5 Recurrent Neural Networks

In the previous section, we talked about deep neural networks (DNNs) and how they help to make predictions by connecting all neurons across layers. However, patient data is often presented in a sequential manner. For example, a patient may have multiple visits over time, and during each visit, diagnoses may be confirmed after lab tests and then medications may be prescribed. These sequential data points can vary in length and complexity, which makes it difficult for traditional DNN setups. Common DNNs are designed for fixed-length inputs and do not consider the inherent order of the features, nor can they adapt to various input lengths. This is a problem because some patients may have over 100 visits, while others may have only one or two. We need a deep learning model that can handle patients with varying numbers of visits and provide accurate results.

To overcome the challenge of processing sequential data, we introduce recurrent neural networks (RNNs). RNNs are capable of processing events one at a time in a recurring manner, which allows them to retain a memory of previous states. This enables RNN models to understand the relationships between events in a lengthy series of input data. RNNs can also adapt to different lengths of patient records using the same model.

We will discuss two practical variations of RNNs: long short-term memory (LSTMs) and gated recurrent units (GRUs), both of which are extensively used in modern research and industrial contexts. Due to their ability to process sequential data, RNN models have become increasingly popular, finding a wide range of applications in the healthcare sector. In this chapter, we will create an RNN model for predicting heart failure diagnosis.

5.1 Introduction to Recurrent Neural Networks

Heart failure onset prediction using longitudinal patient records

Using longitudinal electronic health records (EHR) data to predict heart failure is vital because heart failure is a serious condition that can lead to severe health problems and high medical costs. Treatment can only start after the heart failure is diagnosed, thus by predicting heart failure earlier using EHR data (e.g., blood pressure), doctors can begin treatment sooner. This could include medications, lifestyle changes, and

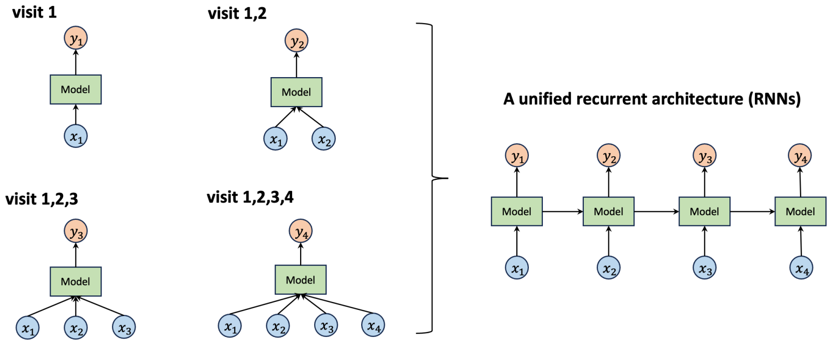

Figure 5.1 The image compares two options for modeling a sequence of visits 1-4 using deep neural networks (DNNs) on the left and recurrent neural networks (RNNs) on the right. It shows that a single DNN is not flexible enough to model sequential data, while a single RNN can handle sequences naturally.

enhancing patient self-management, potentially improving their health outcomes and quality of life.

Let’s dive into the task of predicting the likelihood of heart failure on a weekly basis as a binary classification.

Our initial attempt used deep neural networks (DNNs), as seen in the left part of Figure 5.1. Here, we input the blood pressure reading for the first week ($\mathbf{x}_1$), and the DNN predicts the probability of heart failure for week 2 (y_1). This process is repeated for every subsequent week.

The Challenge of DNN for modeling sequences: However, the input size grows with each passing week, making it incompatible with conventional DNN architectures. One possible solution is to calculate the average blood pressure and standard deviation values as two-dimensional inputs over the preceding weeks. While this is compatible with 2-dimensional DNN architecture, it may ignore weekly progression trends. Our goal is to have the model retain the memory of prior results and integrate new data seamlessly for better decision-making.

RNN’s sequencing modeling ability: The right-hand side of Figure 5.1 shows how we addressed this challenge: by using a recurrent neural network (RNN). The RNN accounts for both present observations (features from the current state) and persistent memory (hidden representations from the preceding state) to generate a prediction. It also creates a new memory representation for the following time step.

For example, some patients have their blood pressure observed for four weeks. One observation is fed into the RNN model each week, producing a new probability prediction output as well as a new memory to be used for the following week. The RNN iteratively runs over the sequential data inputs.

Let’s now introduce the formulation of RNNs.

5.1.1 Architecture of simple RNNs

In this section, we will introduce a simple RNN model that features only one gated function, as shown in Figure 5.2.

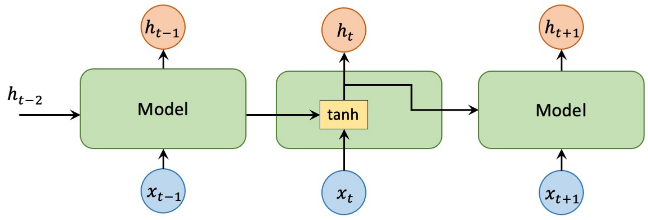

Figure 5.2 A simple RNN model: every input vector $\mathbf{x}_t$ is processed by the same model with Tanh activation function and outputs the memory state $\mathbf{h}_t$; both $\mathbf{x}_{t+1}$ and $\mathbf{h}_t$ are combined as input for next timestep.

The input sequence is represented as $\mathbf{x}_t \in \mathbb{R}^i$, and it can be either a vector or a scalar, such as the weekly blood pressure of a specific patient. The memory state, $\mathbf{h}_t \in \mathbb{R}^m$, is a vector embedding of the previous status. The Tanh activation function combines the input and memory state.

Recurrent update: from $\mathbf{h}_{t-1}$ to $\mathbf{h}_t$

In RNN models, the initial hidden state $\mathbf{h}_0$ is empty. At each time step, the RNN model combines the input $\mathbf{x}_t$ with the previous hidden state $\mathbf{h}_{t-1}$ to generate the new hidden state $\mathbf{h}_t$ by using the Tanh() activation function.

$$\mathbf{h}_0 = 0 \in \mathbb{R}^m, \tag{5.1}$$

$$\mathbf{h}_t = \mathrm{Tanh}(\mathbf{W}_h \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_h) \in \mathbb{R}^m. \tag{5.2}$$

In the above equations, the hidden state $\mathbf{h}_t$ represents the memory state at step t, $\mathbf{W}_h \in \mathbb{R}^{m\times(i+m)}$ are the weight parameters for input and memory in the Tanh activation, while $\mathbf{b}_h \in \mathbb{R}^m$ is the bias term.

Prediction at time t

Given the hidden memory state $\mathbf{h}_t$, one can apply a linear layer with a Sigmoid activation function and generate the output probability $\tilde{y}_t \in \mathbb{R}$ (i.e., t-th week) based on the current memory information, indicating the probability of heart failure at the current week.

$$\tilde{y}_t = \mathrm{Sigmoid}(\mathbf{w}_y \cdot \mathbf{h}_t + b_y). \tag{5.3}$$

Here, $\mathbf{w}_y \in \mathbb{R}^m$ is the weight parameter and $b_y \in \mathbb{R}$ is the bias term for the final prediction layer and $\cdot$ is the inner product operation.

This simple architecture shows the main concepts of RNNs, however, it is not used due to issues such as exploding gradient and vanishing gradient.

The **exploding gradient problem** in recurrent neural networks (RNNs) occurs during training when the gradients of the network’s weights grow exponentially through the backpropagation process. This can lead to excessively large updates to the network weights, causing instability and preventing the model from converging or learning appropriately. The issue is particularly acute in RNNs due to their sequential data processing and can manifest in long sequences where repeated multiplication of large gradients amplifies their values. This can result in model training failures, numerical instability, and poor generalization. There are some simple strategies to alleviate this problem, such as gradient clipping, which involves capping the gradients during backpropagation to prevent them from exceeding a certain threshold.

The **vanishing gradient problem** in recurrent neural networks (RNNs) is another trickier challenge for RNN training, where gradients, which are used in RNN training through backpropagation, become increasingly smaller as they are propagated back through each time step. This occurs especially in long sequences. As a result, the weights in the earlier layers of the network receive very small updates, making it difficult for the network to learn long-term dependencies in the data. This problem impedes the effective training of RNNs on tasks involving long sequences, such as in natural language processing or longitudinal EHR modeling.

Next, we will introduce two advanced RNN architectures that are widely used in modern applications.

5.1.2 Architecture of Long Short Term Memory (LSTMs)

To alleviate the vanishing gradient problems for long sequences, Hochreiter and Schmidhuber [9] introduced the long short-term memory (LSTM) network in 1997. The LSTM architecture is more complex, and it distinguishes between the memory cell $\mathbf{C}_t$ and the hidden state $\mathbf{h}_t$, as shown in Figure 5.3.

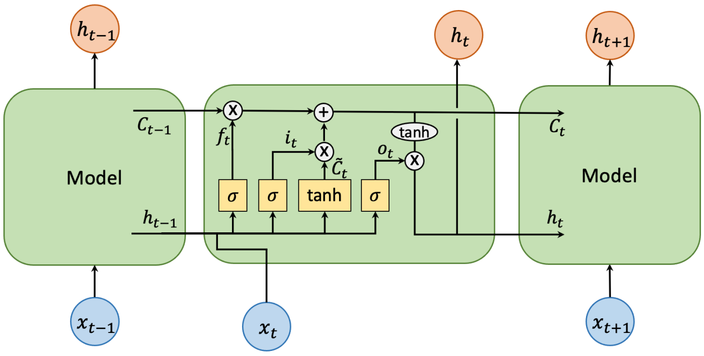

Figure 5.3 The long short-term memory (LSTM) architecture. LSTM maintains a memory cell and a hidden state at each time step enabled by four different gates: a forget gate, an input gate, an output gate, and a cell gate.

The LSTM model includes four gates that regulate the flow of information, unlike the single gate architecture of the simple RNN model described previously. The horizontal memory cell $\mathbf{C}_t$ is a pivotal element within the LSTM architecture that enables long-term memory retention. This memory cell undergoes several adjustments as it transitions from one state (t) to the next state ($t + 1$). Let's now introduce how the information flows through the four gates.

- Forget gate $\mathbf{f}_t$. The forget gate determines what information should be forgotten from the memory cell $\mathbf{C}_{t-1}$. The forget gate uses the hidden representation from the previous state $\mathbf{h}_{t-1} \in \mathbb{R}^m$ and the new input $\mathbf{x}_t \in \mathbb{R}^i$ to make this decision. Here, m is the hidden embedding dimension, and i is the input feature dimension. The forget gate is defined by the following equation:

$$\mathbf{f}_t = \text{Sigmoid}(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \in \mathbb{R}^m, \tag{5.4}$$

  where $\mathbf{W}_f \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix in the forget gate and $\mathbf{b}_f \in \mathbb{R}^m$ is the bias term. The input to the gate is a concatenation of the previously hidden representation $\mathbf{h}_{t-1}$ and the new input $\mathbf{x}_t$.

  The output of the forget gate, $\mathbf{f}_t$, is a vector with elements in the range of (0, 1). If an element in $\mathbf{f}_t$ is close to 0, it means that most of the previous memory on that dimension should be forgotten. On the other hand, if an element is close to 1, it means that the previous memory on that dimension should be kept. This forget gate will be used later in Equation (5.7).
- Input gate $\mathbf{i}_t$. The *input gate* has the same form as the forget gate. However, this gate is trained to determine what we should add to the memory cell $\mathbf{C}_{t-1}$ for updating.

$$\mathbf{i}_t = \text{Sigmoid}(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \in \mathbb{R}^m, \tag{5.5}$$

  where $\mathbf{W}_i \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix in the input gate and $\mathbf{b}_i \in \mathbb{R}^m$ is the bias term. Like the forget gate, the input gate will also be used later in Equation (5.7) to update the cell state.
- Cell gate $\tilde{\mathbf{C}}_t$. The *cell gate* also has a similar structure as forget/input gates but with Tanh as the activation. While the forget and input gates control the flow of information and take on values between 0 and 1 with a Sigmoid activation, the cell gate generates new information to add or to subtract from the existing cell state with Tanh activation.

  This cell gate will generate a candidate for updating the memory $\mathbf{C}_{t-1}$.

$$\tilde{\mathbf{C}}_t = \text{Tanh}(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \in \mathbb{R}^m, \tag{5.6}$$

  where $\mathbf{W}_C \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix in the cell gate and $\mathbf{b}_C \in \mathbb{R}^m$ is the bias term. As shown in the figure, the input gate and the cell gate will be combined together to generate the update and directly add to the previous memory cell $\mathbf{C}_{t-1}$.

  Specifically, we can obtain the new memory $\mathbf{C}_t$ by utilizing the forget gate $\mathbf{f}_t$, input gate $\mathbf{i}_t$, and the cell gate $\tilde{C}_t$. The forget gate $\mathbf{f}_t$ along with the previous memory $\mathbf{C}_{t-1}$ will decide how much previous memory will be preserved, while the input gate $\mathbf{i}_t$ and the current candidate $\tilde{C}_t$ determine how much new information shall be added to the memory state. The update formula writes as,

$$\mathbf{C}_t = \mathbf{f}_t * \mathbf{C}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{C}}_t, \tag{5.7}$$

which is a crucial part of the LSTM architecture. We want to remind readers that “$\cdot$” is used for vector-vector or vector-matrix product, while “$*$” is used for element-wise Hadamard product.

- Output gate $\mathbf{o}_t$. The output gate is responsible for selecting relevant information from the memory cell to create the hidden state $\mathbf{h}_t$ which controls the output information. Once the output gate is computed, we may need to add one more prediction layer (e.g., a sigmoid for binary classification) on top of $\mathbf{h}_t$ to generate the actual output at state t (i.e., $\tilde{y}_t$). The following equations demonstrate how the output gate is computed for the hidden state and output prediction:

$$\mathbf{o}_t = \text{Sigmoid}(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o), \tag{5.8}$$

$$\mathbf{h}_t = \mathbf{o}_t * \text{Tanh}(\mathbf{C}_t), \tag{5.9}$$

$$\tilde{y}_t = \text{Sigmoid}(\mathbf{w} \cdot \mathbf{h}_t + b) \tag{5.10}$$

where $\mathbf{W}_o \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix in the output gate and $\mathbf{b}_o \in \mathbb{R}^m$ is the bias term. Here, $\mathbf{o}_t \in \mathbb{R}^m$ represents the output gate, $\mathbf{h}_t$ is the hidden state, and $\tilde{y}_t \in \mathbb{R}$ is the output prediction.

The forget, input, cell, and output gates constitute the interaction mechanisms that enable the LSTM model to capture extended dependencies among sequential events. This characteristic has propelled the LSTM model to become one of the most prevalent variants of RNNs.

PyTorch Implementation of LSTM

In PyTorch code, the LSTM model is easily accessible[1], and we show the example code below.

```python
# initialize an LSTM
"""
10 as the input size
20 as the hidden embedding size
2 as the LSTM gate neural network layers
"""
lstm = torch.nn.LSTM(10, 20, 2)

# generate sequential input and set default hidden state and memory cell

input = torch.randn(5, 3, 10)
"""
5 steps in the sequence,
3 instances in the mini-batch,
each input vector has 10 features
"""

h0 = torch.randn(2, 3, 20)
```

[1] https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

```
"""
2 layers, 3 instances in the mini-batch, and 20 hidden features.
"""

c0 = torch.randn(2, 3, 20) # cell state is the same size as hidden state.

# get output
output, (hn, cn) = lstm(input, (h0, c0))
```

LSTM is a complex model for RNN. Gated recurrent units (GRUs) are a simpler alternative, which will be introduced next.

5.1.3 Architecture of Gated Recurrent Units (GRUs)

The gated recurrent units (GRUs) [10] offer a more efficient version of the LSTM model. They consolidate the functions of the forget gate and the input gate into a single gate, and merge the memory representation with the hidden representation to form a unified hidden representation. The details of the gates in GRUs will be discussed in the following section.

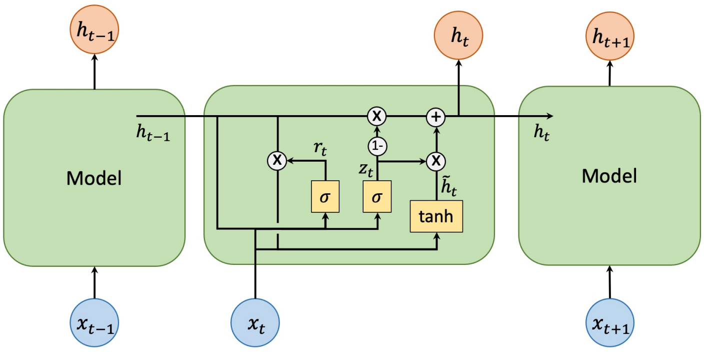

Figure 5.4 The gated recurrent units (GRUs) architecture. GRU is a consolidated version of RNN, which has only two gates: the reset gate and the update gate. It is more flexible and efficient than LSTM.

- *Reset gate* $\mathbf{r}_t$ combines the input data $\mathbf{x}_t$ and the previous memory (i.e., hidden) representations as inputs.

$$\mathbf{r}_t = \text{Sigmoid}(\mathbf{W}_r \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r) \in \mathbb{R}^m, \tag{5.11}$$

where $\mathbf{W}_r \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix in the gate and $\mathbf{b}_r \in \mathbb{R}^m$ is the bias term. This gate will control how much new information should be leveraged to reset the hidden memory state.

Within the reset gate, the GRU model will generate the candidate memory state $\tilde{\mathbf{h}}_t$. It utilizes the reset gate $\mathbf{r}_t$ and the previous memory representation, combined with the data $\mathbf{x}_t$, to generate the candidate memory.

$$\tilde{\mathbf{h}}_t = \text{Tanh}(\mathbf{W}_h \cdot [\mathbf{r}_t * \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h) \in \mathbb{R}^m, \tag{5.12}$$

where $\mathbf{W}_h \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix and $\mathbf{b}_h \in \mathbb{R}^m$ is the bias term.

- *Update gate* $\mathbf{z}_t$. The update gate $\mathbf{z}_t$ generates memory updates in GRUs. It controls how much previous memory and the candidate memory are used to generate the new memory.

$$\mathbf{z}_t = \text{Sigmoid}(\mathbf{W}_z \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z) \in \mathbb{R}^m, \tag{5.13}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) * \mathbf{h}_{t-1} + \mathbf{z}_t * \tilde{\mathbf{h}}_t, \tag{5.14}$$

where $\mathbf{W}_z \in \mathbb{R}^{m\times(m+i)}$ is the weight matrix in the gate, $\mathbf{b}_z \in \mathbb{R}^m$ is the bias term and $*$ is the elementwise multiplication. The new memory in the current state is a linear combination of the previous memory and the memory candidate. The memory will be used for the next state as well as output representation in the current state.

PyTorch Implementation of GRU

In PyTorch, the GRU model can be easily implemented following the documentation[2] and example code below.

```python
# initialize a GRU
"""
10 as the input size
20 as the hidden embedding size
2 as the GRU neural network layers
"""
gru = torch.nn.GRU(10, 20, 2)

# generate sequential input and set default hidden state
"""
5 steps in the sequence,
3 instances in the mini-batch, and
each input vector has 10 features.
"""
input = torch.randn(5, 3, 10)

"""
2 layers, 3 instances in the mini-batch, and
20 hidden features.
"""
h0 = torch.randn(2, 3, 20)

# get output
output, hn = gru(input, h0)
```

Recall previously the return of PyTorch LSTM model has three elements: the output *output*, latest hidden state *hn*, and latest memory cell *cn*. Here the return of PyTorch GRU model only has two elements: the output *output* and the latest hidden state *hn*.

Both LSTM and GRU are commonly used in real-world applications, and their performances are similar.

[2] https://pytorch.org/docs/stable/generated/torch.nn.GRU.html

5.2 Extension and Training the RNN model

5.2.1 Diverse RNN settings

Compared to DNNs, RNNs offer more flexible architectures to address various sequential dependencies. RNN model variants can effectively handle many-to-many configurations beyond the scenarios mentioned earlier, as illustrated in Figure 5.5. While DNN models are designed for one-to-one relationships, RNNs are better suited for the following settings:

- Many-to-many type 1 setting: An RNN model generates predictions on a weekly basis by periodically utilizing weekly data. For example, predicting the status of recovery or relapse over time or forecasting weekly patient recovery progress based on weekly medical observations.
- One-to-many setting: Beginning with a patient’s initial health status, the RNN model predicts their future health status. For example, estimating the stages or severity of the patient’s disease after an initial diagnosis or predicting the progression of a patient’s condition over time.
- Many-to-one setting: Using a patient’s complete longitudinal history, the RNN model generates a single prediction considering all past data. For example, predicting the likelihood of a patient returning to the hospital within a specific timeframe based on their complete medical history.
- Many-to-many type 2 setting: The RNN model forecasts the future progression of a specific health target using past patient data. For example, predicting the evolution of diabetic complications over time based on historical data, such as retinopathy or nephropathy. This setting is also typically used for text translation tasks, such as translating Chinese clinical notes to an English version.

The key difference between type 1 and 2 many-to-many relations: In type 1, input and output sequences have the same length, while in type 2, they can differ in length.

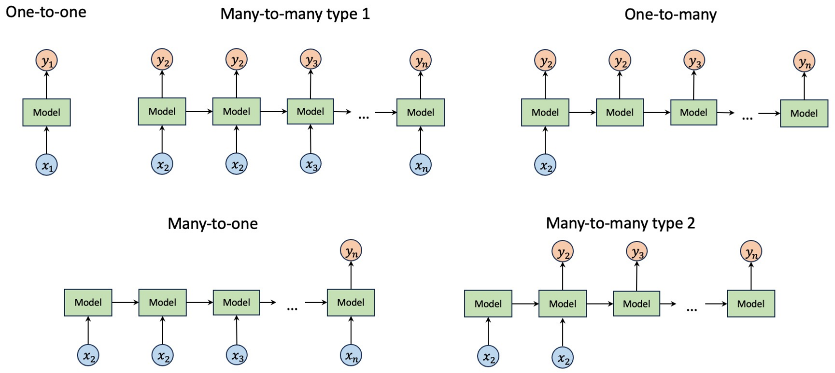

Figure 5.5 Various Sequence-to-Sequence Prediction Types

5.2.2 Training: back-propagation through time (BPTT)

Training RNN models, including LSTMs and GRUs, also employs gradient backpropagation, similar to training a deep neural network. The backpropagation in RNN models is conducted through temporal order which is called backpropagation through time (BPTT), where gradients are computed backwards along the timeline.

However, as sequences can become quite lengthy, potentially reaching 1000 time steps, the propagated gradients may either explode or vanish over long distances. To address this, researchers often partition the long sequence into smaller segments, such as segments of length 20 with overlaps. These segments are treated as independent, allowing gradients to be calculated in parallel per segment. This practical approach is referred to as truncated BPTT, though it may potentially break the model ability on long dependencies in the sequences. Modern deep learning frameworks have built-in truncated BPTT modules, such as PyTorch[3], so users do not have to implement the complicated BPTT update from scratch. Let us look at how to train a GRU model in PyTorch using a simple example below.

Training a GRU model in PyTorch

To prepare the simulation, let us use the sinusoidal function as our data, shown in Figure 5.6.

```
"""
- each input only has 1 dimension
- the output also has 1 dimension
"""
X = torch.arange(10000)
y = torch.sin(X / 500)

plt.plot(X, y)
```

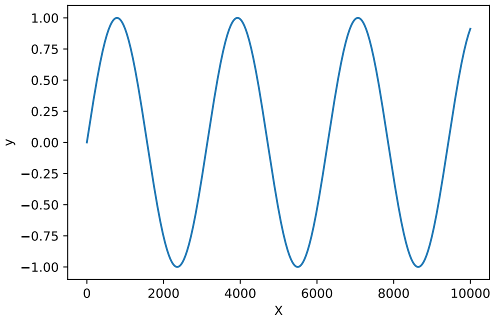

Figure 5.6 The simulated sinusoidal dataset

We then split the dataset into training and test by 80% : 20%. To meet the require-

[3] https://discuss.pytorch.org/t/implementing-truncated-backpropagation-through-time/15500

ments of GRU input formats, we adjust the dimensions of X and y data, so that they could be a 3-dim tensor.

```
# split into train and test
X_train, X_test = torch.FloatTensor(X[:8000].float()), \
                                        torch.FloatTensor(X[8000:].float())
y_train, y_test = torch.FloatTensor(y[:8000]), torch.FloatTensor(y[8000:])

X_train = X_train.unsqueeze(0).unsqueeze(-1)
X_test = X_test.unsqueeze(0).unsqueeze(-1)
y_train = y_train.unsqueeze(0)
y_test = y_test.unsqueeze(0)

X_train.shape, X_test.shape
# torch.Size([1, 8000, 1]), torch.Size([1, 2000, 1])

y_train.shape, y_test.shape
# torch.Size([1, 8000]), torch.Size([1, 2000])
```

Given the data, we could initialize the GRU model from PyTorch, and use mean square error (MSE) loss as the objective and Adam as the optimizer.

```
# input batch size 1, output dimension 1, number of hidden layers 2
gru = torch.nn.GRU(1, 1, 2)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(gru.parameters(), lr=0.01)
```

We are ready to train the GRU model with our sinusoidal dataset. In total, we train the GRU model for 200 epochs. We use full batch gradient descent and feed the whole training data to the model at once during each epoch. Then, we calculate the loss and use Adam optimizer to update the parameters of GRU. The new GRU model is evaluated on the test data at the end of each epoch. The training loss and the test loss are stored per epoch for visualization purposes.

```
train_loss, test_loss = [], []

for epoch in range(200):
    gru.train()
    # predict on the training data
    h0 = torch.randn(2, 8000, 1)
    pre, _ = gru(X_train, h0)

    # calculate the loss
    loss = criterion(pre.squeeze(-1), y_train)
    train_loss.append(loss.item())

    # gradient update
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    gru.eval()
    with torch.no_grad():
        h0 = torch.randn(2, 2000, 1)
        pre, _ = gru(X_test, h0)
        loss = criterion(pre.squeeze(-1), y_test)
        test_loss.append(loss.item())
```

During the whole training process, we use the standard PyTorch training pipeline and do not explicitly leverage the truncated BPTT algorithm, which is handled within the PyTorch internal function calls.

In Figure 5.7, we find that the GRU model gets improved before the 100th epoch, and both curves are decreasing steadily. After that, the model performance does not improve any more.

```
plt.plot(train_loss, label="train_loss")
plt.plot(test_loss, label="test_loss")
```

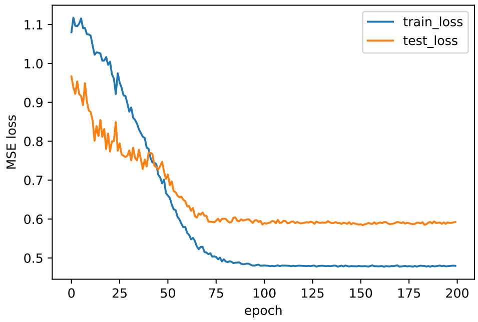

Figure 5.7 The training and test loss curves of GRU

We provide this PyTorch example in the public repository[4].

5.3 Heart Failure Onset Prediction with LSTM

In this section, we aim to predict heart failure using diagnostic codes, leveraging a dataset synthesized from MIMIC-III. The comprehensive process will be facilitated through the implementation of the PyHealth package, streamlining the analysis and prediction procedures.

Task of Heart Failure Prediction

Heart failure is a complex clinical condition arising from structural or functional abnormalities in the ventricular filling or ejection of blood. Detecting the onset of heart failure is crucial and demands prompt medical attention. However, predicting this onset is a formidable task due to the intricate nature of the ailment and the dynamic shifts in physiological parameters. Previous works have encoded the electronic health

[4] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap3-RNN/notebook

records (EHR) data [11] and leveraged the RNN models [6, 12, 13] for predicting the onset of heart failure.

In the heart failure prediction task, the data in this section are obtained from MIMIC-III records and are organized at the patient level. Each patient possesses a sequence of hospital visits, with each visit containing a list of diagnosis codes. For every patient, we extract a binary heart failure label (1 indicates the onset of heart failure, otherwise, 0), and the entire sequences of patient visits serve as the feature inputs.

Within the application of heart failure prediction, the LSTM network can extract temporal patterns in the patient's data. For instance, an increasing trend of blood pressure or severe kidney functioning indicators could serve as crucial signs of an impending heart failure event. The LSTM network has the capability to capture the sequential patterns, offering an early warning signal that allows for timely intervention.

5.3.1 Heart Failure Dataset

The dataset synthesized from MIMIC-III contains several *.pkl* files for both the training and test portion, which could be downloaded from this folder[5]. We show the contents below.

- **pids.pkl**: contains the patient ids;
- **vids.pkl**: contains a list of visit ids for each patient;
- **hfs.pkl**: contains the heart failure label (0: normal, 1: heart failure) for each patient;
- **seqs.pkl**: contains a list of visits (in ICD9 codes) for each patient;
- **types.pkl**: contains the map from ICD9 codes to ICD9 labels;
- **rtypes.pkl**: contains the map from ICD9 labels to ICD9 codes.

Let us load the training data and take a look at a specific patient.

```python
# load the training data
DATA_PATH = "./"
pids = pickle.load(open(os.path.join(DATA_PATH,’train/pids.pkl’), ’rb’))
vids = pickle.load(open(os.path.join(DATA_PATH,’train/vids.pkl’), ’rb’))
hfs = pickle.load(open(os.path.join(DATA_PATH,’train/hfs.pkl’), ’rb’))
seqs = pickle.load(open(os.path.join(DATA_PATH,’train/seqs.pkl’), ’rb’))
types = pickle.load(open(os.path.join(DATA_PATH,’train/types.pkl’), ’rb’))
rtypes = pickle.load(open(os.path.join(DATA_PATH,’train/rtypes.pkl’), ’rb’))

# show the information from the 3rd patient
print("Patient ID:", pids[3])
print("Heart Failure:", hfs[3])
print("# of visits:", len(vids[3]))
for visit in range(len(vids[3])):
    print(f"\t{visit}-th visit id:", vids[3][visit])
    print(f"\t{visit}-th visit diagnosis labels:", seqs[3][visit])
    print(f"\t{visit}-th visit diagnosis codes:",
                    [rtypes[label] for label in seqs[3][visit]])
```

The printed results are shown below.

[5] https://github.com/sunlabuiuc/pyhealth-book/blob/main/chap3-RNN/notebook

```
"""
OUTPUT:
Patient ID: 47537
Heart Failure: 0
# of visits: 2
  0-th visit id: 0
  0-th visit diagnosis labels: [12, 103, 262, 285, 290, 292, 359, 416, 39,
                225, 275, 294, 326, 267, 93]
  0-th visit diagnosis codes: [’DIAG_041’, ’DIAG_276’, ’DIAG_518’,
                ’DIAG_560’, ’DIAG_567’, ’DIAG_569’, ’DIAG_707’, ’DIAG_785’,
                ’DIAG_155’, ’DIAG_456’, ’DIAG_537’, ’DIAG_571’, ’DIAG_608’,
                ’DIAG_529’, ’DIAG_263’]
  1-th visit id: 1
  1-th visit diagnosis labels: [12, 103, 240, 262, 290, 292, 319, 359,
                510, 513, 577, 307, 8, 280, 18, 131]
  1-th visit diagnosis codes: [’DIAG_041’, ’DIAG_276’, ’DIAG_482’,
                ’DIAG_518’, ’DIAG_567’, ’DIAG_569’, ’DIAG_599’, ’DIAG_707’,
                ’DIAG_995’, ’DIAG_998’, ’DIAG_V09’, ’DIAG_584’, ’DIAG_031’,
                ’DIAG_553’, ’DIAG_070’, ’DIAG_305’]
"""

print("number of heart failure patients:", sum(hfs))
print("ratio of heart failure patients: %.2f" % (sum(hfs) / len(hfs)))

"""
OUTPUT:
number of heart failure patients: 548
ratio of heart failure patients: 0.55
"""
```

Since the dataset is already well-prepared and the task is clearly defined, the next step is to reformat this processed data to seamlessly integrate it with the PyHealth package.

5.3.2 Step1 & 2: customized dataset wrapping and split

Using the provided dataset, we can employ the *pyhealth.dataset.SampleEHRDataset* API to transform it into the PyHealth supported structure. As we have learned from Section 4.3.2 that the data samples in PyHealth are stored in JSON structure, the following code will essentially organize the patient ID, visit ID, label, and diagnosis information from this heart failure dataset into one JSON structure, subsequently adding it to the samples list.

```
"""
With this user processed data, we just need to wrap up the data to fit
the pyhealth format.

Task definition: we will use all the visits from the patients as the features,
    then, the labels are given by the hfs list.
"""
samples = []
for pid, vid, hf, seq in zip(pids, vids, hfs, seqs):
    samples.append(
        {
            'patient_id': pid,
            'visit_id': vid[-1],
            'label': hf,
            'diagnoses': [[rtypes[v] for v in visit] for visit in seq],
        }
    )
```

Then, we apply the *pyhealth.dataset.SampleEHRDataset* wrapper and generate a pyhealth dataset object. We show the first sample of this training dataset.

```
from pyhealth.datasets import SampleEHRDataset
train_dataset = SampleEHRDataset(samples)

# check the first sample
train_dataset.samples[0]
"""
{
 'patient_id': 89571,
 'visit_id': 1,
 'label': 1,
 'diagnoses': [
    ['DIAG_250', 'DIAG_285', 'DIAG_682', 'DIAG_730', 'DIAG_531',
    'DIAG_996', 'DIAG_287', 'DIAG_276', 'DIAG_E878', 'DIAG_V45', 'DIAG_996'],
    ['DIAG_250', 'DIAG_276', 'DIAG_285', 'DIAG_998', 'DIAG_996',
    'DIAG_078', 'DIAG_336', 'DIAG_E878', 'DIAG_401', 'DIAG_205']
    ]
}
"""
```

After applying the wrapper, the next steps all follow standard pyhealth operations, such as the dataset split below. Since the above data only contains the training set, we split it into training and validation set by 80% : 20%. We will process the test set in a similar way later.

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

# dataset split by patient id
train_ds, val_ds, test_ds = split_by_patient(train_dataset, [0.8, 0.2, 0])

# obtain train/val/test dataloader, they are <torch.data.DataLoader> object
train_loader = get_dataloader(train_ds, batch_size=64, shuffle=True)
val_loader = get_dataloader(val_ds, batch_size=64, shuffle=False)
```

5.3.3 Step 3: initialize the ML model

Right now, we obtained the training loader and the validation loader, we want to initialize an LSTM model. We call the *pyhealth.model.RNN* API below. The API uses the training dataset, set diagnoses key as the features, choose LSTM as the RNN type, embedding dimension and the hidden dimension are both set at 64, and configure other attributes on need. The initialization steps are similar to the DNN initialization in the last section.

```python
from pyhealth.models import RNN

model = RNN(
    dataset=train_dataset,
    feature_keys=["diagnoses"],
    label_key="label",
    mode="binary",
    rnn_type="LSTM",
    num_layers=2,
    embedding_dim=64,
    hidden_dim=64,
)
```

5.3.4 Step 4: Model training

Similar to previous pipelines, we use the *pyhealth.trainer.Trainer* to handle the model training and evaluation. During the training, we record the precision-recall area under curve (PRAUC), AUROC, and the F1 measure as the evaluation metrics. We set the training epochs to 20 and monitor the AUROC curves to select the best model based on the validation set. The best model will automatically be loaded in the Trainer for the next evaluation step. For more information about the configuration of the Trainer, please refer to this page[6].

```python
from pyhealth.trainer import Trainer

trainer = Trainer(
    model=model,
    metrics=["pr_auc", "roc_auc", "f1"]
)

trainer.train(
    train_dataloader=train_loader,
    val_dataloader=val_loader,
    epochs=20,
    monitor="roc_auc",
) # training ...
```

5.3.5 Step 5: Evaluation on test set

So far, we have processed the training set into train and validation and obtain the best trained LSTM model. Now, we will similarly process the test data into PyHealth format, use the same pyhealth wrapper to get a test data loader and compute the final evaluation metrics. The *trainer.evaluate*() function automatically uses the best trained LSTM model for evaluation.

```python
# load the test data
DATA_PATH = "./"
```

[6] https://pyhealth.readthedocs.io/en/latest/api/trainer.html

```
pids = pickle.load(open(os.path.join(DATA_PATH,’test/pids.pkl’), ’rb’))
vids = pickle.load(open(os.path.join(DATA_PATH,’test/vids.pkl’), ’rb’))
hfs = pickle.load(open(os.path.join(DATA_PATH,’test/hfs.pkl’), ’rb’))
seqs = pickle.load(open(os.path.join(DATA_PATH,’test/seqs.pkl’), ’rb’))
types = pickle.load(open(os.path.join(DATA_PATH,’test/types.pkl’), ’rb’))
rtypes = pickle.load(open(os.path.join(DATA_PATH,’test/rtypes.pkl’), ’rb’))

# wrap up into pyhealth format and obtain the test loader
test_samples = []
for pid, vid, hf, seq in zip(pids, vids, hfs, seqs):
    test_samples.append(
        {
            ’patient_id’: pid,
            ’visit_id’: vid[-1],
            ’label’: hf,
            ’diagnoses’: [[rtypes[v] for v in visit] for visit in seq],
        }
    )

test_dataset = SampleEHRDataset(test_samples, code_vocs=None)
test_loader = get_dataloader(test_dataset, batch_size=64, shuffle=False)

# generate the evaluation output
result = trainer.evaluate(test_loader)
print (result)
```

After running, we receive the following output.

```
"""
OUTPUT:
{’pr_auc’: 0.7694151513845636, ’roc_auc’: 0.7851632456261194,
        ’f1’: 0.7318840579710146, ’loss’: 0.6059411615133286}
"""
```

For this heart failure onset prediction using LSTM model, a complete pipeline is contained in the notebook published at this folder[7].

Questions

- Why are common DNN models not suitable for handling sequence data? What models should we use instead?
- What is a recurrent neural network (RNN) and how is it different from a traditional Neural Network?
- What is the issue of vanishing and exploding gradients in RNNs? How does it affect RNN training?
- What is the purpose of the hidden state in an RNN? How does it assist with processing sequential data?
- Explain the architecture of an LSTM (Long Short-Term Memory) network. How does it solve the long-term dependency problem in RNNs? What is the concept of "gates" in an LSTM? What are the various types of gates and their roles?

[7] https://github.com/sunlabuiuc/pyhealth-book/blob/main/chap3-RNN/notebook/

- How can RNNs be employed for time-series forecasting in healthcare? Can you describe a concrete example how RNNs could be used in clinical decision making?
- Describe a real-world healthcare application where RNNs have been successfully used. What was the impact of the application?
- Programming question:
  - **Datasets and Background:** You are provided with a dataset of hospital admissions derived from MIMIC-III. Each admission is represented as a list of diagnosis codes during the patient's hospital stay. The sample in the dataset consists of a sequence of prior admissions, associated with a length of stay label for the next admission. We categorize the length of stay durations into several bins:

    ```
    if days < 1:
        return 0
    elif 1 <= days <= 7:
        return days
    elif 7 < days <= 14:
        return 8
    else:
        return 9
    ```

    In total, we provide 35,536 samples for training, 4,441 samples for validation, and 4,440 samples for evaluation.
  - **Goal:** Your task is to build an RNN model to predict the length of a patient's hospital stay label based on the given admission sequences. The model should take a sequence as input and output a single number representing the predicted length of stay label in $\{0, 1, \cdots, 9\}$. The performance of the model should be evaluated using *Accuracy* and *F1* between the predicted and actual length of stay labels.
  - **Requirements:** A detailed description of your model architecture (code snippets of PyTorch class), including the type of RNN used, the number of layers, the number of hidden units, and any other relevant details. The code for training the model, including any preprocessing step, the loss function used, the optimization algorithm, and any regularization techniques. A plot of the training and validation loss over time, as well as the final *Accuracy* and *F1* on the test set. A discussion of the results, including any challenges faced, any steps taken to overcome these challenges, and any potential improvements that could be made.
  - **Hint:** Employ an embedding table, learn the embeddings for each diagnosis code, sum up the diagnosis embedding as the admission embedding, and finally leverage the RNNs to model a sequence of admission embeddings.