7 Attention Models

There are a few crucial concepts in recent neural networks and the *attention mechanism* is one of the most popular newly emerging concepts. In this chapter, we will introduce the attention mechanism, which was originally a strong substitution of RNNs in sequence modeling. While the RNN model processes the tokens in a sequence linearly, the attention mechanism has the capacity to learn the token interactions simultaneously regardless of whether two tokens are neighbors. In Chapter 8, we will delve further into another powerful model *Transformer*, which is based on the attention mechanism and has become a backbone in many AI applications including large language models.

In this chapter, we first introduce the sequence-to-sequence (seq2seq) prediction task. This task involves taking a sequence as input, such as a patient's historical disease diagnoses, and generating a new target sequence, such as future medications. We will use the seq2seq problem to introduce the concepts of attention models, which advances the common RNN approaches by allowing the full interactions between input and output sequences.

Subsequently, we show two concrete healthcare predictive examples using attention modules. First, we learn how to use attention modules to get meaningful clinical notes embeddings for classifying the disease codes of a patient. Specifically, we delve into the CAML model [22] and show its PyTorch implementation for handling the task. In another example, we will introduce a simple yet powerful attention-based prediction model called RETAIN [8] and discuss its application in heart failure prediction with its PyHealth implementation. It is worth noting that the attention mechanism is a powerful model that has been widely used in many different neural network architectures, including RNN [8], CNN [22], Graph neural networks [23, 24], Memory networks [25], and Transformer [26].

7.1 Sequence-to-Sequence Task

The task of sequence-to-sequence (seq2seq) is to generate a target sequence based on a source sequence. Typically these two sequences can have different lengths, such as when translating a prescription written in French to English or predicting the list of medications given a sequence of diagnosis codes.

The most common application of seq2seq is translation between different languages.

In recent years, such neural network models have greatly improved the translation systems of different providers like Google Translate and Bing Microsoft Translator.

In healthcare, it is also necessary to "translate" different clinical information from one type of sequence to another. For example, by observing a patient's historical clinical diagnoses, we can predict the corresponding medication prescription for the patient. A Seq2seq model can also assist in longitudinal patient modeling and forecast based on time-series data such as vital signs, lab results, or patient history. These time series are treated as input sequences that predict another sequence of future health conditions.

Seq2seq Problem Formulation

Let us consider that the source sequence is represented by $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m]$. For example, each element $\mathbf{x}_i$ in the sequence can be considered as an embedding vector for individual diagnosis code. On the other hand, the target sequence is represented by $\mathbf{y} = [y_1, y_2, \ldots, y_n]$, which can have a different length from the source sequence. For example, each element y_i can be a medication. The goal of the seq2seq task is to develop a model that can predict the probability of $p(\mathbf{y} \mid \mathbf{X})$ and select the results with the highest probability as the final output, which is mathematically defined by $\mathbf{y}^* = \arg\max_{\mathbf{y}} p(\mathbf{y} \mid \mathbf{X})$. In this healthcare application, we want to predict the sequence of medications based on the sequence of past diagnoses.

7.1.1 Two-RNN approach

Before we introduce the attention mechanism for solving the seq2seq problem, we would like to present a simple RNN-based approach that provides a foundation for understanding the attention mechanism. As we learned from Chapter 5, RNN models are effective for capturing sequential information. However, previous RNN models only consider dynamic input (e.g., all the past visits of a patient) while predicting a fixed output (e.g., the probability of heart failure). To tackle the seq2seq task, which requires a dynamic target sequence, as introduced by Sutskever et al. in 2014 [27], we need to use two distinct RNN models: one for encoding the source sequence and the other for generating the target sequence. For instance, the source RNN could encode a patient's historical visits, while the target RNN decodes the sequence of future medications, as presented in this work [28].

To illustrate this methodology, Figure 7.1 demonstrates its application in **modeling the diagnosis codes in International Classification of Disease version 10 (ICD10) code (as input sequence) to generate a list of recommended medications in Anatomical Therapeutic Chemical (ATC) code (as output sequence)**. The process of generating medication recommendations involves two steps: encoding RNN and decoding RNN. To enhance user understanding, we provide the code snippets of PyTorch implementations after the explanation.

Encoding RNN

First, the *encoder RNN* takes a sequence of past diagnosis codes and generates a final hidden embedding that represents the entire input sequence. This embedding is called

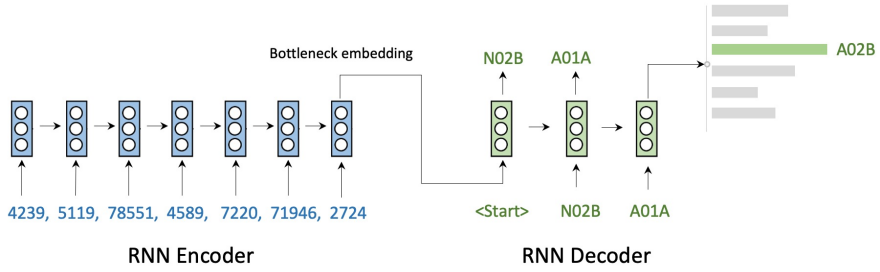

Figure 7.1 This illustration describes the process of using RNNs as an encoder and a decoder to translate a sequence of diagnoses into recommended medications. The encoder takes the diagnosis sequence as input and encodes it into an embedding vector, named "bottleneck embedding", while the decoder generates a list of recommended medications.

the "*bottleneck embedding*" and serves as the input to the decoder model. The quality of decoder generation relies heavily on the quality of the "bottleneck embedding."

```
# user inputs the diagnosis sequence of shape (1, m, 128)
# 1 is the size of batch
# m represents the input sequence length
# 128 is the diagnosis code embedding dimensions
X = ...

encoder = torch.nn.GRU(
    input_size=128,
    hidden_size=256,
    batch_first=True
)

# the output has dimension (1, m, 256)
# 1 is the size of batch
# m represents the sequence length, same value as before
# 256 represents the output dimension of RNN
_, hn = encoder(X)

# we only take the very final encoder output as the bottleneck embedding.
# bottleneck has dimension (1, 256).
# each row of the bottleneck embedding represents one sample.
bottleneck = hn[:, -1]
```

Decoding RNN

During the decoding phase, the decoder RNN utilizes the bottleneck embedding vector to create a list of medication recommendations. To begin, it generates the first medication, such as "N02B" (an ATC-3 level code that means "Antipyretic and Analgesics"), inputting the bottleneck embedding from the encoder RNN and the "start" token "<start>". Subsequently, the generated diagnosis code, "N02B", becomes the next input token used alongside a new hidden embedding to generate the subsequent token, such as "A01A" (another ATC-3 level code that means "Stomatological Prepa-

rations"). This process continues until the "<end>" token is generated. The procedure is shown in Figure 7.1.

This two-RNN approach follows a greedy manner by generating one token at a time with the highest probability and then generating the next token following the same procedure. Mathematically, it operates as

$$\mathbf{h}_0 = \mathbf{c}_0. \tag{7.1}$$

$$\mathbf{h}_{i+1} = \text{RNN}(\hat{y}_i, \mathbf{h}_i), \ \forall\, i \geq 0. \tag{7.2}$$

$$\hat{y}_{i+1} = \arg\max_y p(y \mid \mathbf{h}_{i+1}). \tag{7.3}$$

Equation (7.1) initializes the very first decoder hidden state $\mathbf{h}_0$ by the bottleneck embedding $\mathbf{c}_0$ from the last state of the encoder (also known as the context vector). Equation (7.2) updates the hidden embedding based on the previous hidden embedding and the current input token using the RNN decoder model, where the initial input token is $\hat{y}_0 =< start >$. During decoding, the model follows Equation (7.3), computes probabilities for each token in the vocabulary, and selects the token that has the highest probability as the next predicted token. Finally, the list of output $[\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_i, \cdots]$ are the recommended medications based on the input diagnoses.

```
# initialize the first decoder hidden state by bottleneck embedding
h0 = bottleneck

# the possible ATC-3 level medication code space with size V+2.
# index V is the embedding for <start>
# index V+1 is the embedding for <end>
Med = ["A01A", "A01B", ..., "<start>", "<end>"]

# embedding table for every token
Emb = torch.nn.Embedding(len(Med), 128)

# initialize the decoder model
decoder = torch.nn.GRU(
    input_size=256,
    hidden_size=128,
    batch_first=True
)

# We run the decoding process with <start> token.
cur_token_index = V # indicate
```

With the above preparation, we could use the following PyTorch code snippets for model training. The training loss is calculated as noise contrastive estimation (NCE), that maximizes the dot product of predicted token embedding with the ground truth token embedding while minimizing the dot product of predicted token embedding and random k token embeddings. This loss function is also used in the famous Word2Vec paper [29].

Noise Contrastive Estimation (NCE). Let us specify the NCE loss below. Given the predicted token embedding $\hat{\mathbf{e}}$, the ground truth token embedding $\mathbf{e}_0$, and k randomly sampled token embeddings from the embedding table, $[\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_k]$, we could define

the NCE loss in a softmax form,

$$NCE = \frac{\exp(\langle \hat{\mathbf{e}}, \mathbf{e}_0 \rangle)}{\exp(\langle \hat{\mathbf{e}}, \mathbf{e}_0 \rangle) + \exp(\langle \hat{\mathbf{e}}, \mathbf{e}_1 \rangle) + \cdots + \exp(\langle \hat{\mathbf{e}}, \mathbf{e}_k \rangle)}. \tag{7.4}$$

```
""" for model training """
decoder.train()

gt = ["N02B", "A01A", "A02B", ...] # with length of M
loss = 0

for i in range(n):
    # get embedding for the current token
    cur_input_emb = Emb(cur_token_index)
    # run decoder by giving the current token embedding and hidden embedding
    out, h1 = decoder(cur_token_emb, h0)
    # update hidden state
    h0 = h1
    # get the embedding of the truth token at the current step
    cur_token_index = Med.index(gt[i])
    target_out = Emb(cur_token_index)

    """ calculate the loss (use noise contrastive estimation)
        1. calculating the dot product of out and target_out as d0
        2. same calculation with embeddings of k random tokens as d1, ..., dk
        3. maximize exp(d0) / (exp(d0) + exp(d1) + ... + exp(dk))
    """
    k = 100
    random_idx = torch.randint(len(Med)-2, k)
    random_emb = Emb(random_idx)
    # compute the dot product for out and target_out as d0
    pos = torch.dot(out, target_out)
    # compute the dot product of out and other k random tokens.
    neg = out @ random_emb.T
    # epoch loss calculation
    epoch_loss = torch.exp(pos) / (torch.exp(pos) + torch.exp(neg).sum())

    # accumulate loss and update model after the loop
    loss += epoch_loss
loss.backward()
```

If we have the well-optimized models, similar PyTorch snippets could be employed to generate the recommended medications based on the diagnosis sequence.

```
""" for model evaluation """
model.eval()

predicted_med = []
# the decoding process will stop when the current token is <end>
with torch.no_grad():
    while (cur_token_index != V+1)
        # get embedding for the current token
        cur_input_emb = Emb(cur_token_index)
        # run decoder by current token embedding and hidden embedding
        out, h1 = decoder(cur_token_emb, h0)
        # use the output embedding to find the best matched token
```

```
        # out @ Emb.T performs the dot product of the two embedding vectors
        cur_token_index = torch.max(out @ Emb.T, 0)[1]
        # update the hidden
        h0 = h1

        # add the predicted medication to the results
        if cur_token_idx != V+1:
            predicted_med.append(Med[cur_token_idx])

result = predicted_med
```

7.1.2 Beam search

In the seq2seq modeling task, the above two-RNN model follows a greedy approach, wherein the quality of the entire decoded sentence can be greatly influenced by previously selected words due to error propagation. Merely selecting the most probable words at each stage may not guarantee the best output sentence. To address this limitation, beam search [30], a commonly employed heuristic algorithm, tracks multiple highly probable hypotheses throughout the decoding process.

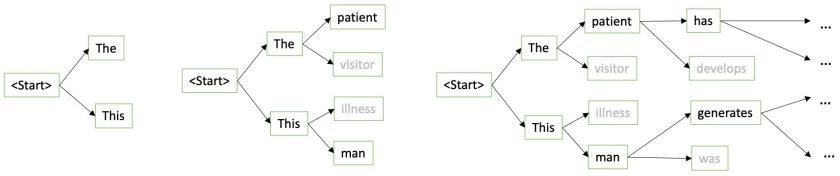

Figure 7.2 Beam search process.

In Figure 7.2, we can see that during the decoding process, each hypothesis is extended, and the top-N most probable continuations are kept. Typically, the beam size falls within the range of 4 to 10. However, going for an even larger beam size can be computationally inefficient and often leads to a drop in output quality. So, it is important to select an optimal beam size that balances computational resources with the desired output quality.

7.2 Cross Attention for Seq-to-seq Task

7.2.1 Inflexibility of two-RNN approach

The above approach uses two Recurrent Neural Networks (RNNs) to process source sentences. However, this approach has significant limitations. Firstly, the encoder RNN may not be able to compress the entire source sentence into a vector-form "bottleneck embedding" without losing essential details. In particular, we often refer to this static embedding vector as **a static context**. Secondly, the decoder needs to interact with

the input sequence more adaptively to receive signals from different parts of the input across different decoding steps. More specifically, we hope to construct **a dynamic context** that depends on the output step.

A more natural way of decoding is to generate one token at a time while considering both the previously decoded words and the corresponding tokens in the source sequence, similar to how humans do it. This idea inspires the development of attention-based seq2seq models.

7.2.2 RNN encoder with attention decoder

The introduction of the attention mechanism in the field (Bahdanau et al., 2014 [31]) marks a significant advancement in sequence-to-sequence tasks.

The section introduces an attention-based seq2seq prediction model. Its core principle involves two key aspects: (i) the attention mechanism allows individual input tokens to possess their own embeddings, different from the conventional method where the RNN encoder generates a single compressed embedding for the entire input source sentence; (ii) during decoding, the model considers not just the current hidden states but also embeddings of all input tokens. This enables the model to dynamically focus on different parts of the input sequence at each decoding step, leveraging this flexibility to generate the most appropriate new token in output.

An illustrative depiction of the attention-based sequence-to-sequence model architecture can be observed in Figure 7.3.

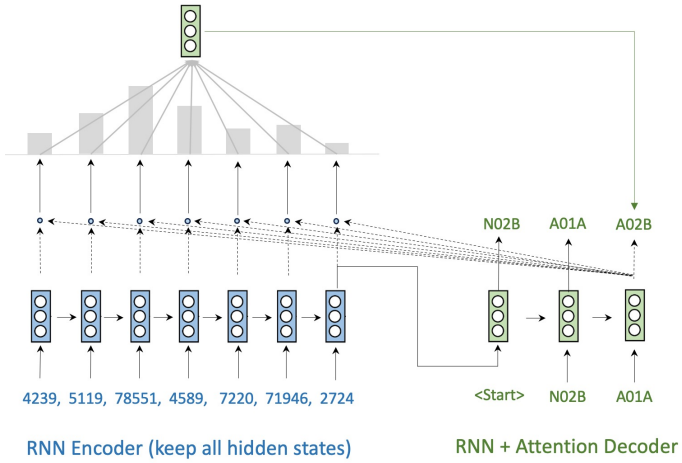

Figure 7.3 RNN encoder with attention-decoder

RNN Encoder

For this new attention-based model, the encoder (shown in Figure 7.3) architecture still follows the common RNN setting, which can be an LSTM or GRU unit.

Let us denote the input sequence as $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ (the sequence of diagnosis codes).

We use the RNN to generate the final hidden states as the bottleneck embedding for the start of the decoder. One difference here is that we also keep the step-wise hidden states for each token as the **source** contextual token embedding, named $\mathbf{S} = [\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_n]$.

```
# user inputs the diagnosis sequence of shape (1, m, 128)
# 1 is the batch size
# m represents the input sequence length
# 128 is the diagnosis code embedding dimensions
X = ... # torch.Tensor([x1, x2, x3, ...])

encoder = torch.nn.GRU(
    input_size=128,
    hidden_size=256,
    batch_first=True
)

# the output has dimension (1, m, 256)
# 1 is the batch size
# m represents the sequence length, the same value as before
# 256 represents the output dimension of RNN
_, hn = encoder(X)

# we only take the very final encoder output as the bottleneck embedding.
# bottleneck has dimension (1, 256).
# each row of the bottleneck embedding represents one sample.
bottleneck = hn[:, -1]
# assign the step-wise token output to S
S = hn
```

Compared to the two-RNN approaches, the encoder in attention-based models leverages the output representation $\mathbf{S}$ for each source token rather than solely relying on the final hidden *bottleneck embedding*.

Decoder: Attention Mechanism Decoder

The decoder model uses another RNN, where the crucial element is integrating the attention mechanism. For each decoding step (generating a new token), the attention mechanism operates in the following manner.

The PyTorch implementation of this decoder is mainly similar to the two-RNN approach in Section 7.1.1 in terms of training and inference, except that we need to modify Line#11 for both training and inference. Line#11 for the two-RNN approach essentially calls the RNN decoder model to update the decoding hidden state and output a predicted token embedding. In this attention-based decoder module, we must update the output token embedding by interacting with the source token embeddings via the attention mechanism.

- First, this model also goes through the RNN recurrent formula to update the hidden state and generate the RNN output for the current step.

```
# cur_token_emb is the embedding for the current input token
# h0 is the hidden state at time (t-1), $h_{t-1}$
output, h1 = decoder(cur_token_emb, h0)
# h1 is the updated state at current time t, $h_t$
```

```
# output is the RNN output
```

- Given the current hidden state $\mathbf{h}_t$ (it is $h1$ shown in the code), the attention mechanism will calculate the attention score between $\mathbf{h}_t$ and the contextual token embeddings in the input sequence, which is a weighted combination of the embedding vectors of all input words. These attention scores determine the weight of each word in the input sequence for generating the next output token.

  Formally, this involves applying an attention function that takes the decoder hidden state $\mathbf{h}_t$ and one-word embedding output $\mathbf{s}_i$, $i = [1, 2, \ldots, m]$, as inputs and produces a scalar value, Attention$(\mathbf{h}_t, \mathbf{s}_i)$. We will introduce different types of attention functions in the next subsection.

$$\text{Attention}(\mathbf{h}_t, \mathbf{s}_i) = \langle \mathbf{h}_t, \mathbf{s}_i \rangle \tag{7.5}$$

```
def attention(h, s):
    return torch.dot(h,s) # will introduce different variants later

# output is the current output state
attentions = [attention(h1, S[i]) for i in range(m)]
```

- These scores are then normalized through a softmax operation to form a probability distribution.

$$Normalized\text{Attention}(\mathbf{h}_t, \mathbf{s}_i) = \frac{\exp(\text{Attention}(\mathbf{h}_t, \mathbf{s}_i))}{\sum_{k=1}^{m} \exp(\text{Attention}(\mathbf{h}_t, \mathbf{s}_k))}; \tag{7.6}$$

```
attentions = np.exp(attentions)
denominator = np.sum(attentions)
attentions = attentions / denominator

""" for example """
# initial attention scores
attentions = [10, 11, 9.5]
attentions = np.exp(attentions) # [22026.46, 59874.14 , 13359.72]
denominator = np.sum(attentions) # 95260.33
attentions = attentions / denominator # [0.2, 0.63, 0.14]
```

Remark. To improve the stability of the exponential function, the above implementation needs to be improved. When the attention score is high before softmax (e.g., 100), the exponential function will blow up (e.g., $exp(100) = e^{100}$). Thus, engineers usually reduce the attention scores by a common large value and then apply the softmax function. Here, a_i represents the attention scores, and t could be an arbitrary real number.

$$\frac{exp(a_i)}{exp(a_1) + exp(a_2) + \cdots} = \frac{exp(a_i - t)}{exp(a_1 - t) + exp(a_2 - t) + \cdots}, \forall\, t \in \mathbb{R}. \tag{7.7}$$

We see the modification of the implementation below.

```
attentions = np.array(attentions)
attentions = attentions - np.max(attentions) # so that the max attention
                                             score is 0
```

```
attentions = np.exp(attentions)
denominator = np.sum(attentions)
attentions = attentions / denominator

""" for example """
# initial attention scores
attentions = [10, 11, 9.5]
attentions = np.array(attentions)
attentions = attentions - np.max(attentions) # subtract the max score to
                                               get [-1, 0, -1.5]
attentions = np.exp(attentions) # [0.36787944, 1.0, 0.22313016])
denominator = np.sum(attentions) # 1.5910096013198722
attentions = attentions / denominator # [0.23, 0.63, 0.14]
```

- Finally, the normalized attention scores and the source token representations from the input embeddings are combined by weighted sum, providing a contextual output embedding.

$$\text{Output} = \sum_{i=1}^{m} Normalized\text{Score}(\mathbf{h}_t, \mathbf{s}_i) \cdot \mathbf{s}_i. \tag{7.8}$$

```
output = attentions * S # vector-matrix product for weighted sum
```

Similar to the two-RNN approach, this new attention-based output: (i) for the training process, it will be used to calculate the NCE loss (defined in Section 7.1.1) for optimizing the model; (ii) for inference, it would be used for retrieving the most similar token from the embedding table by the dot product.

In this way, the attention model showcases enhanced flexibility and a superior capability to enable the long-range interactions between source and target sequences. It could adaptively determine the relative significance of different source elements at each decoder step and harness interactions between any two events between the input sequence and the output sequence.

7.2.3 Attention function

This subsection introduces several common attention functions, $\text{Attention}(\mathbf{h}_t, \mathbf{s}_i)$, that calculates the relevance between hidden decoder state $\mathbf{h}_t$ and the source token hidden embedding $\mathbf{s}_i$.

- **Dot product**: which requires the hidden decoder state $\mathbf{h}_t$ to have the same dimension as the source token embedding $\mathbf{s}_i$. It assumes that two latent embedding spaces are aligned, and higher dot product results mean higher relevance. Dot product attention is the simplest attention function that does not need learnable parameters.

$$\text{Score}(\mathbf{h}_t, \mathbf{s}_i) = \langle \mathbf{h}_t, \mathbf{s}_i \rangle = \mathbf{h}_t^\top \mathbf{s}_i; \tag{7.9}$$

```
def attention(h, s):
    # h and s are two vectors that have the same dimensions
    return torch.dot(h,s)
```

- **Bi-linear function**: which takes a bilinear form with parameter matrix $\mathbf{W}$. It essentially re-weights the relations between every cross-dimensional terms between $\mathbf{h}_t$ and $\mathbf{s}_i$, used in this work [32]. Note that, $\mathbf{h}_t$ and $\mathbf{s}_i$ do not need to follow the same dimension.

$$\text{Score}(\mathbf{h}_t, \mathbf{s}_i) = \mathbf{h}_t^\top \mathbf{W} \mathbf{s}_i; \tag{7.10}$$

```
def attention(h, s, W):
    # h: (d1,), s: (d2,), W: (d1, d2)
    return h.T.reshape(1, -1) @ W @ s.reshape(-1, 1)
```

- **Multi-layer perceptron**: which does not assume any relational form between $\mathbf{h}_t$ and $\mathbf{s}_i$. This general method uses a two-layer neural network to learn the underlying relations between $\mathbf{h}_t$ and $\mathbf{s}_i$, proposed by the original paper [31].

$$\text{Score}(\mathbf{h}_t, \mathbf{s}_i) = \mathbf{w}_2^\top \cdot \tanh(\mathbf{W}_1[\mathbf{h}_t, \mathbf{s}_i]). \tag{7.11}$$

```
def attention(h, s, W1, w2):
    # concatenate h and s and then pass to a two-layer NN.
    layer1 = W1 @ torch.vstack([h, s])
    return w2.reshape(1, -1) @ Tanh(layer1)
```

7.3 CAML: Disease Prediction with Clinical Notes

Now, let us look at a concrete example of applying attention mechanism to improve accuracy in disease prediction using chest X-ray reports.

Clinical notes, such as chest X-ray reports, are documents created by clinicians for each patient encounter. These notes are usually accompanied by medical codes that describe the diagnosis and treatment. However, annotating these codes can be time-consuming and prone to errors. Moreover, the connection between these codes and the text is often unmarked, making it difficult to understand the reasons and details behind specific diagnoses and treatments. A study by Mullenbach et al. [22] proposes an attention-based model (named CAML) that can accurately classify diagnosis codes from clinical notes. We will learn the details of CAML in this section.

7.3.1 Indiana University Chest X-Ray dataset

For this question, we will be using the Indiana University Chest X-Ray dataset. The goal is to predict diseases using chest X-ray reports. The processed datasets could be found in this public folder[1]. The processed dataset contains

- "train_df.csv", "test_df.csv": these two files contain the data used for training and testing. They contain three columns: "Report ID" refers to a unique chest x-ray

[1] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap5-Attention/notebook/IU-chest-Xray-dataset

report, "Text" refers to the clinical report text, and "Label" refers to the disease categories.

- "vocab.csv": this file contains the vocabulary used in the clinical text.

For example, the first chest X-ray report in "train_df.csv" has:

```
Report ID: 1
Text: the cardiac silhouette and mediastinum size are within normal limits.
        there is no pulmonary edema . there is no focal consolidation .
        there are no xxxx of a pleural effusion . there is no evidence of
        pneumothorax . normal chest xxxxx .
Label: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

where label is a multi-hot vector representing the following diseases:

```
normal
cardiomegaly
scoliosis / degenerative
fractures bone
pleural effusion
thickening
pneumothorax
hernia hiatal
calcinosis
emphysema / pulmonary emphysema
pneumonia / infiltrate / consolidation
pulmonary edema
pulmonary atelectasis
cicatrix
opacity
nodule / mass
airspace disease
hypoinflation / hyperdistention
catheters indwelling / surgical instruments / tube inserted / medical device
other
```

So this report 1 is labeled as "normal". The disease classification problem is treated as a multi-label classification problem.

7.3.2 CAML Model Architecture

CAML [22] is a convolutional neural network (CNN)-based model that combines the convolutional architecture with an attention mechanism to model clinical text. It employs a per-label attention mechanism, which allows the model to learn distinct document representations for each label. We break down the model architecture of CAML into four parts below. Figure 7.4 shows an illustration of the model architecture.

- *Input Embedding:* To encode the embedding of X-ray reports, we design a learnable embedding table for all the unique words in the training set. To embed one report, we just load the embedding from the table which is optimized together with other parameters of the model. The word embedding of the clinical document is horizontally concatenated into the input matrix, denoted as $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m]$, where m is the length of the document.

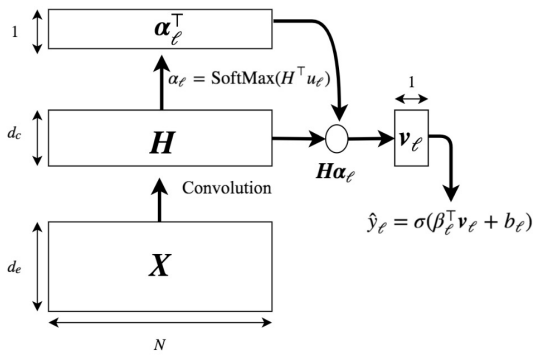

Figure 7.4 CAML model architecture.

```
# raw text of the clinical report
text = [word1, word2, ..., word_m]

# define the embedding table in the initial function
self.embed = nn.Embedding(
    NUM_WORDS,
    embed_size=128,
    padding_idx=0
)
self.embed_drop = nn.Dropout(p=0.8)

def forward_embed(self, text):
    """
    Feed text through the embedding (self.embed) and dropout layer (self.
                                        embed_drop).
    INPUT:
        text: (batch size, seq_len)
    OURPUT:
        text: (batch size, seq_len, embed_size)
    """
    text = self.embed(text)
    text = self.embed_drop(text)
    return text

# embed the text into text embedding
text = self.forward_embed(text)
# (batch size, seq_len, embed_size) -> (batch size, embed_size, seq_len);
text = text.transpose(1, 2)
```

- *Convolution:* We use a convolutional layer to integrate information from adjacent word embeddings. The parameters are $\mathbf{W}_c \in \mathbb{R}^{k \times d_e \times d_c}$, where k is the filter width, d_e the size of the input embedding, and d_c the size of the filter output. At each step n, the convolutional kernel computes

$$\mathbf{h}_n = g(\mathbf{W}_c * \mathbf{X}_{n:n+k-1} + \mathbf{b}_c), \tag{7.12}$$

where $*$ denotes the convolution operator, g is an element-wise nonlinear transformation, and $\mathbf{b}_c \in \mathbb{R}^{d_c}$ is the bias term.

```
# define the cnn module in the initial function
self.conv = nn.Conv1d(
    embed_size=128,
    num_filter_maps=16,
    kernel_size=10,
    padding=5,
)

def forward_conv(self, text):
    """
    Feed text through the convolution layer (self.conv) and tanh
                                            activation function (torch.
                                            tanh)
    INTPUT:
        text: (batch size, embed_size, seq_len)
    OUTPUT:
        text: (batch size, num_filter_maps, seq_len)
    """
    return torch.tanh(self.conv(text))

text = self.forward_conv(text)
# (batch size, num_filter_maps, seq_len) -> (batch size, seq_len,
                                        num_filter_maps);
text = text.transpose(1,2)
```

- *Attention Step:* After convolution, the document embedding is represented as a matrix $\mathbf{H} \in \mathbb{R}^{d_c \times N}$. For each label ℓ, the CAML model computes the matrix-vector product, $\mathbf{H}^\top \mathbf{u}_\ell$, where $\mathbf{u}_\ell \in \mathbb{R}^{d_c}$ is a vector-formed parameterization for label ℓ. The result is passed through a softmax operator, generating a distribution over all words in the document,

$$\mathbf{a}_\ell = \text{Softmax}(\mathbf{H}^\top \mathbf{u}_\ell), \tag{7.13}$$

The attention vector $\mathbf{a}_l$ is then used to re-weight the words in the document, so as to generate a document representation for each label,

$$\mathbf{v}_\ell = \sum_{n=1}^{m} a_{\ell,n} \mathbf{h}_n. \tag{7.14}$$

Here, $a_{\ell,n}$ is the n-th element in vector α_ℓ, and $\mathbf{h}_n$ is the n-th vector in matrix $\mathbf{H}$. Essentially, the words in the document serve as the keys, and each disease diagnosis label becomes a query in this application.

```
# define the attention function in init function
self.U = nn.Linear(num_filter_maps, 20)

def forward_calc_atten(self, text):
    """
    Calculate the attention weights. Be sure to read the documentation for
                                            F.softmax()
    INPUT:
```

```
        text: (batch size, seq_len, num_filter_maps)
    OUTPUT:
        alpha: (batch size, num_class, seq_len), the attention weights
    STEP: 1. multiply 'self.U.weight' with 'text' using torch.matmul();
          2. apply softmax using 'F.softmax()'.
    """
    # (batch size, seq_len, num_filter_maps) -> (batch size,
                                            num_filter_maps, seq_len)
    text = text.transpose(1,2)
    return F.softmax(self.U.weight.matmul(text), dim=2)

def forward_aply_atten(self, alpha, text):
    """
    apply the attention in eq (3) in the paper.
    INPUT:
        text: (batch size, seq_len, num_filter_maps)
        alpha: (batch size, num_class, seq_len), the attention weights
    OUTPUT:
        v: (batch size, num_class, num_filter_maps), vector
                                                representations for each
                                                label
    STEP: multiply 'alpha' with 'text' using torch.matmul().
    """
    return alpha.matmul(text)

alpha = self.forward_calc_atten(text)
v = self.forward_aply_atten(alpha, text)
```

- *Classification:* Finally, given the document representation $\vec{v}_\ell$, the CAML model computes a probability for label ℓ using another linear layer and a sigmoid transformation:

$$\hat{y}_\ell = \sigma(\beta_\ell^\top \mathbf{v}_\ell + b_\ell),$$

where $\beta_\ell \in \mathbb{R}^{d_c}$ is a vector of prediction weights, b_ℓ is a scalar offset, and $\sigma(\cdot)$ is the sigmoid activation function. The total cross-entropy loss across all labels is used $L = -\sum_\ell y_\ell \log(\hat{y}_\ell) + (1 - y_\ell) \log(1 - \hat{y}_\ell)$.

```
# define the final prediction layer in the init function
self.final = nn.Linear(num_filter_maps, NUM_CLASSES)

def forward_linear(self, v):
    """
    apply the final linear classification
    INPUT:
        v: (batch size, num_class, num_filter_maps), vector representations for
                                                each label
    OUTPUT:
        y_hat: (batch size, num_class), label probability
    STEP: 1. multiply 'self.final.weight' v 'text' element-wise using torch.mul
                                            ();
          2. sum the result over dim 2 (i.e. num_filter_maps);
          3. add the result with 'self.final.bias';
          4. apply sigmoid with torch.sigmoid().
    """
```

```
    y = self.final.weight.mul(v).sum(dim=2).add(self.final.bias)
    y_hat = torch.sigmoid(y)
    return y_hat

y_hat = self.forward_linear(v)
```

Results

We train the model using standard PyTorch training and evaluation scripts for 20 epochs. For the standard template, please refer to Section 4.1.3. In the evaluation, precision, recall, and F1-score are used to evaluate the multi-class classification task. We can see that the training loss goes down quickly, while three evaluation metrics improve gradually with more training epochs.

A complete running pipeline including data loading, model construction, training and evaluation could be found in the public repository[2].

```
Epoch: 1   Training Loss: 0.475205
Epoch: 1   Validation p: 0.00, r:0.00, f: 0.00
Epoch: 2   Training Loss: 0.284034
Epoch: 2   Validation p: 0.00, r:0.00, f: 0.00
Epoch: 3   Training Loss: 0.238292
Epoch: 3   Validation p: 1.00, r:0.00, f: 0.00
Epoch: 4   Training Loss: 0.217297
Epoch: 4   Validation p: 0.88, r:0.14, f: 0.24
Epoch: 5   Training Loss: 0.202708
Epoch: 5   Validation p: 0.83, r:0.22, f: 0.35
...
Epoch: 19    Training Loss: 0.112092
Epoch: 19    Validation p: 0.92, r:0.60, f: 0.73
Epoch: 20    Training Loss: 0.109251
Epoch: 20    Validation p: 0.93, r:0.62, f: 0.74
```

7.4 Heart Failure Prediction with RETAIN Model

Longitudinal EHR data are represented as a sequence of patient visits over a period of time. Clinical predictive modeling aims at predicting the occurrence of future events of interest (like heart failure) based on past patient visits. These kind of tasks require machine learning models to be both accurate and interpretable. Attention mechanism, in particular, not only augments the capacity for sequence modeling, but also can be used to understand what part of historical information weighs more in making certain predictions.

In this section, we show how Choi et al. [8] leveraged a two-level attention model to predict heart failure onset using longitudinal EHR data. This section will utilize the comprehensive functionality of the PyHealth package, orchestrating the entire modeling pipeline encompassing data processing, model training, and evaluation. This section unfolds in two segments: firstly, we delve into the intricate architecture of

[2] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap5-Attention/notebook

RETAIN, elucidating its design principles. Subsequently, we offer insights into the implementation pipeline within PyHealth, specifying its practical application in this heart failure prediction context.

7.4.1 Architecture of RETAIN

The main idea of RETAIN is that it mimics physician practice by (1) modeling the patient visits in reverse time order so that recent clinical visits are likely to receive higher attention, (2) applying a two-level attention mechanism to focus on visit and variable level information simultaneously.

The main architecture is shown in Figure 7.5. We explain it briefly in the context of heart failure prediction based on the following steps:

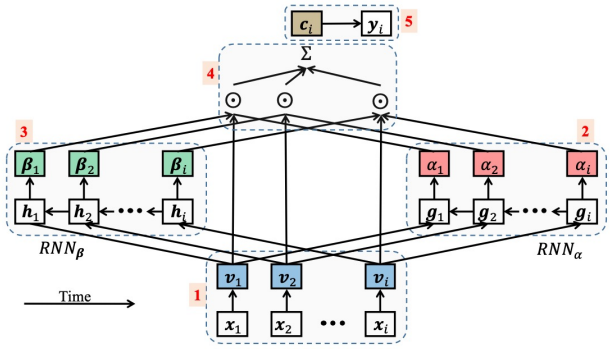

Figure 7.5 Illustration of RETAIN architecture

- Goal: Given a sequence of visits $\mathbf{x}_1, \ldots, \mathbf{x}_t$ from a patient, the goal is to predict whether the patient will develop heart failure in future visits. Here, $\mathbf{x}_i$ is a list of diagnosis codes for each visit i.
- Embed: RETAIN first uses a linear projection layer to embed the discrete diagnosis codes,

$$\mathbf{v_i} = \mathbf{W_{emb}}\mathbf{x_i}, \tag{7.15}$$

  where $\mathbf{W}_{emb} \in \mathbb{R}^{C \times d}$ is an embedding look-up table, C is the total number of unique medical codes and d is the hidden dimension.
- Visit-level Attention: Second, one reverse time RNN (named the Alpha attention) runs from visit t to visit 1 to extract the reverse time sequential visit embeddings $\mathbf{g}_i$.

$$\mathbf{g}_t, \mathbf{g}_{t-1}, \ldots, \mathbf{g}_1 = \text{RNN}_\alpha(\mathbf{v}_t, \mathbf{v}_{t-1}, \ldots, \mathbf{v}_1). \tag{7.16}$$

  Then, the sequential visit embedding is transformed by a linear layer into a scalar indicating the influence of each visit j.

$$e_i = \mathbf{w}_\alpha^\top \mathbf{g}_i + b_\alpha, \quad \forall i = [1, 2, \ldots, t], \tag{7.17}$$

To ensure the influence of all visits forms a probability distribution, the model uses the Softmax function to normalize the influence score.

$$\alpha_1, \alpha_2, \ldots, \alpha_t = \text{Softmax}(e_1, e_2, \ldots, e_t), \tag{7.18}$$

where $\alpha_i \in \mathbb{R}$ denotes the importance of visit $\mathbf{x}_i$.

- Variable-level Attention: Next, another reverse time RNN (named the Beta attention) also runs from visit t to visit 1 to obtain the actual visit embeddings $\mathbf{h}_i$ for each visit.

$$\mathbf{h}_t, \mathbf{h}_{t-1}, \ldots, \mathbf{h}_1 = \text{RNN}_\beta(\mathbf{v}_t, \mathbf{v}_{t-1}, \ldots, \mathbf{v}_1). \tag{7.19}$$

Then, the model uses the non-linearity tanh function to further transform the visit embedding $\mathbf{h}_i$ to $\beta_\mathbf{i}$ as,

$$\beta_\mathbf{i} = \tanh(\mathbf{W}_\beta \mathbf{h}_\mathbf{i} + \mathbf{b}_\beta), \quad \forall \mathbf{i} = [1, 2, \ldots, \mathbf{t}], \tag{7.20}$$

where $\beta_\mathbf{i} \in \mathbb{R}^\mathbf{d}$ denotes the variable-level attention weight for visit $\mathbf{x}_i$. Note the dimension difference between variable-level attention weight $\beta_\mathbf{i} \in \mathbb{R}^\mathbf{d}$ and visit-level attention $\alpha_i \in \mathbb{R}$.

- The final longitudinal embedding $\mathbf{c}_i$ is obtained by using the weighted sum of all visits and the predicted heart failure probability $\hat{y}_i$ is generated with another linear layer.

$$\mathbf{c}_i = \sum_{j=1}^{i} \alpha_j \beta_\mathbf{j} \odot \mathbf{v}_\mathbf{j}, \tag{7.21}$$

$$\tilde{\mathbf{y}} = \text{Sigmoid}(\mathbf{w}_{out}^\top \mathbf{c}_i + b_{out}), \tag{7.22}$$

where $\tilde{\mathbf{y}}$ is the predicted probability of the occurrence of heart failure, $\mathbf{w}_{out}\mathbb{R}^d$ and $b_{out} \in \mathbb{R}$ are two learnable parameters. The predicted heart failure probability is then compared against the true label in the binary cross entropy loss to optimize the model parameters.

7.4.2 Heart Failure Dataset

The dataset employed in this section mirrors the one explored in Section 5.3 and can be obtained from the provided folder[3]. The dataset comprises several essential files, including *pids.csv, vids.csv, hfs.csv, seqs.csv, types.csv, and rtypes.csv*, which we load using the pickle package. This dataset encapsulates information from 1,000 patients, among which 548 individuals have been identified with heart failure.

```
# 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'))
```

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

```
rtypes = pickle.load(open(os.path.join(DATA_PATH,'train/rtypes.pkl'), 'rb'))
```

7.4.3 Step1 & 2: dataset wrapping and split

Utilizing the *pyhealth.dataset.SampleEHRDataset* API, we streamline the transformation process of the aforementioned dataset into a PyHealth-supported structure. Each sample is stored in a JSON format adhering to PyHealth's specifications. The following code snippet showcases the cleanup process for essential attributes: patient ID (pid), visit ID (vid), heart failure label (hf), and diagnosis history (seq). The focal feature herein is the diagnosis sequence, represented as a list of visits, wherein each visit encompasses a series of diagnosis codes.

```
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],
        }
    )
```

Subsequently, employing the *pyhealth.dataset.SampleEHRDataset* API, we proceed to generate a dataset object within the PyHealth package. Below, we showcase the structure of the initial sample from 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']
    ]
}
"""
```

Similar to Section 5.3, we split this dataset into train and validation by 80% : 20%. We will deal with the test set 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)
```

7.4.4 Step 3: Initialize the ML model

This section initializes the RETAIN model using the *pyhealth.model.RETAIN* API, employing both the training loader and the validation loader. The API configuration involves setting "diagnoses" as the key feature within the training dataset. Notably, both the embedding dimension and the hidden dimension are configured to 128, with the ability to customize additional attributes as per specific requirements. The initialization process mirrors the setup steps adopted for other models in a similar manner.

```
from pyhealth.models import RETAIN

model = RETAIN(
    dataset=train_dataset,
    feature_keys=["diagnoses"],
    label_key="label",
    mode="binary",
    embedding_dim=128,
)
```

7.4.5 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. The configuration of Trainer could refer to this page[4].

```
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",
)
```

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

7.4.6 Step 5: Evaluation on test set

Next, we similarly process the test data into Json format and fit into pyhealth dataset. The *trainer.evaluate*() function automatically uses the best trained RETAIN model for evaluation.

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

We could obtain the output values as below.

```
"""
OUTPUT:
{'pr_auc': 0.7582368833149129, 'roc_auc': 0.7667034026725444,
    'f1': 0.7500000000000001, 'loss': 0.595910519361496}
"""
```

The complete pyhealth implementation pipeline is provided in this folder [5].

Questions

- Explain the concept of a sequence-to-sequence task in neural networks. How does it differ from other problem setting? Could you give two real-world healthcare examples for sequence-to-sequence prediction?
- Explain the two-RNN approach for handling sequence-to-sequence tasks. What does

[5] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap5-Attention/notebook

the encoder module do? What does the decoder module do? Could you explain the training and evaluation phases?

- What are two major problems in two-RNN approach? Why could attention mechanism solve this issue? Could you explain how the attention mechanism is applied in the decoder for solving the issues?
- Please explain the module architecture in the seq-to-seq attention paper [31], step by step.
- What are three common ways to compute attention scores? Why will researchers apply softmax to normalize the attention score in the end?
- Could you come up with new functions to calculate the attention score? In which scenarios will the new function work better than three attention functions mentioned in the section.
- Could you explain the RETAIN model? What are the novelties in RETAIN that are different from previous RNN models.
- Please follow the instruction of after-class programming questions in Chapter 2 RNN and use RETAIN to finish the length-of-stay prediction tasks. Could you make the RETAIN work better than LSTM?