What is The Meaning of Life?

Unknown[1], Anirbit Mukherjee [2]

[1]Nowhere
[2]Department of Computer Science, The University of Manchester, UK

*To whom correspondence should be addressed; E-mail:

Contents

1 Example for Referencing **2**

2 Introduction **2**

3 Setup and Main Results **2**

4 Notes on PAC learning **2**

1 Example for Referencing

[SPW18] is a great paper!

2 Introduction

3 Setup and Main Results

4 Notes on PAC learning

D_X	Some distribution over X where $X \in \mathbb{R}$
$f : X \to \{0, 1\}$	A function mapping X values to real numbers between 0 and 1
S	Collection of samples generated by D i.i.d and labeled by f
$h : X \to Y$	A hypothesis which maps X to Y
H	Hypothesis class which contains all chosen finite hypothesis h
$L_S(h) = \frac{\lvert\{i \in [m] : h(x_i) \neq y_i\}\rvert}{m}$	The loss function
$\epsilon, \delta \in (0, 1)$	$Epsilon$ and $Delta$ are the accuracy and confidence parameters respectively
$m_H(\epsilon, \delta) : (0, 1)^2 \to \mathbb{N}$	The minimum number of samples needed to produce a hypothesis given ϵ and δ.

DEFINITION (PAC Learnability)
A hypothesis class H is PAC learnable if there exists a function m_H and a learning algorithm with the following property: For every ϵ,$\delta \in (0, 1)$, for every distribution D_X, and for every labeling function f, if the realizable assumption holds with respect to H, D, f, then when running the learning algorithm on $m \geq m_H(\epsilon, \delta)$ samples S, the algorithm return a hypothesis h such that, with probability of at least 1 - δ, $L_{(D,f)}(h) \leq \epsilon$.

DEFINITION (Agnostic PAC Learning)
From now on let D be a probability distribution over X x Y, where formally X is our domain set and Y is a set of labels. That is, D is a joint distribution over over domain points and labels. The distribution can be viewed as being split into two parts, D_x being the probability distribution over unlabeled domain points, and a conditional probability over labels for each domain point, $D((x, y)|x)$

5 How the transformer works

In this section, we will be going through the architecture of the transformer section by section and showing how it works using a translation task as an example.

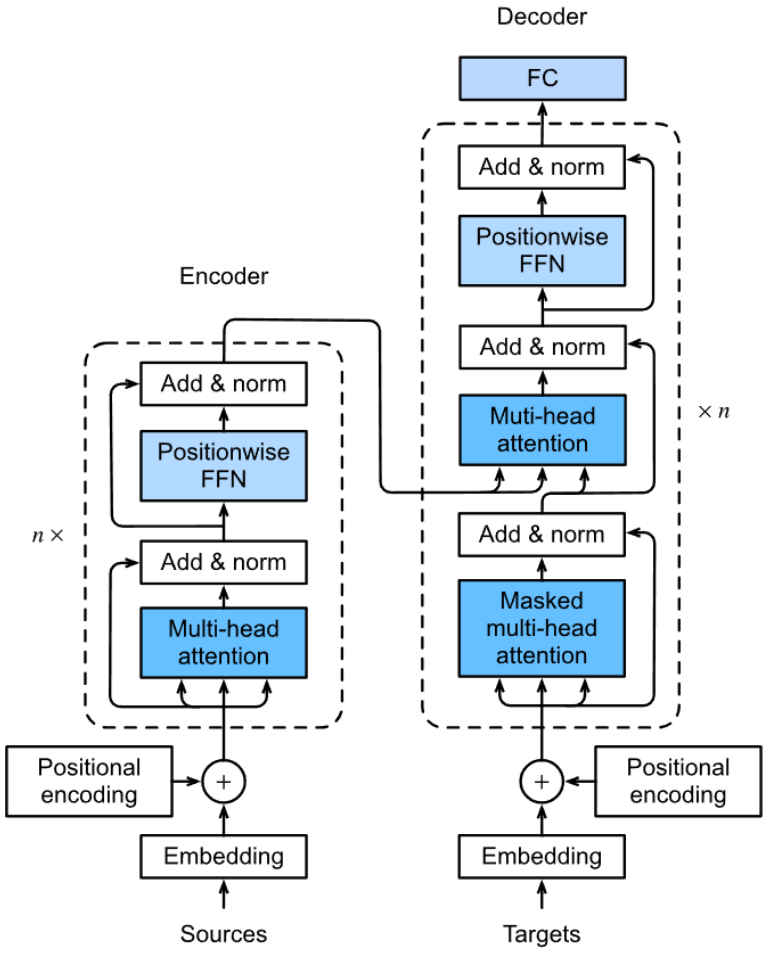

Figure 1: The Transformer Architecture

5.1 The training data

To explain how a transformer works, I will be using a English to French translation task as an example. We will go through the transformer architecture step by step and observe how the data is handled throughout. Figure 2 shows an example of data that is used for training the transformer. The transformer should aim to be able to predict the correct French translation given an English sequence of inputs.

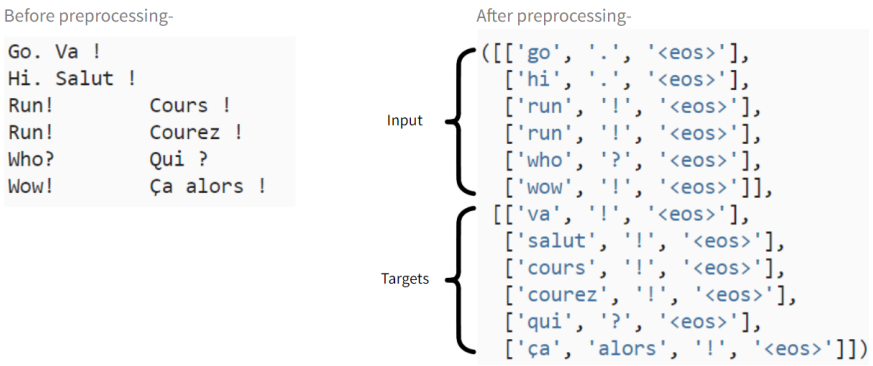

Figure 2: Data example

A singular training data would be a pair of $X_i^{(\text{enc})}$ and $X_i^{(\text{dec})}$.

```
source: ['hi', '.', '<eos>']
target: ['salut', '!', '<eos>']
```

Figure 3: A pair of preprocessed words

Before being passed into the transformer, the training data is padded or truncated with a padding token up to size N, which is determined before training. A separate list called valid_lens $\in \mathbb{N}^N$ is also generated during this process which tells us the length of the english words before being padded (['hi','.','<eos>'] would be 3 in valid_lens).

```
source: ['hi', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>']
target: ['<bos>', 'salut', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>']
```

Figure 4: Preprocessed words after padding

The words are then replaced with the dictionary index corresponding to the word. e.g. 'go' would be replaced by 71, and '.' by 2. Note: The two languages do not share the same dictionary, so the index number 3 may correspond to '<eos>' in the english dict whilst in the french dict it is '<bos>'

```
source: tensor([71,  2,  3,  4,  4,  4,  4,  4,  4], dtype=torch.int32)
target: tensor([  3, 176,   2,   4,   5,   5,   5,   5,   5], dtype=torch.int32)
```

Figure 5: Preprocessed words after being replaced by index

5.2 Embedding

In the translation example, the transformer calculates the attention between the English word and French word in order to calculate a probability that the French word is predicted given the English word. In order to do that the words used in the calculations must be represented using vectors, and that is the purpose of the Embedding layer.

The Embedding layer stores a look-up table of words used within the vocabulary, and for each call to a word it's corresponding vector representation is returned. Below shows an example of a call to a word "go" at index position 2.

```
['go'].index = 2
lookup_table = [[8.09, 3.12, 3.12, 5.21, 5.48],
 [5.01, 8.15, 2.85, 1.21, 5.01],
 [8.70, 1.01, 3.66, 4.52, 5.80], #<-- 'go'
 ...]
```

Figure 6: Example lookup table

This lookup table is a trainable parameter that is updated throughout the training process.

5.3 Positional Encoding

After the embedding layer, our words will be represented using the matrix $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, where D is the dimensions of the word vector(This value is determined before the training process) and N is the maximum data length. N is determined during data preprocessing and shorter sequences are padded whilst longer sequences are truncated.

As the word vectors themselves do not have their positions represented within the vector, we will need the positional encoding layer to add that component. This layer performs $\boldsymbol{X} + \boldsymbol{P}$, with $\boldsymbol{X}$ being the matrix containing the word vectors and P being a matrix constructed with the below conditions.

$$\boldsymbol{P}_{i,2j} = \sin\left(\frac{i}{10000^{2j/D}}\right),$$
$$\boldsymbol{P}_{i,2j+1} = \cos\left(\frac{i}{10000^{(2j+1)/D}}\right).$$

For every even j position, its value is given by $\sin\left(\frac{i}{10000^{2j/d}}\right)$ and for odd j positions, $\cos\left(\frac{i}{10000^{2j/d}}\right)$. The output of this layer will be a matrix of the same size, but with a positional value added to its matrix representation. Note: D is the dimension of any word vector.

The reason this formula is used can be attributed to multiple reasons. Think about it, if you were to use a positional encoding of just i, why would that not work (i is the position of word i in the sentence)? Although each position is unique, for larger numbers the positional encoding would dominate the values of the original encoding (For position $i = 10$, the result of $\boldsymbol{X} + \boldsymbol{P}$ could be 10.14). What about i/N? Although the positional vector would only go up to 1, for sentences of different length, the positional vector would be different, as N would change. So for sentences of different length, the word $i + \phi$ where phi is some offset, this number would be different, and make it hard for the transformer to "learn" this relationship (for example, nouns followed by verbs).

The formula we use for positional encoding overcomes these two issues, as the values of $\boldsymbol{P}$ stay between -1 and 1, and the relationship between words of offset ϕ can be calculated using a linear transformation. Each word in different positions are also represented by a unique sin or cos function.

5.4 Scaled Dot Product Attention

The main attention function we will be using is dot product attention, or more specifically, scaled dot product attention. It takes in three matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times D}$ and outputs a singular matrix of size $N \times D$. $\boldsymbol{Q}, \boldsymbol{K}$ and $\boldsymbol{V}$ is different depending on where in the transformer it is called, so we will not be going into detail on what they are for now.

$$\text{softmax}(\tilde{\boldsymbol{X}}) = [\ldots, \frac{e^{\tilde{\boldsymbol{X}}}_{ij}}{\sum_{j=1}^{D} e^{\tilde{\boldsymbol{X}}}_{ij}}, \ldots]$$

$$\text{ScaledDotProductAttention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}(\frac{(\boldsymbol{Q}\boldsymbol{K}^T)}{\sqrt{D}})\boldsymbol{V} \in \mathbb{R}^{N \times D}$$

The scaled dot product attention function is used to measure the similarity between the query and key matrix, and higher weighting is given to values with high similarity scores.

The result of this function applied to some query key and value matrix would be the matrix containing a weighted sum of word vectors. In other words, the output matrix would be a re-weighted matrix, with the weights calculated using query and key, before multiplying the weights with the value matrix.

```
[8]  test = torch.tensor([[0, 1, 4],[3, 4, 5]], dtype = torch.float32)
     masked_softmax(test, None)

     tensor([[0.0171, 0.0466, 0.9362],
             [0.0900, 0.2447, 0.6652]])
```

Figure 7: Example of softmax over test matrix

Note that the softmax function sums over rows or the feature dimension of each word vector.

5.5 Multi-Headed Attention

The Multi-Headed Attention layer performs Scaled Dot Product Attention multiple times instead of just once, allowing for the transformer to attend to different parts of the word vector representations with each heads. It takes in a query, key and value matrix of size $N \times D$, performs a linear transformation between the matrices and some trainable weights then splits them into h(head) matrices of size $N \times \frac{D}{H}$. Figure 8 shows an example of this split.

The output of head h_i can be given as:

$$h_i = \text{ScaledDotProductAttention}(\boldsymbol{Q}_i\boldsymbol{W}_i^{(q)}, \boldsymbol{K}_i\boldsymbol{W}_i^{(k)}, \boldsymbol{V}_i\boldsymbol{W}_i^{(v)}) \in \mathbb{R}^{N \times \frac{D}{H}}$$

$$h_i = \text{softmax}(\frac{((\boldsymbol{Q}_i\boldsymbol{W}_i^{(q)})(\boldsymbol{K}_i\boldsymbol{W}_i^{(k)})^T)}{\sqrt{D}})(\boldsymbol{V}_i\boldsymbol{W}_i^{(v)}) \in \mathbb{R}^{N \times \frac{D}{H}}$$

something is looking odd here - where is the dependency on the data $\boldsymbol{X}$?

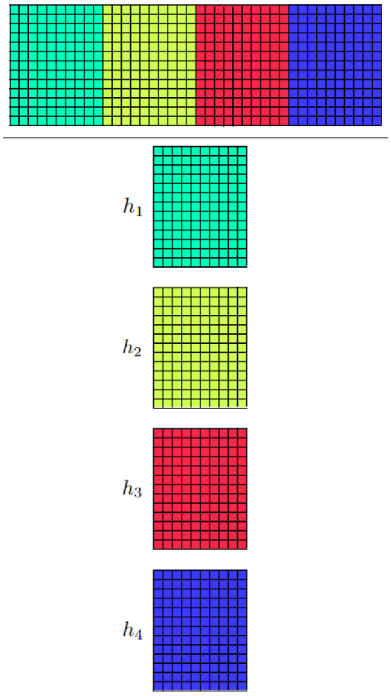

Figure 8: Example of data being split into $H = 4$ heads

The outputs of each head are then concatenated into one matrix, and another linear transformation $\mathbf{W}_o$ is applied before returning the final matrix.

$$\begin{bmatrix} h_1 \\ \vdots \\ h_H \end{bmatrix} \boldsymbol{W}_o \in \mathbb{R}^{N \times D}$$

$$\begin{bmatrix} \text{softmax}(\frac{((\boldsymbol{Q}_1 \boldsymbol{W}_1^{(q)})(\boldsymbol{K}_1 \boldsymbol{W}_1^{(k)})^T)}{\sqrt{D}})(\boldsymbol{V}_1 \boldsymbol{W}_1^{(v)}) \\ \vdots \\ \text{softmax}(\frac{((\boldsymbol{Q}_H \boldsymbol{W}_H^{(q)})(\boldsymbol{K}_H \boldsymbol{W}_H^{(k)})^T)}{\sqrt{D}})(\boldsymbol{V}_H \boldsymbol{W}_H^{(v)}) \end{bmatrix} \boldsymbol{W}_o \in \mathbb{R}^{N \times D}$$

5.6 Position-wise Feed-Forward-Network

This neural network consists of an input layer, a ReLU activation layer and an output layer. This layer can be represented by:

$$\mathcal{N}(\boldsymbol{X}) = \max(0, \boldsymbol{X}\boldsymbol{W}_1)\boldsymbol{W}_2$$

where $W_1 \in \mathbb{R}^{D \times B}$ and $W_2 \in \mathbb{R}^{B \times D}$ are learnable parameters (B is the number of layers for this neural network decided before training). This layer helps the model capture more complex relationships between words.

5.7 Add and Normalise

The add and normalisation layer takes the input and output of the multi-headed attention layer or the position-wi se FFN layer and performs the following operation on them:

$$\text{LayerNorm}(\boldsymbol{Y} + \boldsymbol{X}), \text{ where}$$

$$\text{LayerNorm}(\boldsymbol{X}_{ij}) = \left[\ldots, \frac{\boldsymbol{X}_{ij} - \hat{\mu}_i}{\hat{\sigma}_i}, \ldots\right]$$

$$\hat{\mu}_i = \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{X}_{ij}$$

$$\hat{\sigma}_i = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(\boldsymbol{X}_{ij} - \hat{\mu}_i)^2 + \epsilon}$$

With Y and X being the two inputs into the layer, the layer sums them up and normalises them using the mean and standard deviation of the result of the sum. The purpose of this layer is to make sure information is not lost between either the FFN or the attention by adding the unprocessed matrix back to the output matrix. Note: ϵ is a small number to prevent rooting 0 and n is the size of $\boldsymbol{X}$.

5.8 Encoder

Referring back to Figure 1, we can see that the encoder block consists of one multi-head attention followed by an add and normalise layer, being fed into a positionwise FFN and another add and normalise layer. For any transformer, this encoder block can be ”stacked” on top of each other to form multiple encoder blocks, which can help capture more complex relationships.

The multi-headed attention takes in three matrices, and in the encoder the three query key value matrices is just $\boldsymbol{X}$, it is only after these matrices are multiplied by the trainable parameters do they differ.

TBC

5.9 Decoder

5.10 Loss function

The loss function of this transformer uses mini-batch gradient descent with cross-entropy loss. It can be formulated as such:

$$\hat{\boldsymbol{Y}} \in \mathbb{R}^{S\times N\times \text{vocab-size}}, \boldsymbol{Y} \in \mathbb{R}^{S\times N}$$

$$\frac{1}{S}\sum_{i=1}^{S}\left[-\frac{1}{N}\sum_{j=1}^{N}\ln\left(\hat{\boldsymbol{Y}}_{i,j,(\boldsymbol{Y}_{i,j})}\right)\right]$$

Where S is the total amount of data used for training and validation and N is the maximum data length. N is determined during data preprocessing and shorter sequences are padded whilst longer sequences are truncated. $\hat{\boldsymbol{Y}}$ is the output of the tranformer, where i represents the i^{th} training data and j representing the j^{th} token. $\boldsymbol{Y}$ is the matrix containing the dictionary index of the French words excluding the ¡bos¿ token. So if the first word in the first french training data $(i, j = 1, 1)$ is ”va”, and the dictionary index for ”va” is 117, $Y_{1,1}$ would be 117.

Here we go into more detail on how $\hat{\boldsymbol{Y}}$ is obtained. For each training data pair $(X_i^{(\text{enc})}, X_i^{(\text{dec})})$ up to S training data, the following is performed.

$$\hat{\boldsymbol{Y}}_i = \text{Decoder}(X_i^{(\text{dec})}, \text{Encoder}(X_i^{(\text{enc})}, \text{valid\_lens}), \text{valid\_lens})$$

The result is S number of $N \times$ vocab-size matrices, as the output of the Decoder is a $N \times$ vocab-size matrix.

Algorithm 1 Embed

Input:
$A \in \mathbb{R}^N$
Output:
$B \in \mathbb{R}^{N \times D}$
$A \to B$

Algorithm 2 PositionalEncoding

Input:
$\boldsymbol{X} \in \mathbb{R}^{N \times D}$
Output:
$\boldsymbol{X} + \tilde{\boldsymbol{P}} \in \mathbb{R}^{N \times D}$

$\boldsymbol{P}_{ij} \in \mathbb{R}^{N \times D}$
for $i = 1$ to N **do**
  for $j = 1$ to D **do**
    $\boldsymbol{P}_{ij} \leftarrow \frac{i}{10000^{2j/D}}$
  end for
end for
$\tilde{\boldsymbol{P}}_{ij} \in \mathbb{R}^{N \times D}$
for $i = 1$ to N **do**
  for $j = 1$ to D **do**
    if j is even **then**
      $\tilde{\boldsymbol{P}}_{ij} \leftarrow \sin(\boldsymbol{P}_{ij})$
    else
      $\tilde{\boldsymbol{P}}_{ij} \leftarrow \cos(\boldsymbol{P}_{ij})$
    end if
  end for
end for

Decoder is WIP

Algorithm 3 AddNorm

Input:
$\boldsymbol{X} \in \mathbb{R}^{N \times D}$
$\boldsymbol{Y} \in \mathbb{R}^{N \times D}$
Output:
$\tilde{\boldsymbol{X}} \in \mathbb{R}^{N \times D}$

$\tilde{\boldsymbol{X}} \leftarrow \boldsymbol{X} + \boldsymbol{Y}$
$\hat{\mu}, \hat{\sigma} \in \mathbb{R}^{N}$
for $i = 1$ to N **do**
    $\hat{\mu}_i = \frac{1}{D} \sum_{j=1}^{D} \tilde{\boldsymbol{X}}_{ij}$ ▷ Compute mean for each layer
    $\hat{\sigma}_i = \sqrt{\frac{1}{D} \sum_{j=1}^{D} (\tilde{\boldsymbol{X}}_{ij} - \hat{\mu}_i)^2 + \epsilon}$ ▷ Compute variance for each layer
    for $j = 1$ to D **do**
        $\tilde{\boldsymbol{X}}_{ij} = \frac{\tilde{\boldsymbol{X}}_{ij} - \hat{\mu}_i}{\hat{\sigma}_i}$ ▷ Normalize each element
    end for
end for

Algorithm 4 MultiHeadedAttention

Input:
$\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times D}$
Condition:
$D \in \mathbb{N}, H \in \mathbb{N}, \frac{D}{H} \in \mathbb{N}$
Output:
$\boldsymbol{O} \in \mathbb{R}^{N \times D}$

$\tilde{\boldsymbol{Q}}, \tilde{\boldsymbol{K}}, \tilde{\boldsymbol{V}} \in \mathbb{R}^{N \times D}$
$\boldsymbol{W}^{(q)}, \boldsymbol{W}^{(k)}, \boldsymbol{W}^{(v)} \in \mathbb{R}^{D \times D}$
$\tilde{\boldsymbol{Q}} \leftarrow \boldsymbol{Q}\boldsymbol{W}^{(q)}$
$\tilde{\boldsymbol{K}} \leftarrow \boldsymbol{K}\boldsymbol{W}^{(k)}$
$\tilde{\boldsymbol{V}} \leftarrow \boldsymbol{V}\boldsymbol{W}^{(v)}$ ▷ Linear projections
$\tilde{\boldsymbol{Q}}_H, \tilde{\boldsymbol{K}}_H, \tilde{\boldsymbol{V}}_H \in \mathbb{R}^{H \times N \times \frac{D}{H}}$
$\tilde{\boldsymbol{Q}}_H \leftarrow [q_1, q_2, ..., q_H] \in \mathbb{R}^{H \times N \times (D/H)}$
$\tilde{\boldsymbol{K}}_H \leftarrow [k_1, k_2, ..., k_H] \in \mathbb{R}^{H \times N \times (D/H)}$
$\tilde{\boldsymbol{V}}_H \leftarrow [v_1, v_2, ..., v_H] \in \mathbb{R}^{H \times N \times (D/H)}$
$q, k, v \in \mathbb{R}^{N \times (D/H)}$
$q_{i(n,d)} \leftarrow \tilde{\boldsymbol{Q}}_{(n,(d+i(D/H)))}$
$k_{i(n,d)} \leftarrow \tilde{\boldsymbol{K}}_{(n,(d+i(D/H)))}$
$v_{i(n,d)} \leftarrow \tilde{\boldsymbol{V}}_{(n,(d+i(D/H)))}$
$\boldsymbol{O}_H[i] \in \mathbb{R}^{H \times N \times \frac{D}{H}}$
for $i = 1$ to H **do**
    $\boldsymbol{O}_H[i] \leftarrow \text{DotProductAttention}(\tilde{\boldsymbol{Q}}_H[i], \tilde{\boldsymbol{K}}_H[i], \tilde{\boldsymbol{V}}_H[i])$
end for
$\tilde{\boldsymbol{O}} \in \mathbb{R}^{N \times D}$
$\tilde{\boldsymbol{O}} \leftarrow \text{Concatenate}(\boldsymbol{O}_H)$ ▷ Concatenate along D dimension
$\boldsymbol{W}^{(o)} \in \mathbb{R}^{D \times D}$
$\boldsymbol{O} = \tilde{\boldsymbol{O}}\boldsymbol{W}^{(o)}$ ▷ Final linear projection

Algorithm 5 Encoder

Hyperparameters

D, ffn_num_hiddens, H, num_blks $\in \mathbb{N}$ ▷ D decides the dimensionality of the

Input:

$X \in \mathbb{N}^N$

valid_lens $\in \mathbb{N}^N$

Output:

$\boldsymbol{Y}_o \in \mathbb{N}^{N \times D}$

$\boldsymbol{X}_e \leftarrow \text{PositionalEncoding}(\text{Embed}(X) \times \sqrt{D}) \in \mathbb{R}^{N \times D}$ ▷ Embedding and Positional Encoding

$\boldsymbol{Y} \leftarrow \text{AddNorm}(\boldsymbol{X}_e, \text{MultiHeadedAttention}(\boldsymbol{X}_e, \boldsymbol{X}_e, \boldsymbol{X}_e, \text{valid\_lens})) \in \mathbb{R}^{N \times D}$ ▷ Self-attention

$\boldsymbol{Y}_o \leftarrow \text{AddNorm}(\boldsymbol{Y}, \text{PositionWiseFFN}(\boldsymbol{Y})) \in \mathbb{R}^{N \times D}$ ▷ Final Feedforward

Algorithm 6 Decoder

Hyperparameters:

vocab_size, num_step $\in \mathbb{N}$

Input:

$X \in \mathbb{R}^N$

$\boldsymbol{O}_{\text{enc}} \in \mathbb{R}^{N \times D}$ ▷ Output of encoder

$\text{valid\_lens}_{\text{enc}} \in \mathbb{N}^N$

$\text{state}_{\text{dec}}$

Initialization:

$\boldsymbol{O}_{\text{enc}} \in \mathbb{R}^{N \times D}$

$\text{valid-lens}_{\text{enc}} \in \mathbb{N}^N$

dec-states = None on the first input

$X_e = \text{PositionalEncoding}(\text{Embed}(X) \times \sqrt{D}) \in \mathbb{R}^{N \times D}$

key-values $\in \mathbb{R}^{(\text{num-steps} \times N) \times D}$

if dec-states = None **then**

    key-values = X_e

else

    key-values = $\text{Concatenate}_1(\text{state}['\text{dec-states}'][i], X_e)$

end if

dec-states = key-values

$X_o = \text{MultiHeadedAttention}(X_e, \text{key-values}, \text{key-values}, \text{dec-valid-lens}) \in \mathbb{R}^{N \times D}$

$Y = \text{AddNorm}(X_e, X_o) \in \mathbb{R}^{N \times D}$

$Y_o = \text{MultiHeadedAttention}(Y, \text{enc-outputs}, \text{enc-outputs}, \text{enc-valid-lens}) \in \mathbb{R}^{N \times D}$

$Z = \text{AddNorm}(Y, Y_o) \in \mathbb{R}^{N \times D}$

$Z_o = \text{AddNorm}(Z, \text{PositionWiseFFN}(Z)) \in \mathbb{R}^{N \times D}$

$W \in \mathbb{R}^{D \times \text{vocab-size}}$

$Output = Z_o W \in \mathbb{R}^{N \times \text{vocab-size}}$