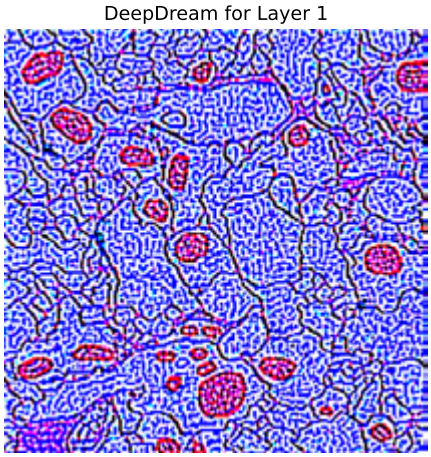


Figure 3-26: The DeepDream corresponding to the first layer

In this DeepDream corresponding to layer 1 of the neural network, the abundance of dot-like patterns and textures reflects this layer's focus on detecting basic features such as edges and simple textures in the input image.

Refactoring the DeepDreams Code as a Function

To improve the organization and reusability of your code, you can refactor the DeepDreams code into the `deepdream()` function shown in Listing 3-64.

```
def deepdream(im, layer_index, iter_num=100, eta=0.1):
    """Generate DeepDream."""
    mean = np.array([0.485, 0.456, 0.406], dtype=np.float32)
    std = np.array([0.229, 0.224, 0.225], dtype=np.float32)

    low = torch.tensor((-mean / std).reshape(1, -1, 1, 1))
    high = torch.tensor(((1 - mean) / std).reshape(1, -1, 1, 1))
```

```
    im_tensor = image_to_tensor(im, mean, std)

    hookdata = {}
    --snip--
    layer = model.features[layer_index]
    for _ in range(iter_num):
        --snip--

    im_deepdream = tensor_to_image(im_tensor, mean, std)

    plt.imshow(im_deepdream)
    plt.title(f"DeepDream for Layer {layer_index}")
    plt.axis("off")
    plt.show()
```

Listing 3-64: The function to generate a DeepDream (by modifying Listing 3-61 and combining it with Listings 3-59 and 3-63)

This function accepts as inputs the image (`im`), the index of the layer whose activation is maximized by the DeepDreams algorithm (`layer_index`), the iteration number (`iter_num`), and the step size (`eta`). The rest of the code is sourced from Listings 3-59, 3-61, and 3-63.

When you call this function with

```
deepdream(im, layer_index=1, iter_num=100, eta=0.1)
```

you should get the DeepDream shown in Figure 3-26.

Thanks to this refactoring of the code, next you can easily add new functionalities, as you'll do in the following sections.

Improving the DeepDreams Function with a Context Manager

You can now use a context manager for the hook. A *context manager* in Python is a special construct that provides a clean way to automatically allocate and release resources. It allows you to set up a temporary context for your code, ensuring that resources are properly managed and cleaned up after the code block is executed, typically using the `with` statement.

To do so, write the `Fwd_Hook` class, as shown in Listing 3-65.

```
class Fwd_Hook():
    """Forward hook."""

    def __init__(self, layer):
        """Initialize forward hook."""
        self.hook = layer.register_forward_hook(self.hook_func)

    def hook_func(self, layer, input, output):
        """Save activations."""
        self.activations = output

    def __enter__(self, *args):
```

```
        """Enter context management."""
    ❶ return self

    def __exit__(self, *args):
        """Exit context management and remove hook."""
    ❷ self.hook.remove()
```

Listing 3-65: The class to context-manage the hook to the activation of a layer

When the `Fwd_Hook` class is initialized with the `__init__()` method, the class registers a forward hook on the specified `layer`, which ensures that the designated `hook_func()` method is called every time the layer completes its forward pass. This hook method captures the output of the layer, storing it in the `activations` attribute for later use. The methods `__enter__()` and `__exit__()` define the class as a context manager. When entering the context (using a `with` statement), an instance of `Fwd_Hook` is returned ❶, and upon exiting, the context manager ensures cleanup by removing the hook ❷. This approach provides an elegant and Pythonic way to temporarily attach and safely remove hooks.

To use this context manager, you need to modify the `deepdream()` function as shown in Listing 3-66.

```
def deepdream(im, layer_index, iter_num=100, eta=0.1):
    --snip--
    im_tensor = image_to_tensor(im, mean, std)

    layer = model.features[layer_index]
    for _ in range(iter_num):
        with Fwd_Hook(layer) as fh:
            _ = model(im_tensor)

        loss = fh.activations.mean()
        loss.backward()
        --snip--
```

Listing 3-66: The function to generate a DeepDream with a context manager for the hook (by modifying Listing 3-64)

Verify that this updated `deepdream()` function still works:

```
deepdream(im, layer_index=1, iter_num=100, eta=0.1)
```

This should again generate the DeepDream in Figure 3-26.

Creating DeepDreams from Deeper Layers

Now it's time to play with your code and generate DeepDreams from deeper layers:

```
for layer_index in [1, 3, 6, 8, 11, 13, 15, 18, 20, 22, 25, 27, 29]:
    deepdream(im, layer_index, iter_num=100, eta=0.1)
```

Figure 3-27 showcases a collection of DeepDreams generated from various layers of the neural network, visualizing the complex patterns and features that each layer has learned to recognize.

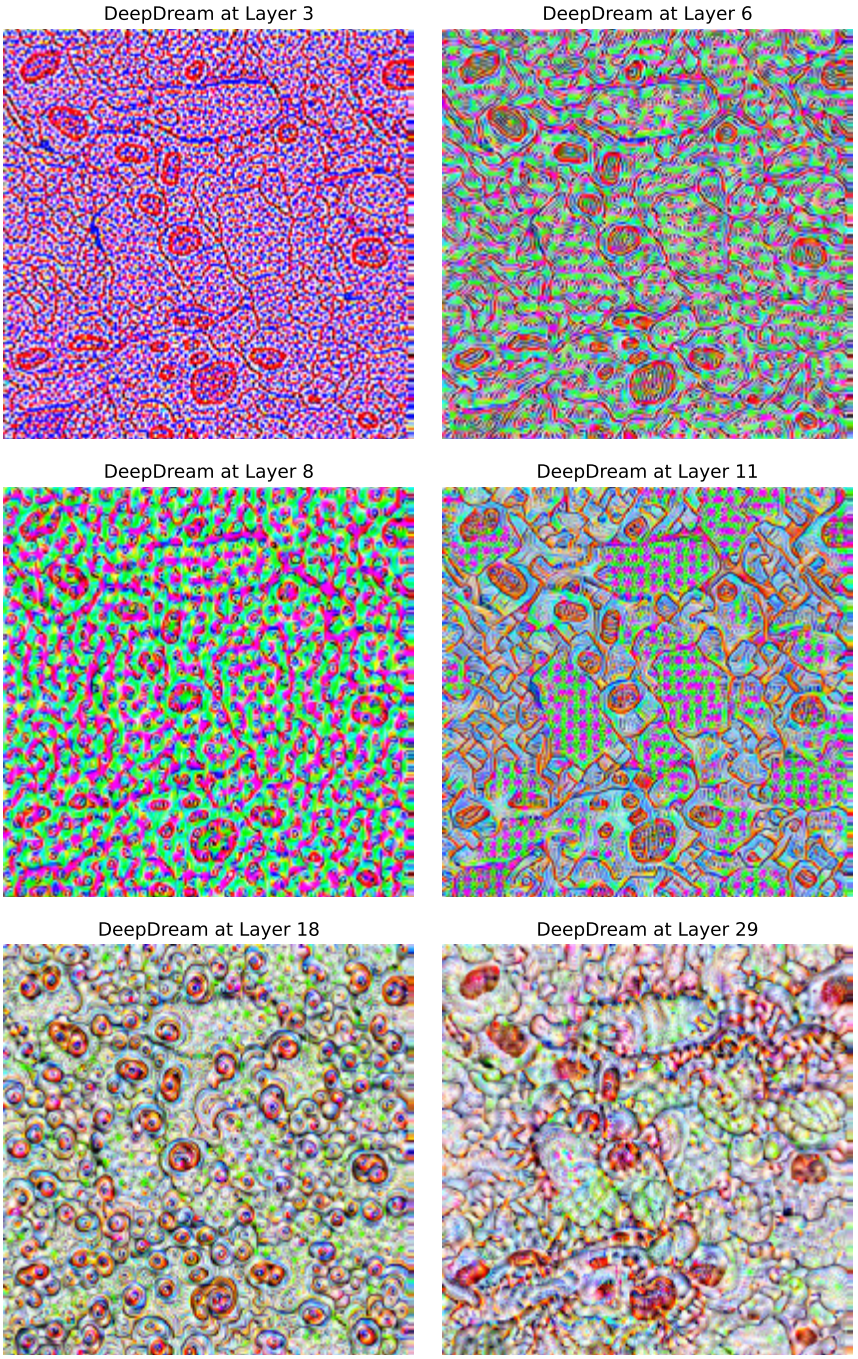


Figure 3-27: DeepDreams maximizing the activation of deeper layers

In the early layers, such as layers 3 and 6, the patterns are simple and repetitive, focusing on basic shapes and edges with vibrant colors. As you progress to intermediate layers like 8 and 11, the complexity increases, with the DeepDreams beginning to feature more-abstract motifs that suggest a transition from simple to more-complex feature recognition.

In the deeper layers, 18 and 29, the images become increasingly surreal, with elaborate, dreamlike amalgamations of shapes and textures that resemble eyes and biological forms, illustrating the network's higher-level feature detection that interprets the visual data in more-abstract ways.

Using Multiple Layers at Once

You can also generate DeepDreams by maximizing the activation of multiple layers at once. To do this, you need to capture the activations of multiple layers, which requires modifications to the context manager class, as shown in Listing 3-67.

```
class Fwd_Hooks():
    """Forward hooks."""

    def __init__(self, layers):
        """Initialize forward hooks."""
        self.hooks, self.activations_list = [], []
        for layer in layers:
         ❶ self.hooks.append(layer.register_forward_hook(self.hook_func))

    def hook_func(self, layer, input, output):
        """Save activations."""
        self.activations_list.append(output)

    --snip--

    def __exit__(self, *args):
        """Exit context management and remove hooks."""
        for hook in self.hooks:
         ❷ hook.remove()
```

Listing 3-67: The class to context-manage the hooks to the activations of multiple layers at once (by modifying Listing 3-65)

This new context manager class contains, as attributes, lists of hooks and activations, one for each layer. When the context manager is initialized, a hook function is attached to each layer ❶ and, when the context manager exits, all the functions are removed ❷.

Finally, modify the `deepdream()` function further to accept multiple input layers, as shown in Listing 3-68.

```
def deepdream(im, layer_indices, iter_num=100, eta=.1):
    --snip--
    layers = [model.features[layer_index] for layer_index in layer_indices]
```

```
for _ in range(iter_num):
❶ with Fwd_Hooks(layers) as fh:
        _ = model(im_tensor)

    losses = [activations.mean() for activations in fh.activations_list]
❷ loss = torch.stack(losses).sum()
    loss.backward()
    --snip--
    plt.title(f"DeepDream for Layers {layer_indices}")
    --snip--
```

Listing 3-68: The function to generate a DeepDream from multiple layers (by modifying Listing 3-66)

This is the definitive version of the `deepdream()` function. It accepts as input multiple layers' indices and extracts the corresponding `layers`. Then it uses the context manager for hooks in multiple layers ❶. The loss is now calculated by summing up the `losses` for the activation of each layer ❷.

Now you can get a DeepDream that maximizes feature activations from multiple layers of VGG16 simultaneously:

```
deepdream(im, layer_indices=[1, 8, 11, 18, 25, 27, 29], iter_num=100, eta=0.1);
```

This should generate the DeepDream in Figure 3-28.

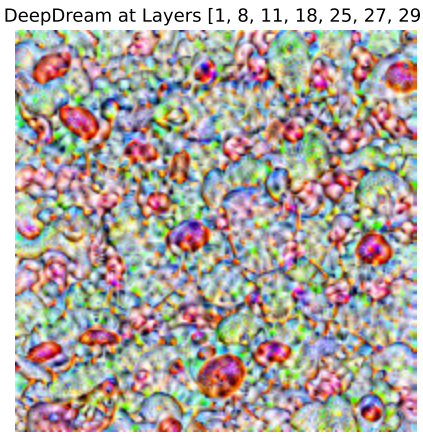


Figure 3-28: The DeepDream maximizing the activation of multiple layers at once

This multilayered approach creates a densely packed tapestry of features that vary in complexity. The image is a vibrant collage of textures and patterns, with visual elements that range from the geometric shapes found in the earliest layers to the more-complex, almost biological structures reminiscent of eyes and organic forms that emerge from the deeper layers. This visual complexity is a result of targeting a combination of layers, each contributing its own learned representations to the final image.

NOTE *Code Example 3-C, "Creating DeepDreams," is available at* https://github.com/DeepTrackAI/DeepLearningCrashCourse. *Navigate to the* Ch03_CNN *folder and then* ec03_C_deepdream. *The* deepdream.ipynb *notebook provides a complete code example that loads an image and the VGG16 pretrained neural network, and uses them to create DeepDreams.*

Project 3D: Transferring the Style of Images

In this last project of this chapter, you'll explore *neural style transfer*, which answers this question: How can the stylistic elements of one image be imposed onto the content of another image while preserving the original content's structure?

For example, imagine you have a photograph (content image) and a painting (style image). You want to re-create the photograph in the unique artistic style of the painting. You can achieve this by using a deep neural network trained to understand and identify various features in images. During neural style transfer, you ask the neural network to keep the large-scale structures from the photograph, but to paint them with the textures and patterns you see in the painting.

As you've seen many times by now, a deep network comprises multiple layers, each responsible for recognizing different levels of the image's details. The first layers might detect simple edges and textures typically more related to the style. The last layers capture the larger-scale complex structures that define the content. You can then instruct the computer to minimize the difference between the content of your photograph and the content recognized in a particular layer of the network, while simultaneously minimizing the difference between the style of your painting and the style recognized in other layers.

Through an iterative process of adjustments, the neural network blends the content and the style, keeping the essential elements of the photograph's structure while casting it in the colors, brush strokes, and textures of the painting, creating a hybrid image that looks like a painted version of the photograph. You'll see how this works in more detail in the rest of this project. You'll learn how to apply the texture of the Trencadís Lizard (created by Antoni Gaudí in Barcelona's Parc Güell) to a microscopic image of the neural tissue of a fruit fly larva.

Before you begin this project, you should have a solid understanding of the concepts introduced in Projects 3A and 3C, particularly regarding the use of hooks and context managers to access the activations of layers.

Loading the Content and Style Images

Start by loading the content image:

```
from PIL import Image

content = (Image.open("neuraltissue_with_colorlabels.png").convert("RGB")
           .crop((100, 170, 100 + 256, 170 + 256)))
```

This is the same image you used in Project 3C to generate DeepDreams. The only difference is that you are now cropping a square of 256×256 pixels from the image.

Then plot this crop:

```
import matplotlib.pyplot as plt

plt.imshow(content)
plt.axis("off")
plt.show()
```

The content image is shown on the left side of Figure 3-29.

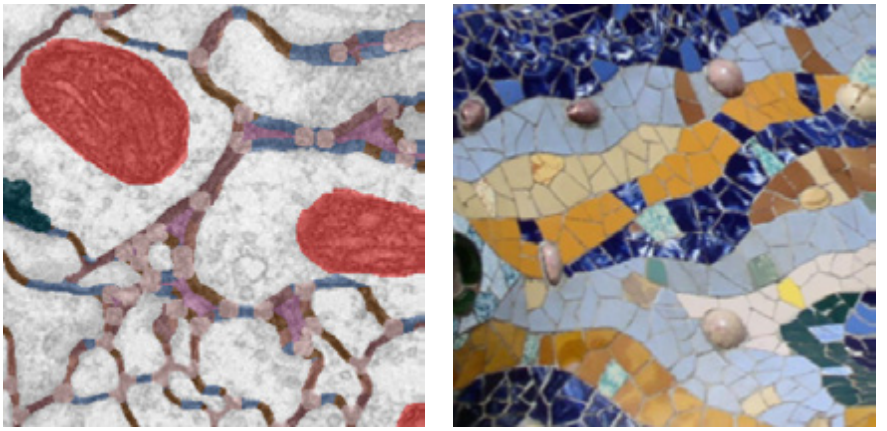


Figure 3-29: The content (left) and style (right) images for the neural style transfer

Next, load the image you'll use for the style. You'll use a photograph of a detail of Gaudí's lizard, which is in the *lizard.png* image file:

```
style = Image.open("lizard.png").convert("RGB").resize((256, 256))
```

This loads the style image, converts it to RGB, and finally resizes it to 256×256 pixels.

You can then plot the style image:

```
plt.imshow(style)
plt.axis("off")
plt.show()
```

The resulting image is shown on the right side of Figure 3-29.

Loading a Pretrained Neural Network

Now you need to import a pretrained neural network. As with the DeepDreams in Project 3C, you'll import the VGG16 model and freeze its weights with Listing 3-57. The printout with the architecture of the VGG16 model is

in Listing 3-58. Furthermore, since this is a pretrained model, it needs normalization, which is again the same as in Project 3C (Listing 3-59).

Implementing Style Transfer

To implement style transfer, you'll reuse several tools developed already for the DeepDreams in Project 3C. In particular, you'll reuse the `Fwd_Hooks` class (Listing 3-67) and the `image_to_tensor()` and `tensor_to_image()` functions (Listings 3-60 and 3-62).

Start by implementing the additional `gram()` function to calculate the Gram matrix between all the activations of a specific layer, as shown in Listing 3-69.

```
from torch import bmm

def gram(tensor):
    """Gram matrix."""
 ❶ batch_size, num_channels, height, width = tensor.size()
 ❷ features = tensor.view(batch_size, num_channels, height * width)
    gram_matrix = bmm(features, features.transpose(1, 2)) / (height * width)
    return gram_matrix
```

Listing 3-69: The function to calculate the Gram matrix

The *Gram matrix* represents the correlations between different feature maps (or channels) of a convolutional layer's output. The `gram()` function unpacks the dimensions of the input tensor ❶. Then `gram()` reshapes the tensor into a 2D matrix ❷ (the `batch_size` will be equal to 1). Finally, the function computes the Gram matrix as the product of the matrix by its transpose, normalizing it by the number of elements in each feature map (`height * width`).

You can implement the style transfer with the `style_transfer()` function in Listing 3-70, which is quite similar in structure to the `deepdream()` function in Listing 3-68.

```
def style_transfer(image, content, style, content_layers, style_layers,
                   lr=1, iter_num=100, beta=1e3):
    """Perform style transfer."""
    mean = np.array([0.485, 0.456, 0.406], dtype=np.float32)
    std = np.array([0.229, 0.224, 0.225], dtype=np.float32)

    image_tensor = image_to_tensor(image, mean, std)

    with Fwd_Hooks(content_layers) as fh:
        _ = model(image_to_tensor(content, mean, std))
 ❶ con_activ_list = [activ.detach() for activ in fh.activations_list]

    with Fwd_Hooks(style_layers) as fh:
        _ = model(image_to_tensor(style, mean, std))
 ❷ target_gram_list = [gram(activ.detach()) for activ in fh.activations_list]
```

```
    optimizer = torch.optim.LBFGS([image_tensor], lr=lr)
    mse_loss = torch.nn.MSELoss(reduction="sum")

    def closure():
        """Closure function for the optimizer."""
        --snip--
        return total_loss

    for i in range(iter_num):
        print(f"iteration {i}")
     ❸ optimizer.step(closure)

        plt.imshow(tensor_to_image(image_tensor, mean, std))
        plt.title(f"Iteration {i}")
        plt.axis("off")
        plt.show()
```

Listing 3-70: The function for implementing style transfer

This function transforms the input image into a PyTorch tensor with the `image_to_tensor()` function, normalizing the image with the predefined mean and standard deviation values. Then forward hooks are used within context managers to capture content activations from specified layers when the content image is passed through the model ❶, and to calculate the Gram matrix representations of style activations when the style image is processed ❷.

As `optimizer`, this code uses the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm because of its efficiency in handling the large number of variables involved in style transfer. This optimizer is responsible for iteratively tweaking the image tensor to minimize the loss function that measures the difference between the content and style features of the current image and the desired targets. The `closure()` function is essential in this context because L-BFGS requires the reevaluation of the loss within each iteration to update the weights, and this function provides the mechanism to recalculate the loss and its gradients every time the optimizer takes a step, as you'll see in detail in Listing 3-71.

In each interaction, after the optimization step ❸, the current state of the image tensor is converted back to an image via the `tensor_to_image()` function and displayed, allowing for visually tracking the style-transfer process.

Now, add the necessary `closure()` function to the `style_transfer()` function provided in Listing 3-71.

```
def style_transfer(...):
    --snip--
    mse_loss = torch.nn.MSELoss(reduction="sum")

    def closure():
        """Closure function for the optimizer."""
     ❶ optimizer.zero_grad()
```

```
    with Fwd_Hooks(content_layers) as fh:
        _ = model(image_tensor)
    im_con_activ_list = fh.activations_list

    content_loss = 0
    for im_con_activ, con_activ in zip(im_con_activ_list, con_activ_list):
        num_feats = im_con_activ.shape[1]
     ❷ content_loss += mse_loss(im_con_activ, con_activ) / num_feats ** 2
❸ content_loss = content_loss / len(im_con_activ_list)

    with Fwd_Hooks(style_layers) as fh:
        _ = model(image_tensor)
❹ im_gram_list = [gram(activ) for activ in fh.activations_list]

    style_loss = 0
    for im_gram, target_gram in zip(im_gram_list, target_gram_list):
        num_feats = im_gram.shape[1]
     ❺ style_loss += mse_loss(im_gram, target_gram) / num_feats ** 2
❻ style_loss = style_loss / len(im_gram_list)

    print(f"content_loss={content_loss} style_loss={style_loss}")

 ❼ total_loss = content_loss + beta * style_loss
 ❽ total_loss.backward()
    return total_loss
--snip--
```

Listing 3-71: The closure function for the optimizer of the style transfer (by modifying Listing 3-70)

This closure function is evaluated several times during each step of the L-BFGS optimization loop to compute the gradient of the loss function and provide it to the optimizer. This function performs the following operations:

1. Resets the optimizer's gradients to 0 ❶. This is necessary to prevent the accumulation of gradients from multiple backward passes.
2. Computes the content loss by comparing the activations from the current image tensor to those from the original content image. The function computes the MSE loss between each corresponding set of activations and then normalizes it by the number of features squared ❷. The total content loss is the average of these losses over all content layers ❸.
3. Calculates the style loss in a similar way, but instead of using the raw activations, it uses the Gram matrices of the activations to capture the style information ❹. The function calculates the MSE loss between the Gram matrix of the current image tensor and the Gram matrix of the style image, normalizes the loss by the number of elements in the Gram matrix squared ❺, and averages over all style layers ❻.

4. Combines the content loss and style loss to form the total loss ❼, with the style loss being scaled by the `beta` regularization factor to balance the two types of losses.
5. Calls the `backward()` method on the total loss ❽, which computes the gradient of the loss function with respect to the image tensor.
6. Returns the total loss.

Creating an Image in Gaudí's Style

You are finally ready to create a new version of the neural tissue image in Gaudí's style. You can do this with the content image as a starting point, using Listing 3-72.

```
style_transfer(
    image=content, content=content, style=style,
    content_layers=[model.features[l] for l in [14]],
    style_layers=[model.features[l] for l in [0, 2, 5, 7, 10]],
    lr=1, iter_num=50, beta=1e5,
)
```

Listing 3-72: Transforming the style of an image

This code transforms the original `content` image into an image that retains the content seen by the 14th VGG16 layer, but features the style of the `style` image seen by layers 0, 2, 5, 7, and 10 of VGG16.

Figure 3-30 shows the resulting style images in the first and last iterations.

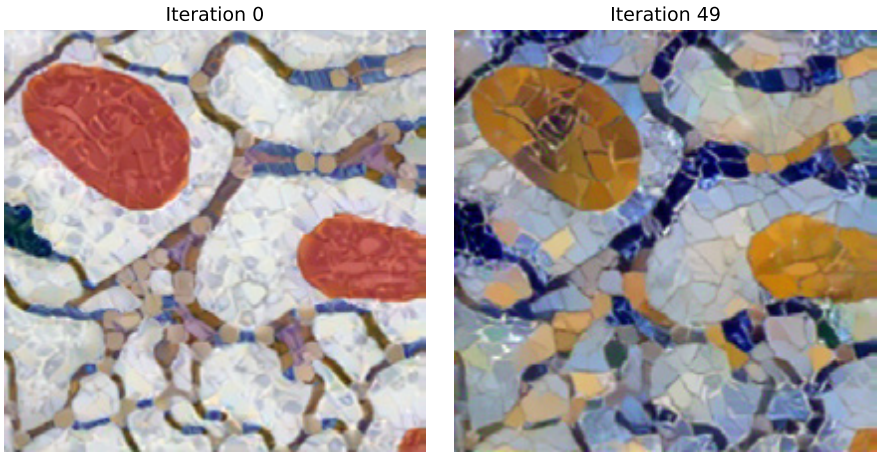


Figure 3-30: The progression of the style transfer

As the iterations proceed, the output image becomes increasingly similar in style to Gaudí's lizard.

NOTE *Code Example 3-D, "Transferring Image Styles," is available at* https://github.com/DeepTrackAI/DeepLearningCrashCourse. *Navigate to the* Ch03_CNN *folder and then* ec03_D_style_transfer. *The* style_transfer.ipynb *notebook provides a complete code example that loads the content image, the style image, and the VGG16 pretrained neural network, and uses them to transfer the style to the image.*

Summary

In this chapter, you learned to use convolutional neural networks, which are essential tools for image analysis. You began by exploring the concept of convolution and its application to both 1D and 2D data. Then you continued with the practical implementation of convolutional neural networks, using PyTorch. You gained hands-on experience in constructing convolutional layers, integrating ReLU activation functions, and mastering pooling and upsampling techniques, all crucial for processing and transforming images. The inclusion of dense layers and their role in image classification further enriched your understanding.

In Project 3A, you applied convolutional neural networks to the task of classifying malaria-infected blood smears. This project highlighted the medical application of convolutional neural networks, involving loading and preprocessing complex datasets, deploying both dense and convolutional neural networks, and visualizing the convolutional filters and layer activations to gain some understanding of the network's decision-making process.

In Project 3B, you ventured into video analysis, using convolutional neural networks to localize microscopic particles in video data. This project highlighted the critical importance of having a reliable ground truth for neural network training. To achieve this, you engaged in the demanding task of manually annotating the particle images, providing the network with ground-truth labels. You also explored the route of generating simulated particle images, offering an alternative means of creating a robust training dataset.

Project 3C introduced you to the artistic side of convolutional neural networks with the creation of DeepDreams. This project illustrated the network's ability to enhance and alter images, revealing the patterns learned by the network.

Project 3D explored the fascinating world of neural style transfer. You learned to merge the content of one image with the style of another, further showcasing the creative potential of convolutional neural networks.

Looking forward, this knowledge serves as a stepping stone to the next chapters, where you'll delve deeper into advanced neural network architectures and techniques. Your newly acquired understanding of convolutional neural networks paves the way for exploring encoder-decoders for latent space manipulation, U-Nets for image transformation, and even generative adversarial networks and diffusion models for image synthesis.

Seminal Works and Further Reading

Yann LeCun et al. introduced the LeNet-5 architecture in "Gradient-Based Learning Applied to Document Recognition," published in 1998 in *Proceedings of the IEEE* (volume 86, pages 2,278–2,324). LeNet-5 demonstrated that backpropagation could be effectively used to train convolutional neural networks, providing one of the first successful applications of convolutional neural networks for image recognition.

A significant leap came with "ImageNet Classification with Deep Convolutional Neural Networks" by Alex Krizhevsky et al., published in 2012 in *Advances in Neural Information Processing Systems* (*NeurIPS*, volume 25). This work introduced AlexNet, which won the 2012 ImageNet competition by a large margin, demonstrating the power of deeper convolutional architectures combined with GPU acceleration for training large models. AlexNet's success was a major catalyst for the deep learning revolution in computer vision.

"Very Deep Convolutional Networks for Large-Scale Image Recognition" by Karen Simonyan and Andrew Zisserman, published in 2014 on arXiv (article number 1409.1556) and presented at the 2015 International Conference on Learning Representations (ICLR), introduced the VGG network. This architecture showed that increasing the depth of convolutional neural networks by using smaller convolutional filters could yield significant improvements in image-classification accuracy.

Christian Szegedy et al. introduced the Inception architecture (also known as GoogLeNet) in 2015 in "Going Deeper with Convolutions," published in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*, pages 1–9). Inception utilized multiple filter sizes at each layer, allowing for deeper networks while maintaining computational efficiency.

The introduction of the ResNet architecture in "Deep Residual Learning for Image Recognition" by Kaiming He et al., published in 2016 in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*, pages 770–778), addressed the vanishing gradient problem in deep networks by using residual connections. This allowed for the successful training of very deep networks, resulting in breakthrough performance in image classification and numerous other computer vision tasks.

Project 3D is based on "Digital Video Microscopy Enhanced by Deep Learning" by Saga Helgadottir et al., published in *Optica* (volume 6, pages 506–513). This work uses convolutional neural networks to accurately measure the position of microscopic particles.

Project 3D is inspired by "Image Style Transfer Using Convolutional Neural Networks" by Leon A. Gatys et al., published in 2016 in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*, pages 2,414–2,423). The authors introduced an artistic algorithm that was able to separate and recombine the image content and style of natural images.

CHALLENGE PROJECTS

3-1: Advanced medical image classification Implement a convolutional neural network to classify medical images beyond the malaria-infected blood smears. Choose a medical imaging dataset from a Kaggle challenge to classify diseases or conditions (for example, the Chest CT-Scan Images dataset or the COVID-19 Chest CT Image Augmentation GAN dataset). Compare your results to those of the challenge participants.

3-2: Traffic sign recognition Develop a convolutional neural network to recognize and classify traffic signs from real-world images. You can use the German Traffic Sign Recognition Benchmark (GTSRB) dataset. Create a model that can accurately identify various traffic signs from multiple angles, distances, and lighting conditions. Compare the performance of your model against traditional image-recognition methods.

3-3: Artistic style transfer with custom styles Extend the neural style transfer project to explore how well your model can adapt to various styles and how it handles style transfer with complex and abstract images.