4 Deep Neural Networks

Deep neural networks are a type of machine learning model that have been around since the 1940s and 1950s [2]. However, it was not until the emergence of big data and hardwares optimized for parallel computing, such as graphical processing units (GPUs), when deep neural network models became more competitive in terms of their speed and accuracy in real-world applications.

The main concept behind deep neural networks is the use of multiple layers of interconnected neurons that can map input features to an output label. For example, the input can be a set of lab tests of a patient, and the output can be the probability of the patient having heart disease. There are many variants of deep neural networks that have been introduced for different types of data such as convolutional neural networks (CNN) for images such as X-ray images, and recurrent neural networks (RNN) for event sequences such as longitudinal electronic health records. We will introduce them one by one in later parts of the book.

In this chapter, we will cover fundamental concepts of neural networks such as neurons, activation functions, input/output layers, and forward/back-propagation. We will also illustrate how these concepts can be utilized to address real-world issues in healthcare. Specifically, we will guide you through constructing a neural network for predicting patient hospital readmission based on patient variables. By the end of this chapter, you will have a practical understanding of deep neural networks, their training procedures, and their applications to real-world healthcare problems.

4.1 Train A Single Neuron

In this section, we introduce the core building block of neural networks, a **neuron**, also known as a **perceptron**.

Example in Healthcare

A neuron can help make simple decisions in healthcare applications. It can become a "hypertension detector" by estimating the risk that a patient has hypertension based on their blood pressure and age. By feeding enough patient cases to the neuron, it can be trained to output an accurate probability estimate of whether a random patient has hypertension. Let us see the details below.

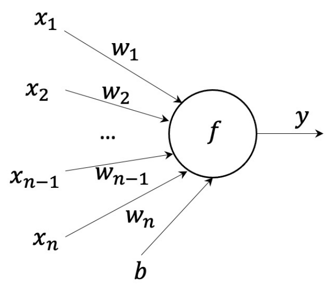

Figure 4.1 Illustration of a single neuron

4.1.1 Structure of a single neuron

A neuron processes a set of features such as blood pressure and age, represented as $\mathbf{x}$, by assigning weights to them, denoted as $\mathbf{w}$, adding a bias, denoted as b, and then passing the result through an activation function to generate an output. The weights and bias in a neuron are determined during the training process. The primary aim of this process is to *identify an optimal set of weights and a bias* that effectively reduces the difference between the predicted output (i.e., the neuron's output) and the actual outcome (i.e., whether the patient has hypertension or not).

Let's define the architecture of a single neuron formally. As illustrated in Figure 4.1, we have an **input feature vector** $[x_1, x_2, \ldots, x_{n-1}, x_n]$, denoted as a vector $\mathbf{x}$. The single neuron has corresponding parameters to weigh each input feature, which are $[w_1, w_2, \ldots, w_{n-1}, w_n]$, also denoted as a **weight vector** $\mathbf{w}$. Usually, a neuron also has a **bias** term b to adjust the weighted inputs. The input features, weights, and bias term are combined to obtain the **pre-activation term** $\sum_{i=1}^{n} x_i \cdot w_i + b$. Finally, after applying the **activation function** $f(\cdot)$ on the pre-activation term, we get the output y of the neuron:

$$y = f(\langle \mathbf{x}, \mathbf{w} \rangle + b) = f\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right). \tag{4.1}$$

There are different types of activation functions, which will be introduced next. For example, an activation function $f(\cdot)$ can transform the pre-activation term into a desirable range (e.g., [0, 1]). High output values mean high activation, while low values indicate the neuron is not activated. Next, we will introduce some common activation functions.

4.1.2 Activation Functions

Activation functions are mathematical functions that convert a scalar value (pre-activation term) into a corresponding scalar activation value within a certain range using a nonlinear transformation. There are several types of activation functions, and some of the most well-known ones are listed in this section. Figure 4.2 provides an illustration of these activation functions.

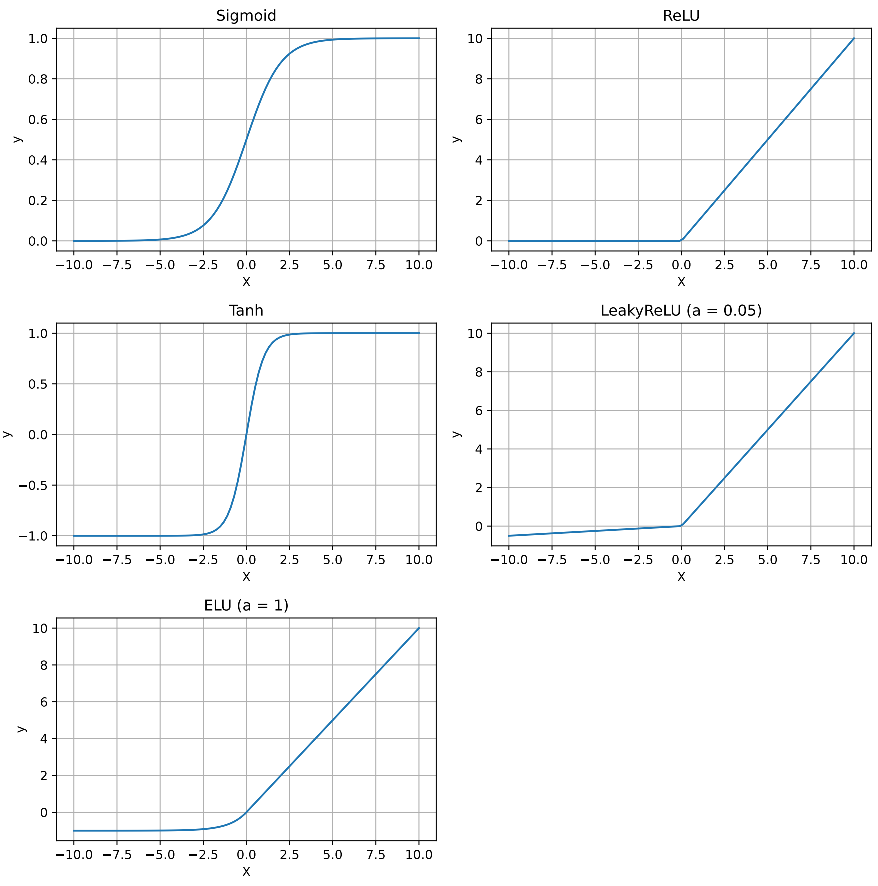

Figure 4.2 Illustration of different activation functions (Sigmoid, ReLU, Tanh, Leaky ReLU, ELU)

- **Sigmoid activation**. The Sigmoid activation function is commonly used for binary classification problems. It can transform a scalar input into an output in the range of $(0, 1)$. The function compresses extreme values by squeezing them towards 1. However, it has a high resolution for small activation values. Since the output range aligns with the probability range $(0, 1)$, the activation values can often be interpreted as predicted probabilities. Additionally, the Sigmoid activation function does not have any parameter.

$$\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.2}$$

As shown in the figure, the sigmoid function can capture the subtle changes near "0" and ignore the big changes in extreme positive or negative regions. The

Python implementation is also straightforward. More information could refer to PyTorch documentation [1].

```python
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

- **Rectified linear unit (ReLU) activation**. Through ReLU function, the output will remain the same as the input if input is non-negative, otherwise, it will be truncated to 0. While Sigmoid is a widely used activation function in the output layer, ReLU is widely used between hidden layers of the neural network. More information could refer to PyTorch documentation [2].

$$\text{ReLU}(x) = \max(x, 0). \tag{4.3}$$

```python
def relu(x):
    return max(x, 0)
```

- **Hyperbolic tangent (Tanh) activation**. The tanh function transforms any scalar input into $(-1, 1)$, which will squeeze the extremely large or small inputs to approximately the boundary values (-1 or 1), a similar effect as the Sigmoid function (which is 0 or 1). For more information, pleas refer to PyTorch documentation. [3].

$$\text{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}. \tag{4.4}$$

```python
def tanh(x):
    return (math.exp(x) - math.exp(-x)) / (math.exp(x) + math.exp(-x))
```

- **Leaky ReLU activation**. Leaky ReLU is a variant of ReLU activation. The simple ReLU activation will zero-out all inputs that are negative, which may lose negative information. Leaky ReLU preserves information in the negative region by using a smaller coefficient to scale the input and creating a piece-wise linear activation, which overall is still non-linear but can distinguish the positive and negative information. More information could refer to PyTorch documentation [4].

$$\text{LeakyReLU}(x) = \max(x, 0) + a \cdot \min(x, 0), \tag{4.5}$$

where a is a constant scalar. Usually researchers use $a = 0.01$ or $a = 0.05$. The minor difference between ReLU and LeakyReLU can be observed from Figure 4.2.

```python
def leakyrelu(x, a=0.05):
    return max(x, 0) + a * min(x, 0)
```

[1] https://pytorch.org/docs/stable/generated/torch.sigmoid.html
[2] https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html
[3] https://pytorch.org/docs/stable/generated/torch.tanh.html
[4] https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html

- **Exponential linear unit (ELU) activation**. The ELU function is also an improved version of ReLU. ELU makes the transition on $x = 0$ more "smooth" by adopting an exponential function on the negative x-axis. Especially, $a = 1$ makes the derivatives of the whole activation function continuous on $x = 0$. Commonly, ELU is also designed to help mitigate the vanishing gradient problem. The shape could be found in Figure 4.2. More information about the activation function could refer to PyTorch documentation [5].

$$\mathrm{ELU}(x) = \max(x, 0) + a \cdot (\exp(\min(x, 0)) - 1), \tag{4.6}$$

where a is a constant scalar, usually researchers use $a = 1$ as in torch package.

```python
def elu(x, a=1.0):
    return max(x,0) + a * (math.exp(min(x, 0)) - 1)
```

In addition to the common activation functions, there are several other activation functions tailored to meet specific needs. For example, Softmax activation is widely used for multi-class classification, which needs vector-formed inputs and will be discussed in the later section. It's important to note that all these activation functions share a common characteristic: *they are easily differentiable*. This property is crucial as it allows the gradient to be back-propagated through them, facilitating the adjustment of weights and biases during the training process. The choice of activation function can significantly impact the performance of a neural network, and it often depends on the specific requirements of the task at hand.

PyTorch Model Initialization: Hypertension Detector

In our hypertension example with patient blood pressure and age as features and "having hypertension or not" as target, the neuron detector has one weight term for the blood pressure, one weight term for the age, and one bias term. These three terms are parameters that should be optimized based on the existing patient data (i.e., training data). The suitable activation function here is Sigmoid for binary classification. We should learn how to implement the hypertension detector in PyTorch following the PyTorch templates[6].

The model class in the scripts uses a single neuron layer that takes 2-dimensional inputs - blood pressure as one dimension and age as the other. The output is a 1-dimensional value transformed by the Sigmoid activation function to generate a hypertension activation value between 0 and 1.

```python
import torch
import torch.nn as nn

class HypertensionDetector(nn.Module):
    def __init__(self):
        # input dimension: 2
        # output dimension: 1
```

[5] https://pytorch.org/docs/stable/generated/torch.nn.ELU.html
[6] https://pytorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html

```
        self.neuron = nn.Linear(2, 1)

    def forward(self, x):
        y = torch.sigmoid(self.neuron(x))
        return y

model = HypertensionDetector()
```

4.1.3 Train a Sigmoid neuron

Next, we show how to train a single neuron with Sigmoid activation. Follow the previous notation, the input to the neuron is $x_i,\ i = 1, \ldots, n$, the weights in the neuron is $w_i,\ i = 1, \ldots, n$. The bias term is b. Using the Sigmoid activation, we have the end-to-end neural function as:

$$\hat{y} = f(x_1, \ldots, x_n; w_1, \ldots, w_n, b) = \frac{1}{1 + \exp(-\sum_{i=1}^{n} x_i \cdot w_i - b)}. \tag{4.7}$$

Within $f(\cdot)$, the x_i are the input features, and w_i and b are the parameters. They are separated by a semicolon ";". Adjusting the weight of the input feature x_i is the intuitive function of w_i, while the bias b is an adjustable offset that helps the neuron to activate or deactivate independently from the input features x_i.

Let us assume the label is binary $y \in \{0, 1\}$, we use the binary cross entropy as the objective (which is typically used together with Sigmoid activation) to optimize the parameters of the neuron.

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]. \tag{4.8}$$

The gradient for each parameter w_i and b is

$$\nabla_{w_i} = \frac{\partial \mathcal{L}}{\partial w_i} = \frac{d\mathcal{L}}{df} \frac{\partial f}{\partial w_i}, \tag{4.9}$$

$$\nabla_b = \frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{df} \frac{\partial f}{\partial b}. \tag{4.10}$$

We have some intermediate variables to calculate. We can start with the first one, which is given by the equation:

$$\frac{d\mathcal{L}}{df} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}, \tag{4.11}$$

where y and $\hat{y}$ are values of the output and predicted output, respectively.

Next, we need to calculate the partial derivatives of f with respect to the weights w_i and bias b. These derivatives are given by the following equations:

$$\frac{\partial f}{\partial w_i} = -\hat{y}(1 - \hat{y})x_i, \tag{4.12}$$

$$\frac{\partial f}{\partial b} = -\hat{y}(1 - \hat{y}). \tag{4.13}$$

Using these partial derivatives, we can calculate the gradients as follows:

$$\nabla_{w_i} = ((1-y)\hat{y} - y(1-\hat{y}))x_i, \tag{4.14}$$

$$\nabla_b = ((1-y)\hat{y} - y(1-\hat{y})). \tag{4.15}$$

Alternatively, we can write the gradient of weights in vector form as:

$$\nabla_{\mathbf{w}} = ((1-y)\hat{y} - y(1-\hat{y})) \cdot \mathbf{x}, \tag{4.16}$$

where **w** is the vector of weights and **x** is the input vector. The notation $\cdot$ represents Hadamard product (scalar-vector product here).

To update the weights and bias term, we can use **gradient descent** with learning rate λ,

$$\mathbf{w} \leftarrow \mathbf{w} - \lambda \nabla_{\mathbf{w}}, \tag{4.17}$$

$$b \leftarrow b - \lambda \nabla_b, \tag{4.18}$$

and the gradients can be calculated from all the dataset (called full gradient descent) or from a random batch of samples (called stochastic gradient descent). After parameter optimization, Equation (4.7) could be used to make binary predictions on new data points.

PyTorch Model Training

In the case of our hypertension example, by using PyTorch for building the model, we no longer have to manually calculate complex gradients. Instead, the model updates itself automatically through the optimizer, which takes care of the underlying operations. The process of training the model follows a standardized template, which minimizes the need for significant modifications when adapting the model to different application scenarios. We show the complete training process in the notebook [7].

```
"""
Suppose we are given 500 patient records and the decision.
X: (500, 2), y: (500,1)
"""
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01) # lr is the learning rate

# Training loop
for epoch in range(200):
    y_hat = model(X)
    loss = criterion(y_hat, y)

    # Clear old gradient from last step - otherwise gradient would accumulate
    optimizer.zero_grad()
    # Computes the gradient of the loss with respect to the model's parameters.
    loss.backward()
    # Updates the model's parameters based on the computed gradients.
    optimizer.step()
```

[7] https://github.com/sunlabuiuc/pyhealth-book/blob/main/chap2-DNN/notebooks

After running the above code, the model is optimized to the training data. We can obtain the blood pressure and age information from new patients and use the detector to find high-risk patients for hypertension.

4.2 Train Deep Neural Network (DNN)

Deep neural networks (DNNs) are more powerful than single neurons because they are constructed using multiple layers of neurons. A basic form of a multi-layer DNN is known as a feed-forward neural network. In this type of network, information can only move in one direction, from the input layer, through the hidden layers, to the output layer, without any cycles. We will focus on a DNN that has two hidden layers and show how to implement such a neural network to improve the hypertension detection model.

4.2.1 Structure of a deep neural network

Each layer in a DNN serves a specific purpose. The input layer receives the raw data and passes it to the first hidden layer. The hidden layers, which form the core of the network, perform complex computations on the inputs received, extracting and learning features from the data. The output layer then takes these learned features and uses them to produce the final prediction.

Training a DNN involves adjusting the weights and biases of the neurons in these layers to minimize the difference between the predicted and actual outputs. This is done using techniques such as back-propagation and gradient descent, which we have discussed earlier for a single neuron and will continue to explore in detail in this section for layers of neurons. By understanding how to train a DNN, you will be equipped with the knowledge to tackle complex problems that require high-level feature learning and abstraction.

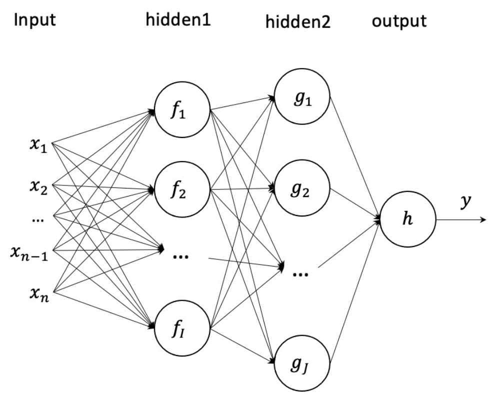

Figure 4.3 Illustration of Deep Neural Networks

We illustrate the two-layer deep neural networks (DNNs) in Figure 4.3.

- **Input:** Here, we use the same notations for input data, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, which represent n feature inputs (such as blood pressure value and age).
- **First hidden layer:** We use the notation $f_i(\cdot),\ i = 1, \ldots, I$ to denote the first layer hidden neuron with the activation functions. Their weight parameters can be written together as a large weight matrix $\mathbf{W}_1 \in \mathbb{R}^{n\times I}$, where the columns of $\mathbf{W}_1$ are parameters for each neuron. Their bias terms can be written as a vector $\mathbf{b}_1 \in \mathbb{R}^{I\times 1}$. All the neurons in the first layer are connected to the inputs.
- **Second hidden layer:** We use the notation $g_j(\cdot),\ j = 1, \ldots, J$ to denote the second layer hidden neuron. Their weight parameters can be written together as a large weight matrix $\mathbf{W}_2 \in \mathbb{R}^{I\times J}$, where the columns of $\mathbf{W}_2$ are the weights for each neuron. Their bias terms can be written as a vector $\mathbf{b}_2 \in \mathbb{R}^{J\times 1}$. All the neurons in the second layer are connected to all the neurons in the first layer.
- **Output layer:** We use the notation $h(\cdot)$ to denote the output layer neuron, which has parameters $\mathbf{w}_3 \in \mathbb{R}^{J\times 1}$ and bias term $b_3 \in \mathbb{R}$. This output layer will aggregate the results from all neurons in the second layer.

We use ReLU activation between the hidden layer and use the Sigmoid activation for the output neuron to generate the final prediction.

Forward Propagation. The process of forward propagation involves transforming the input $\mathbf{x}$ through the first layer, second layer, output layer, and finally generating a scalar output $\hat{y}$.

- The first layer transforms the input using the ReLU activation function as follows:

$$\mathbf{h}_1 = \max(\mathbf{W}_1^\top \mathbf{x} + \mathbf{b}_1, 0) \in \mathbb{R}^I$$

  Note that the symbol $^\top$ represents the transpose of a matrix. When a matrix is transposed, its rows become columns and its columns become rows.
- The second layer then transforms the output from the first layer using ReLU activation function:

$$\mathbf{h}_2 = \max(\mathbf{W}_2^\top \mathbf{h}_1 + \mathbf{b}_2, 0) \in \mathbb{R}^J$$

- Finally, the output layer generates the scalar output, $\hat{y}$, by applying the sigmoid activation function to the output from the second layer:

$$\hat{y} = \frac{1}{1 + \exp(-\mathbf{w}_3^\top \mathbf{h}_2 - b_3)}$$

4.2.2 Train a simple deep neural network

Given the above equations, we use the typical back-propagation to update the model parameters. The DNN model has three parameter matrices/vectors $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{w}_3$ and three bias term vectors/scalars $\mathbf{b}_1$, $\mathbf{b}_2$, b_3. The updating of these parameters follows chain rules. Assume the ground truth label is $y \in \{0, 1\}$, we follow the binary cross-entropy loss,

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})].$$

Back-propagation is a widely used algorithm for training deep neural networks. It is used to update the parameters of the model by minimizing the loss function. The algorithm works by propagating the error back through the network (i.e., its gradients), starting from the output layer and moving back towards the input layer. At each layer, the error is used to update the weights and biases of the neurons. This process is repeated multiple times until the algorithm converges to a set of parameters that minimize the loss function.

Gradient for $\mathbf{w}_3, b_3$

The gradient is calculated backward, starting from the output layer. By definition, the derivative for $\mathbf{w}_3$ and b_3 are calculated (refer to Equation 4.16),

$$\nabla_{\mathbf{w}_3} = \frac{\partial \mathcal{L}}{\partial \mathbf{w}_3} = ((1 - y)\hat{y} - y(1 - \hat{y})) \cdot \mathbf{h}_2, \tag{4.19}$$

$$\nabla_{b_3} = \frac{\partial \mathcal{L}}{\partial b_3} = ((1 - y)\hat{y} - y(1 - \hat{y})). \tag{4.20}$$

Gradient for $\mathbf{W}_2, b_2$

For $\mathbf{W}_2$, we need to calculate the gradient through the ReLU activation. To formalize this, we introduce the auxiliary variable $\mathbf{z}_2 = \text{Sign}(\mathbf{h}_2)$, which is a binary vector, denoting the sign of the hidden units. The derivative of $\mathcal{L}$ with respect to $\mathbf{W}_2$ and $\mathbf{b}_2$ are (Here, the notation $\circ$ represents the vector-vector outer product),

$$\nabla_{\mathbf{W}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = ((1 - y)\hat{y} - y(1 - \hat{y})) \cdot \mathbf{h}_1 \circ (\mathbf{w}_3 \cdot \mathbf{z}_2)^\top, \tag{4.21}$$

$$\nabla_{\mathbf{b}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{b}_2} = ((1 - y)\hat{y} - y(1 - \hat{y})) \cdot (\mathbf{w}_3 \cdot \mathbf{z}_2). \tag{4.22}$$

Gradient for $\mathbf{W}_1, b_1$

For $\mathbf{W}_1$, the derivative calculation is similar following the chain rules. If we use $\mathbf{z}_1 = \text{Sign}(\mathbf{h}_1)$ to denote the binary sign vector of $\mathbf{h}_1$, we can denote the derivative for $\mathbf{W}_1$ and $\mathbf{b}_1$ as

$$\nabla_{\mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = ((1 - y)\hat{y} - y(1 - \hat{y})) \cdot \mathbf{x} \circ (\mathbf{W}_2(\mathbf{w}_3 \cdot \mathbf{z}_2) \cdot \mathbf{z}_1)^\top, \tag{4.23}$$

$$\nabla_{\mathbf{b}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} = ((1 - y)\hat{y} - y(1 - \hat{y})) \cdot (\mathbf{W}_2(\mathbf{w}_3 \cdot \mathbf{z}_2) \cdot \mathbf{z}_1). \tag{4.24}$$

Later, the parameters can be updated by stochastic gradient descent (SGD) on batch data with λ as the learning rate.

$$\mathbf{W}_1 \leftarrow \mathbf{W}_1 - \lambda \nabla_{\mathbf{W}_1}, \tag{4.25}$$

$$\mathbf{b}_1 \leftarrow \mathbf{b}_1 - \lambda \nabla_{\mathbf{b}_1}, \tag{4.26}$$

$$\mathbf{W}_2 \leftarrow \mathbf{W}_2 - \lambda \nabla_{\mathbf{W}_2}, \tag{4.27}$$

$$\mathbf{b}_2 \leftarrow \mathbf{b}_2 - \lambda \nabla_{\mathbf{b}_2}, \tag{4.28}$$

$$\mathbf{w}_3 \leftarrow \mathbf{w}_2 - \lambda \nabla_{\mathbf{w}_2}, \tag{4.29}$$

$$b_3 \leftarrow b_3 - \lambda \nabla_{b_3}. \tag{4.30}$$

PyTorch DNN Implementation

We can use PyTorch to implement the above DNN as our hypertension detector based on patient’s blood pressure and age information. We need to make minor modifications from the *HypertensionDetector* class and use exactly the same training process to optimize the parameters as shown in Section 4.1.3.

```python
class DNNHypertensionDetector(nn.Module):
    def __init__(self):
        # input dimension: 2
        # first layer neuron: 16
        # second layer neuron: 8
        # output dimension: 1

        self.f = nn.Linear(2, 16)
        self.g = nn.Linear(16, 8)
        self.h = nn.Linear(8, 1)

    def forward(self, x):
        neurons_l1 = torch.relu(self.f(x))
        neurons_l2 = torch.relu(self.g(neurons_l1))
        output = torch.sigmoid(self.h(neurons_l2))
        return output

model = DNNHypertensionDetector()
```

In this public notebook [8], we have provided a complete PyTorch implementation of DNN on a synthetic hypertension detection dataset.

4.3 Build DNN model for Readmission Prediction

We have previously discussed the structure and training of a single neuron and deep neural networks (DNNs). Now, we will focus on their practical application in the healthcare industry. In this section, we will work with our readers to create a simple DNN model using the public MIMIC-IV dataset [3] for hospital readmission prediction tasks.

Manually processing the datasets and using the PyTorch package to build, train,

[8] https://github.com/sunlabuiuc/pyhealth-book/blob/main/chap2-DNN/notebooks

and evaluate deep learning models can be time-consuming and takes several hours to complete. Fortunately, this section (and the entire book) will use PyHealth[9] as the primary tool to implement the entire pipeline, from data processing to model evaluation.

PyHealth [4] is a Python library specially designed for healthcare data analysis. It provides a range of tools and functionalities that simplify the process of building, training, and evaluating healthcare models. For this specific task, PyHealth has a suite of API modules that cover MIMIC-IV data processing, DNN model initialization, training, and evaluation.

MIMIC-IV dataset

The MIMIC-IV (Medical Information Mart for Intensive Care IV)[10] [3] is sourced from two in-hospital database systems. It is a custom hospital wide EHR and an ICU specific clinical information system. This dataset is publicly available developed by the MIT Lab for Computational Physiology, comprising de-identified health-related data associated with over forty thousand patients who stayed in critical care units of the Beth Israel Deaconess Medical Center between 2001 and 2012. This dataset is widely adopted for various clinical prediction tasks on electronic health records, including drug recommendation, readmission prediction, length of stay prediction, patient phenotyping, etc.

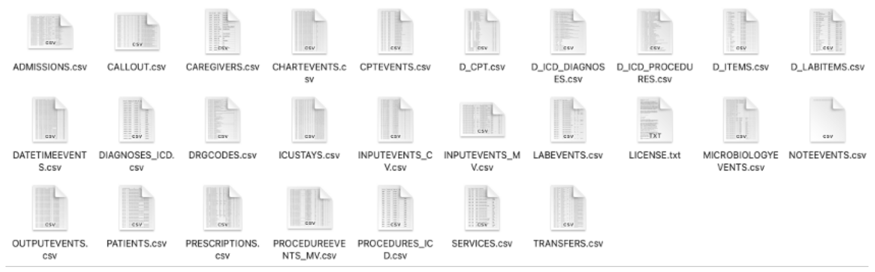

Figure 4.4 Raw files from MIMIC-III dataset

The rich MIMIC-IV dataset includes information such as demographics, vital signs, laboratory tests, medications, and more. The raw files are shown in Figure 4.4, and different information aspects are stored in separate CSV files linked by patient encounter id. In this section, we will use the *procedures, medications, and diagnoses information* from the datasets for readmission task. As the background, **medical diagnosis** is the process of determining which disease or condition explains a person’s symptoms and signs. **Medical procedures** are conducted with the intention of determining, measuring, or diagnosing a patient condition or parameter and is also called a medical test. Major medical procedures include surgery to the organs of the head, chest and abdomen. **Medications** are prescribed by doctors after the visit and used to diagnose,

[9] https://pyhealth.readthedocs.io/en/latest
[10] https://physionet.org/content/mimiciv/2.0/

cure, treat, or prevent disease. All these information are strong indicators of patient's current health status.

Readmission prediction task

The readmission prediction task aims to predict whether a patient will be readmitted to the ICU within 7 days after discharge, based on the procedure, medication, and diagnosis codes from the current hospital visit [5, 6, 7]. Since this task is essentially a binary classification task, the objective function is to minimize the cross-entropy loss between the predicted probability and the actual label for readmission within 7 days.

The task can be broken down into the following parts:

- **Input**: The electronic health records (EHR) of the current hospital visit, including procedure, medication, and diagnosis data sources from the MIMIC-IV dataset. The diagnosis and procedure codes are generated on hospital discharge for billing purposes, and the medication codes record the prescribed medications. Each patient input is represented by a list of medical codes (of type diagnosis, procedure, and medication).
- **Output**: A binary prediction of whether the patient will be readmitted to the ICU within 7 days after discharge. Each patient output is a binary label (e.g., 0 or 1).
- **Objective Function**: The cross-entropy loss between the predicted probability and the actual label for readmission within 7 days is minimized to improve the accuracy of readmission prediction.

Accurate prediction of readmissions can help healthcare providers identify patients at risk and intervene early, potentially improving patient outcomes and reducing healthcare costs.

To accomplish the given task, we will be utilizing the PyHealth package model APIs to construct a three-layer DNN. This model will use patient data extracted from the MIMIC-IV dataset as input, and it will be passed through two hidden layers for learning complex representations. Finally, the output layer will generate a prediction that will indicate the probability of readmission.

4.3.1 Step 1: Data Processing using *pyhealth.datasets*

To start with, we need to preprocess the MIMIC-IV data. The MIMIC-IV and other EHR databases have complex and heterogeneous file structures. CSV tables are maintained for each type of clinical events, and these tables are connected by a unique patient visit ID. Researchers typically join tables together and filter invalid records, among other things, before feeding the dataset into deep learning models. For the readmission prediction tasks, we utilize the diagnosis, procedure, and medication tables as feature entries from the MIMIC-IV dataset.

Fortunately, the *MIMIC4Dataset* API from the *pyhealth.datasets* module can assist us in processing the dataset into a patient-visit-event nested dictionary structure, as illustrated in Figure 4.5. This data structure is adaptable to various clinical predictive tasks and can be post-processed to suit the readmission prediction task (which we will discuss in Step 2).

To call the *MIMIC4Dataset* API, we require the root of the MIMIC-IV database and the data tables we want to use for features (diagnoses, procedures, and prescriptions).

```python
from pyhealth.datasets import MIMIC4Dataset

# STEP 1: dataset processing
mimic4_ds = MIMIC4Dataset(
    root="/srv/local/data/physionet.org/files/mimiciv/2.0/hosp",
    tables=["diagnoses_icd", "procedures_icd", "prescriptions"],
)
mimic4_ds.stat()
```

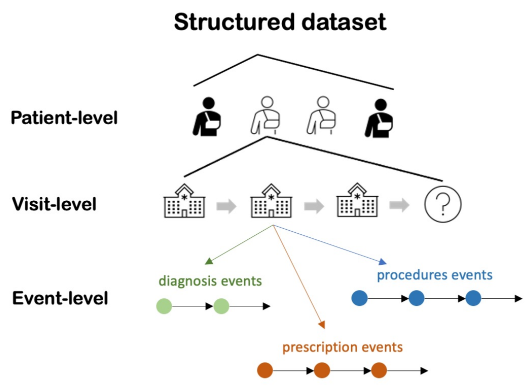

Figure 4.5 The MIMIC-IV dataset is organized in a hierarchical structure. At the top level are patients, each with multiple visits over time. Within each visit, there are various types of events, such as diagnoses, procedures, and prescription medications.

After loading data, we can use the *.stat*() function to view stats. The output includes patient count, visit count, and feature frequencies per visit.

```
"""
Statistics of base dataset:
        - Dataset: MIMIC4Dataset
        - Number of patients: 190279
        - Number of visits: 454324
        - Number of visits per patient: 2.3877
        - Number of events per visit in diagnoses_icd: 11.0205
        - Number of events per visit in procedures_icd: 1.5498
        - Number of events per visit in prescriptions: 35.6424
"""
```

To provide more details about the nested data structure, we can briefly introduce the resulting object and its data query process. A natural way of storing the EHR database is by utilizing a hierarchical "Patient-Visit-Event" structure in Figure 4.5.

- The **top-level** is a list of patients with patient-specific information. They are the *python.data.Patient*[11] object, used for the top-level structure. Each object stores the overall information for one patient, including patient id, birth date time, death date time, gender, ethnicity, etc.

```
>>> patients = mimic4_ds.patients
>>> patient_id, patient_obj = list(patients.items())[2] # 3-rd patient
# patient_obj.birth_datetime
# patient_obj.death_datetime
# patient_obj.gender
# patient_obj.ethnicity
```

- In the **middle-level**, each patient contains a list of hospital visits as well as visit-specific information. They are the *python.data.Visit* [12] object, used for the middle-level structure, which records the visit id, patient id, encounter time, etc.

```
>>> visits = patient_obj.visits
>>> visit_id, visit_obj = list(visits.items())[5] # the 6-th visit
# visit_obj.encounter_time
# visit_obj.discharge_time
# visit_obj.discharge_status
```

- In the most **granular-level**, each hospital visit contains multiple event lists of all types of clinical activities, such as diagnosis events, procedures, and prescriptions. *python.data.Event*[13] is used for the granular-level structure.

```
>>> diagnoses = visit_obj.get_event_list("diagnoses_icd")
>>> diagnosis = diagnoses[0] # the 1-st diagnosis
# diagnosis.code
# diagnosis.vocabulary
# daignosis.timestamp
>>> procedures = visit_obj.get_event_list("procedures_icd")
>>> procedure = procedures[2] # the 2-nd procedure
# procedure.code
# procedure.vocabulary
# procedure.timestamp
```

We will learn about the Patient, Visit, Event objects in details in Chapter **??**.

4.3.2 Step 2: Define Healthcare Task using *pyhealth.tasks*

PyHealth's *pyhealth.tasks*[14] module offers a range of healthcare AI task functions. Each of them defines a unique healthcare AI task. All task functions *input a pyhealth.data.Patient object* and then *output a list of samples containing the "X" and "Y" for supervised learning*. If the task is defined as a patient-level prediction task, then the output is one sample; if it is a visit-level prediction task, then the output is a list of samples, one for each visit.

[11] https://pyhealth.readthedocs.io/en/latest/api/data/pyhealth.data.Patient.html
[12] https://pyhealth.readthedocs.io/en/latest/api/data/pyhealth.data.Visit.html
[13] https://pyhealth.readthedocs.io/en/latest/api/data/pyhealth.data.Event.html
[14] https://pyhealth.readthedocs.io/en/latest/api/tasks.html

For example, **readmission prediction** is a visit-level clinical predictive task. Given the above structured dataset, we need to extract the diagnosis, procedure, and prescription information as the features and compare the time gap between the current visit and next visit to decide whether the patient will be readmitted into hospital within 7 days. If the answer is Yes, then we set label to be 1; otherwise, the label is 0. It is a binary classification task. In this case, PyHealth provides a default readmission prediction task with time window as an argument (In this task, the threshold is 7 days).

```python
def readmission_prediction_mimic4_fn(patient, time_window=7):
    """Processes a single patient for the readmission prediction task.
    ...
    """
    samples = []

    # we will drop the last visit since we cannot tell its label
    for i in range(len(patient) - 1):
        visit = patient[i]
        next_visit = patient[i + 1]

        # get time difference between current visit and next visit
        time_diff = (next_visit.encounter_time
                    - visit.encounter_time).days
        readmission_label = 1 if time_diff < time_window else 0

        conditions = visit.get_code_list(table="diagnoses_icd")
        procedures = visit.get_code_list(table="procedures_icd")
        drugs = visit.get_code_list(table="prescriptions")
        # exclude: visits without condition, procedure, or drug code
        if len(conditions) * len(procedures) * len(drugs) == 0:
            continue
        samples.append(
            {
                "visit_id": visit.visit_id,
                "patient_id": patient.patient_id,
                "conditions": conditions, # or [conditions]
                "procedures": procedures, # or [procedures]
                "drugs": drugs, # or [drugs]
                "label": readmission_label,
            }
        )

    return samples
```

By simply calling the *.set_task*() function and utilizing the above readmission prediction task function, the PyHealth package will automatically help you query the nested dictionary structure and re-organize it to obtain the clean form for downstream machine learning models.

```python
# from pyhealth.tasks import readmission_prediction_mimic4_fn

# STEP 2: define the readmission prediction task
readmission_dataset = mimic4_ds.set_task(
    lambda x: readmission_prediction_mimic4_fn(x, time_window=7)
)
```

```
readmission_dataset.stat()
```

Let us dive deep into the *readmission_prediction_mimic4_fn*() task function.

```
def readmission_prediction_mimic4_fn(patient: Patient, time_window=15):
    samples = []

    # we will drop the last visit
    for i in range(len(patient) - 1):
        visit: Visit = patient[i]
        next_visit: Visit = patient[i + 1]

        # get time difference between current visit and next visit
        time_diff = (next_visit.encounter_time - visit.encounter_time).days
        readmission_label = 1 if time_diff < time_window else 0

        conditions = visit.get_code_list(table="diagnoses_icd")
        procedures = visit.get_code_list(table="procedures_icd")
        drugs = visit.get_code_list(table="prescriptions")
        # exclude: visits without condition, procedure, or drug code
        if len(conditions) * len(procedures) * len(drugs) == 0:
            continue
        samples.append(
            {
                "visit_id": visit.visit_id,
                "patient_id": patient.patient_id,
                "conditions": [conditions],
                "procedures": [procedures],
                "drugs": [drugs],
                "label": readmission_label,
            }
        )
    # no cohort selection
    return samples
```

As shown above, this task function takes a single patient object as input and returns a list of samples (of type dictionary). For the readmission prediction case, the sample is defined at visit level, so each patient can have more than one samples. Note that the last visit is discarded as we do not know whether the patient is readmitted in the given time window or not. For each visit, we gather the diagnosis, procedure, and medication codes and store them in a sample dictionary.

For simplicity, users can call the readmission prediction function directly from *pyhealth.tasks*. They can also modify the function with further cohort selections or apply application-specific filters to the function.

To obtain the data summary of the post-processed dataset, we can call the .*stat*() function. In this processed dataset, we have 59,273 patients with a total of 132,301 hospital visits, resulting in 132,301 training samples. There are 18,856 unique diagnosis (conditions) codes, 10220 unique procedure codes, and 5,458 different medication (drugs) codes. For the readmission prediction task, out of 132,301 visits, 67,860 visits show that the patient did not return to the hospital within 7 days, while in the remaining 64,441 visits, the patient was readmitted within 7 days after discharge.

```
"""
Statistics of sample dataset:
        - Dataset: MIMIC4Dataset
        - Task: <lambda>
        - Number of samples: 132301
        - Number of patients: 59273
        - Number of visits: 132301
        - Number of visits per patient: 2.2321
        - conditions:
                - Number of conditions per sample: 13.5967
                - Number of unique conditions: 18858
                - Distribution of conditions (Top-10):
                    [('4019', 34015), ('2724', 25635), ('53081', 18716),
                    ('E785', 15686), ('25000', 15530), ('41401', 14924),
                    ('4280', 14814), ('I10', 14065), ('42731', 13960),
                    ('Z87891', 13776)]
        - procedures:
                - Number of procedures per sample: 2.7078
                - Number of unique procedures: 10220
                - Distribution of procedures (Top-10):
                    [('3893', 8320), ('3897', 6165), ('3995', 6035),
                    ('02HV33Z', 5966), ('8856', 5171), ('0040', 4588),
                    ('966', 4479), ('9925', 4310), ('4513', 3395),
                    ('5491', 3313)]
        - drugs:
                - Number of drugs per sample: 29.6466
                - Number of unique drugs: 5458
                - Distribution of drugs (Top-10):
                    [('0', 127393), ('63323026201', 79098),
                    ('00904224461', 70124), ('00338004904', 53800),
                    ('00904516561', 53187), ('00338011704', 44555),
                    ('00409672924', 40443), ('00409490234', 39312),
                    ('00406055262', 36542), ('51079000220', 34618)]
        - label:
                - Number of label per sample: 1.0000
                - Number of unique label: 2
                - Distribution of label (Top-10):
                    [(0, 67860), (1, 64441)]
"""
```

The generated machine learning samples are arranged in a dictionary format, containing information at the patient or visit level. Users can customize the attributes while defining the task function. The details include visit ID, patient ID, binary label for readmission, and codes for diagnosis, procedures, and medication.

```
>>> readmission_dataset[0]
{
    'visit_id': '22595853',
    'patient_id': '10000032',
    'conditions': ['5723', '78959', '5715', '07070', '496', '29680',
            '30981', 'V1582'],
    'procedures': ['5491'],
    'drugs': ['0', '63323026201', '19515089452', '00245004101',
            '63739054410', '51079007220', '00904198861', '00006022761',
            '00173068224', '61958070101', '00135019502', '00487980125',
            '51079007320'],
    'label': 0
}
```

The following is a code snippet that contains a dictionary of patient data, which includes:

- "conditions": A list of conditions that have been *diagnosed* for the patient. The codes used follow the ICD-9-CM classification of diseases and injuries, such as "5723" represents portal hypertension, "78959" represents other ascites, "07070" represents unspecified viral hepatitis C without hepatic coma.
- "procedures": A list of *medical procedures* that have been performed on the patient. The codes used also follow the ICD-9-CM classification of procedures, such as "5491" represents the Percutaneous abdominal drainage.
- "drugs": A list of *drugs* that have been prescribed to the patient during their visit. The codes used for the drugs follow the National Drug Code (NDC) classification, such as "63323026201" is a type of heparin sodium, "00245004101" is the potassium chloride, "00904198861" represents extra strength Mapap tablet.

These samples are ready to be used in the machine learning models. We then use PyHealth utility functions to split the whole dataset by patients (ensuring that patient does not overlap in the train/val/test datasets). We split the dataset by 80% : 10% : 10%. Then, we call *get_dataloader*, which essentially specifies how to aggregate a list of samples to batch, and set the batch size to 32.

```python
from pyhealth.datasets import split_by_patient, get_dataloader

# split the dataset into train/val/test
train_dataset, val_dataset, test_dataset = split_by_patient(
    readmission_dataset, [0.8, 0.1, 0.1]
)
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)
```

4.3.3 Step 3: Define a DNN Model using *pyhealth.models*

The *pyhealth.models*[15] module provides a large variety of ML models for users to choose from. These methods can be broadly categorized into: **(i) general deep learning models**, like DNN, Convolutional Neural Network (CNN), Recurrent Neural Network (RNN) and **(ii) healthcare specific deep learning models**, like RETAIN [8] and GCT [6].

In this section, we will use the simple three-layer multi-layer perceptron (MLP) in PyHealth, specifying the number of layers, the number of hidden neurons in each layer, and the activation functions. More advanced models and examples can be found in later sections or by visiting the PyHealth website[16].

[15] https://pyhealth.readthedocs.io/en/latest/api/models.html
[16] https://pyhealth.readthedocs.io/en/latest/

In model initialization, we input the keys listed in the sample dictionary. To leverage three types of clinical event information, we use "conditions", "procedures", "drugs" as keywords for features, and we input the "label" as the readmission prediction label.

```python
from pyhealth.models import MLP

# STEP 3: define model
model = MLP(
    dataset=readmission_dataset,
    feature_keys=["conditions", "procedures", "drugs"],
    label_key="label",
    mode="binary",
    embedding_dim=128,
    hidden_dim=128,
    n_layers=3,
    activation="relu",
)
```

We input the readmission dataset as one argument to inform the neural network about the basic feature types and dimensions while the data sample information will not be used in initializing the model.

4.3.4 Step 4 & 5: Model Training and Evaluation by *pyhealth.trainer* and *pyhealth.metrics*

In PyHealth, the *pyhealth.trainer.Trainer*[17] module serves as a powerful tool for managing the training process of all models. This trainer simplifies logging during training, automatically selects available devices, and conveniently saves the best model to disk. The trainer module offers a range of selectable arguments for customization. Let's proceed with the programming steps outlined below.

```python
from pyhealth.trainer import Trainer

# STEP 4: define trainer
trainer = Trainer(model=model)

trainer.train(
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    epochs=5,
    monitor="roc_auc",
) # model is training ...
```

Following model training, we offer two evaluation methods for the optimized deep learning model through the *pyhealth.metrics* module[18]. PyHealth offers an array of metrics suitable for assessing binary classification, multi-class classification, and multi-label classification problems, alongside fairness metrics and advanced model calibration metrics.

[17] https://pyhealth.readthedocs.io/en/latest/api/trainer.html
[18] https://pyhealth.readthedocs.io/en/latest/api/metrics.html

```python
# method 1
result = trainer.evaluate(test_dataloader)
print (result)

# method 2
from pyhealth.metrics.binary import binary_metrics_fn

y_true, y_prob, loss = trainer.inference(test_dataloader)
binary_metrics_fn(
    y_true,
    y_prob,
    metrics=["pr_auc", "roc_auc", "f1"]
)

"""
{"pr_auc": 0.6081, "roc_auc": 0.6360, "f1": 0.5797, "loss": 5.0959}
"""
```

Overall, this five-step pipeline can be entirely managed using PyHealth, and each step operates quite independently. We've provided example codes for this process on GitHub[19]. For users with varying levels of expertise in Python programming, there's also the option to utilize PyHealth's data processing or deep learning model scripts alone. They can easily integrate their own code in between these pre-existing components.

Questions

1. What defines a single neuron in the domain of deep learning? How many parameters does a single neuron have?
2. Why are activation functions necessary in the output of a single neuron? What are the common activation functions, and what ranges do their outputs fall within?
3. How do ReLU and LeakyReLU differ? In what scenarios would you choose ReLU over LeakyReLU, and vice versa?
4. What distinguishes a single neuron from a deep neural network (DNN)? Why are DNNs generally more powerful compared to a single neuron?
5. Could you find a more efficient way to implement the Tanh activation function compared to Equation 4.4?
6. Can you identify three medical applications where DNNs might potentially outperform doctors?
7. Dr. Cross needs help building a classifier to detect lung cancer based on patient age, binary gender indicator (0 for women, 1 for men), and years of smoking history. Using a neuron with sigmoid activation for the features $[age, gender, smoke\ years]$, could you implement this classifier? What are the parameters?
8. Following the previous question, suppose we aim to create a stronger classifier using

[19] https://github.com/sunlabuiuc/pyhealth-book/blob/main/chap2-DNN/notebooks

a deep neural network (DNN) with two hidden layers, each consisting of 8 units. How would you construct this DNN model, and what parameters would it involve?

9. We have provided a synthetic dataset accessible here: Synthetic Dataset Link [20]. This dataset contains a three-dimensional vector $[age, gender, smoke\ years]$ for each patient and a binary label (0 or 1). The dataset is divided into 5000 for training, 800 for validation, and 500 for testing. Could you load this dataset and use $pyhealth.models$ to train a DNN model for this task? What is the best evaluation performance achieved for the 500 patients, using AUROC as the evaluation metric?

[20] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap2-DNN/after-chapter-questions