8 Transformers

The attention mechanism is an immensely versatile concept in neural network methods. It has been widely used to improve prediction accuracy and provide model interpretation. This chapter will introduce the **transformer model**, which is one of the most powerful neural network models based on the attention mechanism. The transformer model serves as the foundation for many advanced neural network methods, including large language models. In healthcare, transformer models have been widely used for various applications such as rare disease detection [33], clinical outcome prediction [34], and synthetic patient data generation [35].

In Chapter 7, we explored the origin of the attention mechanism in the sequence-to-sequence (Seq2Seq) prediction setting. Traditional RNN-based models had notable challenges in training for Seq2Seq problems, mainly due to computational intensity and optimization difficulties. Since each step in the model relied on the previous step's output, it hindered parallel training and exacerbated the vanishing gradient problem. Fortunately, the attention mechanism alone without the sequence dependency from RNN can overcome these computational issues. This chapter will introduce the Transformer, which was proposed in the famous paper "Attention Is All You Need" [36] and relies primarily on attention modules to establish global dependencies between input and output. The Transformer's architecture has been integral to the development of models like BERT [37] and has become the cornerstone for large language models such as ChatGPT. Furthermore, its versatility extends beyond text, showing promise as a unified framework for processing multi-modal data.

Towards the end of this chapter, we will provide two practical examples of using transformers for clinical prediction tasks. First, we will demonstrate the use of ClinicalBERT [38] for the classification of ICD codes. Second, we will showcase the use of the transformer encoder model for recommending drugs based on a patient's diagnosis, medical procedures, and past medications [25].

8.1 Introduction to the Transformer Model

8.1.1 Transformer Model

Let's start by discussing the vanilla Transformer, which was introduced in the paper "Attention Is All You Need" [36]. The Transformer model follows an Encoder-Decoder

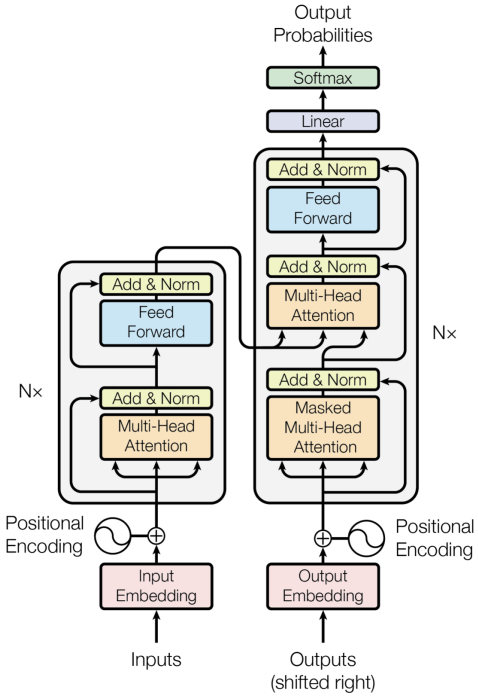

Figure 8.1 The Vanilla Transformer model is composed of two main modules - the encoder module on the left-hand side and the decoder module on the right-hand side. This model is built with multiple layers of multi-head attention and feedforward layers, represented as N× in the figure. N× indicates that N-layers of the same architecture are stacked on top of each other. An example of using the transformer model in healthcare is to input progress notes documented during a visit and output a discharge summary given to the patient at the time of their discharge.

architecture, similar to the Seq2Seq model as shown in Figure 8.1. However, unlike the Seq2Seq model which uses two Recurrent Neural Network (RNN) models, the Transformer model simplifies the architecture by removing the RNN models and only using attention mechanisms. This simplification allows for parallelization during training which results in greater efficiency and the ability to train larger and deeper Transformer models.

In the **Encoder module**, each Transformer layer consists of two sub-layers. The first sub-layer is a multi-head self-attention mechanism, and the second sub-layer is a position-wise fully connected feed-forward network. The two sub-layers are connected using a residual connection and layer normalization, which was originally introduced in the ResNet model (refer to Chapter 6.2.5 for more information). We will provide a detailed explanation of these sub-layers in this chapter.

In contrast, in the **Decoder module**, the Transformer layer involves a third sub-layer, which performs multi-head attention over the output of the encoder stack. This cross-attention sub-layer (two arrows from the inputs and one arrow from the output

as shown in Figure 8.1) is inserted in between the self-attention sub-layer and the feed-forward sub-layer. Furthermore, the self-attention sub-layer is (i.e., masked multi-head attention in Figure 8.1) modified with an auto-regressive mask for the output to prevent each position from attending to subsequent positions (i.e., position i can only attend to positions less than or equal to i). Figure 8.1 depicts the overall vanilla Transformer architecture, which is borrowed from the original paper [36].

The Self-Attention Mechanism

The Transformer model introduced the self-attention model, which separates attention from RNNs and has had a significant impact on neural networks and artificial intelligence in general.

Self-Attention. Revisiting sequence representation learning, our objective is to transform an input sequence $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_l)$ (e.g., progress notes from a patient visit) into a contextually richer embedding $\mathbf{X}' = (\mathbf{x}'_1, \ldots, \mathbf{x}'_l)$, where $\mathbf{X}, \mathbf{X}' \in \mathbb{R}^{l \times d}$. One could use Recurrent Neural Networks (RNNs) to achieve this through a recurrent operation, where each step's output is dependent on the previous step. In contrast, the Transformer model eliminates this temporal dependency by introducing a self-attention mechanism, which involves the following steps:

1. **Linear Transformation**: To transform the input $\mathbf{X}$, three linear embedding layers are used to generate $\mathbf{K}$, $\mathbf{Q}$, and $\mathbf{V}$ matrices. These matrices are obtained by multiplying $\mathbf{X}$ with learnable projection matrices $\mathbf{W}^K$, $\mathbf{W}^Q$, and $\mathbf{W}^V$, respectively. Each projection matrix has dimensions of $\mathbb{R}^{d \times d}$.

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}^K$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}^V$$

   The advantage of this approach is that it breaks the dependency between the projection matrices $\mathbf{W}^K$, $\mathbf{W}^Q$, and $\mathbf{W}^V$, allowing them to be updated in parallel for more efficient updates.

2. **Similarity Search**: We can use three embedding matrices $\mathbf{K}$, $\mathbf{Q}$, and $\mathbf{V}$ to compute the similarity between a query $\mathbf{Q}$ and key $\mathbf{K}$ and retrieve the weighted sum of value $\mathbf{V}$. This similarity search step calculates the similarity between the query

$$\mathbf{Q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^d$$

   which is the i-th row of $\mathbf{Q}$ and all key elements $\mathbf{K}$ for each token $\mathbf{x}_i$ in the sequence. To compute the similarity scores, a scaled dot product is used

$$\mathbf{e}_i = \frac{\mathbf{K}\mathbf{Q}_i}{\sqrt{d}}$$

   where d is a scaling factor that prevents the softmax function from having extremely small gradients in high dimensions. The similarity logit vector $\mathbf{e}_i \in \mathbb{R}^l$ is obtained from this step. Similar to the standard attention mechanism, the attention weight is obtained by normalizing the similarity logit vector, as $\mathbf{a}_i = \text{Softmax}(\mathbf{e}_i)$.

3. **Value Retrieval**: With the attention scores $\mathbf{a}_i$, the model computes the weighted sum of $\mathbf{V}$ to produce the output embedding $\mathbf{x}'_i = \mathbf{V}^\top \mathbf{a}_i$.

The updating of all $\{\mathbf{x}'_i\}$ could be calculated together to be in matrix form. The scaled dot-product attention used by Transformer is given by,

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}, \tag{8.1}$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{l\times d}$, and $\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right) \in \mathbb{R}^{l\times l}$. Note that the softmax function is applied to each row of the matrix resulting from $\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}$, turning the scores into a probability distribution. We refer to the process of similarity search as retrieval because it involves computing attention scores which can be used to identify the rows in matrix $\mathbf{V}$ that are similar to the query $\mathbf{Q}_i$. When the attention scores are close to 0 or 1, the weighted sum will provide the subset of rows in $\mathbf{V}$ that are most similar to the query. The whole process can be efficiently computed using matrix multiplications in parallel.

The left panel in Figure 8.2 illustrates this process. Note that the "Mask" operation is optional, and is used to avoid attending to padding inputs when doing batch calculation.

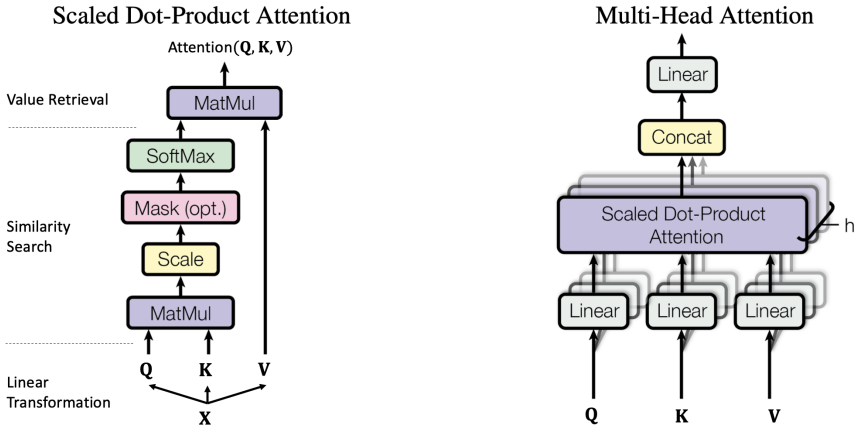

Figure 8.2 Transformer's self-attention mechanism. The left-hand side introduces a single self-attention model, while the right-hand side represents the multi-head attention model with h heads.

```
class Attention(nn.Module):
    def forward(self, query, key, value, mask=None, dropout=None):
        """ Score Calculation """
        # query in [batch size, sequence length l, features d],
        # query.size(-1) is number of features d
        # key in [batch size, sequence length l, features d]
        # after applying transpose(-2, -1), the shape becomes [batch size,
                                                features d, sequence length l]
```

```
        # torch.matmul  does batched matrix multiplication between these two
                                                tensors
        # scores in [batch size, sequence length, sequence length]
        scores = torch.matmul(query, key.transpose(-2, -1)) / \
                                        math.sqrt(query.size(-1))
        """ Masking (Optional) """
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        """ Softmax Normalization """
        p_attn = torch.softmax(scores, dim=-1)
        if mask is not None:
            p_attn = p_attn.masked_fill(mask == 0, 0)
        """ Optional Dropout: """
        if dropout is not None:
            p_attn = dropout(p_attn)

        """ Value Weighting and Aggregation """
        # p_attn in [batch size, sequence length 1, sequence length 1]
        # value in [batch size, sequence length 1, features d]
        return torch.matmul(p_attn, value), p_attn
```

The code snippet above implements the self-attention mechanism. The Attention class, when instantiated as a module within a neural network, takes four parameters: *query*, *key*, *value*, and *optionally mask* and *dropout*. Here is a breakdown of its operation:

- **Scores Calculation**: It calculates the attention scores by performing a scaled dot product of the query (`query`, **Q**) and the key (`key`, **K**) matrices. The scaling factor is the square root of the dimension of the query vectors (i.e., query.size(-1)), which helps in stabilizing the gradients. This is a direct implementation of the Similarity Search step in the self-attention mechanism.
- **Masking (Optional)**: If a mask `mask` is provided, the scores are adjusted such that positions with 0 in the mask have their values set to -1e9 (a very small number close to negative infinity). This ensures that these positions do not contribute to the softmax normalization, effectively removing them from consideration. This is particularly useful for ignoring padding tokens in a batch of sequences of varying lengths.
- **Softmax Normalization**: The `scores` are then normalized using a softmax function across the last dimension into `p_attn`. This transforms the scores into attention weights that sum up to 1, making them into probabilities indicating the importance of each key-value pair.
- **Optional Dropout**: If a dropout module is provided, it is applied to the attention weights. Dropout is a regularization technique that randomly zeroes some of the elements with a certain probability, helping to prevent overfitting by making the model less sensitive to the specific weights of neurons.
- **Value Weighting and Aggregation**: Finally, the attention mechanism computes the output by multiplying the attention weights with the value matrix (`value`, **V**). This step combines the values weighted by their computed importance, producing the output embedding that is a contextually enriched representation of the input.

The code block above produces a pair consisting of two elements. The first element is the attention-weighted sum of the value vectors, which transforms the input and provides a new representation. The second element is the attention weights themselves. These weights can be helpful in interpreting the behavior of the model by identifying which parts of the input are being attended to for a given output.

Encoder Attention v.s. Decoder Attention

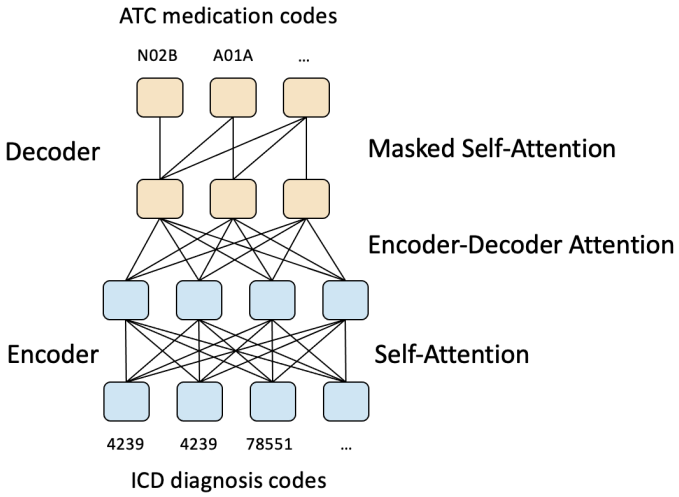

Figure 8.3 Illustration of the attention mechanisms in the encoder and decoder module. We use the diagnosis codes (ICD) to map medication codes (ATC) as an example. In particular, the input is a sequence of ICD codes (4239, 4239, 78551, ...), and the output is a sequence of medication codes (N02B, A01A, ...). The self-attention layers in the encoder and the one from encoder to decoder are all fully connected without masking. The self-attentions within the decoder are masked self-attentions, where each medication is only connected to the ones that before it. We call this autoregressive masking or causal masking.

In the Transformer model, the attention mechanism is applied differently within the encoder and decoder modules. Figure 8.3 illustrates the different attention mechanisms in the encoder and decoder module.

In the encoder, the attention is bidirectional: each token can attend to all tokens in the previous encoder layer. This allows the encoder to fully capture the context information in the input. For example, all the ICD codes in Figure 8.3 are all connected.

While **in the decoder**, there are two types of attention: encoder-decoder attention and masked self-attention. The encoder-decoder attention is similar to self-attention in the encoder module, which allows the decoder to attend to the entire encoder output. *However, the decoder process runs iteratively one token at a time.* When decoding a token (such as A01A in the figure), the query is the output embedding of the previous token (such as N02B in the figure), and the keys and values remain the same, which are from the output of the encoder module. This mechanism enables the decoder to

focus on relevant parts of the input sequence during the generation process, effectively using the encoder's context to inform its predictions.

The masked self-attention operates similarly to self-attention but with a crucial difference: it is masked to ensure that the predictions for a given position can only depend on the known outputs at positions before it in the sequence. This prevents the decoder from "looking ahead" at the output it has not generated yet, enforcing the auto-regressive property necessary for sequential generation.

In Figure 8.3, we use the example of mapping diagnosis codes (ICD) to medication codes (ATC). The input is a sequence of ICD codes (4239, 4239, 78551, ...) while the output is a sequence of medication codes (N02B, A01A, ...). The self-attention layers in the encoder and the one from the encoder to the decoder are all fully connected without masking. However, the self-attentions within the decoder are masked self-attentions, where each medication is only connected to the ones that come before it. We refer to this as autoregressive masking or causal masking. For example, medication N02B is the first decoder output, which only attends to itself, while medication A01A is the second output and attends to both N02B and A01A from the previous layer.

Multi-head Attention. Beyond the naive self-attention, the Transformer model also introduces a more advanced version: *multi-head attention*. Instead of performing a single attention function with d-dimensional keys, values, and queries, it is beneficial to linearly project with h different sets of query, key, and value matrices. On each query, key, and value matrices, we then perform the attention function in parallel, yielding h different output values independently as shown on the right-hand side of Figure 8.2. The resulting embeddings are concatenated and once again projected via an output layer $\mathbf{W}^O$, resulting in the final values. The multi-head attention can be summarized by the following equations,

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)\mathbf{W}^O \tag{8.2}$$

$$\text{where head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V), \tag{8.3}$$

where $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$ are three learnable parameters for head i.

```python
class MultiHeadedAttention(nn.Module):
    """ Initialization """
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0

        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h

        # We combine the projection matrices for different heads together for
                                                parallel computing
        self.linear_layers = nn.ModuleList(
            [nn.Linear(d_model, d_model, bias=False) for _ in range(3)]
        )
        self.output_linear = nn.Linear(d_model, d_model, bias=False)
        self.attention = Attention()
```

```
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        # query, key, value in [batch size, sequence length l, features d]
        # mask in [batch size, sequence length l, sequence length l]
        batch_size = query.size(0)

        """ Linear Projections """
        # 1) Do all the linear projections in batch from d_model => h x d_k
        # Then, view it as (batch size, seq len, num heads h, per head
                                                dimension d_k)
        # Lastly, transpose it into (batch size, num heads h, seq len, per head
                                                 dimension d_k)
        query, key, value = [
            l(x).view(batch_size, -1, self.h, self.d_k).transpose(1, 2)
            for l, x in zip(self.linear_layers, (query, key, value))
        ]

        """ Parallel Attention Processing """
        # 2) Apply attention on all the projected vectors in batch.
        if mask is not None:
            mask = mask.unsqueeze(1)
        x, attn = self.attention(query, key, value, \
            mask=mask, dropout=self.dropout)

        """ Concatenation and Final Projection """
        # 3) "Concat" using a view and apply a final linear.
        # First, transpose it into (batch size, seq len, num heads h, per head
                                                dimension d_k)
        # Then, call contiguous() to ensure that the returned tensor occupies a
                                                 contiguous block of memory
        # Lastly, view it as (batch size, seq len, hidden dimension d)
        x = x.transpose(1, 2).contiguous().view(batch_size, \
            -1, self.h * self.d_k)

        return self.output_linear(x)
```

The code snippet above implements the multi-head attention mechanism. Here is a breakdown of its operations.

- **Initialization**: The MultiHeadedAttention class initializes with the number of heads `h`, the model dimension `d_model`, and an optional dropout rate. It asserts that `d_model` is divisible by `h`, ensuring that each head can have an equal portion of the model's dimensionality. The division of `d_model` by `h` determines the dimensionality `d_k` of keys, values, and queries within each head.
- **Linear Projections**: For each head, the queries, keys, and values are linearly projected from `d_model` dimensions into `d_k` dimensions. This is done in parallel for efficiency, using a set of linear layers stored in `self.linear_layers`. Each input (query, key, value) is transformed by these layers and then reshaped to prepare for parallel attention processing across the heads.
- **Parallel Attention Processing**: The reshaped queries, keys, and values are then processed by the attention function, `self.attention`, in parallel across all

heads. This is where the attention mechanism computes the attention scores, applies optional masking, and generates weighted sums of values for each head. If a mask is provided, it is adjusted to match the dimensions expected by the attention operation.

- **Concatenation and Final Projection**: After attention is applied separately in each head, the outputs are concatenated back together. This concatenation reverses the earlier reshaping, combining the results from all heads into a single matrix. This matrix is then linearly transformed by `self.output_linear` to produce the final output of the multi-head attention layer.
- **Optional Dropout Application**: Dropout can be applied within the attention function to the attention weights before they are used to weight the values. This helps in regularizing the model and preventing overfitting.

The multi-head attention mechanism allows the Transformer to better capture different types of relationships in the data by processing the inputs in parallel across multiple heads, each able to learn different aspects or different parts of the sequence.

This self-attention approach allows the Transformer to capture contextual information without the constraints of sequential dependencies, which allows parallelization via matrix product and makes large-scale pre-training possible. In addition to self-attention, the transformer model has several modules that further optimize the models including 1) position-wise feed-forward networks, 2) residual connection and normalization, and 3) position encodings.

Position-wise Feed-forward Networks (FFN)

The position-wise FFN is a fully connected feed-forward network, which is applied to each position separately and identically.

$$\text{FFN}(\mathbf{H}) = \text{ReLU}(\mathbf{H}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \tag{8.4}$$

where $\mathbf{H} \in \mathbb{R}^{l \times d}$ is the output of the previous sub-layer, and $\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2$ are learnable parameters. In the original paper, the input and output dimension of the position-wise FFN is 512, while the intermediate dimension is 2048.

```
class PositionwiseFeedForward(nn.Module):
    """ Initialization """
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.l1 = nn.Linear(d_model, d_ff)
        self.l2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = nn.ReLU()

    def forward(self, x, mask=None):
        # x: (batch size, sequence length, hidden dimension)
        x = self.l2(self.dropout(self.activation(self.l1(x))))
        if mask is not None:
            # convert mask from [batch size, sequence length 1, sequence length
                                                    1] to [batch size,
                                                   sequence length 1
            mask = mask.sum(dim=-1) > 0
```

```
            x[~mask] = 0
        return x
```

The code snippet above provides an implementation of the position-wise FFN. Here's a breakdown of its operation:

- **Initialization**: The PositionwiseFeedForward class is initialized with the size of the input/output dimension `d_model`, the size of the hidden layer `d_ff`, and an optional dropout rate. These parameters define the structure of the feed-forward network, where `d_model` matches the dimensionality of the Transformer model's embeddings and `d_ff` is typically much larger, allowing the network to capture more complex relationships.
- **Forward Pass**: First Linear Transformation: The input `x` is first passed through a linear layer `self.l1` that expands its dimensionality from `d_model` to `d_ff`. This expansion allows the network to learn more complex functions.
- **Activation**: A ReLU (Rectified Linear Unit) activation function is applied to the output of the first linear layer. ReLU is used for introducing non-linearity into the model, enabling it to learn more complex patterns in the data.
- **Dropout**: Dropout is applied after the activation function as a form of regularization to prevent overfitting.
- **Second Linear Transformation**: The dropout-modified activations are then passed through a second linear layer `self.l2` that projects them back to the original dimensionality `d_model`. This ensures that the output of the FFN can be added to the input of the layer in a residual connection, a critical component for effective training of deep networks.
- **Optional Masking**: If a mask is provided, it is used to zero-out the output for specific positions within the sequence. This can be useful for ignoring padding tokens or other positions that should not contribute to the model's output. The mask is applied by first summing it across the last dimension to identify positions to be masked, then setting the output to zero for these positions.

The position-wise FFN is a crucial element of the Transformer's architecture, allowing the model to apply complex transformations at each position in the sequence independently.

Residual Connection and Normalization

In order to build a deep model, Transformer employs a residual connection around each of the two sub-layers, followed by layer normalization, as

$$\mathbf{H} = \text{LayerNorm}(\text{Attention}(\mathbf{x}) + \mathbf{x}), \tag{8.5}$$

$$\mathbf{H} = \text{LayerNorm}(\text{FFN}(\mathbf{H}) + \mathbf{H}). \tag{8.6}$$

```
class SublayerConnection(nn.Module):
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = nn.LayerNorm(size)
        self.dropout = nn.Dropout(dropout)
```

```
    def forward(self, x, sublayer):
        # x: (batch size, sequence length, hidden dimension)
        return x + self.dropout(sublayer(self.norm(x)))
```

The code snippet above implements the residual connection and layer normalization mechanism. Here is how the implementation works:

- **Initialization**: The `SublayerConnection` class is initialized with the dimension size of the layer and a dropout rate.
- **Normalization**: The input `x` is first normalized using layer normalization (`self.norm`). Layer normalization is applied across each feature vector independently, rather than across the batch dimension. This helps to stabilize the hidden layer distributions, making the model's training more predictable and consistent.
- **Sublayer Application**: The normalized input is then passed to the sublayer, which could be any function representing a Transformer sublayer, such as a multi-head self-attention mechanism or a position-wise feed-forward network. This design makes the sublayer connection module highly versatile and reusable across different parts of the Transformer model.
- **Dropout**: Dropout is applied to the output of the sublayer to prevent overfitting.
- **Residual Connection**: Finally, the original input `x` is added to the dropout-modified output of the sublayer. This residual connection allows the input to bypass the sublayer, directly adding to its output. Residual connections are a key innovation that enables deep neural networks to be trained more effectively.

The combination of residual connections and layer normalization is a hallmark of the Transformer architecture, contributing significantly to its ability to learn deep representations without degradation in performance.

Position Encodings

The Transformer model does not inherently understand the order of the input sequence because of its reliance on self-attention mechanisms, which are *permutation-invariant.* To overcome this limitation, the model incorporates position encodings into the input embeddings, providing the model with information about the position of each token in the sequence. Specifically, the goal is to represent each position with an embedding vector of dimension d.

The original paper proposes adding learned or fixed position encodings to the input embeddings before processing by the self-attention layers. The position encodings have the same dimension `d_model` as the embeddings, allowing the two to be summed directly. The most common form of position encoding uses sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \tag{8.7}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \tag{8.8}$$

where pos is the position and i is the dimension. That is, each dimension of the position encoding corresponds to a sine wave and a cosine wave with different frequencies. Intuitively, the choice of sine and cosine functions for these embeddings has several benefits: (1) the periodic nature of the sine and cosine functions allows the model to generalize to sequence lengths that it has not seen during training; (2) the use of sine and cosine functions enables the model to learn and encode relative positions easily. Because these functions are continuous and smooth, changes in position result in predictable changes in the encoding.

Overall Transformer Layer

Now we can combine the previous sub-layers and build the Transformer layer.

```
class TransformerBlock(nn.Module):
    def __init__(self, hidden, attn_heads, dropout):
        super(TransformerBlock, self).__init__()
        self.attention = MultiHeadedAttention(h=attn_heads, d_model=hidden)
        self.feed_forward = PositionwiseFeedForward(
            d_model=hidden, d_ff=4 * hidden, dropout=dropout
        )
        self.input_sublayer = SublayerConnection(size=hidden, dropout=dropout)
        self.output_sublayer = SublayerConnection(size=hidden, dropout=dropout)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x, mask=None):
        # x: (batch size, sequence length, hidden dimension)
        x = self.input_sublayer(x, lambda _x: \
                        self.attention(_x, _x, _x, mask=mask))
        x = self.output_sublayer(x, lambda _x: \
                        self.feed_forward(_x, mask=mask))
        return self.dropout(x)
```

The code snippet above defines the class `TransformerBlock`. The constructor of the `TransformerBlock` class takes three parameters: `hidden`, `attn_heads`, and `dropout`. These parameters configure the size of the hidden layers, the number of attention heads in the multi-head attention mechanism, and the dropout rate for regularization, respectively. The forward method implements the data flow through the transformer block:

- It first applies the multi-headed attention mechanism to the input `x` along with any optional mask. This is wrapped in an `input_sublayer` call, which applies a residual connection followed by layer normalization.
- The output of the attention mechanism is then passed through a position-wise feed-forward network. This step is also wrapped in an `output_sublayer` call, ensuring that the output of the feed-forward network is combined with the original input through a residual connection and then normalized.
- Finally, dropout is applied to the output of the second sublayer connection for regularization.

The optional mask parameter in the forward method allows the block to selectively ignore certain parts of the input sequence, which is crucial for tasks like sequence-to-sequence models where certain inputs should not influence the output at specific times (e.g., to prevent future information from affecting the prediction of the current state in sequence generation).

8.1.2 Model Complexity

The Transformer model's computational complexity primarily arises from its multi-head self-attention mechanism and the position-wise feed-forward networks.

- Self-attention complexity: The complexity of the self-attention mechanism for each head is $O(l^2 d)$ due to the pairwise dot products computed for each element in the input sequence of length l in the d-dimensional space.
- Feed-forward network complexity: Each layer in the feed-forward network has a complexity of $O(nd)$. Since this operation is applied independently to each position, the complexity scales linearly with the sequence length l.

8.1.3 Recent Improvement over the Vanilla Transformer Model

Recently, many works have sought to improve upon the original Transformer model by addressing some of its limitations, such as computational efficiency and handling of long sequences. We summarize a few key improvements below:

Efficiency Improvements

Efficiency improvements aim to reduce the computational and memory costs of the Transformer, enabling faster training and inference, as well as the ability to process longer sequences. Examples include:

- **Sparse Attention Mechanisms**: Methods like the Reformer [39] and Longformer [40] introduce sparsity into the attention mechanism, reducing the quadratic complexity with respect to sequence length to something more manageable for long documents.
- **Factorized Embeddings**: Techniques that reduce the dimensionality of embeddings before processing them through the network layers, as seen in models like ALBERT [41], which also shares parameters across layers to further reduce memory consumption.
- **Efficient Attention**: Models like Linformer [42] and Performer [43] propose efficient attention mechanisms that approximate the standard attention computation in a way that significantly decreases the complexity, especially beneficial for long sequences.

Long Sequence Handling

To better manage long sequences, several models have introduced novel architectures and mechanisms:

- **Memory Mechanisms**: Models such as the Transformer-XL [44] incorporate a memory mechanism that allows the model to remember information from distant tokens, significantly improving its performance on tasks requiring long-term context.
- **Hierarchical Structures**: Some approaches, like the Sparse Transformer [45], organize the sequence hierarchically, allowing the model to efficiently capture dependencies at multiple scales.

8.2 From Transformer to BERT

The development of the BERT (Bidirectional Encoder Representations from Transformers) model, based on the Transformer architecture, has marked a significant step forward in the field of natural language processing (NLP). The Transformer model introduced a powerful self-attention mechanism for processing data sequences. BERT, on the other hand, leveraged this mechanism to generate deep contextualized word embeddings, paving the way for a new era of pre-trained models.

Pre-training is not a novel concept in NLP. In fact, researchers have proposed models such as word2vec [46] and GloVe [47] to learn non-contextualized word embeddings. These models attempt to learn a distinct embedding vector for each word, regardless of the context in which it is used. For instance, the word "bank" in the sentence "I deposited money into the bank" and in the sentence "The river bank was flooded" would have the same embedding representation, despite their vastly different meanings in context. These embeddings are then utilized as fixed inputs to downstream NLP models, providing a basic form of semantic understanding. However, the limitation of such models is their inability to capture the polysemantic nature of language, where the meaning of a word can change dramatically based on its context.

The emergence of pre-trained models such as BERT brought about a significant change towards contextualized word embeddings. Unlike previous models such as word2vec and GloVe, BERT and its successors generate embeddings that adapt based on the surrounding words. This enables a more nuanced understanding of language. This shift from non-contextualized to contextualized embeddings has been one of the most impactful advancements in Natural Language Processing (NLP), allowing for models to achieve unprecedented performance on a wide range of tasks.

This section introduces how BERT builds upon the Transformer model, its unique features, and its impact on NLP tasks, especially in the clinical domain.

8.2.1 BERT's Architecture

The architecture of BERT is based on the encoder of the Transformer. However, unlike the original Transformer model, BERT uses only the encoder and not the decoder.

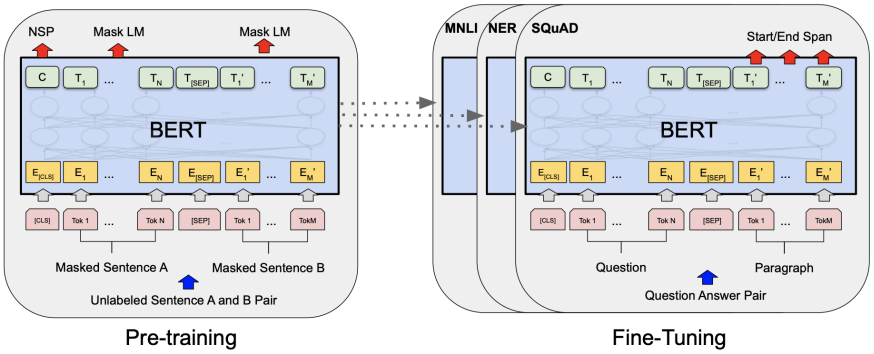

Figure 8.4 Bidirectional Encoder Representations from Transformers (BERT). This figure is from [48].

This is why the model is named "Bidirectional Encoder", as each token attends to all other tokens before and after itself. The key innovation of BERT is its approach to pre-training on a large corpus of text using two new tasks: masked language modeling (MLM) and next sentence prediction (NSP).

8.2.2 Pre-Training

BERT's training process consists of two stages: pre-training and fine-tuning. During pre-training, BERT is trained on a large corpus of text, such as the *BooksCorpus* [1] and English Wikipedia [2], using the MLM and NSP tasks. This stage allows BERT to learn a general language representation.

Masked Language Modeling (MLM)

In MLM, a certain percentage of the input tokens are randomly masked, and the model's objective is to predict these masked tokens (modeled as a multi-class classification problem over the vocabulary) based on their context. This task forces the model to understand the bidirectional context of the masked word, thus learning deep contextualized word embeddings. The original BERT paper masks 15% of all tokens in each sequence for this purpose.

Next Sentence Prediction (NSP)

The NSP task involves presenting the model with pairs of sentences and asking it to predict whether the second sentence is the logical and immediate successor of the first one. This task helps BERT learn relationships between sentences, which is crucial for tasks that involve understanding the relationship between sentences, such as question answering and natural language inference.

[1] https://huggingface.co/datasets/bookcorpus
[2] https://en.wikipedia.org/wiki/Wikipedia:Database_download

8.2.3 Fine-tuning

In the fine-tuning stage, BERT is adapted to specific NLP tasks by continuing the training process on a smaller dataset specific to the task, such as sentiment analysis, question answering, or named entity recognition. The fine-tuning adjusts the weights of the pre-trained model to make it perform well on the target task.

8.2.4 Impact on NLP Tasks

BERT has markedly advanced the state-of-the-art across a broad spectrum of NLP tasks. Its capacity to produce deep, contextualized word embeddings has resulted in significant performance enhancements, particularly for tasks that require an understanding of context and the relationships between sentences. Before BERT, the focus was predominantly on training models specific to each task. However, BERT showcased the effectiveness of large-scale pre-training followed by fine-tuning for specific tasks. This approach inaugurated a new era in the field of NLP, paving the way for the development of pre-trained models, which later evolved into what are now known as **foundation models** (such as GPT).

8.2.5 Variants and Extensions

Following BERT's success, several variants and extensions have been developed to address its limitations and adapt it to different languages and domains. Some notable examples include:

- RoBERTa [49] is an optimized version of BERT that has undergone changes in its pre-training procedure. The RoBERTa model dynamically changes the masked positions during training, which helps in learning more robustly from varied contexts. In contrast, the original BERT model has the same masking for the training data. RoBERTa also removes the next sentence prediction objective, which was originally part of BERT's training but was found to have a negligible performance impact. Additionally, RoBERTa utilizes a much larger training dataset compared to BERT.
- DistilBERT [50]: A smaller, faster, and lighter version of BERT designed to maintain similar performance with significantly reduced size and computational cost. DistilBERT is based on a technique called knowledge distillation, where a smaller model (the "student") is trained to reproduce the behavior of a larger pre-trained model (the "teacher"). In this case, DistilBERT learns from BERT, effectively capturing its knowledge. Specifically, during training, DistilBERT not only learns from the hard labels of the original training data but also from the soft labels (the output distributions) produced by BERT, enabling it to distill BERT's knowledge.
- ALBERT (A Lite BERT) [51]: A light version of BERT with several new tricks: 1) Factorized embedding parameterization: ALBERT decouples the size of the hidden layers from the size of the vocabulary embeddings, significantly reducing the number of parameters. 2) Cross-layer parameter sharing: ALBERT shares

model parameters from self-attention layers and feed-forward networks across all layers of the model, not only reducing the model's size but also helping with training stability. 3) Loss modification: ALBERT modifies the learning loss to focus more on inter-sentence coherence, introducing a sentence-order prediction (SOP) task that improves the model's understanding of sentence relationships.

8.2.6 Customized BERT Models for clinical NLP tasks

The application of BERT in clinical NLP tasks represents a significant leap forward in processing and understanding medical texts, ranging from electronic health records (EHRs) to medical literature. Clinical NLP tasks pose unique challenges, including dealing with specialized terminology, understanding complex sentence structures that convey nuanced medical information, and ensuring patient privacy. BERT's deep contextualized embeddings have proven particularly beneficial in addressing these challenges.

Given the specialized nature of clinical language, BERT has been developed to several domain-specific versions to better capture medical semantics:

- BioBERT [52]: Adapted from BERT for biomedical text mining, BioBERT is pre-trained on large-scale biomedical corpora, including *PubMed abstracts*, *PubMed Central full-text articles*, and books from the *NCBI bookshelf*. It has shown remarkable performance improvements in tasks such as disease name recognition, chemical entity extraction, and biomedical relation extraction.
- ClinicalBERT [53]: Specifically fine-tuned for clinical notes, ClinicalBERT has been trained on notes from the *MIMIC-III* database, which contains de-identified EHRs. It has demonstrated improved performance in patient outcome prediction and medical coding tasks compared to non-specialized BERT models.

8.3 Transformer Beyond Text Data

The Transformer architecture has proven to be highly adaptable and effective, inspiring its use beyond natural language processing. This versatility has extended its utility to various data types and domains. In the healthcare domain, the Transformer model can be adapted to other modalities, which we will briefly introduce in the following section.

8.3.1 Transformer for Sequence Data

Electronic Health Records (EHRs) are rich repositories of patient health information, including diagnoses, treatments, and outcomes over time. Transformers can be employed to analyze EHRs for tasks such as patient outcome prediction, disease progression modeling, and personalized treatment recommendations. By considering the sequence of medical events recorded in EHRs as comparable to natural language sentences, Transformers can uncover complex patterns in patient journeys and identify

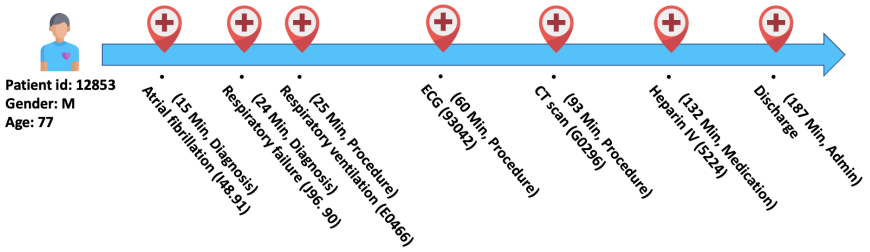

Figure 8.5 An example of the patient EHR record, which consists of a sequence of events.

how different health conditions and treatments interact. This application aids in understanding individual patient experiences, identifying effective treatment strategies, and potential risks.

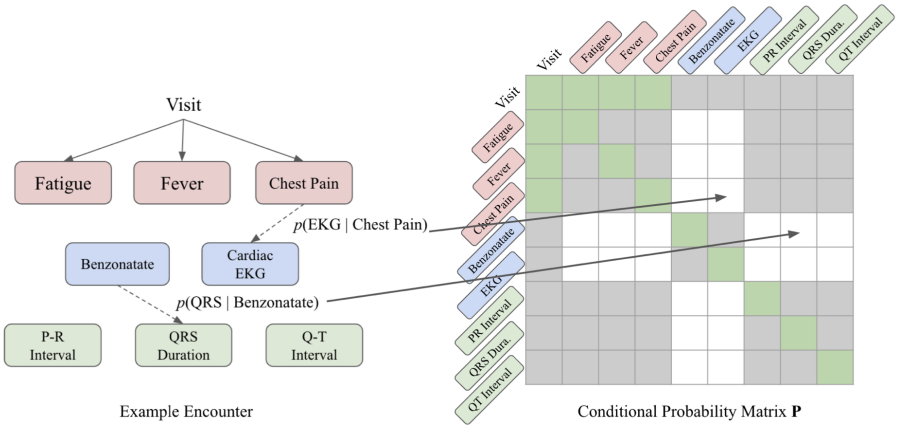

Figure 8.6 Graph Convolutional Transformer (GCT) [6].

The Graph Convolutional Transformer (GCT) [6] is an example of a model that utilizes the Transformer architecture to effectively model the relationship between various medical codes present in electronic health records (EHR) data. The attention matrix is modified in GCT to allow only a few known connections, indicating that the Transformer architecture is a suitable model for learning the hidden structure in sequential EHR data. This work provides evidence of the effectiveness of the Transformer model in analyzing EHR data.

8.3.2 Transformer for Imaging Data

Medical imaging, such as X-rays, MRI, and CT scans, is a cornerstone of modern diagnosis and treatment planning. Adapting Transformer models to process and analyze medical images can enhance their interpretability and diagnostic accuracy.

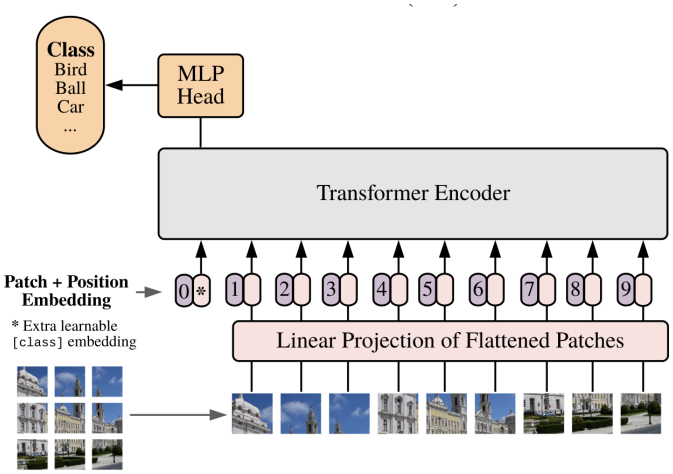

Figure 8.7 Vision Transformer (ViT) [54].

Vision Transformers (ViTs) [54] for instance, divide images into patches and process these patches to identify features and patterns critical for diagnosis.

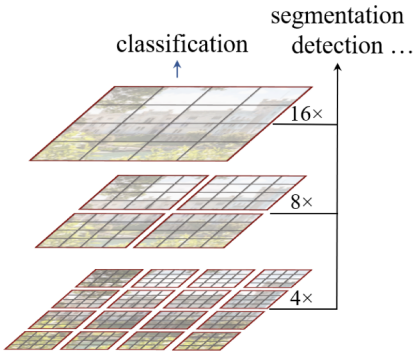

Figure 8.8 Swin Transformer [55].

Another example is the Swin Transformer [55], which introduces a hierarchical structure that allows for modeling at various scales and significantly reduces computational complexity. It uses shifted windows to limit self-attention computation to non-overlapping local windows while still enabling cross-window connection.

In medical imaging, this approach can improve the detection and classification of diseases by enabling the model to focus on relevant image regions and integrate contextual information across the entire image. This capability is particularly beneficial for conditions where subtle features distributed across an image are critical for accurate diagnosis.

8.3.3 Transformer for Timeseries Data

Transformers have also been extended to time series modeling due to their ability to capture long-term dependencies.

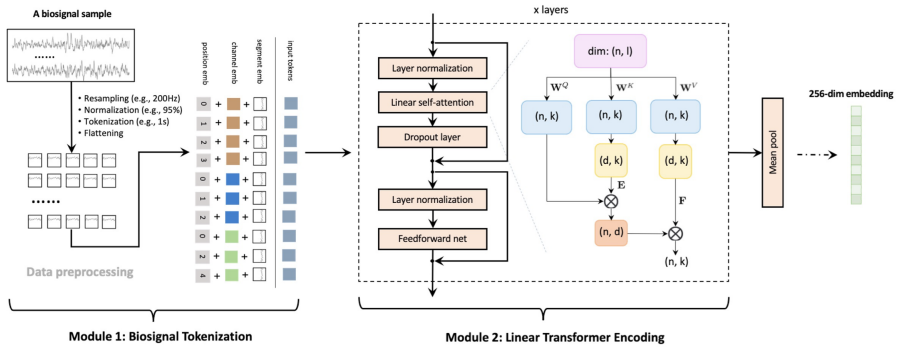

Figure 8.9 Biosignal Transformer (BIOT) [56].

For example, Biosignal Transformer (BIOT) [56] tokenizes each channel separately into fixed-length segments containing local signal features and then re-arranges the segments to form a long "sentence". Channel embeddings and relative position embeddings are added to each segment (viewed as a "token") to preserve spatio-temporal features.

8.4 ICD coding with the ClinicalBERT model

ICD coding assigns codes from the International Classification of Diseases (ICD) to medical notes. This task is commonly performed to support billing, computer-assisted diagnosis, or epidemiological studies. This section will showcase how to automate this process with the ClinicalBERT [53] model using the PyHealth package.

8.4.1 Step1: Dataset loading

The initial step involves loading the data into PyHealth's internal structure. This process is straightforward: import the appropriate dataset class from PyHealth and specify the root directory where the raw dataset is stored. PyHealth will handle the dataset processing automatically.

Specifically, we will be using the medical transcription data scraped from mtsamples.com. This dataset includes the transcripted medical reports and the corresponding medical category. You can find more information about the dataset at this url[3].

```
from pyhealth.datasets import MedicalTranscriptionsDataset

```

[3] https://www.kaggle.com/datasets/tboyle10/medicaltranscriptions

```

root = "/content/MedicalTranscriptions"
base_dataset = MedicalTranscriptionsDataset(root)

base_dataset.stat()
"""
Statistics of MedicalTranscriptionsDataset:
Number of samples: 4999
Number of classes: 40
Class distribution: Counter({' Surgery': 1103, ' Consult - History and Phy.':
                                        516, ' Cardiovascular / Pulmonary': 372
                                        , ...
"""

base_dataset.patients[0]
"""
{'description': ' A 23-year-old white female presents with complaint of
                                        allergies.',
 'medical_specialty': ' Allergy / Immunology',
 'sample_name': ' Allergic Rhinitis ',
 'transcription': 'SUBJECTIVE:,  This 23-year-old white female presents with
                                         complaint of allergies. She used to
                                         have allergies when she lived in
                                         Seattle but she thinks they are worse
                                         here. In the past, she has tried
                                         Claritin, ...',
 'keywords': 'allergy / immunology, allergic rhinitis, allergies, asthma, nasal
                                          sprays, rhinitis, nasal, erythematous
                                         , allegra, sprays, allergic,'}
 """
```

8.4.2 Step 2. Define the Task

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

For this dataset, PyHealth offers a default task specifically for transcription classification. This task takes the transcription text as input and aims to predict the medical categories associated with it.

```
base_dataset.default_task
"""
MedicalTranscriptionsClassification(task_name='
                                        MedicalTranscriptionsClassification',
                                        input_schema={'transcription': 'text'},
                                         output_schema={'label': 'label'})
"""

sample_dataset = base_dataset.set_task()
```

Here is an example of a single sample, represented as a dictionary. The dictionary

contains keys for feature names, label names, and other metadata associated with the sample.

```
sample_dataset[0]
"""
{'transcription': 'SUBJECTIVE:,  This 23-year-old white female presents with
                                    complaint of allergies.  She used to
                                    have allergies when she lived in
                                    Seattle but she thinks they are worse
                                    here. In the past, she has tried
                                    Claritin...',
 'label': ' Allergy / Immunology'}
"""
```

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

```
from pyhealth.datasets import split_by_sample
from pyhealth.datasets import get_dataloader


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

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)
```

8.4.3 Step 3. Define the Model

Next, we will define the deep learning model we want to use for our task. PyHealth supports all major language models available in the Huggingface's Transformer package. You can load any of these models using the *model_name* argument.

```
from pyhealth.models import TransformersModel


model = TransformersModel(
    model_name="emilyalsentzer/Bio_ClinicalBERT",
    dataset=sample_dataset,
    feature_keys=["transcription"],
    label_key="label",
    mode="multiclass",
)
```

8.4.4 Step 4. Training

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

```
from pyhealth.trainer import Trainer


trainer = Trainer(model=model)
```

Now, let's start the training process. The trainer will automatically track the best model based on the metric you set to monitor (e.g., accuracy).

```
trainer.train(
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    epochs=1,
    monitor="accuracy"
)
"""
Training:
INFO:pyhealth.trainer:Training:
Batch size: 32
INFO:pyhealth.trainer:Batch size: 32
Optimizer: <class 'torch.optim.adam.Adam'>
INFO:pyhealth.trainer:Optimizer: <class 'torch.optim.adam.Adam'>
Optimizer params: {'lr': 0.001}
INFO:pyhealth.trainer:Optimizer params: {'lr': 0.001}
Weight decay: 0.0
INFO:pyhealth.trainer:Weight decay: 0.0
Max grad norm: None
INFO:pyhealth.trainer:Max grad norm: None
Val dataloader: <torch.utils.data.dataloader.DataLoader object at
                                        0x7f8769161e40>
INFO:pyhealth.trainer:Val dataloader: <torch.utils.data.dataloader.DataLoader
                                        object at 0x7f8769161e40>
Monitor: accuracy
INFO:pyhealth.trainer:Monitor: accuracy
Monitor criterion: max
INFO:pyhealth.trainer:Monitor criterion: max
Epochs: 1
INFO:pyhealth.trainer:Epochs: 1

INFO:pyhealth.trainer:
Epoch 0 / 1: 100\%
109/109 [02:27<00:00, 1.34s/it]
--- Train epoch-0, step-109 ---
INFO:pyhealth.trainer:--- Train epoch-0, step-109 ---
loss: 3.1713
INFO:pyhealth.trainer:loss: 3.1713

INFO:pyhealth.trainer:--- Eval epoch-0, step-109 ---
accuracy: 0.2540
INFO:pyhealth.trainer:accuracy: 0.2540
f1_macro: 0.0116
INFO:pyhealth.trainer:f1_macro: 0.0116
f1_micro: 0.2540
INFO:pyhealth.trainer:f1_micro: 0.2540
loss: 2.9753
INFO:pyhealth.trainer:loss: 2.9753
New best accuracy score (0.2540) at epoch-0, step-109
```

```
INFO:pyhealth.trainer:New best accuracy score (0.2540) at epoch-0, step-109
Loaded best model
INFO:pyhealth.trainer:Loaded best model
"""
```

8.4.5 Step 5. Evaluation

At the end of training, the trainer will automatically load the best save model weights. So that we can easily evaluate the ResNet model on the test set. This can be done using PyHealth's Trainer.evaluate() function.

```
print(trainer.evaluate(test_dataloader))
"""
{'accuracy': 0.22032193158953722, 'f1_macro': 0.010030228084638637, 'f1_micro':
                                        0.22032193158953722, 'loss': 3.
                                       106539271771908}
"""
```

The complete pyhealth implementation pipeline is provided in this folder [4].

8.5 Drug Recommendation with the Transformer Model

Drug recommendation is an essential task of AI for healthcare. The goal is to recommend a set of medications for a particular patient based on the patient's health conditions. Specifically, the input is a patient EHR data which captures comprehensive medical histories of patients in the format of longitudinal vectors of medical codes (e.g., diagnoses, procedures, and drugs). And the output is a set of medications. This is a multilabel classification problem.

Next, we will show how we can use the Transformer model to perform this task with the help of the PyHealth package.

8.5.1 Step1: Dataset loading

We will be using a synthetic version of the MIMIC-III dataset. The original MIMIC-III dataset is a single-center database containing 53K ICU records from Beth Israel Deaconess Medical Center. We created a synthetic version on top of it for easy distribution.

The initial step involves loading the data into PyHealth. To do this, we will use the MIMIC3Dataset API.

```
from pyhealth.datasets import MIMIC3Dataset

mimic3_ds = MIMIC3Dataset(
        root="https://storage.googleapis.com/pyhealth/Synthetic_MIMIC-III/",
        tables=["DIAGNOSES_ICD", "PROCEDURES_ICD", "PRESCRIPTIONS"],
        code_mapping={"NDC": ("ATC", {"target_kwargs": {"level": 3}})},
```

[4] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap6-Transformer/notebook

```
        dev=True
)
```

Here, "root" is the arguments directing to the data folder. "tables" is a list of table names from raw databases, which specifies the information that will be used in building the pipeline. "code_mapping " asks for a dictionary input, setting the new coding systems for each data table. For example, "NDC": ("ATC", "target_kwargs": "level": 3) means that our pyhealth will automatically change the codings from NDC into ATC-3 level for tables if any. We set dev to True only to load a small set of patients.

8.5.2 Step 2: Define the task

This step assigns a task function to the dataset for data loading pyhealth.tasks. The task function specifics how to process each patient's data into a set of samples for the downstream machine learning models.

```
from pyhealth.tasks import drug_recommendation_mimic3_fn

mimic3_ds = mimic3_ds.set_task(task_fn=drug_recommendation_mimic3_fn)
# stats info
mimic3_ds.stat()
```

Specifically, let us delve into the *drug_recommendation_mimic3_fn* task function.

```
    def drug_recommendation_mimic3_fn(patient: Patient):
    """Processes a single patient for the drug recommendation task."""
    samples = []
    for i in range(len(patient)):
        visit: Visit = patient[i]
        conditions = visit.get_code_list(table="DIAGNOSES_ICD")
        procedures = visit.get_code_list(table="PROCEDURES_ICD")
        drugs = visit.get_code_list(table="PRESCRIPTIONS")
        # ATC 3 level
        drugs = [drug[:4] for drug in drugs]
        # 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,
                "drugs_hist": drugs,
            }
        )
    # exclude: patients with less than 2 visits
    if len(samples) < 2:
        return []
    # add history
    samples[0]["conditions"] = [samples[0]["conditions"]]
    samples[0]["procedures"] = [samples[0]["procedures"]]
```

```
    samples[0]["drugs_hist"] = [samples[0]["drugs_hist"]]

    for i in range(1, len(samples)):
        samples[i]["conditions"] = samples[i - 1]["conditions"] + [
            samples[i]["conditions"]
        ]
        samples[i]["procedures"] = samples[i - 1]["procedures"] + [
            samples[i]["procedures"]
        ]
        samples[i]["drugs_hist"] = samples[i - 1]["drugs_hist"] + [
            samples[i]["drugs_hist"]
        ]

    # remove the target drug from the history
    for i in range(len(samples)):
        samples[i]["drugs_hist"][i] = []

    return samples
```

The code snippet above demonstrates that, given all the EHR data for an individual patient, how to generate a list of samples, where each sample is a key-value dictionary. Specifically, we first loop through each visit for the patient. We extract the diagnoses (conditions), procedure, and drug codes. Visits without any code or patients with less than two visits are discarded. Then, we apply another for loop to include the diagnosis and procedure history up to the current visit. Our goal is to recommend a set of medications for the current visit based on the diagnosis and procedure information from the past and current visits.

8.5.3 Step 3: Initialize the ML model

Now we initialize the Transformer model. "dataset" specifies the dataset to train the model. It is used to query specific information such as the set of all tokens. "feature_keys" is a list of keys in samples to use as features. "label_key" is the key in samples to use as label. "mode" should be one of binary, multiclass, or multilabel.

```
from pyhealth.models import Transformer

model = Transformer(
    dataset=mimic3_ds,
    feature_keys=["conditions", "procedures"],
    label_key="drugs",
    mode="multilabel",
)
```

In the code snippet above, we set the dataset as the synthetic patient dataset we loaded. We use the conditions and procedure codes as input and try to predict drugs as labels. The problem is multilabel classification.

8.5.4 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 monitor the sample-level the precision-recall area under curve (PRAUC) score. We set the training epochs to 3. The best model will automatically be loaded in the Trainer for the next evaluation step. The configuration of Trainer could refer to this page[5].

```
from pyhealth.trainer import Trainer

trainer = Trainer(model=model)
trainer.train(
    train_dataloader=train_loader,
    val_dataloader=val_loader,
    epochs=3,
    monitor="pr_auc_samples",
)
```

8.5.5 Step 5: Evaluation on test set

Next, we use pyhealth to evaluate the trained model. We can either use the built-in evaluation metric stored in the trainer. Or, we can use any appropriate metrics for multilabel classification provided in pyhealth.metrics.

```
# option 1: use our built-in evaluation metric
score = trainer.evaluate(test_loader)
print (score)

# option 2: use our pyhealth.metrics to evaluate
from pyhealth.metrics.multilabel import multilabel_metrics_fn
y_true, y_prob, loss = trainer.inference(test_loader)
multilabel_metrics_fn(y_true, y_prob, metrics=["pr_auc_samples"])
```

We could obtain the output values as below.

```
"""
OUTPUT:
{'pr_auc_samples': 0.24748548437566276, 'loss': 0.8778110146522522}
{'pr_auc_samples': 0.24748548437566276}
"""
```

The complete pyhealth implementation pipeline is provided in this folder [6].

[5] https://pyhealth.readthedocs.io/en/latest/api/trainer.html
[6] https://github.com/sunlabuiuc/pyhealth-book/tree/main/chap6-Transformer/notebook

8.6 Takeaways

- **Attention is all you need:** The Transformer architecture is based solely on attention mechanisms, eliminating recurrence in RNN models. This allows more parallelization during training.
- **Several ideas in Transformer:** Key components of Transformers include multi-head self-attention, feedforward networks, residual connections and layer normalization, and position encodings.
- **Encoder and decoder modules:** Transformers have an encoder-decoder structure. The decoder attends to the encoder output and uses causal masking for autoregressive generation.
- **BERT an encoder-only transformer:** BERT introduced masked language modeling and next sentence prediction during pre-training, advancing NLP applications.
- **Variants of BERT:** Variants like RoBERTa and ClinicalBERT improve optimization and domain generalization of BERT.
- **Transformer for healthcare applications** Beyond NLP, Transformers are adaptable for sequence, image, and time series data in healthcare applications.
- **PyHealth Transformer code examples:** Transformers enable clinical tasks like ICD coding, outcome prediction, and personalized drug recommendation from EHRs.

This chapter presents the Transformer architecture, pre-training strategies employed by models like BERT and diverse applications in healthcare enabled by this breakthrough technique. The key components and innovations of Transformers are the foundation for many AI applications including large language models.

Questions

- Describe the role of self-attention in the Transformer model. How does it allow the model to weigh the importance of different parts of the input data?
- Explain the multi-head attention mechanism in the Transformer architecture. How does splitting the attention mechanism into multiple "heads" improve the model's performance?
- Detail the feedforward neural networks in the Transformer blocks. What role do they play, and how are they configured within the Transformer architecture?
- Discuss the layer normalization and residual connections in Transformers. How do these components contribute to the model's training stability and performance?
- Explain the encoder-decoder structure of the original Transformer model. How do they interact during the model's training and inference phases?
- What are some of the main challenges and limitations of the Transformer model, and how have subsequent models (e.g., BERT) addressed these issues?

- Describe how the Transformer model can be adapted for tasks beyond sequence-to-sequence problems, such as classification and regression. Provide examples of such adaptations.
- Find some cool applications of Transformer in healthcare. How have they influenced the development of new models and applications in healthcare domains?