Throughout this book, we have built several deep learning models that can perform different kinds of tasks for us. For example, a handwritten digit classifier, an image-caption generator, a sentiment classifier, and more. Although we have mastered how to train and evaluate these models using PyTorch, we do not know what precisely is happening inside these models while they make predictions. Model interpretability or explainability is that field of machine learning where we aim to answer the question, why did the model make that prediction? More elaborately, what did the model see in the input data to make that particular prediction?

In this chapter, we will use the handwritten digit classification model from *Chapter 1*, *Overview of Deep Learning Using PyTorch*, to understand its inner workings and thereby explain why the model makes a certain prediction for a given input. We will first dissect the model using only PyTorch code. Then, we will use a specialized model interpretability toolkit, called **Captum**, to further investigate what is happening inside the model. Captum is a dedicated third-party library for PyTorch that provides model interpretability tools for deep learning models, including image- and text-based models.

This chapter should provide you with the skills that are necessary to uncover the internals of a deep learning model. Looking inside a model this way can help you to reason about the model's predictive behavior. At the end of this chapter, you will be able to use the hands-on experience to start interpreting your own deep learning models using PyTorch (and Captum).

This chapter is broken down into the following topics:

Model interpretability in PyTorch

Using Captum to interpret models

Model interpretability in PyTorch

In this section, we will dissect a trained handwritten digits classification model using PyTorch in the form of an exercise. More precisely, we will be looking at the details of the convolutional layers of the trained handwritten digits classification model to understand what visual features the model is learning from the handwritten digit images. We will look at the convolutional filters/kernels along with the feature maps produced by those filters.

Such details will help us to understand how the model is processing input images and, therefore, making predictions. The full code for the exercise can be found in our github repository [13.1] .

Training the handwritten digits classifier – a recap

We will quickly revisit the steps involved in training the handwritten digits classification model, as follows:

1. First, we import the relevant libraries, and then set the random seeds to be able to reproduce the results of this exercise:

```
import torch
np.random.seed(123)
torch.manual_seed(123)
```

1. Next, we will define the model architecture:

```
class ConvNet(nn.Module):
    def __init__(self):
    def forward(self, x):
```

1. Next, we will define the model training and testing routine:

```
def train(model, device, train_dataloader, optim,  epoch):
def test(model, device, test_dataloader):
```

1. We then define the training and testing dataset loaders:

```
train_dataloader = torch.utils.data.DataLoader(...)
test_dataloader = torch.utils.data.DataLoader(...)
```

1. Next, we instantiate our model and define the optimization schedule:

```
device = torch.device("cpu")
model = ConvNet()
optimizer = optim.Adadelta(model.parameters(), lr=0.5)
```

1. Finally, we start the model training loop where we train our model for 20 epochs:

```
for epoch in range(1, 20):
    train(model, device, train_dataloader, optimizer, epoch)
    test(model, device, test_dataloader)
```

This should output the following:

```
epoch: 1 [0/60000 (0%)]  training loss: 2.324445
epoch: 1 [320/60000 (1%)]        training loss: 1.727462
epoch: 1 [640/60000 (1%)]        training loss: 1.428922
epoch: 1 [960/60000 (2%)]        training loss: 0.717944
epoch: 1 [1280/60000 (2%)]       training loss: 0.572199
                                 ⋮
epoch: 19 [58880/60000 (98%)]    training loss: 0.016509
epoch: 19 [59200/60000 (99%)]    training loss: 0.118218
epoch: 19 [59520/60000 (99%)]    training loss: 0.000097
epoch: 19 [59840/60000 (100%)]   training loss: 0.000271

Test dataset: Overall Loss: 0.0387, Overall Accuracy: 9910/10000 (99%)
```

Figure 13.1 – Model training logs

1. Finally, we can test the trained model on a sample test image. The sample test image is loaded as follows:

```
test_samples = enumerate(test_dataloader)
b_i, (sample_data, sample_targets) = next(test_samples)
plt.imshow(sample_data[0][0], cmap='gray', interpolation='none')
plt.show()
```

This should output the following:

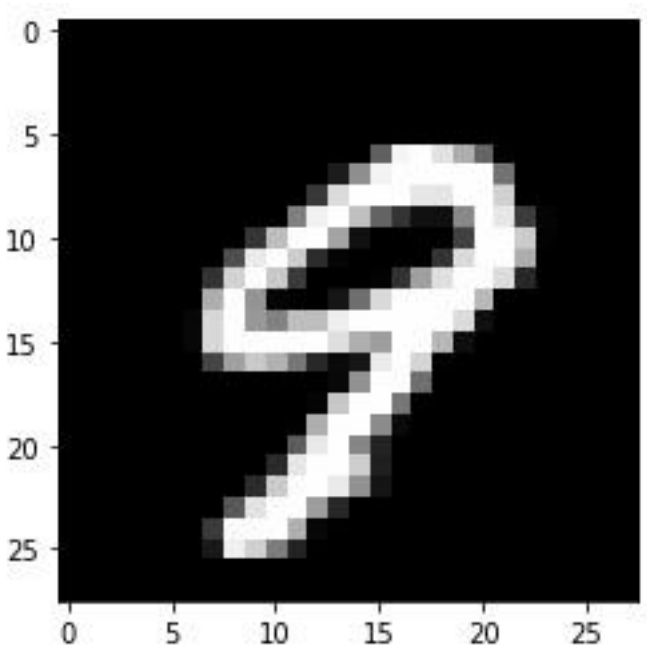

Figure 13.2 – An example of a handwritten image

1. Then, we use this sample test image to make a model prediction, as follows:

```
print(f"Model prediction is : {model(sample_data).data.max(1)[1][0]}")
print(f"Ground truth is : {sample_targets[0]}")
```

This should output the following:

```
Model prediction is : 9
Ground truth is : 9
```

Figure 13.3 – Model prediction

Therefore, we have trained a handwritten digits classification model and used it to make inference on a sample image. We will now look at the internals of the trained model. We will also investigate what convolutional filters have been learned by this model.

Visualizing the convolutional filters of the model

In this section, we will go through the convolutional layers of the trained model and look at the filters that the model has learned during training. This will tell us how the convolutional layers are operating on the input image, what kinds of features are being extracted, and more:

1. First, we need to obtain a list of all the layers in the model, as follows:

```
model_children_list = list(model.children())
convolutional_layers = []
model_parameters = []
model_children_list
```

This should output the following:

```
[Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1)),
 Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1)),
 Dropout2d(p=0.1, inplace=False),
 Dropout2d(p=0.25, inplace=False),
 Linear(in_features=4608, out_features=64, bias=True),
 Linear(in_features=64, out_features=10, bias=True)]
```

Figure 13.4 – Model layers

As you can see, there are 2 convolutional layers that both have 3x3-sized filters. The first convolutional layer uses **16** such filters, whereas the second convolutional layer uses **32**. We are focusing on visualizing convolutional layers in this exercise because they are visually more intuitive. However, you can similarly explore the other layers, such as linear layers, by visualizing their learned weights.

1. Next, we select only the convolutional layers from the model and store them in a separate list:

```
for i in range(len(model_children_list)):
    if type(model_children_list[i]) == nn.Conv2d:
        model_parameters.append(model_children_list[i].w      eight)
        convolutional_layers.append(model_children_list[i])
```

In this process, we also make sure to store the parameters or weights learned in each convolutional layer.

1. We are now ready to visualize the learned filters of the convolutional layers. We begin with the first layer, which has 16 filters of size 3x3 each. The following code visualizes those filters for us:

```
plt.figure(figsize=(5, 4))
for i, flt in enumerate(model_parameters[0]):
    plt.subplot(4, 4, i+1)
    plt.imshow(flt[0, :, :].detach(), cmap='gray')
    plt.axis('off')
plt.show()
```

This should output the following:

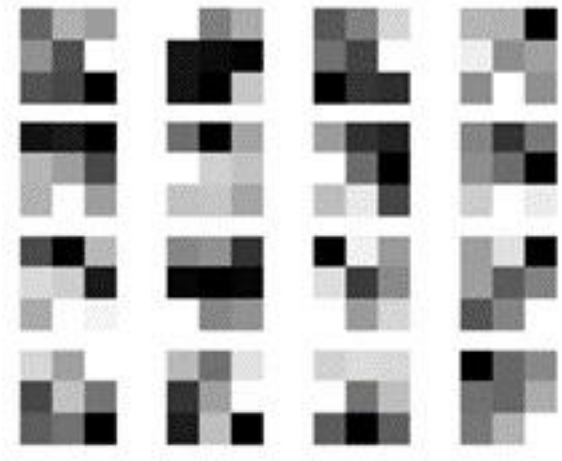

Figure 13.5 – The first convolutional layer's filters

Firstly, we can see that all the learned filters are slightly different from each other, which is a good sign. These filters usually have contrasting values inside them so that they can extract some types of gradients when convolved around an image. During model inference, each of these 16 filters operates independently on the input grayscale image and produces 16 different feature maps, which we will visualize in the next section.

1. Similarly, we can visualize the 32 filters learned in the second convolutional layer using the same code, as in the preceding step, but with the following change:

```
plt.figure(figsize=(5, 8))
for i, flt in enumerate(model_parameters[1]):
plt.show()
```

This should output the following:

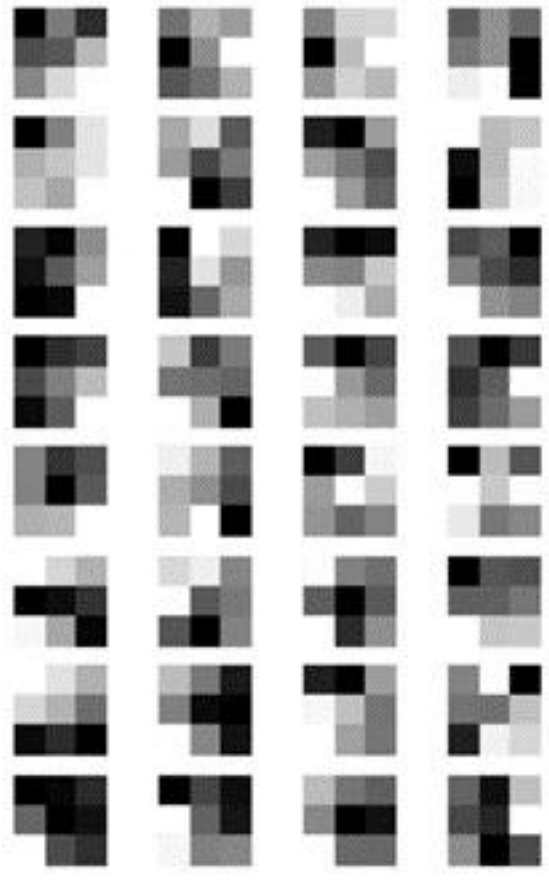

Figure 13.6 – The second convolutional layer's filters

Once again, we have 32 different filters/kernels that have contrasting values aimed at extracting gradients from the image. These filters are already applied to the output of the first convolutional layer, and hence produce even higher levels of output feature maps. The usual goal of CNN models with multiple convolutional layers is to keep producing more and more complex, or higher-level, features that can represent complex visual elements such as a nose on a face, traffic lights on the road, and more.

Next, we will take a look at what comes out of these convolutional layers as these filters operate/convolve on their given inputs.

Visualizing the feature maps of the model

In this section, we will run a sample handwritten image through the convolutional layers and visualize the outputs of these layers:

1. First, we need to gather the results of every convolutional layer output in the form of a list, which is achieved using the following code:

```
per_layer_results = [convolutional_layers[0](sample_data)]
for i in range(1, len(convolutional_layers)):
    per_layer_results.append(convolutional_layers[i](per_layer_results[-1]))
```

Notice that we call the forward pass for each convolutional layer separately while ensuring that the *n*th convolutional layer receives as input the output of the (*n-1*)th convolutional layer.

1. We can now visualize the feature maps produced by the two convolutional layers. We will begin with the first layer by running the following code:

```
plt.figure(figsize=(5, 4))
layer_visualisation = per_layer_results[0][0, :, :, :]
layer_visualisation = layer_visualisation.data
print(layer_visualisation.size())
for i, flt in enumerate(layer_visualisation):
    plt.subplot(4, 4, i + 1)
    plt.imshow(flt, cmap='gray')
    plt.axis("off")
plt.show()
```

1. This should output the following:

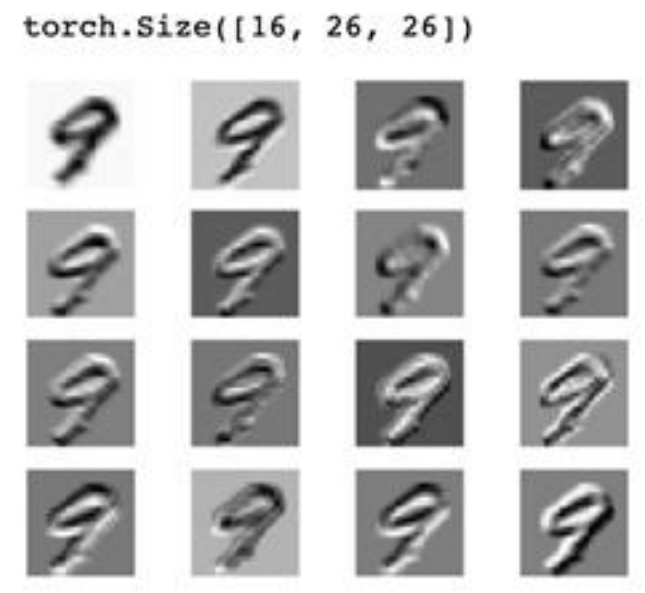

Figure 13.7 – The first convolutional layer's feature maps

The numbers, **(16, 26, 26)**, represent the output dimensions of the first convolution layer. Essentially, the sample image size is (28, 28), the filter size is (3,3), and there is no padding. Therefore, the resulting feature map size will be (26, 26). Because there are 16 such feature maps produced by the 16 filters (please refer to *Figure 13.5*), the overall output dimension is (16, 26, 26).

As you can see, each filter produces a feature map from the input image. Additionally, each feature map represents a different visual feature in the image. For example, the top-left feature map essentially inverts the pixel values in the image (please refer to *Figure 13.2*), whereas the bottom-right feature map represents some form of edge detection.

These 16 feature maps are then passed on to the second convolutional layer, where yet another 32 filters convolve separately on these 16 feature maps to produce 32 new feature maps. We will look at these next.

1. We can use the same code as the preceding one with minor changes (as highlighted in the following code) to visualize the 32 feature maps produced by the next convolutional layer:

```
plt.figure(figsize=(5, 8))
layer_visualisation = per_layer_results[1][0, :, :, :]
    plt.subplot(8, 4, i + 1)
plt.show()
```

This should output the following:

```
torch.Size([32, 24, 24])
```

Figure 13.8 – The second convolutional layer's feature maps

Compared to the earlier 16 feature maps, these 32 feature maps are evidently more complex. They seem to be doing more than just edge detection, and this is because they are already operating on the outputs of the first convolutional layer instead of the raw input image.

In this model, the 2 convolutional layers are followed by 2 linear layers with (4,608x64) and (64x10) number of parameters, respectively. Although the linear layer weights are also useful to visualize, the sheer number of parameters (4,608x64) is, visually, a lot to get your head around. Therefore, in this section, we will restrict our visual analysis to convolutional weights only.

And thankfully, we have more sophisticated ways of interpreting model prediction without having to look at such a large number of parameters. In the next section, we will explore Captum, which is a machine learning model interpretability toolkit that works with PyTorch and helps us to explain model decisions within a few lines of code.

Using Captum to interpret models

Captum [13.2] is an open source model interpretability library built by Facebook on top of PyTorch, and it is currently (at the time of writing) under active development. In this section, we will use the handwritten digits classification model that we had trained in the preceding section. We will also use some of the model interpretability tools offered by Captum to explain the predictions made by this model. The full code for the following exercise can be found in our github repository [13.3] .

Setting up Captum

The model training code is similar to the code shown under the *Training the handwritten digits classifier – a recap* section. In the following steps, we will use the trained model and a sample image to understand what happens inside the model while making a prediction for the given image:

1. There are few extra imports related to Captum that we need to perform in order to use Captum's built-in model interpretability functions:

```
from captum.attr import IntegratedGradients
from captum.attr import Saliency
from captum.attr import DeepLift
from captum.attr import visualization as viz
```

1. In order to do a model forward pass with the input image, we reshape the input image to match the model input size:

```
captum_input = sample_data[0].unsqueeze(0)
captum_input.requires_grad = True
```

1. As per Captum's requirements, the input tensor (image) needs to be involved in gradient computation. Therefore, we set the `requires_grad` flag for input to `True`.
2. Next, we prepare the sample image to be processed by the model interpretability methods using the following code:

```
orig_image = np.tile(np.transpose((sample_data[0].cpu().detach().numpy() / 2) + 0.5,
(1, 2, 0)), (1,1,3))
_ = viz.visualize_image_attr(None, orig_image, cmap='gray', method="original_image",
title="Original Image")
```

This should output the following:

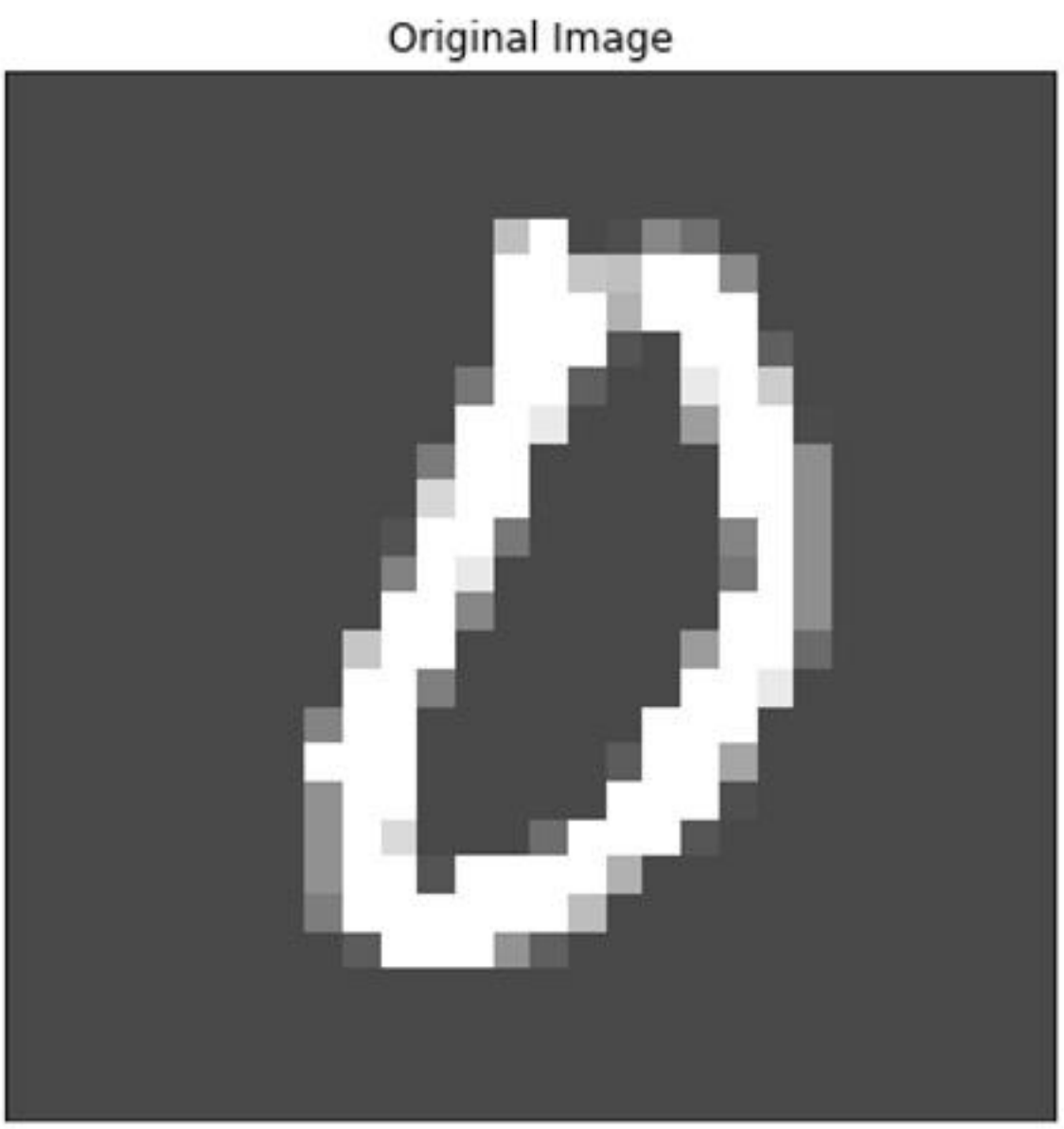

Figure 13.9 – The original image

We have tiled the grayscale image across the depth dimension so that it can be consumed by the Captum methods, which expect a 3-channel image.

Next, we will actually apply some of Captum's interpretability methods to the forward pass of the prepared grayscale image through the pretrained handwritten digits classification model.

Exploring Captum's interpretability tools

In this section, we will be looking at some of the model interpretability methods offered by Captum.

One of the most fundamental methods of interpreting model results is by looking at saliency, which represents the gradients of the output (class 0, in this example) with respect to the input (that is, the input image pixels). The larger the gradients with respect to a particular input, the more important that input is. You can read more about how these gradients are exactly calculated in the original saliency paper [13.4] . Captum provides an implementation of the saliency method:

1. In the following code, we use Captum's `Saliency` module to compute the gradients:

```
saliency = Saliency(model)
gradients = saliency.attribute(captum_input, target=sample_targets[0].item())
gradients = np.reshape(gradients.squeeze().cpu().detach().numpy(), (28, 28, 1))
_ = viz.visualize_image_attr(gradients, orig_image, method="blended_heat_map",
sign="absolute_value",
show_colorbar=True, title="Overlayed Gradients")
```

This should output the following:

Figure 13.10 – Overlayed gradients

In the preceding code, we reshaped the obtained gradients to size `(28,28,1)` in order to overlay them on the original image, as shown in the preceding diagram. Captum's `viz` module takes care of the visualizations for us. We can further visualize only the gradients, without the original image, using the following code:

```
plt.imshow(np.tile(gradients/(np.max(gradients)), (1,1,3)));
```

We will get the following output:

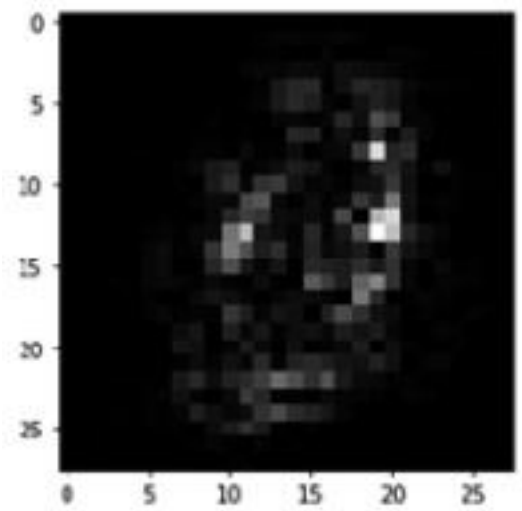

Figure 13.11 – Gradients

As you can see, the gradients a re spread across those pixel regions in the image that are likely to contain the digit `0`.

1. Next, using a similar code fashion, we will look at another interpretability method – integrated gradients. With this method, we will look for **feature attribution** or **feature importance**. That is, we'll look for what pixels are important to use when making predictions. Under the integrated gradients technique, apart from the input image, we also need to specify a baseline image, which is usually set to an image with all of the pixel values set to zero.

An integral of gradients is then calculated with respect to the input image along the path from the baseline image to the input image. Details of the implementation of integrated gradients technique can be found in the original paper [13.5] . The following code uses Captum's `IntegratedGradients` module to derive the importance of each input image pixel:

```
integ_grads = IntegratedGradients(model)
attributed_ig, delta=integ_grads.attribute(captum_input, target=sample_targets[0],
baselines=captum_input * 0, return_convergence_delta=True)
attributed_ig = np.reshape(attributed_ig.squeeze().cpu().detach().numpy(), (28, 28, 1))
_ = viz.visualize_image_attr(attributed_ig, orig_image,
method="blended_heat_map",sign="all",show_colorbar=True, title="Overlayed Integrated
Gradients")
```

Copy Explain

This should output the following:

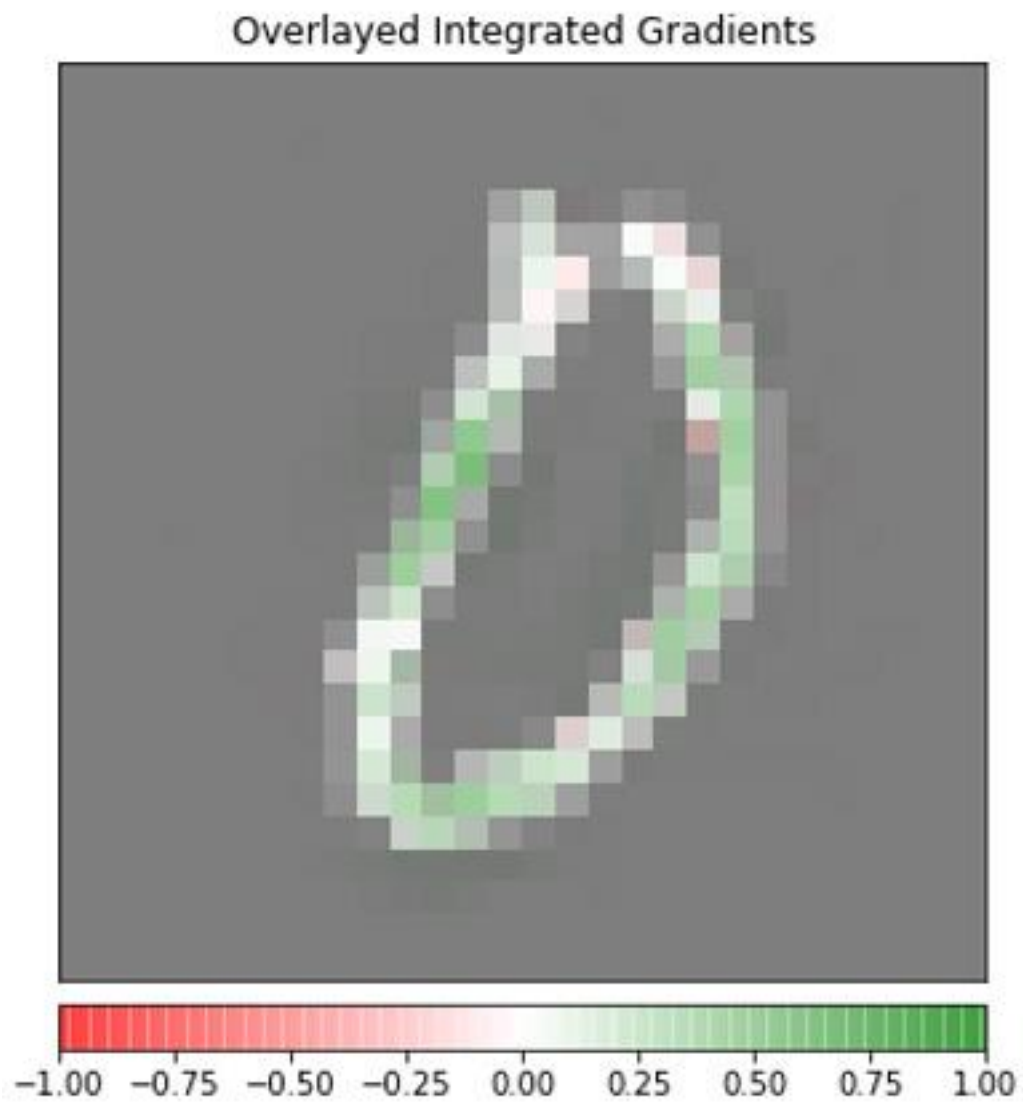

Figure 13.12 – Overlayed integrated gradients

As expected, the gradients are high in the pixel regions that contain the digit `0`.

1. Finally, we will look at yet another gradient-based attribution technique, called **deeplift**. Deeplift also requires a baseline image besides the input image. Once again for the baseline, we use an image with all the pixel values set to zero. Deeplift computes the change in non-linear activation outputs with respect to the change in input from the baseline image to the input image (*Figure 13.9*). The following code uses the `DeepLift` module provided by Captum to compute the gradients and displays these gradients overlayed on the original input image:

```
deep_lift = DeepLift(model)
attributed_dl = deep_lift.attribute(captum_input, target=sample_targets[0],
baselines=captum_input * 0, return_convergence_delta=False)
attributed_dl = np.reshape(attributed_dl.squeeze(0).cpu().detach().numpy(), (28, 28,
1))
_ = viz.visualize_image_attr(attributed_dl, orig_image,
method="blended_heat_map",sign="all",show_colorbar=True, title="Overlayed DeepLift")
```

Copy Explain

You should see the following output:

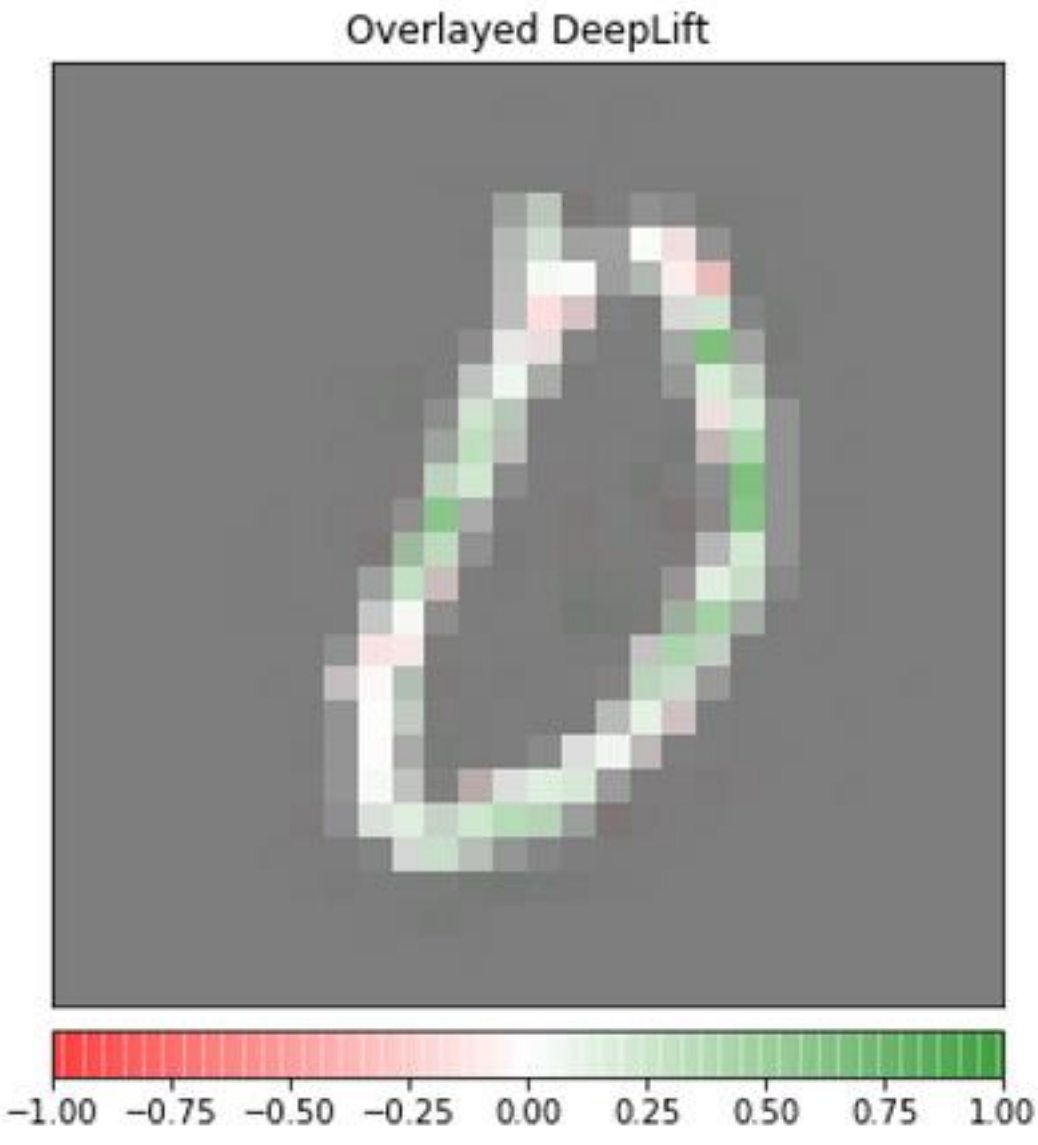

Figure 13.13 – Overlayed deeplift

Once again, the gradient values are extreme around the pixels that contain the digit `0`.

This brings us to the end of this exercise and this section. There are more model interpretability techniques provided by Captum, such as *LayerConductance*, *GradCAM*, and *SHAP [13.6]* . Model interpretability is an active area of research, and hence libraries such as Captum are likely to evolve rapidly. More such libraries are likely to be developed in the near future, which will enable us to make model interpretability a standard component of the machine learning life cycle.

Summary

In this chapter, we have briefly explored how to explain or interpret the decisions made by deep learning models using PyTorch.

In the next chapter of this book, we will learn how to rapidly train and test machine learning models on PyTorch – a skill that is useful for quickly iterating over various machine learning ideas. We will also discuss a few deep learning libraries and frameworks that enable rapid prototyping with PyTorch.

Previous Chapter

Next Chapter