6 Convolutional Neural Networks

In the previous chapters, we introduced how Deep Neural Networks (DNNs) can help us learn complex feature interactions, and how Recurrent Neural Networks (RNNs) can be used to encode patient event sequences. However, patient data often involves multiple modalities that can also present a grid-like structure. These modalities include physiological signals such as vital signs, electroencephalogram (EEG) and electrocardiogram (ECG) signals, textual information like admission notes and discharge summaries, and imaging data such as X-rays or CT scans. We are excited to explore Convolutional Neural Networks (CNNs) in this chapter, which specialize in analyzing grid-based data such as time series and imaging data.

In this section, we will discuss the key concepts of CNNs, which include the convolutional operation, the reasons for using CNNs, and the pooling operation. Additionally, we will introduce some of the popular CNN architectures. Finally, we will explore one specific healthcare applications of CNNs, X-ray classification.

6.1 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs), drawing inspiration from the human visual system, have gained significant prominence owing to their exceptional effectiveness in processing image and time series data. Fukushima & Miyake [14] proposed Neocognitron in 1980, which could be regarded as the predecessor of CNN. In 1989, LeCun et al. [15] established the modern framework of CNN to classify handwritten digits from raw pixels. CNNs provide specialized architectures adept at discerning patterns within grid-like data structures. Examples include 1D grid-like data such as EEG signals, 2D grid-like data such as chest X-Ray images, and 3D grid-like data such as CT scans. Figure 6.1 provides several examples of grid-like data that can be used with CNNs.

Consider the example of analyzing X-ray data using machine learning models to make classification decisions. These models interpret the images purely as arrays or matrices of numbers, without understanding the essence of the image. One possible solution is to flatten the 2D image (e.g., 224x224) into a 1D array (e.g., 50,176) and use a simple DNN architecture for modeling. However, the naive application of DNN models is not effective in image and time series applications, because the DNN method treats each pixel as an individual feature, resulting in too many parameters and ignoring the spatial relations of those pixels, where neighboring pixels usually

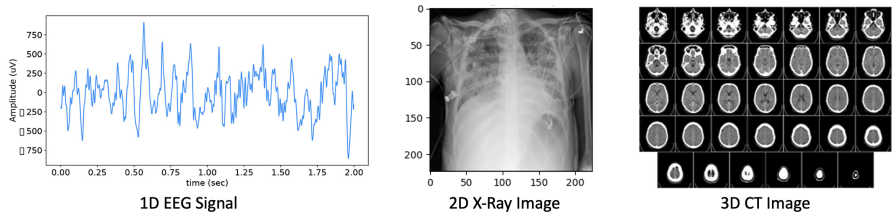

Figure 6.1 Examples of different formats of grid-like medical data that can be used with CNNs.

have similar color intensities. In this section, we will show how convolution operations imitate the human visual system's processing of images, opening the door to powerful new CNN architectures.

6.1.1 Convolution Operation

Let us continue with the 2D image data example and learn about the convolution operation. An illustration is shown in Figure 6.2.

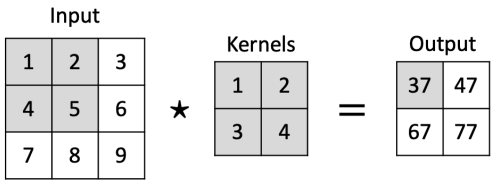

Figure 6.2 An example of 2D convolution.

At a higher level, convolution involves sliding a small matrix, called a "kernel" or "filter", over an input image and computing dot products to produce a new matrix. Mathematically, the convolution operation for a 2D input matrix I and a kernel K is defined as:

$$(I * K)(x, y) = \sum_i \sum_j I(x + i, y + j) \cdot K(i, j), \tag{6.1}$$

where

- I is the input matrix, representing the 2D image data;
- K is the kernel or filter, a smaller 2D matrix that we slide over the input image;
- (x, y) are the coordinates in the output matrix (or feature map). For each position (x, y) in the output, we align the top-left corner of the kernel K with the (x, y) position of the input image I;
- $I(x + i, y + j)$ represents the value of the input image at a specific position. As we slide the kernel over the image, $(x + i, y + j)$ corresponds to the coordinates in

the input image that are currently under the kernel. Here, $x + i$ and $y + j$ are the coordinates in the input image I that align with the current kernel element $K(i, j)$;

- The Summations $\sum_i \sum_j$ indicate that we compute the dot product by summing over all elements of the kernel. For each position of the kernel on the image, we multiply each element of the kernel $K(i, j)$ with the corresponding element of the image $I(x + i, y + j)$ and sum all these products. This sum gives us a single value for each position (x, y) in the output matrix.

In summary, during convolution, we slide the kernel over each position of the input image, perform element-wise multiplications between the kernel and the corresponding overlapping region of the input image, and sum these values to produce each element of the output matrix. This operation effectively filters the input image to extract features, patterns, or information encoded in the spatial structure of the data. For example, in Figure 6.2, the shaded portions are input and kernel elements used to calculate the first output element: $37 = 1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4$. Similarly, the rest of the output elements are calculated as follows,

$$47 = 2 \times 1 + 3 \times 2 + 5 \times 3 + 6 \times 4, \tag{6.2}$$

$$67 = 4 \times 1 + 5 \times 2 + 7 \times 3 + 8 \times 4, \tag{6.3}$$

$$77 = 5 \times 1 + 6 \times 2 + 8 \times 3 + 9 \times 4. \tag{6.4}$$

The kernel moves across an image, multiplies its values with the corresponding image values each time it moves, and aggregates them to produce a new output. This iterative process extracts diverse visual characteristics from the image.

Sobel filter for edge detection

The convolution operation's beauty is its ability to detect specific features in the input, depending on the kernel. Back to the old days, researchers handcrafted customized kernels to detect specific features. An example is the Sobel filter, which is commonly used in classic image processing. Sobel filter is a discrete differentiation operator. It computes an approximation of the gradient of the image intensity function. By using the Sobel filter, one can detect edges within images, specifically highlighting the regions where rapid intensity changes occur. For example, the Sobel filter for detecting horizontal edges is represented by the following kernel:

$$S_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

When this filter is convolved with an image, it highlights the horizontal edges, as can be seen in Figure 6.3. Specifically, it responds maximally to edges running vertically in the image since it captures the vertical change in intensity. In contrast, for detecting vertical edges, the Sobel filter is:

$$S_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}.$$

This filter, when convolved with an image, accentuates the vertical edges. It responds maximally to edges running horizontally in the image, as it captures the horizontal change in intensity. We show an illustration of Sobel edge detectors in Figure 6.3.

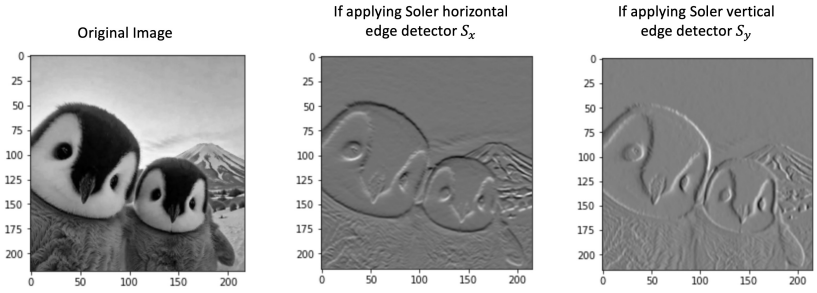

Figure 6.3 Sobel horizontal and vertical edge detectors

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt

img = Image.open("penguin.jpg") # load any image
convert_img = transforms.ToTensor()
img = convert_img(img)[0] # tensor of shape (height, weight)

filters1 = torch.Tensor([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
#reshape img to fit the input shape for F.conv2d
img_conv1 = F.conv2d(img.unsqueeze(0).unsqueeze(0), \
                        filters1.unsqueeze(0).unsqueeze(0))[0,0]

filters2 = torch.Tensor([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
img_conv2 = F.conv2d(img.unsqueeze(0).unsqueeze(0), \
                        filters2.unsqueeze(0).unsqueeze(0))[0,0]
```

In the above implementation, the unsqueeze(0) method is used twice on img. This is done to reshape the image tensor to match the expected input shape for the $F.conv2d$ function in PyTorch. The $F.conv2d$ function expects a 4-dimensional tensor as input, with shape (batchsize, numchannels, height, width). The original img tensor is likely 2-dimensional (since it represents a single-channel grayscale image), with shape (height, width). The first $.unsqueeze(0)$ adds a dimension at the front, representing the batch size (even though there is only one image in this batch). The second $.unsqueeze(0)$ adds a dimension for the number of channels (again, there is only one channel in this case). This reshaping results in a tensor with shape (1, 1, height, width).

In the early days of image processing and computer vision, convolution kernels played a vital role in feature extraction. However, these kernels were designed heuristically, and experts crafted them manually based on their experiences and insights to capture specific image characteristics. Fortunately, with the introduction of Con-

volutional Neural Networks (CNNs), the concept of learnable parameters replaced manually designed kernels. This means that during the training process, the network itself can determine the optimal values for these kernels based on the data, eliminating the need for human intervention in kernel design. This often results in more robust and adaptable feature extraction mechanisms.

The code snippets we provided in this subsection and later part will be collectively provided at this our GitHub at [1].

6.1.2 Motivation for CNNs

Before we delve into the CNNs, let us take a step back and discuss the motivation for CNNs: Why do we need such convolution operations? Why can't we just use the DNNs?

Consider the task of training a Deep Neural Network (DNN) on two-dimensional images represented by $\mathbf{X} \in \mathbb{R}^{w \times h}$, where w and h denote the width and height of the image, respectively. A Multi-layer Perceptron (MLP) treats each pixel in the input image as an independent feature. However, this approach is not suitable for image data as images usually have spatial hierarchies and local patterns. Pixels that are close to each other in an image form structures such as edges, textures, and shapes, which are further combined to form more complex patterns. MLPs do not inherently capture this localized and hierarchical nature of image data.

In contrast, Convolutional Neural Networks (CNNs) are designed to recognize and leverage the spatial hierarchies in images, making them more appropriate for image classification tasks. Two fundamental motivations underpin the design and success of CNNs: Translation Invariance and Locality.

Translation Invariance

Translation Invariance ensures that regardless of the position of a feature in an image, the network can detect and recognize it. In other words, if an object appears in different parts of an image, a CNN is still capable of identifying it without any loss of performance. This is only possible if the kernel parameter $\mathbf{W} \in \mathbb{R}^{w \times h}$ does not depend on the location (i, j). In other words, the CNN model slides the same kernel $\mathbf{W}$ across the entire input image. This is crucial for tasks like image recognition, where the exact position of an object in an image shouldn't affect the network's ability to recognize it.

Locality

Locality emphasizes the importance of capturing local features in the data. Instead of focusing on the entire image at once, CNNs assess small, localized regions. This allows the network to detect intricate patterns and structures, such as edges and textures, which are then pieced together to form a more holistic understanding of the image. Under this motivation, at each location (i, j), we can only focus on a small location region around the location (i, j) instead of the entire image. As a result, the kernel parameter $\mathbf{W}$ can reduce its size to $\mathbb{R}^{k_w \times k_h}$ where k_w and k_h denote the local range.

[1] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap4-CNN/notebook

These two motivations significantly reduce the total number of learnable parameters. The resulting operation is called a convolutional layer. CNNs stack multiple such convolutional layers together to extract complex representations from an image.

6.1.3 Convolutional Layer

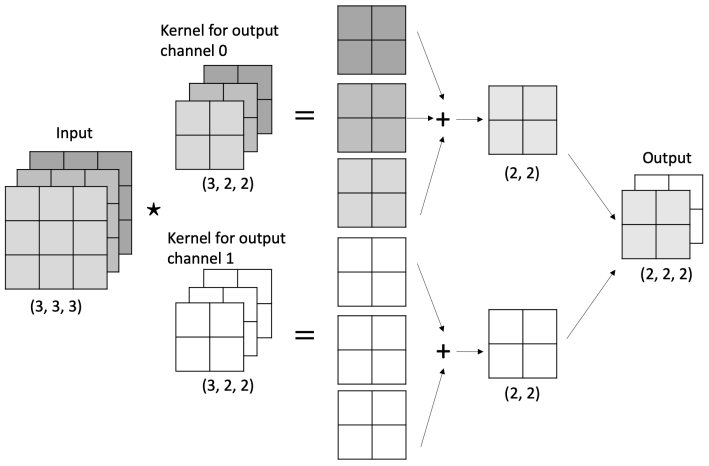

Figure 6.4 Convolution for multiple input and output channels.

We will now introduce the convolutional layer formally. Let's say we have an input image with a shape of $c_i \times w_i \times h_i$ (i refers to "input" index), where c_i represents the number of color channels (3 for RGB images and 1 for grey images), w_i represents the width, and h_i represents the height of the image. We have c_i kernels of a shape of $w_k \times h_k$, and the output to have a shape of $c_o \times w_o \times h_o$. The convolutional layer will maintain a separate set of learnable kernels $\mathbf{W} \in \mathbb{R}^{c_i \times w_k \times h_k}$ for each output channel. The result for each output channel is calculated from the convolution kernels corresponding to that output channel and takes in all input channels.

Figure 6.4 shows an example of an input image of shape 3x3x3, output of shape 2x2x2, and a kernel of size 2x2. Typically, the convolutional layer also involves the **padding** and **striding** operations.

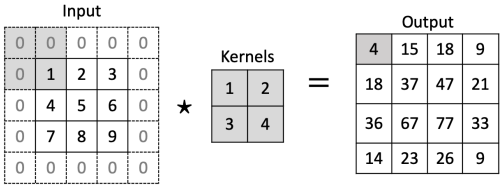

Figure 6.5 The padding operation.

Padding:

Padding is a technique used in CNN to add extra rows and columns on the boundary of an input matrix or image. These additional rows and columns are typically filled with zeros and help to ensure that the spatial dimensions of the feature map remain consistent after the convolution process is applied. Padding can also be used to achieve desired output dimensions.

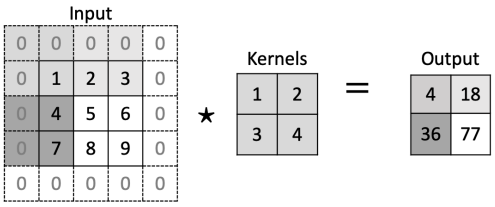

Figure 6.6 The striding operation (stride is 2 here).

Striding

The term stride refers to the distance traveled by a convolutional filter as it moves over the input matrix. If the stride is set to one, the filter moves one unit at a time, which corresponds to one pixel in a 2D image. On the other hand, a stride of two skips every alternative cell in the input matrix. By adjusting the stride value, we can control the spatial resolution of the output, where larger strides produce coarser representations, effectively reducing the size of the output feature map.

Below, we showcase how to define a convolutional layer in PyTorch.

```
import torch.nn as nn

# Define a Conv2d layer
conv_layer = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1,
                                                    padding=1)
# 'in_channels' refers to the depth of the input (e.g., 3 for RGB images)
# 'out_channels' is the number of filters/kernels
# 'kernel_size' is the size of each filter
# 'stride' controls the step size when moving the filter
# 'padding' is added to the input to control the spatial size of the output

# Assuming we have a single-channel image (e.g., grayscale) with a size of
                                                    28x28 pixels
# Create a dummy input tensor with shape (batch_size, channels, height, width)
# Here, batch_size=1, channels=1, height=width=28
input_tensor = torch.randn(1, 1, 28, 28)

# Apply the conv_layer to the input_tensor
output_tensor = conv_layer(input_tensor)

# Print the shape of the output tensor
print("Output shape:", output_tensor.shape)
# Output shape will typically be (1, 32, 28, 28) if padding is set to 'same' or
```

```
                                    equivalent, keeping the height and
                                width unchanged, and the channels equal
                                to out_channels of the conv_layer
```

The above code defines a convolutional layer with 1 input channel, 32 output channels (or filters), a 3x3 kernel size, a stride of 1, and padding of 1.

6.1.4 Pooling Layer

In CNNs, pooling is a crucial down-sampling operation aimed at reducing the spatial dimensions of feature maps, thereby making the network more computationally efficient and less prone to over-fitting.

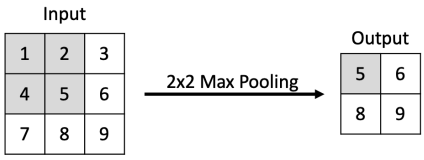

Figure 6.7 An example of max pooling.

Pooling works by sliding a window, often referred to as a "pooling window", over the image and aggregating the values within this window into a single value. The most common pooling methods are max pooling and average pooling. Max pooling takes the maximum value from within the window, while average pooling computes the average. Figure 6.7 shows an example of 2x2 max pooling over a 3x3 input image. The shaded portions are input elements used to calculate the first output element: $5 = \max(1, 2, 4, 5)$. Similarly, the rest of the output elements are calculated as follows,

$$6 = \max(2, 3, 5, 6), \tag{6.5}$$

$$8 = \max(4, 5, 7, 8), \tag{6.6}$$

$$9 = \max(5, 6, 8, 9). \tag{6.7}$$

By performing pooling, the network retains the most essential features while discarding redundant spatial information, leading to a condensed representation. Pooling layers interspersed between convolutional layers ensure that deeper layers of the network operate on progressively smaller spatial dimensions, focusing more on high-level abstract features rather than granular details.

When humans look at images, we read the image part by part and tend to forget irrelevant details gradually but keep the most essential information from each part to get a holistic view of the whole image. Below, we showcase how to define a pooling layer in PyTorch. We also apply the same 2x2 MaxPooling repetitively on the same image and the effect is shown in Figure 6.8. The details of the image disappear gradually while the main object stays recognizable.

```
1 import torch
```

```python
import torchvision.transforms as transforms
from PIL import Image
import matplotlib.pyplot as plt

# Load an image
image_path = 'penguin.jpg'  # Update this to your image path
image = Image.open(image_path)

# Transform the image to tensor
transform = transforms.ToTensor()
image_tensor = transform(image).unsqueeze(0)  # Add batch dimension

# Define max pooling operation
pool = torch.nn.MaxPool2d(kernel_size=2)

# Apply max pooling operations
pooled_image1 = pool(image_tensor)
pooled_image2 = pool(pooled_image1)
pooled_image3 = pool(pooled_image2)

# Function to plot images
def plot_image(tensor, title):
    plt.figure()
    tensor = tensor.squeeze(0)  # Remove batch dimension
    tensor = transforms.ToPILImage()(tensor)
    plt.imshow(tensor)
    plt.title(title)

# Plot original and pooled images
plot_image(image_tensor, 'Original Image')
plot_image(pooled_image1, '1st Pooling')
plot_image(pooled_image2, '2nd Pooling')
plot_image(pooled_image3, '3rd Pooling')

plt.show()
```

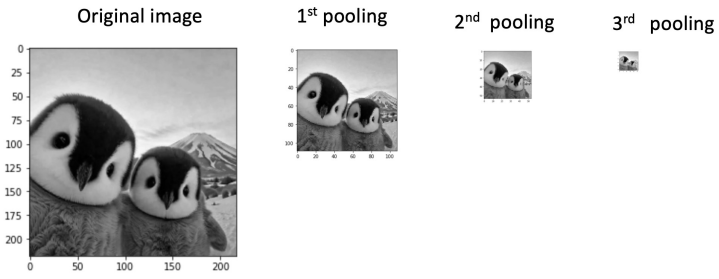

Figure 6.8 Apply 2x2 max pooling consecutively.

6.1.5 CNN for 1D and 3D Grid-like Data

Besides 2D images, CNNs can also be applied to 1D and 3D grid-like data.

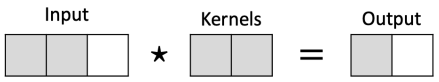

Figure 6.9 An example of the convolution operation on 1D data.

For 1D grid-like data, such as EEG, ECG, or vital signs, convolution involves sliding a one-dimensional kernel over the input data from left to right. At each step, the kernel weights are multiplied by the corresponding input values, and the results are summed up to produce a single output value. The shaded portions are input and kernel elements are used to calculate the first output element,

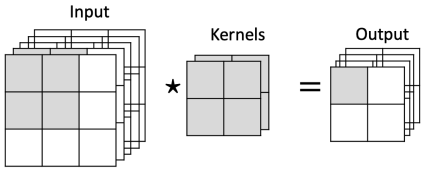

Figure 6.10 An example of the convolution operation on 3D data.

For 3D grid-like data, such as MRI and CT scans, the kernel itself is a three-dimensional cube, which moves across the width, height, and depth (or time) dimensions of the input data. At each location, the convolution operation remains the same as 1D and 2D convolution: element-wise product and sum. The shaded portions are input and kernel elements used to calculate the first output element.

6.2 Modern CNNs

Many different CNN architectures have been proposed over the years where the main components are the convolutional layers and pooling layers. Next, we introduce a few popular CNN architectures.

6.2.1 LeNet

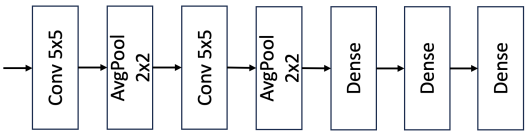

Figure 6.11 An illustration of the LeNet framework.

LeNet [16] is one of the earliest convolutional neural networks that successfully

demonstrated the power of convolutional layers. LeNet's design was relatively simple, consisting of alternating convolutional and subsampling (pooling) layers, followed by fully connected layers, setting a foundational pattern for future CNN architectures. Its key innovations included the use of convolutional layers for efficient feature extraction, shared weights for reduced model complexity, pooling layers for improved translation invariance, and an end-to-end trainable system combining feature extraction with classification. The adoption of backpropagation for training enabled the network to learn complex patterns effectively.

Demonstrated in real-world applications like postal digit recognition, LeNet laid the foundational principles for modern CNN architectures and significantly propelled the field of deep learning forward, especially in computer vision.

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # Convolutional layer (sees 32x32x1 image tensor)
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5,
                                                stride=1, padding=2)
        # Convolutional layer (sees 14x14x6 image tensor)
        self.conv2 = nn.Conv2d(6, 16, 5, stride=1)
        # Average pooling layer with a 2x2 window
        self.pool = nn.AvgPool2d(kernel_size=2, stride=2)
        # Fully connected layer (sees 5x5x16 image tensor)
        self.fc1 = nn.Linear(16*5*5, 120)
        # Fully connected layer (sees 120-dimensional vector)
        self.fc2 = nn.Linear(120, 84)
        # Output layer (sees 84-dimensional vector)
        self.fc3 = nn.Linear(84, 10)  # 10 output classes for MNIST

    def forward(self, x):
        # Apply a ReLU activation function to convolutional layer
        x = F.relu(self.conv1(x))
        # Apply average pooling
        x = self.pool(x)
        # Apply a ReLU activation function to convolutional layer
        x = F.relu(self.conv2(x))
        # Apply average pooling
        x = self.pool(x)
        # Flatten image input
        x = x.view(-1, 16*5*5)
        # Fully connected layer with a ReLU activation function
        x = F.relu(self.fc1(x))
        # Fully connected layer with a ReLU activation function
        x = F.relu(self.fc2(x))
        # Final output layer
        x = self.fc3(x)
        return x


model = LeNet()
```

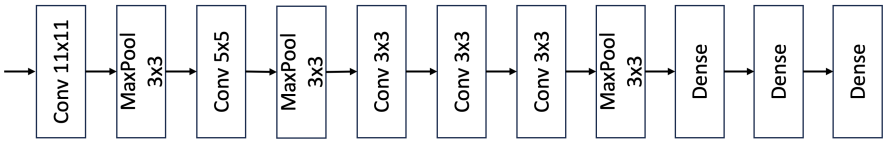

Figure 6.12 An illustration of the AlexNet framework.

6.2.2 AlexNet

AlexNet [17], a seminal model in deep learning, significantly outperformed competitors in the 2012 ImageNet challenge, largely due to its deeper architecture and pioneering techniques. It was one of the first models to incorporate the ReLU activation function, which enhanced training efficiency and resolved vanishing gradient issues. The model's use of GPU computing was crucial for managing its larger scale, allowing for effective parallel computation. AlexNet also introduced key regularization strategies like dropout and overlapping max pooling, which substantially reduced overfitting. Complemented by data augmentation methods, these innovations greatly improved the model's ability to generalize, marking a turning point in the application of CNNs in computer vision and sparking extensive research in the field of deep learning.

```
import torchvision.models as models
alexnet_model = models.alexnet()
```

6.2.3 VGG

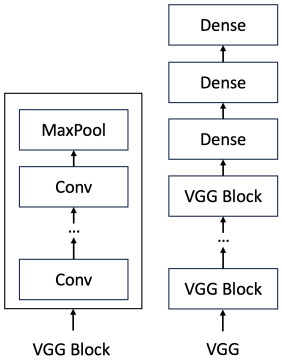

Figure 6.13 An illustration of the VGG framework.

VGGNet [18] , known for its simplicity and depth, demonstrated the effectiveness of deep CNNs using small (3x3) convolution filters. Its architecture, featuring multiple layers of these small filters, achieved a receptive field akin to larger filters but with fewer parameters, enhancing learning capacity. The model's uniform structure,

with up to 19 layers, allowed for capturing complex features at different abstraction levels, significantly improving image recognition performance. This approach not only showcased the importance of network depth but also influenced future CNN designs. VGGNet's notable success in the 2014 ImageNet challenge validated its architectural effectiveness.

```
vgg_model = models.vgg16()
```

6.2.4 GoogLeNet

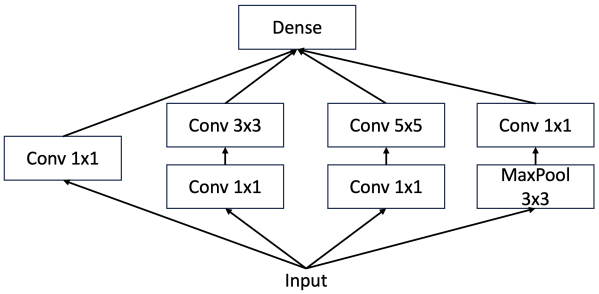

Figure 6.14 An illustration of the GoogLeNet framework.

GoogLeNet [19] allows the network to automatically learn the appropriate filter sizes (1x1, 3x3, 5x5 convolutions) alongside pooling layers, all concatenated into a single output. This innovative design enabled the network to efficiently capture information at various scales while controlling the computational cost, allowing GoogLeNet to be deeper and more accurate without a substantial increase in parameters. This architecture significantly improved performance in image recognition tasks and influenced the design of subsequent deep learning models.

```
googlenet_model = models.googlenet()
```

6.2.5 ResNet

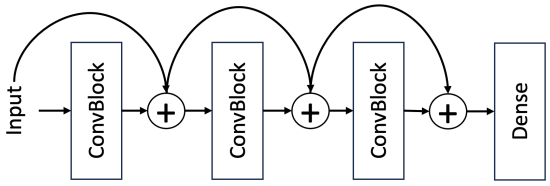

Figure 6.15 An illustration of the ResNet framework.

ResNet, short for Residual Network [20], addressed the problem of training very deep neural networks through the introduction of residual blocks. These blocks allow for the training of networks that are much deeper than was previously possible by using identity shortcut connections that skip one or more layers. The key idea is that these shortcuts enable the direct propagation of gradients through the network, mitigating the vanishing gradient problem and allowing the network to learn identity mappings as needed, thereby improving training efficiency and effectiveness for deep networks.

```
resnet_model = models.resnet50()
```

6.2.6 DenseNet

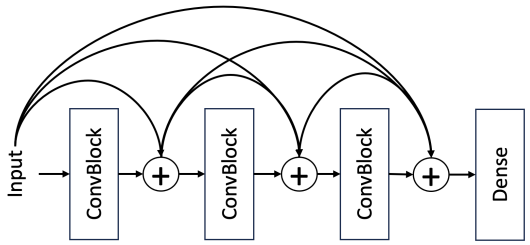

Figure 6.16 An illustration of the DenseNet framework.

DenseNet, short for Densely Connected Convolutional Networks [21], further innovated in the direction of improving information flow and feature reuse in deep networks by introducing a densely connected architecture. In DenseNet, each layer receives inputs from all preceding layers and passes its own feature maps to all subsequent layers, leading to a highly interconnected network. This design encourages feature reuse, significantly reduces the number of parameters (making the network more efficient), and improves gradient flow through the network, which is beneficial for model training.

```
densenet_model = models.densenet121()
```

6.3 X-ray disease classification with CNN

In this section, we will explore how to use PyHealth to analyze chest X-ray images and classify them into various chest diseases. Again, PyHealth will help to handle the key steps in the chest disease classification pipeline: data processing, task definition, model initialization, training, and inference.

6.3.1 Background

Chest diseases such as Lung Opacity, COVID-19, and Viral Pneumonia present a significant public health challenge due to their high prevalence and the severity of

their impact on the human respiratory system. The early detection and classification of these diseases using chest X-rays are vital for initiating effective treatment protocols and improving patient outcomes. The radiographic examination of the chest provides a non-invasive method to investigate the underlying pulmonary conditions, but the interpretation of these images requires a high level of expertise and is subject to inter-observer variability.

Convolutional Neural Networks (CNNs) have emerged as a powerful tool for analyzing visual data and have shown remarkable success in medical image analysis. The application of CNNs in the classification of chest diseases involves training the network using a vast dataset of annotated chest X-ray images, where each image is labeled with one of the disease categories: Normal, Lung Opacity, COVID-19, or Viral Pneumonia.

The CNN model learns to identify distinctive patterns and features within the X-ray images that correlate with the specific diseases. By learning from multiple layers of convolution and pooling operations, the network can discern subtle variations in lung tissue opacity, patterns of fluid accumulation, and other radiographic signs that are indicative of these conditions. This automated and sophisticated analysis holds the promise of providing rapid, consistent, and accurate disease classification, which is crucial in a clinical setting, especially during high-demand periods such as the COVID-19 pandemic.

6.3.2 Step 1: Dataset Processing

The dataset we will use here is the *COVID-19 CXR dataset* collected by a team of researchers from Qatar University, Doha, Qatar, and the University of Dhaka, Bangladesh along with their collaborators from Pakistan and Malaysia with X-ray images for COVID-19 positive cases along with Normal and Viral Pneumonia images. The whole dataset contains 3616 COVID-19 positive cases along with 10,192 Normal, 6012 Lung Opacity (Non-COVID lung infection), and 1345 Viral Pneumonia images and corresponding lung masks.

The initial step involves loading the data into PyHealth, for which, we will directly use the $pyhealth.datasets.COVID19CXRDataset$ API. Users are required to specify the root directory where the raw dataset is stored. PyHealth will handle the dataset processing automatically.

```
from pyhealth.datasets import COVID19CXRDataset


root = "/content/COVID-19_Radiography_Dataset"
base_dataset = COVID19CXRDataset(root)
base_dataset.stat()
"""
Output:
Statistics of COVID19CXRDataset:
Number of samples: 21165
Number of classes: 4
Class distribution: Counter({'Normal': 10192, 'Lung Opacity': 6012, 'COVID':
                                        3616, 'Viral Pneumonia': 1345})
```

```
"""

base_dataset.patients[0]
"""
Output:
{'path': '/content/COVID-19_Radiography_Dataset/COVID/images/COVID-1.png',
 'url': 'https://sirm.org/category/senza-categoria/covid-19/',
 'label': 'COVID'}
"""
```

6.3.3 Step 2: Dataset

The next step is to define the machine learning task. This step instructs the package to generate a list of samples with the desired features and labels based on the data for each individual patient. Please note that in this dataset, patient identification information is not available. Therefore, we will assume that each chest X-ray belongs to a unique patient.

For this dataset, PyHealth offers a default task specifically for chest X-ray classification. This task takes the image as input and aims to predict the chest diseases associated with it.

```
base_dataset.default_task
"""
Output:
COVID19CXRClassification(task_name='COVID19CXRClassification', input_schema={'
                                        path': 'image'}, output_schema={'label
                                        ': 'label'})
"""

sample_dataset = base_dataset.set_task()
```

Additionally, let's define a transformation function for the X-ray image. This function will resize the chest X-ray image to 224x224 pixels, convert it to grayscale, and normalize the pixel values. These transformation steps ensure that all the images have the same formats and are compatible with the CNN model.

```
from torchvision import transforms

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.Grayscale(),
    transforms.Normalize(mean=[0.5862785803043838], std=[0.27950088968644304])
])


def encode(sample):
    sample["path"] = transform(sample["path"])
    return sample


sample_dataset.set_transform(encode)
```

Here is an example of a single image sample, represented as a dictionary. The dictionary contains keys for feature names, label names, and other metadata associated with the sample.

```
sample_dataset[0]
"""
Output:
{'path': tensor([[[-0.0679, -1.8486, -2.0976,  ..., -2.0858, -1.9487, -0.6912],
          [-1.5782, -2.0419, -2.0976,  ..., -2.0847, -2.0557, -1.7701],
          [-2.0739, -2.0895, -2.0976,  ..., -2.0816, -2.0286, -1.8701],
          ...,
          [-1.1680, -1.2323, -1.0222,  ..., -2.0809, -2.0149, -1.9473],
          [-0.8756, -1.1191, -0.9243,  ..., -2.0930, -1.9579, -1.4143],
          [-0.1718, -0.9416, -0.8654,  ..., -2.0953, -1.8020, -0.2139]]]),
 'url': 'https://sirm.org/category/senza-categoria/covid-19/',
 'label': 'COVID'}
"""
```

We can also check the input and output schemas, which specify the data types of the features and labels.

```
sample_dataset.input_schema
"""
Output:
{'path': <pyhealth.datasets.featurizers.image.ImageFeaturizer at 0x7884ae6fc430
                                                >}

sample_dataset.output_schema
{'label': ValueFeaturizer()}
"""
```

Below, we plot the sample number distributions over all classes, and visualize one sample for each class.

```
from collections import defaultdict
import matplotlib.pyplot as plt

label_counts = defaultdict(int)
for sample in sample_dataset.samples:
    label_counts[sample ["label"]] += 1
print(label_counts)
plt.bar(label_counts.keys(), label_counts.values())
```

```
import random

label_to_idxs = defaultdict(list)
for idx, sample in enumerate(sample_dataset.samples):
    label_to_idxs[sample["label"]].append(idx)

fig, axs = plt.subplots(1, 4, figsize=(15, 3))
for ax, label in zip(axs, label_to_idxs.keys()):
    ax.set_title(label, fontsize=15)
    idx = random.choice(label_to_idxs[label])
    sample = sample_dataset[idx]
    image = sample["path"][0]
    ax.imshow(image, cmap="gray")
```

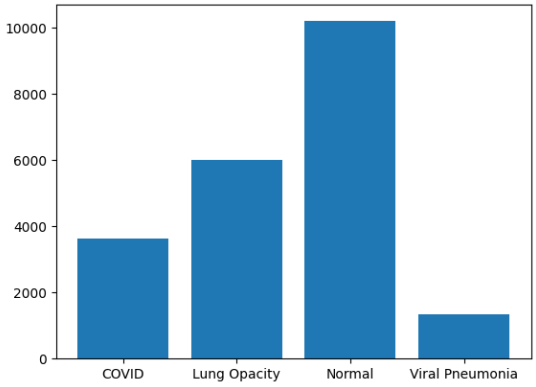

Figure 6.17 Number of samples per each disease class

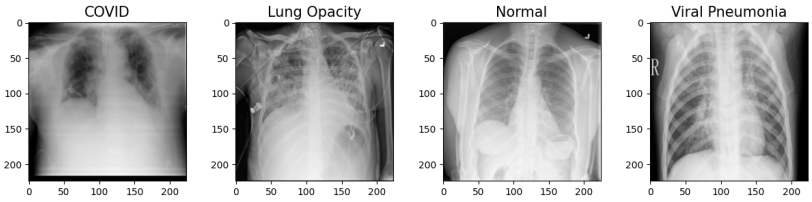

Figure 6.18 Visualization of each disease class

Finally, we will split the entire dataset into training, validation, and test sets using the ratios of 70%, 10%, and 20%, respectively. We will then obtain the corresponding data loaders for each set.

```
from pyhealth.datasets import split_by_sample

train_dataset, val_dataset, test_dataset = split_by_sample(
    dataset=sample_dataset,
    ratios=[0.7, 0.1, 0.2]
)

from pyhealth.datasets import get_dataloader

train_dataloader = get_dataloader(train_dataset, batch_size=32, shuffle=True)
val_dataloader = get_dataloader(val_dataset, batch_size=32, shuffle=False)
test_dataloader = get_dataloader(test_dataset, batch_size=32, shuffle=False)
```

6.3.4 Step 3. Define the Model

Next, we will define the deep learning model we want to use for our task. PyHealth supports all major vision models available in the Torchvision package. You can load any of these models using the "model_name" argument.

```python
from pyhealth.models import TorchvisionModel

resnet = TorchvisionModel(
    dataset=sample_dataset,
    feature_keys=["path"],
    label_key="label",
    mode="multiclass",
    model_name="resnet18",
    model_config={"weights": "DEFAULT"}
)
```

6.3.5 Step 4. Training

In this step, we will train the model using PyHealth's Trainer class, which simplifies the training process and provides standard functionalities.

```python
from pyhealth.trainer import Trainer

trainer = Trainer(model=resnet)
trainer.train(
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    epochs=1,
    monitor="accuracy"
)
```

6.3.6 Step 5. Evaluation

Lastly, we can evaluate the ResNet model on the test set. This can be done using PyHealth's Trainer.evaluate() function.

```python
print(trainer.evaluate(test_dataloader))
"""
Output:
{
    'accuracy': 0.7184030238601464,
    'f1_macro': 0.7293848976927099,
    'f1_micro': 0.7184030238601463,
    'loss': 0.9326339725376969
}
"""
```

Below, we show a confusion matrix of the trained ResNet model.

```python
from sklearn.metrics import confusion_matrix
import seaborn as sns

y_true, y_prob, loss = resnet_trainer.inference(test_dataloader)
y_pred = y_prob.argmax(axis=1)
cf_matrix = confusion_matrix(y_true, y_pred)
ax = sns.heatmap(cf_matrix, linewidths=1, annot=True, fmt='g')

```

```
ticks_name = resnet.label_tokenizer.convert_indices_to_tokens(list(range(4)))
ax.set_xticklabels(ticks_name)
ax.set_yticklabels(ticks_name)
ax.set_xlabel("Pred")
ax.set_ylabel("True")
```

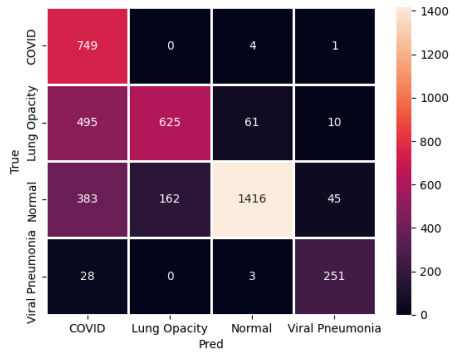

Figure 6.19 Confusion matrix of the CNN predictions

To practice this Chest X-ray image classification task with PyHealth implementation, we have provided the complete pipeline in a public repository at GitHub [2].

Questions

- What is the key difference between CNNs and other deep learning models, such as DNN and RNN? In which scenarios would you prefer CNN over other model architectures?
- What is the convolution operation? Given the following matrix $\mathbf{A}$ and a filter K, what is the output dimension of the convolution results? Could you write down the output?

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \tag{6.8}$$

- What is a Sobel edge detector? What are they used for? Could you use the codes provided in the text and detect the vertical edges and horizontal edges for your own image?
- Could you design other kernel matrix for novel purpose? Such as how to detect the diagonal edges?
- What is a pooling function? What are the pooling functions that we have learned

[2] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap4-CNN/notebook

from the section? Given the same **A** matrix here, could you generate the output for using a 2x2 MaxPooling layer and a 2x2 MeanPooling layer, respectively?

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \tag{6.9}$$

- What other pooling functions could you design? Why do you think this new pooling function is better than MaxPooling or MeanPooling? In which scenarios?
- What is padding? When to use padding in CNN model? Given the same matrix **A** and filter **K**, what is the output of the convolution now if we add zero-padding on four sides of the matrix **A**?
- What is the output of the MaxPooling on **A** if we add zero-padding on four side of **A**?
- what other padding method can you design? In which scenario this new padding method will work better than zero-padding?
- What is stride? How does stride affect the convolution and pooling operations?
- Let us set padding to be 1 on four sides of **A** and use the same convolution kernel **K**. What is the output matrix if the stride is 2?
- Additionally, follow the above question, we also use stride 2 for a MeanPooling layer right after the convolution, what is the output?
- Coding Challenge: let us improve the pipeline in Section 6.3. Here are the additional requirements: (i) set train/val/test split into 80% : 10% : 10%; (ii) we want to use pre-trained resnet-34 for handling this task, please improve the model initialization; (iii) during the training, we want to monitor the metric "F1 macro" and use it to select the best model on validation set.