Homework 4 Part 2

Attention-based End-to-End Speech-to-Text Deep Neural Network

11-785: Introduction to Deep Learning (Fall 2024)

OUT: **November 8, 2024, at 11:59 PM ET**
Preliminary Submission (and Canvas Quiz): **November 22, 2024, at 11:59 PM ET**
DUE: **December 6, 2024, at 11:59 PM ET**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.
- **Submission:**
  - You need to go through the writeup and finish the Canvas quiz by the early submission deadline.
  - You need to implement what is mentioned in the write-up (although you are free and encouraged to explore further) and submit your results to Kaggle . We will share a Google form or an Autolab link after the Kaggle competition ends for you to submit your code.
- **TL; DR:**
  - In this homework you will work on a sequence-to-sequence conversion problem, in which you must train models to transcribe speech recordings into word sequences, spelled out alphabetically.
  - As in previous HWP2s, this homework too is conducted as a Kaggle competition. You can access the kaggle for the homework at this link.
  - You can download the training and test data directly from Kaggle using their API or this link. Your download will include the following:
    - Train Data: The primary training data is located in the train-clean-100 folder. It includes data pairs comprising sequences of audio feature vectors and their transcriptions, stored in the fbank and text folders, respectively.
    - Text-for-LM: This is a large text-only dataset located in the text-for-LM folder. It is used for pre-training the Transformer Decoder as a language model (LM).
    - Validation Data: The validation data is located in the dev-clean folder. This dataset contains both filterbank features and transcripts for model validation.
    - Test Data: The test data is located in the test-clean folder. It contains only filterbank features and is used strictly for evaluation.
  - You must use the train-clean-100 data to train the ASR model. The text-for-LM dataset can only be used for pre-training the decoder as a language model. Data in the dev-clean folder should be used only for validation and not for training.
  - You must transcribe the utterances in the test-clean folder, format your results according to the specifications in submission.csv, and upload the file to Kaggle for evaluation.
  - You will be graded by your performance. Additional details will be posted on piazza.

Homework objectives

If you complete this homework successfully, you would ideally have learned:

- To implement a Transformer architecture to solve a sequence-to-sequence problem:
  - How to setup an encoder with mutlihead-attention and self-attention.
  - How to setup a decoder with cross-attention, multihead-attention and self-attention.
  - How to use attention mechanism with masking.
  - How to use positional encoding.
  - How to train your transformer with different strategies
- You would have learned how to address some of the implementation details
  - How to setup a teacher forcing for the decoder training.
  - How to use search techniques, such as greedy search, for inference in the transformer decoder to generate the output sequence.
  - How to pad-pack the variable length data
- To explore architectures and hyperparameters for the optimal solution
  - To tune parameters and hyperparameters that affect your solution.
  - To effectively explore the search space and strategically finding the best solution.
- As a side benefit you would also have learned how to construct a speech recognition system using the transformer encoder-decoder architecture.

Important: A note on presentation style and pseudocode

The following writeup is intended to be reasonably detailed and explains several concepts through pseudocode. However there is a caveat; the pseudocode is intended to be *illustrative*, not *exemplary*. It conveys the concepts, but you cannot simply convert it directly to python code. For instance, most of the provided pseudocode is in the form of functions, whereas your python code would use classes (in fact, we've written the pseudocode as functions with the express reason that you *cannot* simply translate it to your code. Function-based DL code will be terribly inefficient. You *must* use classes). Please write reusable code wherever possible, and avoid redundant calculations to make your code more efficient.

Nonetheless, we hope that when you read the entire write-up, you will get a reasonable understanding of how to implement (at least the baseline architecture of) the homework.

Contents

Checklist

Here is a checklist page that you can use to keep track of your progress as you go through the write-up and implement the corresponding sections in your starter notebook. As you complete your unit tests for each component to verify your progress, you can check the corresponding boxes aligned with each section and go through this writeup and starter notebook simultaneously step by step.

1. Read Introduction and Notebook Walkthrough.
2. Read the Writeup, and finish the Canvas Quiz.
3. Dataset and Dataloader.
    - ☐ Dataset Description.
4. Encoder
    - ☐ Encoder and Transformer Encoder Description.
    - ☐ Implement the Encoder Layer.
    - ☐ Implement Transformer Encoder.
5. Decoder
    - ☐ Speech Transformer Decoder Description.
    - ☐ Implement Transformer Decoder (Decoder Layer)).
    - ☐ Implement Decoder (combine Decoder modules).
6. Training
    - ☐ Training Description.

1 Introduction

Key new concepts: Sequence-to-sequence conversion, Encoder-decoder architectures, Attention, Transformer.

Core Ideas: Positional Encoding, Multihead-attention, Cross-attention, Self-attention, Masking.

Mandatory implementation: Encoder-decoder architecture. Your solution *must* implement an encoder-decoder architecture *with* attention. While it may be possible to solve the homework using other architectures, you will not get marks for implementing those.

Restrictions: You may not use any data besides that provided as part of this homework. You may not use the validation data for training.

1.1 Overview

In this homework you will learn to build neural networks for sequence-to-sequence conversion or transduction.

"Sequence-to-sequence" conversion refers to problems where a sequence of inputs goes into the system, which emits a sequence of outputs in response. The need for such conversion arises in many problems, e.g.

- Speech recognition: A sequence of speech feature vectors is input. The output is a sequence of characters writing out what was spoken.
- Machine translation: A sequence of words in one language is input. The output is a sequence of words in a different language.
- Dialog systems: A user's input goes in. The output is the system's response.

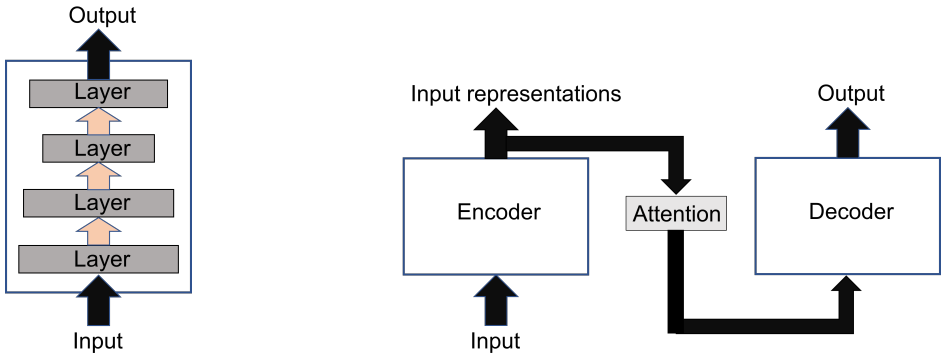
For this homework, we will work on the speech recognition problem. Unlike HW1P2 and HW3P2, where the task was to predict *phonemes* in the speech, in this homework, we will learn to directly spell the spoken sentence out in the English alphabet.

A key characteristic of sequence-to-sequence problems is that there may be no obvious correspondence between the input and the output sequences. For instance, in a dialog system, a user input "my screen is blank" may elicit the output "check the power switch". In a speech recognition system the input may be the (feature vector sequence for the) spoken recording for the phrase "know how". The system output must be the character sequence "k", "n", "o", "w", " ", "h", "o", "w" (note the blank space " " between know and how – the output is supposed to be readable, and must include blank spaces as characters). There is no obvious audio corresponding to the silent characters "k" and "w" in *know*, or the blank space character between *know* and *how*, nevertheless these characters must be output.

As a consequence of the absence of correspondence between the input and output sequences, the usual "vertical" architectures that we have used so far in our prior homeworks (Figure 1a), where the input is sequentially processed by layers of neurons until the output is finally computed, can no longer be used.

Instead, we must use a two-component model, comprising two *separate* network blocks, one to process the input, and another to compute the output. Such architectures are called *encoder-decoder* architectures. The encoder processes the input sequence to compute feature representations (embeddings) of the input sequence. The decoder subsequently uses the input representations computed by the encoder to *sequentially* compute the output *de novo*, *i.e.* from scratch (Figure 1b).

In this homework, we will build a **transformer architecture**, which is an example of encoder-decoder architectures. The transformer encoder takes an input sequence and produces feature representations, while the decoder uses these representations to generate the target sequence.



(a) Vertical architecture

(b) Encoder-decoder architecture

Figure 1: (a) Standard "vertical" architectures we have seen in previous homeworks. The input is sequentially passed through several layers until the output is computed from the final layer. *This kind of architecture is not useful for sequence-to-sequence problems with no input-output correspondence, such as the problem we deal with in this homework.* (b) General encoder-decoder architecture, comprising two separate modules, an encoder to process the input and a decoder to generate the output. The two are linked through an attention module.

Unlike the LSTM and RNN models we saw in HW3P2, which process sequences step-by-step, the transformer can process the entire sequence at once. This allows for much faster training by taking advantage of parallel computing. **But how does the transformer achieve parallel computation?** It does so through the attention mechanism, a technique that allows the model to focus on specific parts of the input when producing the output. This mechanism dynamically weighs the importance of different input elements to capture their relationships This features in two places:

i. ***Self-attention* – the attention that the encoder/decoder pay to themselves:** In computing their respective outputs, both the encoder (which derives feature representations from the input) and the decoder (which predicts outputs) must also pay attention to their own internal context, which in the case of the encoder would be adjacent inputs, and for the decoder would be previous outputs. This is *self-attention* – the attention a module pays to its own internal variables.

ii. ***Cross-attention* – the attention the decoder pays to the encoder:** Although the decoder's output may not have a direct correspondence with the input, each individual output symbol (character) nonetheless relates more to some parts of the input than others, *e.g.* the blank character " " in our above speech recognition example corresponds primarily to the audio at the boundary between the words "know" and "how" in the recording. When outputting the " ", the decoder must somehow know to "focus" on, or "pay attention" to this region of the input. In practice, this is implemented as an "attention" that the decoder pays to feature representations of the inputs derived by the encoder. Since this involves attention paid by one module, the decoder, to representations derived by another module (the encoder) this is referred to as *cross-attention.*

The attention mechanism allows the model to selectively focus on relevant parts of the input and ignore irrelevant information. This is particularly useful in tasks like machine translation, where the relationship between input and output elements is not always linear, and different parts of the input may be relevant for different parts of the output.

But how does the transformer know the order of the sequence? The transformer model understands the order of the sequence through a technique called **positional encoding**. Since the transformer processes the entire sequence at once and doesn't have any inherent sense of order, positional encoding is added to give the model information about the position of each element

in the sequence. Positional encoding is typically implemented by adding a vector to each input embedding, where the vector is generated based on the position of the element in the sequence. These vectors follow a specific pattern that helps the model distinguish between different positions. For more details refer to section 9.1

In this homework, you will learn how to build an encoder to effectively extract features from a speech signal, how to construct a Transformer-based decoder that can generate the transcription of the audio, and how to implement an attention mechanism between the Transformer decoder and the encoder.

Although the specifics of the homework will be targeted towards speech recognition, please note that the general framework (with appropriate modification of the encoder and decoder architectures) should also apply to other types of sequence-to-sequence problems, or even to image or video captioning (you would only need to modify the encoder to derive features from images or video respectively).

1.2 Problem specifics

In this problem of speech recognition you will learn how to convert a sequence of feature vectors of speech to a sequence of characters that spells out its transcription. To achieve this, you will implement a Speech Transformer model which is effective in capturing complex patterns in audio data using its self-attention, multi-head attention and cross-attention mechanisms. You will be implementing the two major parts of a transformer architecture which are the encoder and decoder.

- **Transformer Encoder:** In this task, you are responsible for building the EncoderLayer and Encoder classes, which together form the core of a Transformer encoder. The EncoderLayer class represents a single layer within the encoder, and it is designed to capture contextual relationships between tokens through a series of operations. Each encoder layer consists of a multi-head self-attention mechanism, followed by a feed-forward network (FFN) to capture non-linear transformations. Each step in the layer includes dropout regularization and a residual connection, allowing the model to maintain information from previous layers while applying normalization to stabilize training.

  The Encoder class is responsible for stacking multiple EncoderLayer instances and preparing input sequences with positional encoding and dropout. This class enables the encoder to generate high-dimensional representations of input sequences by passing them through each layer iteratively. In addition to the core stack of layers, the Encoder class includes a CTC (Connectionist Temporal Classification) head, which maps the output representations to the target vocabulary.

- **Transformer Decoder:** Similarly, in this task, you are responsible for building the DecoderLayer and Decoder classes. The DecoderLayer class combines self-attention, cross-attention (encoder-decoder attention), and a feed-forward network (FFN) to progressively refine its predictions. Each layer begins with masked multi-head self-attention, which operates solely over the decoder's input sequence and prevents the model from attending to future tokens, thus preserving the autoregressive nature of sequence generation. Cross-attention follows, attending to the encoder's output to incorporate context from the input sequence. The final component in each layer is the FFN, which captures complex interactions in the combined representations.

  The Decoder class applies positional encodings to the target embeddings, providing necessary information on token order. The sequence then passes through a stack of DecoderLayers, with each layer refining the representations via self-attention and cross-attention mechanisms. The Decoder concludes with a linear projection layer that maps the refined representations to the target vocabulary, enabling token prediction.

- **Attention**. Both the encoder and decoder are expected to employ self-attention to compute

their respective outputs. You will be expected to use masked multi-head attention and cross-attention between the encoder and decoder

- **Speech Transformer:** You will be expected to combine the encoder and decoder to build a speech-transformer capable of performing speech recognition task.
- **Training:** You will be training the transformer you built above using the training data provided. You will be using teacher forcing during training which involves always using the true previous output as input for the next step, ensuring direct learning. You will also be provided with three training strategies.
- **Inference:** In previous assignments your forward method will give the full sequence outputs of your inputs.However here, during inference, the decoder behaves as an auto-regressive network meaning the predicted sequence so far serves as input to predict the next token and this behavior is different from the forward during training. You will be given a greedy search algorithm for the decoding the decoder outputs and you can optionally implement the beam search algorithm.

In this homework, we will be implementing the speech transformer architecture based on Dong et al. [2018], Vaswani et al. [2017] (which we highly recommend that you read), and describe the key components of it later in Section 4. We will provide you with the code for some components, and you will implement the others yourself.

- **Components provided:**
  i. Position Encoding Module to add positional information to the input embeddings.
  ii. Tokenizer Class with different strategies
  iii. SpeechEmbedding Class for downsampling and embedding purpose
  iv. Speech Transformer Module that combines the encoder and decoder modules to create a full speech transformer architecture.
  v. We will also provide the training, validation and testing code.
- **Your task is to implement these components:**
  i. SpeechDataset
  ii. Masks
  iii. Encoder Layer Module to use in the encoder module. Here you will use the Multi-Head Attention and Feed Forward Modules that we provided, along with a LayerNorm from PyTorch.
  iv. Encoder Module that stacks multiple encoder layers. Here you will use positional encoding and two masking functions that were provided to you.
  v. Decoder Layer Module to use in the decoder module. Here you will use the Multi-Head Attention and Feed Forward Modules that we provided, along with a LayerNorm from PyTorch.
  vi. Decoder Module that stacks multiple decoder layers. Here you will use positional encoding and three masking functions that were provided to you.

Transformer models are difficult to train naively, and can result in suboptimal performance or even fail to converge, if improperly designed. It's crucial to carefully select and tune various hyperparameters to achieve optimal performance. However we assure you that there are many configurations where they will be possible to train well. To find one of these you must explore the following:

- **Number of attention heads**: In the multi-head attention mechanism it is a critical hyper-parameter, with a typical range being 2 to 10 heads. Increasing the number of heads allows

the model to capture different aspects of the input data in parallel, potentially improving its ability to learn complex patterns.

- **Number of layers**: The Number of layers in both the encoder and decoder parts of the transformer also significantly impacts its performance. A range of 2 to 8 layers is common, with more layers providing greater representational capacity at the cost of increased computational complexity and potential over-fitting.
- **Model dimensions**: Including the convolution dimension (128-256), the feedforward dimension (512-2048), and the encoder-decoder attention dimension (256-1024), are crucial for determining the size and capacity of the model. Larger dimensions can enable the model to capture more information but also increase the risk of over-fitting and the computational burden.
- **Optimizers**: The choice of optimizer is another important factor, with options such as Adam, AdamW, and SGD (Stochastic Gradient Descent) being popular. Each optimizer has its own characteristics and may perform differently depending on the specific task and data.
- **Learning rate schedulers**: like the cosine annealing learning rate and ReduceLR, help in adjusting the learning rate during training to improve convergence and prevent overshooting. The cosine annealing scheduler reduces the learning rate following a cosine curve, while ReduceLR decreases the learning rate when a metric has stopped improving.

By experimenting with these hyperparameters and understanding their impact on the transformer model, it's possible to find a configuration that leads to effective training and strong performance on the target task.

Believe it or not, although this may seem like a lot, you will manage to get all this done in the course of this homework. In the following sections, we will outline how to complete the work.

1.3 Resources

An important list of resources that will help you implement the transformer architecture in your homework:

i. The transformer lecture in this course by Akshat Gupta. **Don't miss this amazing lecture.**

ii. The Illustrated Transformer by Jay Alammar.

iii. Batool Haider did three episodes on transformers, providing in-depth explanations and practical insights into the architecture and its applications.

iv. PyTorch's Transformer Documentation.

v. TensorFlow's Transformer Model Tutorial.

vi. An amazing lecture about attention and awareness from neuroscience perspective by Nancy Kanwisher.

By leveraging these resources, you will gain a comprehensive understanding of transformer architectures and how to implement them effectively in your homework.

2 Notebook Walkthrough

This section serves to provide a brief overview of the different sections in the starter notebook.

i. **Configurations.** This section lays out the major hyperparameters inside the notebook. Ranges associated with some of the parameters are given to you to narrow down your experimental search space. Please get comfortable will the parameters listed and their significance to different aspects of the model. Key comments are added to aid you in your ablations.

ii. **LibriSpeech Dataset and Tokenizer** This section includes the Tokenizer, SpeechDataset, and TextDataset classes as well as the instantiation of the tokenizer results. Our Tokenizer provides multiple tokenization strategies, including character-level tokenization and pre-trained tokenizers (1k, 10k, 50k vocab sizes), which can be selected based on the configuration.

   The SpeechDataset class handles the loading and processing of audio featuresand the corresponding text transcriptions.

   The TextDataset class loads and tokenizes text transcripts from files, applies start-of-sequence and end-of-sequence tokens, and provides methods to retrieve and batch data. You will use this to pre-train a Transformer decoder as an LM before optionally using it along with the Encoder for the ASR task

iii. **Introduction.** This section provides a description of the Transformer architecture used in this homework. Additionally, this section defines some necessary utilities and modules that are extremely helpful for the implementation of the model. Please read through the code and comments to understand all the utilities and modules given. This section also provides three strategies you can used to train your transformer.

   NOTE: You don't have to change any of the given code; however, you must know what each module is doing to correctly implement the model architecture and answer the Canvas quiz.

iv. **Transformer Modules.** In this section, we provides some very important transformer modules which are listed as following:

   A. **PositionalEncoding** class.

   B. The **SpeechEmbedding** class with **BiLSTMEmbedding**, **Conv2DSubsamplin** modules.

   Positional encoding is a technique introduced in the "Attention Is All You Need" paper to allow transformers to retain information about the order of input tokens, as transformers lack inherent sequential understanding due to their non-recurrent architecture.

   The embedding integrates downsampling to make training more efficient and tractable, particularly when dealing with long sequence lengths. You will have to manage the tradeoff between training tractability and model performance by tweaking the amount of feature-wise and time-wise downsampling.

v. **The Transformer.** In this section, you will implement both the encoder and decoder class as well as their forward methods, and integrate these components with the provided Transformer class.

   Your specific tasks in this section include:

   A. implement the **initialization** and **forward** method of the **EncoderLayer** class.

   B. implement the **initialization** and **forward** method of the **Encoder** class.

   C. implement the **initialization** and **forward** method of the **DecoderLayer** class.

   D. implement the **initialization** and **forward** method of the **Decoder** class.

E. Finalize your transformer model with the components you implemented.

vi. **Experiments.** This is the final, experimental portion of the homework, where you can experiment with all kinds of pre-train strategies. This is more open-ended, and we encourage you to be creative!

3 Dataset

You will be working on a similar dataset as in HW1P2 and HW3P2. The training set will comprise input-output pairs, where each input consists of a sequence of 80-dimensional filter bank (FBank) spectral vectors derived from a spoken recording, and the corresponding output consists of the spelled-out text transcription. The transcriptions are in terms of 31 characters (the 26 characters in the English alphabet, blank space, "," [comma] and three special tokens: "<sos>," "<eos>," and "<pad>," representing the start and end of a sentence and padding respectively).

In this assignment, we use FBank (filter bank) features instead of MFCC (Mel-Frequency Cepstral Coefficients). FBank features are derived directly from the Mel-filtered spectrogram of audio, preserving more detailed spectral information by retaining all frequency components. In contrast, MFCC features are obtained by further applying a discrete cosine transform (DCT) to FBank features, which reduces dimensionality by keeping only the low-frequency components, discarding some of the high-frequency details. FBank features are often preferred in end-to-end speech recognition models because they retain more acoustic information however using MFCC's might allow you to explore more complex architectures by making training time more tractable. This decision is left up to you.

In addition, we use a separate unpaired text-only dataset for optional pre-training. You are given the `TextDataset` class, which loads and tokenizes text transcripts from files, applies start-of-sequence and end-of-sequence tokens, and provides methods to retrieve and batch data. You will use this to pre-train a Transformer decoder as an LM before optionally using it along with the Encoder for the ASR task

3.1 Dataloader

The `SpeechDataset` class and `Dataloader` for this homework are pre-designed and you will be required to complete it. The dataloader prepares the input for the speech transformer's decoder by shifting the transcriptions. Each decoder input sequence starts with a <SOS> token to signal the beginning and is used to process the subsequent characters. Conversely, the target sequence for comparison is shifted in the opposite direction to include the <EOS> token at the end, indicating the end of a sentence. This setup represents a full teacher forcing strategy, where the actual target sequences are always provided to the decoder during training. The decoder constructs the autoregressive matrix using an attention mechanism and a triangular mask to prevent future characters from influencing the prediction of the current character. The dataloaders will supply you with the shifted target sequence, e.g., "<SOS> H E L L O," and the corresponding golden target, e.g., "H E L L O <EOS>."

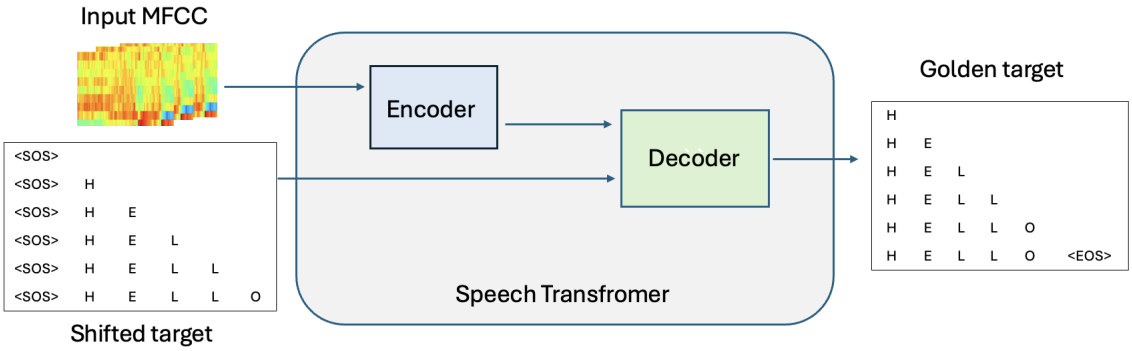


Figure 2: The input Mel-Frequency Cepstral Coefficients (MFCC) are fed into the encoder. The shifted target sequence, beginning with a start-of-sentence token (<SOS>) and excluding the end-of-sentence token (<EOS>), is used by the decoder to predict the subsequent token in the sequence. The golden target sequence is the true next token the model aims to predict at each timestep, including the <EOS>token but excluding the initial <SOS>.

4 Speech Transformer

In this homework, you will build a Transformer model Vaswani et al. [2017], Dong et al. [2018] for speech recognition. Unlike the original application of the Transformer for machine translation, we will adapt the architecture to transcribe spoken language into text. The Transformer model will be divided into three main components:

By the end of this homework, you will have implemented a complete Transformer model tailored for the task of speech recognition, gaining a deeper understanding of this powerful architecture and its applications in natural language processing. Specifically, we will be implementing the Speech-Transformer in Figure 3

We acknowledge that a transformer architecture consists of multiple components, which might be confusing. However, the subsequent sections will guide you through this transformer, step by step. We recommend familiarizing yourself with the architecture before beginning to code, as this understanding is crucial for working with the starter notebook without encountering problems.

The above architecture can be divided into two main parts. The Encoder and the Decoder.

4.1 Encoder

The encoder in a speech recognition Transformer model processes the input speech features, such as Mel-Frequency Cepstral Coefficients (MFCCs), and converts them into a sequence of embeddings that capture the contextual information of the speech. These embeddings capture meaningful information from the speech, which the decoder then uses to predict characters in the English alphabet. The encoder consists of convolutional layers, positional encoding, and multiple encoder layers each comprising a self-attention mechanism and feed-forward networks. More specifically, this section walks through the portion of the architecture as shown in Figure 4

4.1.1 Encoder Layers

Congratulations, now you are ready to extract features and add positional encoding for a transformer encoder. Your inputs are ready and we can now dive into the encoder layers which are the main components of the transformer encoder. As shown in Figure 5, this part of the architecture - the encoder consists of multiple EncoderLayer modules stacked together. Each EncoderLayer includes the following components:

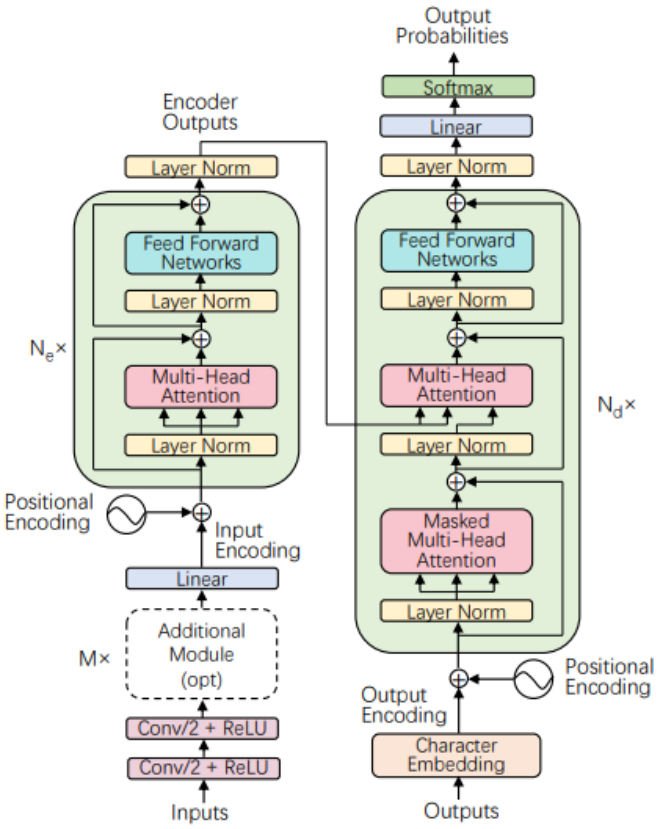


Figure 3: Baseline Model: Speech Transformer

i. **Positional Encoding:** Since transformers lack a built-in mechanism for sequence ordering, we use positional encoding to add information about token positions in the sequence. This helps the model understand the order and temporal relationships within speech data.

ii. **Layer Normalization:** Layer normalization is applied at multiple points within the EncoderLayer to stabilize training and normalize feature dimensions. This ensures more stable gradients and better generalization.

iii. **Multi-Head Attention:** This mechanism allows each token to attend to every other token in the input sequence, enabling the model to capture relationships across the sequence. Masks (padding and attention masks) are applied to ignore padding tokens and ensure correct attention weight calculations. Please refer to section D for more information.

iv. **Feed-Forward Network (FFNs):** serve as a critical component following the multi-head attention mechanism. Each FFN consists of two linear transformations with a nonlinear activation function in between. This structure allows the model to introduce additional complexity and nonlinearity, enabling the encoding of more abstract representations of the speech data. The FFNs operate on each position separately and identically, ensuring that the model can enhance its understanding of the speech input by applying the same transformation across all positions, thereby enriching the feature space before passing it onto the next layer.

Now that we understand the various parts of the encoder layers let us see how to put it together. In the starter notebook, we have created a module called the EncoderLayer which you are to complete.

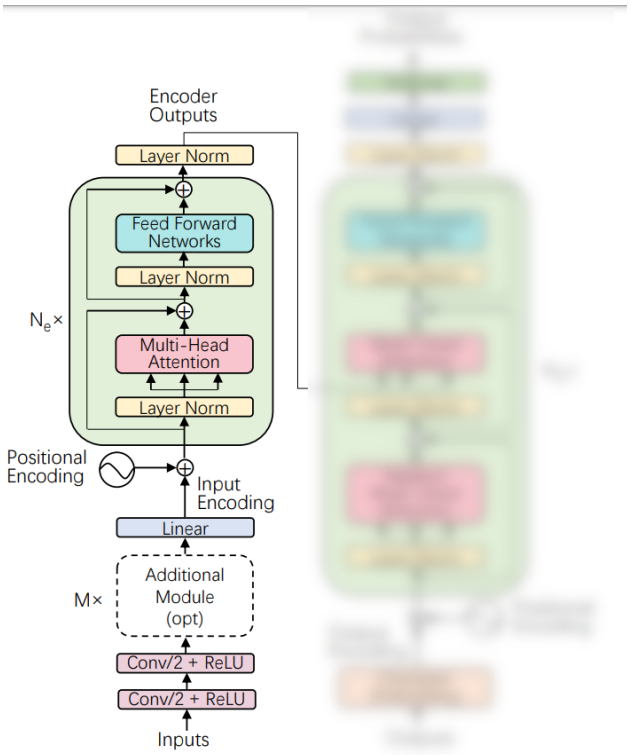


Figure 4: Encoder part of Transformer

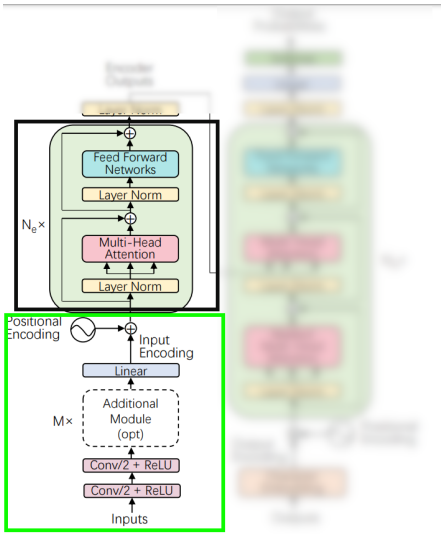


Figure 5: Encoder Layers

i. **Module Initialization:** First, you need to initialize all the modules required for the encoder layer. This includes:

   A. **Layer Normalization:** You will create two layer normalization modules (norm1 and norm2). These modules are applied before and after key components to stabilize the training process.

   B. **Multi-Head Self-Attention Module:** This module allows tokens to attend to others

in the sequence, enabling the model to capture dependencies across the sequence.

C. **Feed-Forward Neural Network (FFN):** The FFN contains two fully connected layers with a GELU activation in between, followed by dropout for regularization. It introduces non-linearity and increases the model's expressiveness.

D. **LDropout Module:** Dropout is applied after the self-attention and feed-forward network to prevent overfitting.

Refer to the provided module definitions in the notebook to see how to initialize them. Ensure that the parameters required for these modules are correctly passed to the `__init__` method of your EncoderLayer module.

ii. **Forward Method** In the forward method, you will implement the flow of data through the EncoderLayer step by step:

A. **Layer Norm 1:** As mentioned above normalizing our inputs to a module is very important. In the forward method of the encoder layer, we receive the input encoding which is the combination of the extracted features and positional encoding as described in the previous sections. This input encoding passes through the first layer normalization module.

B. **Multi-Head Self-Attention Module:** Next, we need to pass the output of our first layer norm to the multi-head self-attention module. In this section, make a minor modification by using a mask with the multi-head attention module. The reason for this is, that our inputs are padded and to be able to attend well, we need to mask the padding we added to the input. Hence we also receive the pad mask and the self-attention mask as parameters to the forward method of the EncoderLayer. The padding mask is set to one for all tokens except for the pad token, which is set to zero. The self-attention mask is also a padding mask but for the attention weights. However, in this case, it is the opposite: the positions of non-pad tokens are set to zero, and the pad token is set to one. This is because, during the attention computation, we will fill the positions with ones (representing pad tokens) with infinity to ensure that the attention scores for padding tokens are zero. As a result, padding tokens do not contribute to the encoder representation. After the layer norm, we pad the outputs and use these as parameters for our multi-head attention module forward method which then produces the attention scores.

C. **Residual Connection 1:** As indicated in the architecture, the output of the multi-head attention is combined with the input encoding. These connections help in alleviating the vanishing gradient problem, enabling deeper models to be trained more effectively.

D. **Layer Norm 2**: Similar to the first layer norm, the input and the multi-head attention output combined are passed through the second layer normalization.

E. **Feed Forward Networks:** The results from layer norm 2 is passed through the feed-forward network. Its benefits are as described above.

F. **Residual Connection 2:** Finally, the output of the feed-forward network is combined with the residual connection 1 result which was the combination of the inputs and the attention output.

This concludes the encoder layer. Refer to the provided module definitions in the notebook to see how to initialize them. Ensure that the parameters required for these modules are correctly passed to the `__init__` method of your EncoderLayer module. Take a moment to reflect on the process through which the input encoding passes through. Does it make sense? If not please visit OH or ask a question of Piazza to get clarification.

4.1.2 Transformer Encoder

Now that we understand all the various building blocks for the transformer encoder, let us put them together to form our Speech-Transformer Encoder.

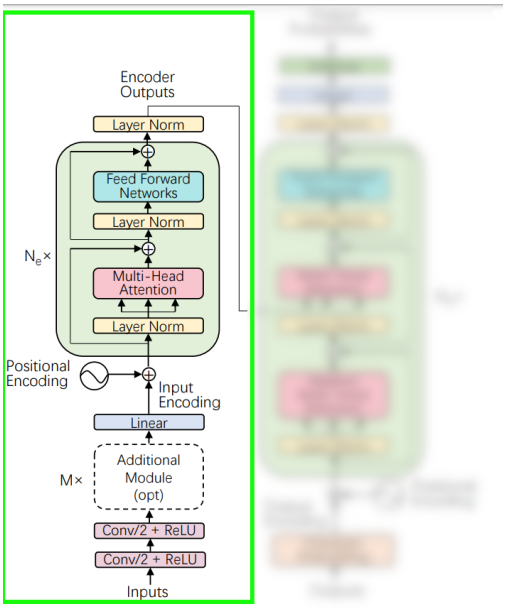


Figure 6: Speech-Transformer Encoder

The encoder part which we will now put together is shown in Figure 6. To complete this module, we divide the process into two main steps:

i. **Module Initialization:** First, you need to initialize all the modules required for the encoder layer. This includes:

   A. **Positional Encoding:** Since transformers do not inherently understand sequence order, the positional encoding module provides the necessary sequence information. Review the provided implementation and consider parameters such as sequence length and embedding dimensions.

   B. **Encoder Layers:** Multiple EncoderLayer modules will be stacked to refine the input representations. While the initialization is pre-defined for you, understanding this process is crucial, especially if you later experiment with varying the number of layers or freezing/unfreezing layers during pre-training. Consider how increasing or decreasing the number of layers impacts model performance.

   C. **CTC Head (Optional):** The encoder includes an optional linear layer (CTC head) for auxiliary training with the CTC objective. This component can improve alignment during training.

   D. **Layer Normalization:** A final layer normalization is applied after the encoder layers. This step ensures the output is well-normalized before being passed to the decoder.

ii. **Forward Method**

   A. **Positional Encoding**: We create a positional encoding of the input using the Positional encoding module we initialized earlier. You can also apply dropout after positional encoding for regularization.

   B. **PAD Mask:** We have provided a function to help create a padding mask. Check the utilities section of the notebook and also ask yourself why are we creating this. How does

padding impact sequences of variable lengths during training? Consider scenarios where your input sizes vary greatly.

C. **Attention Mask**: We have provided a function to create the attention mask, please have a look at it and ask yourself why we created an attention mask. Remember we are dealing with Multi-Head self-attention. How does attention masking help in ignoring irrelevant tokens during multi-head attention, particularly when training on variable-length sequences?

D. **Encoder Layers:** We initialized our encoder layers earlier. If we have more than one encoder layer, the output of the first layer serves as input to the second layer.

E. **Layer normalization:** The output of the final encoder layer is passed through a layer norm layer to prepare it for our decoder.

Congratulations! You have successfully built a transformer encoder. Take a moment to reflect through the steps, what happens in each module, and how the modules are connected. Feel free to ask any questions for more clarification if there is any part you don't understand.

4.2 Decoder

The decoder in a speech recognition Transformer model takes as input the embeddings generated by the encoder and decodes them into sequences of characters representing the transcribed text. It includes positional encoding, multiple decoder layers each comprising self-attention and cross-attention mechanisms, and feed-forward networks. During training, the decoder predicts the next token in the target sequence based on the previously generated tokens. This is achieved through self-attention and cross-attention mechanisms, where the decoder attends not only to the encoder outputs but also to its own previously generated outputs.

In this section we will be exploring the different components in the decoder part of the architecture as shown in Figure 7

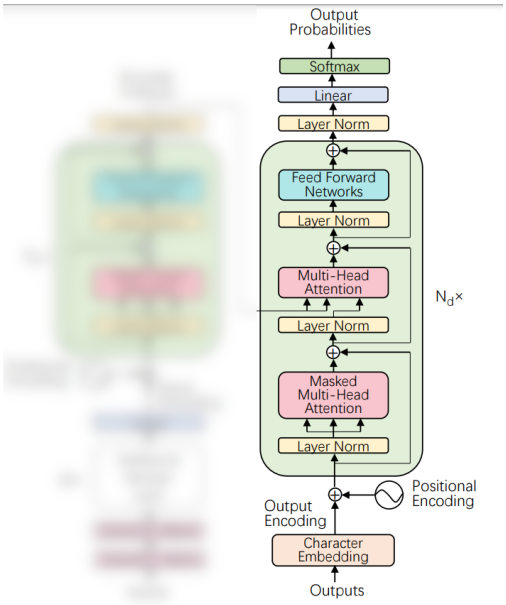


Figure 7: Decoder Part of Speech Transformer

Similar to section 4.1 on the encoder, we will go through the components that make up the decoder one after the other. Let's get started

4.2.1 Decoder inputs

The decoder learns to decode the inputs to some target output. During training, the decoder accepts inputs from the encoder as well as a shifted target (indicated as outputs in the decoder side of the architecture as in Figure 7). During inference, outputs are the generated tokens from the decoder itself. For more on shifted targets, please refer to Section 3. Consider how shifted targets ensure the autoregressive property of the decoder during training.

4.2.2 Output Encoding

Similar to the encoder encoding, we need to encode one of the inputs to the decoder. We do this by first encoding the targets using character embedding. Since this representation is already standard and does not need to be learned, we use an already existing neural network. The character embedding is combined with the positional encoding to produce the output encoding which is one of the inputs to the decoder layers.

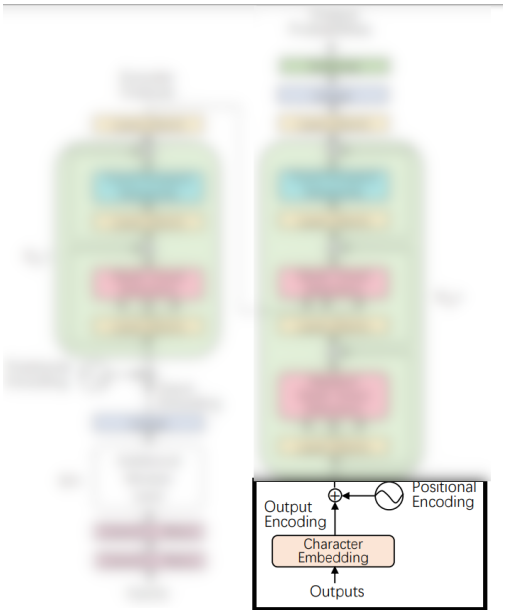


Figure 8: Decoder Output Embedding

4.2.3 Decoder Layers

The Transformer Decoder is designed to take into account the output of the Transformer Encoder and generate the target sequence. It also contains self-attention layers, but with a slight modification to prevent positions from attending to subsequent positions. This masking ensures that the predictions for position i can only depend on the known outputs at positions less than i. We discuss this in more details in 9.3

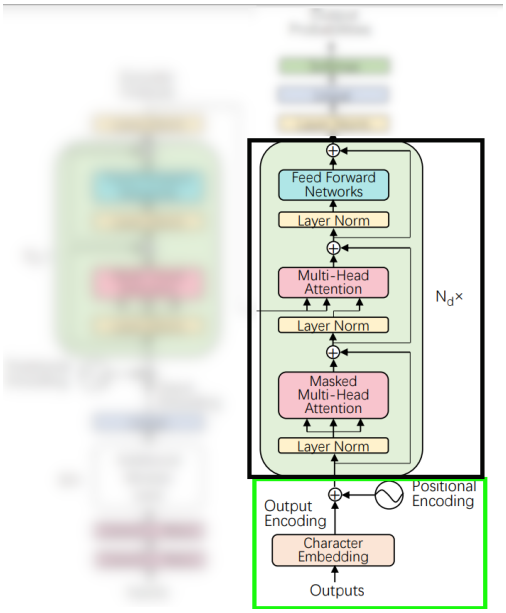


Figure 9: Decoder Layer

There are four main modules in a single decoder layer;

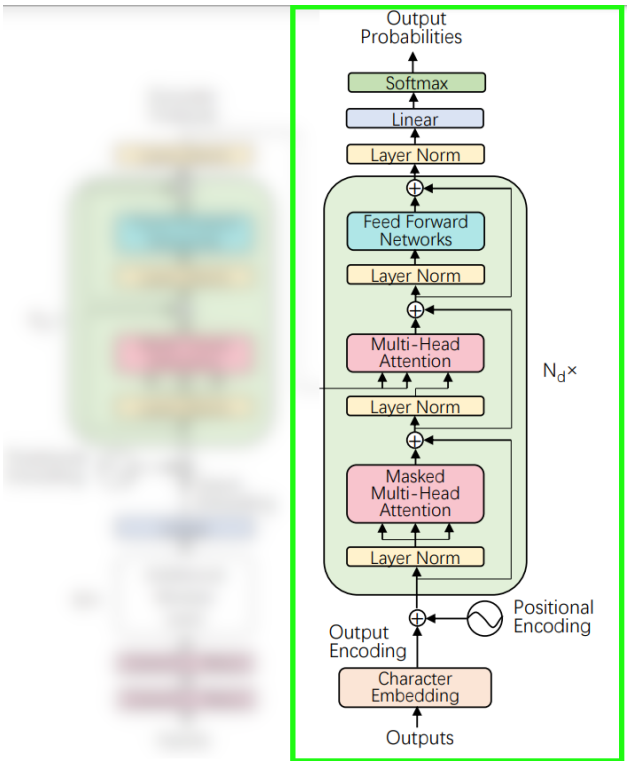


Figure 10: Transformer-Decoder

i. **Layer Normalization:** Same as explained in the Encoder

ii. **Masked Multi-Head Self-Attention:**

In the decoder part, the masked multi-head self-attention module uses a lookahead mask, also known as a causal mask, in addition to the padding mask, unlike the encoder which was using attention mask. This lookahead mask is crucial for ensuring that the decoder can only attend to earlier positions in the output sequence, preventing it from seeing future tokens during training and inference. This is essential for maintaining the autoregressive property of the decoder, where each token is predicted based on the preceding tokens. How does this lookahead mask influence the decoding process compared to encoder self-attention?

iii. **Feed-Forward Network (FFNs):** Same as explained in the encoder section.

iv. **Multi-Head Self-Attention (Cross-Attention):** Cross-attention plays a crucial role in this process by allowing each step of the decoder to attend to different parts of the encoder's output. This is achieved by calculating attention scores that determine how much focus should be placed on each part of the encoder output when generating the current element of the decoder output. For more information please check Section 9

4.2.4 Transformer Decoder

Now that you have all your modules ready. To build the decoder part, we follow the same approach as with the encoder. Initializing all modules required and connecting them as shown in the figure 10

If your have problems with the connection please check how it is done in the encoder section. However, note that the outputs of the outputs of the decoder layer passes through another layer normalization followed by a linear layer and finally a Softmax to output the probabilities. How do we convert these probabilities to actual tokens ? please check the section 5.2 on inference.

4.3 Combining it All

Now we have all parts of our transformer implemented and working. We need to combine the encoder and decoder. We have provided a module called Speech Transformer which does this combination. It is as easy as initializing the encoder and decoder module, writing a forward function which receives inputs and target output, passes the inputs to the encoder layer and get the encoder outputs, use these encoder outputs in the decoder.

5 Training

Training an encoder-decoder transformer model for speech tasks involves several key steps. The model leverages self-attention mechanisms without using recurrent neural networks, capturing dependencies in the sequential input.

During the forward pass, the encoder processes the input sequence, producing a set of representations. These representations are passed to the decoder, which takes both the encoder's output and the target sequence (shifted right) to generate the output sequence. The decoder utilizes:

- **Self-Attention**: to understand relationships within the target sequence, and
- **Cross-Attention**: to focus on relevant parts of the input sequence representations.

5.1 Training Strategies

We incorporate three distinct training strategies, each designed to enhance the model's learning efficiency and effectiveness. Depending on the approach, you may dynamically adjust learning rates, schedulers, and training epochs.

- **Pretrain Approach 1: Decoder Language Model (LM)**:
  - Train the decoder as a language model (LM) on the Librispeech 960 text-only dataset for 20-30 epochs. Once pre-training is complete, unfreeze the entire model and proceed to train the full Transformer.
- **Pretrain Approach 2: Decoder Conditional LM with Speech Embeddings**:
  - Train the decoder as a conditional language model, using simple embeddings from Librispeech 100 for 20-30 epochs. Then, freeze the embeddings and decoder and train the encoder for 3-5 epochs. Finally, unfreeze everything and train the full Transformer.
- **Full Transformer Training**:
  - Directly train the entire model without any pre-training. Alternatively, after completing either of the above pre-training strategies, attempt freezing certain components (e.g., embeddings, decoder) and train with the unfrozen encoder for 3-5 epochs before unfreezing everything.

5.1.1 Loss Function

Several loss functions are commonly used depending on the model's configuration and the task's requirements:

- **Cross-Entropy Loss (CE)**: Cross-entropy loss, widely used in sequence prediction tasks, calculates the difference between the predicted output sequence and the ground-truth sequence on a per-token basis. For a sequence of length T with ground-truth tokens y_t and predicted probabilities $\hat{y}_t$, the cross-entropy loss is given by:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{T}\sum_{t=1}^{T} \log \hat{y}_t(y_t)$$

  This loss is particularly effective in well-aligned, token-by-token tasks.

- **Connectionist Temporal Classification (CTC) Loss**: CTC is suitable for tasks like speech recognition where exact alignment between input and output sequences is not provided. It works by summing the probabilities of all valid alignments between the input sequence and target sequence. Given $\mathcal{A}(X, Y)$ as the set of all valid alignments for input X and output Y, the CTC loss is:

$$\mathcal{L}_{\text{CTC}} = -\log \sum_{a \in \mathcal{A}(X,Y)} p(a|X)$$

- **Negative Log Likelihood (NLL) Loss**: Often used with log probabilities, NLL loss maximizes the likelihood of correct outputs. For a predicted distribution $\hat{y}$ over possible labels y_t for each timestep t:

$$\mathcal{L}_{\text{NLL}} = -\sum_{t=1}^{T} \log \hat{y}_t(y_t)$$

In some configurations, combining CTC loss with Cross-Entropy or NLL Loss (often called a hybrid loss) can enhance performance, especially in tasks with varying alignment challenges.

Starter Notebook The starter notebook provides an adaptation for both CTC and Cross-Entropy losses. We encourage students to experiment further by implementing Negative Log Likelihood (NLL) loss or creating hybrid configurations to explore the effects of each on model performance.

5.1.2 Overall Training Procedure

When combining losses, ensure dynamic adjustment of learning rates, schedulers, and training epochs based on the chosen strategy. The overall procedure is outlined as follows, with support for any single loss or a combination of both.

```
# X is the input sequence, T_shifted is the ground-truth output sequence shifted right,
    T_golden is the ground-truth output sequence.
# model is the transformer model (encoder-decoder)
function gradient = ComputeGradient(X, T_shifted, T_golden, model, loss_type)
    # Forward pass through encoder and decoder
    encoded = model.encoder(X)
    decoded = model.decoder(encoded, T_shifted)

    # Compute loss based on specified loss type
    if loss_type == "CTC":
        L = CTCLoss(decoded, T_golden)
    elif loss_type == "CrossEntropy":
        L = CrossEntropyLoss(decoded, T_golden)
    elif loss_type == "NLL":
        L = NLLLoss(decoded, T_golden)
    elif loss_type == "Hybrid":
        L_ctc = CTCLoss(decoded, T_golden)
        L_ce = CrossEntropyLoss(decoded, T_golden)
        L = 0.5 * L_ctc + 0.5 * L_ce # Example of combining CTC and CE losses

    # Backpropagate the computed loss
    gradient = BackPropagate(L)
end
```

Here, `model.encoder` and `model.decoder` represent the forward passes through the encoder and decoder. The `loss_type` argument determines the type of loss function, allowing for flexibility across different tasks and training configurations. `BackPropagate` performs gradient backpropagation to update model parameters.

This setup accommodates various loss functions, optimizing for either alignment flexibility (CTC) or strict sequence matching (Cross-Entropy, NLL), and enables flexibility in selecting the most appropriate training strategy.

5.2 Inference

Inference in the Speech Transformer model is a sequential process, particularly during the generation of output sequences. Initially, the encoder encodes the entire input audio sequence into a set of hidden representations using self-attention.

During the decoding phase, the model generates the transcription sequence (sequence of characters) one element at a time in an autoregressive manner. This means that the generation of each element in the output sequence is conditioned on the previously generated elements. The decoder uses the encoded representations and the partially generated output sequence to predict the next element. This process continues iteratively until a predefined end-of-sequence (¡EOS¿) token is produced or a maximum output length is reached.

The autoregressive nature of the inference process ensures that the model takes into account the context provided by the preceding elements in the output sequence, which is crucial for maintaining coherence and relevance in the generated sequence.

So the entire process can be encapsulated by the following pseudocode:

Listing 1 Inference

```
# X is the input audio feature vector sequence
# T is the <SOS> token

function O = Inference(X)
    E = Encoder(X)
    T = [<SOS>]
    for i in range(maximum_sequence_length):
        NT = Decoder(E, T)
        T += NT
        if NT == <EOS>:
            stop
end
```

5.2.1 Greedy Search

The naive approach to decoding is **Greedy Search**, the simplest method for inference. In Greedy Search, we select the character with the highest probability at each time step from the model's output distribution for the entire batch of predictions. The pseudocode is as follows:

Listing 2 Greedy Decoding

```
# X is the input audio feature vector sequence
# T is the <SOS> token

function O = Inference(X)
    E = Encoder(X)
    T = [<SOS>]
    for i in range(maximum_sequence_length):
        NT = Decoder(E, T)
        NT = argmax(NT)
        T += NT
        if NT == <EOS>:
            break
end
```

While Greedy Search is straightforward and often produces reasonable results, it does not guarantee

the most probable output sequence due to the model's autoregressive nature. To obtain a more likely final output, alternative methods like **Beam Search** (discussed below) can provide improved decoding by exploring multiple possible paths.

5.2.2 Beam Search

Beam Search is a more advanced decoding technique that considers multiple hypotheses at each time step, aiming to find the most likely sequence overall rather than selecting the highest-probability character step-by-step as in Greedy Search. In Beam Search, we maintain a fixed number of candidate sequences (the "beam width") and keep only the top k sequences with the highest cumulative probabilities. This method significantly improves accuracy for sequence generation tasks by balancing exploration with exploitation.

Listing 3 Beam Decoding

```
# X is the input audio feature vector sequence
# T is the <SOS> token
# k is the beam width

function O = BeamInference(X, k)
    E = Encoder(X)
    beams = [(T=[<SOS>], score=0)]

    for i in range(maximum_sequence_length):
        all_candidates = []

        for beam in beams:
            NT_candidates = Decoder(E, beam.T)
            for j in range(len(NT_candidates)):
                new_T = beam.T + NT_candidates[j]
                new_score = beam.score + log_probability(NT_candidates[j])
                all_candidates.append((new_T, new_score))

        # Sort all candidates by score and keep the top k
        beams = sorted(all_candidates, key=lambda x: x[1], reverse=True)[:k]

        # Stop if all candidates have <EOS> as last token
        if all([candidate[0][-1] == <EOS> for candidate in beams]):
            break

    # Return the sequence with the highest score
    return beams[0][0]
```

Advantages of Beam Search Beam Search offers a more balanced approach, allowing exploration of multiple paths and improving the chances of finding a more probable sequence. However, it is more computationally expensive than Greedy Search, with performance depending on the choice of beam width k. Higher values of k typically yield better results but require more computation.

Comparison of Decoding Methods While Greedy Search is efficient and straightforward, it may miss the optimal sequence. Beam Search, by retaining multiple hypotheses, is more effective in achieving higher accuracy for tasks where sequence coherence is crucial. For best performance, experiment with beam width and explore trade-offs between computational cost and decoding quality.

6 Evaluation

In evaluating the performance of our speech transformer model, we use metrics designed to measure transcription accuracy. The primary metrics include Word Error Rate (WER), Character Error Rate (CER), and Levenshtein Distance (LD). **In the Kaggle competition, CER will be used as the main metric for evaluation across the entire test set. You will be reporting the percentage value of the CER ratio which must be equal to or below 60% to qualify for the early submission.**

6.0.1 Word Error Rate (WER)

Word Error Rate (WER) is a common metric for evaluating the accuracy of speech-to-text systems. It measures the average number of word-level errors per reference transcription. Given a reference transcription R and a hypothesis transcription H, WER is calculated as:

$$\text{WER} = \frac{S + D + I}{N}$$

where:

- S: the number of substitutions,
- D: the number of deletions,
- I: the number of insertions, and
- N: the total number of words in the reference transcription R.

6.0.2 Character Error Rate (CER)

Character Error Rate (CER) is similar to WER but operates at the character level, making it suitable for languages with small word boundaries or shorter transcriptions. CER is computed as:

$$\text{CER} = \frac{S + D + I}{N}$$

where:

- S: the number of character substitutions,
- D: the number of character deletions,
- I: the number of character insertions, and
- N: the total number of characters in the reference transcription.

6.0.3 Levenshtein Distance (LD)

Levenshtein Distance (LD) measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one sequence into another. For a reference sequence R and a hypothesis sequence H, the Levenshtein Distance $\text{LD}(R, H)$ can be computed via dynamic programming. The relationship between CER and LD for a single transcription pair is given by:

$$\text{CER} = \frac{\text{LD}(R, H)}{N}$$

where N is the length of the reference sequence R.

This evaluation strategy ensures a detailed analysis of model performance at both the word and character levels, with CER as the primary focus for final model selection.

7 Common Errors and Fixes

During the implementation of the Transformer model for speech recognition, students may encounter several common issues. Here are some guidelines to help address these problems:

7.1 Managing Tradeoffs

This assignment centers on balancing the trade-offs between training efficiency and model performance. Training a Transformer can demand substantial resources, so you'll work on managing this through careful feature selection, tokenization, pre-training methods, and embedding choices. This section outlines common issues you might encounter and provides a prioritized list of strategies to address them. In summary, the goal is to balance training time, memory usage, and model accuracy by tuning feature selection and downsampling.

Memory Optimization Steps

If you encounter **Out-of-Memory (OOM)** errors, try the following adjustments in order:

1. **Increase time-downsampling**: Consider values between 2 and 4.
2. **Increase feature-downsampling**: Adjust similarly within the range of 2 to 4.
3. **Reduce the number of features**:
   - For filter banks, set a minimum of 20.
   - For MFCCs, set a minimum of 12.

Performance Optimization Steps

If you aren't experiencing OOM errors but training is slow, try the following adjustments:

- **If convergence is slow:**
  a. Decrease time-downsampling if applied.
  b. Decrease feature-downsampling if applied.
  c. Increase the number of features:
     - Cap at 20 for MFCCs.
     - Cap at 80 for filter banks.

7.2 Gradient Clipping

Gradient clipping is a technique used to prevent the exploding gradient problem in neural networks. It is recommended to implement gradient clipping with a norm of 1 or 2. This will add a ceiling to your gradient, ensuring that it does not exceed the maximum norm value. To clarify, gradient clipping is optional but highly recommended for stabilizing training.

7.3 Debugging NaN Loss

Encountering NaN (Not a Number) loss during training can be a frustrating issue. Here are some steps to debug and fix this problem:

(a) **Check Learning Rate:** A high learning rate can cause the model to diverge, leading to NaN values. Try reducing the learning rate and observe if the issue persists.

(b) **Inspect Weight Initialization:** Improper weight initialization can lead to NaN loss. Ensure that the weights are initialized using a method appropriate for the activation functions in your network (e.g., Xavier initialization for sigmoid activations).

(c) **Monitor Gradients:** Check if the gradients are exploding, which can lead to NaN values. You can print the gradients or use a tool like TensorBoard to visualize them. If you notice large gradients, consider implementing gradient clipping to prevent them from exceeding a certain threshold.

(d) **Use a Different Optimizer:** Sometimes, switching to a different optimizer can help stabilize training. For example, if you're using SGD, try using Adam or RMSprop and see if the issue is resolved.

(e) **Add Small Constants:** In operations that involve division or logarithms, adding a small constant (e.g., 1e-8) to the denominator or argument can prevent division by zero or taking the logarithm of zero, which can lead to NaN values.

(f) **Check for Inf Values:** Ensure that there are no infinite values in your data or intermediate computations. Infinite values can propagate through the network and result in NaN loss.

(g) **Reduce Model Complexity:** A very complex model might be more prone to numerical instability. Try simplifying your model architecture and see if the issue persists.

By systematically addressing each of these potential causes, you can identify and fix the source of NaN loss in your training process.

7.4 Out Of Memory (OOM)

Out of Memory errors occur when your model requires more memory than is available on your GPU. To resolve this, you can reduce the batch size, use gradient accumulation, or simplify your model architecture.

7.5 Dataloader and Dataset

For training, use 100 hours of data . Ensure that the dataloader is correctly configured to provide the expected values, including the correct input feature dimensions and batch size.

7.6 Concatenating the Context Vector with the Embeddings

To concatenate the context vector with the embeddings, you can use a function like `torch.cat()` in PyTorch. Ensure that the dimensions of the context vector and embeddings align correctly for concatenation.

7.7 Training Duration and Monitoring

The training time for one epoch of a Transformer model can vary based on factors such as the size of the model, the size of the dataset, and the hardware used for training. On Google Colab, using a standard GPU, you can expect an approximate training time of 1-2 hours per epoch for a moderate-sized Transformer model on a medium-sized speech dataset. However, this is just an estimation, and actual training times may vary.

Additionally, it is important to save checkpoints regularly during training. This practice helps to avoid losing progress in case of interruptions and allows for resuming training from the last saved state. Implementing a checkpointing mechanism is especially important when training on platforms like Google Colab, where sessions can be terminated after a certain period of inactivity.

Debugging common issues involves carefully examining the loss curves, checking the gradients for any abnormalities, and ensuring that the data pipeline is functioning correctly. If the model is not

converging, consider adjusting the learning rate, experimenting with different optimization algorithms, or revising the model architecture.

8 Conclusion

This homework is a true litmus test for your understanding of concepts in Deep Learning. It is not easy to implement, and the competition is going to be tough. But we guarantee you, that if you manage to complete this homework by yourself, you can confidently claim that you are now a **Deep Learning Specialist**! Good luck and enjoy the challenge!

9 An important note on Attention

Since the *key* feature of this homework is the *attention* mechanism, it is useful to first explain it briefly. You will need this information for your homework. You have already encountered attention in Lecture 18. We assume that you have an idea of attention, and the concepts behind it. If not, we recommend that you review lectures 18 and 19 and recitation 10. Assuming this prior knowledge, we quickly recap the key concepts of attention that apply to this homework problem before we continue.

9.1 An attention head

The encoder in an encoder-decoder model takes in an input sequence of vectors $X_0, \cdots, X_{N-1}$ and produces a sequence of *embeddings*, $E_0, \cdots, E_{M-1}$. The length of the embedding sequence M need not be the same as that of the input sequence, N, but each embedding vector E_i effectively represents a section of the input. Assuming all the embeddings to be *row* vectors (in the standard python format), the set of embedding vectors can be vertically stacked into a matrix $\mathbf{E}$.

In the typical implementation of attention, from the embeddings E_i the encoder computes two terms, a *key* K_i and a *value* V_i. Typically these are computed through a linear transform. Stacking all the key vectors (assumed to be row vectors) vertically into a matrix $\mathbf{K}$, and the value vectors into a matrix $\mathbf{V}$, these can be computed as $\mathbf{V} = \mathbf{E}\mathbf{W}_V$ and $\mathbf{K} = \mathbf{E}\mathbf{W}_K$, where $\mathbf{W}_V$ and $\mathbf{W}_K$ are learnable parameters.

In the decoder, corresponding to each ouput O_k, we first compute a *query* vector $\mathbf{q}_k$, which is used to "query" the encoder outputs to determine which portion of the input to pay most attention to, to generate O_k. When the decoder is a recurrent network, the query is typically computed as a linear transform of the recurrent state $\mathbf{s}_{k-1}$, and optionally also O_{k-1}: $\mathbf{q}_k = \mathbf{s}_{k-1}\mathbf{W}_Q$, or $\mathbf{q}_k = [\mathbf{s}_{k-1}, O_{k-1}]\mathbf{W}_Q$ where $\mathbf{W}_Q$ is a learnable parameter, O_{k-1} is the (embedding or one-hot representation of the) $(k-1)^{\text{th}}$ output, and $[\mathbf{s}_{k-1}, O_{k-1}]$ represents the concatenation of $\mathbf{s}_{k-1}$ and O_{k-1}.

In order to produce the k^{th} output symbol O_k, the decoder computes a *context* C_k from the encoder-derived value vectors: $C_k = \mathbf{w}_k\mathbf{V}$. $\mathbf{w}_k$ is a vector of *attention weights* for the k^{th} output, computed as $\mathbf{w}_k = softmax(\mathbf{q}_k\mathbf{K}^\top)$, and has the property that (a) it has as many components as the number of vectors in $\mathbf{E}$, (b) all of its components are non-negative, and (c) they sum to 1.0. Ideally, the components of $\mathbf{w}_k$ corresponding to Value vectors representing the region of the input most relevant to O_k will be high, and the rest will be low. The context C_k is input to the decoder (along with any other recurrent state values and previous outputs that your architecture considers) to generate O_k.

9.2 Multi-head attention

The attention computed using the above method (Section 9.1) computes a *single* context vector C_k to compute output O_k. C_k derives and focuses on a single specific aspect of the input that is most relevant to O_k. It is reasonable to assume that there are, in fact, *multiple* separate aspects of the input that must all be considered to compute O_k.

Multi-head attention works on this principle to derive multiple context vectors $C_k^1, C_k^2, \cdots, C_K^L$. Each context vector C_k^l is computed using a separate attention module of the form explained above (Section 9.1), with its own learnable parameters $\mathbf{W}_K^l, \mathbf{W}_V^l, and \mathbf{W}_Q^l$. The complete module that computes an individual context vector C_k^l is called an *attention head*, thus the overall attention framework that computes multiple context vectors is referred to as *multi-head* attention.

The final context vector C_k that is used to compute O_k is obtained by concatenating the individual context vectors: $C_k = [C_k^1, C_k^2, \cdots, C_k^L]$ (where L is the number of attention heads).

In this homework you will also investigate the relative advantages of multi-head attention over simple attention and the benefits to be obtained through increasing the number of attention heads.

9.3 Masking

In transformers, the concept of masking plays a crucial role in controlling the flow of information, particularly for sequence processing. There are three primary types of masks utilized:

(a) **Causal Mask:** The causal mask is utilized within the decoder to ensure that the prediction for a given token can only be influenced by the tokens that precede it. This maintains the autoregressive property of language generation. It is mathematically represented as:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j, \\ -\infty & \text{otherwise.} \end{cases} \tag{1}$$

Here, i and j denote the positions in the sequence. The use of zero values permits attention to previous tokens, while $-\infty$ blocks attention to future tokens.

(b) **Attention Mask:** This mask directly modifies the attention scores, enabling the model to selectively focus on or ignore specific parts of the input. This mechanism is versatile, allowing for dynamic adjustments of the model's focus during training and inference.

(c) **Padding Mask:** Since sequences are batched together for efficient computation, they are padded to a uniform length. The padding mask ensures that the model does not attend to these padding tokens, which carry no meaningful information. This is crucial for maintaining the integrity of the model's output. The padding mask is applied as follows:

$$P_{ij} = \begin{cases} 0 & \text{for actual sequence positions,} \\ 1 & \text{for padding positions.} \end{cases} \tag{2}$$

Together, these masks ensure that the transformer model's attention mechanism operates correctly, focusing only on relevant parts of the input and adhering to the desired sequence generation constraints.

Appendices

A Character Based vs Word Based

We are giving you raw text in this homework, so you have the option to build a character-based or word-based model. Word-based models won't have incorrect spelling and are very quick in training because the sample size decreases drastically. The problem is, it cannot predict rare words (i.e words in testing set that are not present in your training set), thus resulting in a drop in performance. You would have to come up with strategies to handle OOV (Out of Vocabulary) words.

The paper describes a character-based model. Character-based models are known to be able to predict some really rare words but at the same time they are slow to train because the model needs to predict character by character. In this homework, we **strongly recommend** you to implement a character based model, since most TAs are familiar with the character-based model, so you can receive more help for debugging.

B Transcript Processing

HW4P2 transcripts contain letters. You should make use of the `LETTER_LIST` array (consisting of all characters/letters in the vocabulary) provided in the starter notebook and convert the transcripts into their corresponding numerical indices to train your model.

Each transcript/utterance is a separate sample that is a variable length. We want to predict all characters, so we need a [start] and [end] character added to our vocabulary, which are `<sos>` and `<eos>`, respectively.

Hints/Food for thought:

- Think about special characters that are included in the vocabulary.
- You need to build a mapping from char to index, and vice versa. Reverse mapping is required to convert output, i.e., index, back to char.
- How do you deal with the length of the final output when evaluating on validation/test sets?

C Single Head Attention (Minibatch)

The following pseudocode describes context computation, as outlined in Section **??**, for a minibatch. We use loops in the pseudocode – actual python code would be smarter and use tensor operations.

Listing 4 Attention (minibatch)

```
# E(:,:,:) = batchsize x max_sequence_length x dim array of embeddings for a minibatch
# mask(:,:) = batchsize x max_sequence_length array of binary masks (1 for valid, 0 for
    invalid)
# q(:,:) = batchsize x Dq matrix of query vectors
# C(:,:) = batchsize x Dv matrix of computed context vectors
# attention_weights = batchsize x max_sequence_length array of attention weights
function C, attention_weights = Attend(E, D, mask=None)
    # WK, WV, WQ must also be specified in actual code
    query_length = length(D(1,:))
    for b = 0:batchsize-1 # Goes over the entire minibatch
        instance_embeddings = E(b,:,:)
        K = instance_embedding * WK      # WK is a dim x Dq matrix
        V = instance_embedding * WV      # WV is a dim x Dv matrix
        Q = D(b,:,:) * WQ
        raw_weights = (1/sqrt(query_length)) * Q * transpose(K)
        if mask != None:
            # Mask attention weights manually here if not using ignore_index, or if you
                require a causal mask in the case of a transformer
        attention_weight = softmax(raw_weights) # Renormalize to sum to 1
        attention_weights(b,:) = attention_weight
        C(b,:) = attention_weight * V
    end
    return C, attention_weights
end
```

D Multi-Head Attention

The attention computed in (Section **??**) computes a *single* context vector C_k to compute output O_k. C_k derives and focuses on a single specific aspect of the input that is most relevant to O_k. It is reasonable to assume that there are *multiple* separate aspects of the input that must all be considered to compute O_k.

Multi-head attention works on this principle to derive multiple context vectors $C_k^1, C_k^2, \cdots, C_K^L$. Each context vector C_k^l is computed using a separate attention module of the form explained above (Section **??**), with its own learnable parameters $\mathbf{W}_K^l, \mathbf{W}_V^l, and \mathbf{W}_Q^l$. The complete module that computes an individual context vector C_k^l is called an *attention head*, thus the overall attention framework that computes multiple context vectors is referred to as *multi-head* attention.

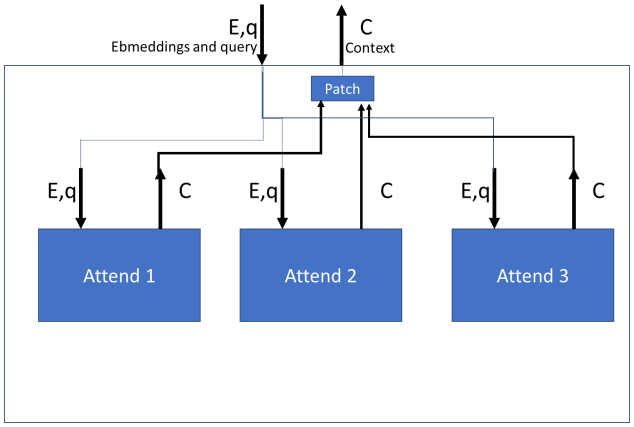


Figure 11

The final context vector C_k that is used to compute O_k is obtained by concatenating the individual context vectors: $C_k = [C_k^1, C_k^2, \cdots, C_k^L]$ (where L is the number of attention heads).

Following is the pseudocode for multi-head attention:

Listing 5 Multihead Attention

```
# E = (sequence_length, dim) array of speech embeddings for \textit{single} input
# D = (sequence_length, dim) array of character embeddings for \textit{single} input
# context = context vector computed (1,Dv) row vector)
# no_of_heads = number of attention heads
function Attention = MultiHead(E, D, mask=None)
    key = E * WK # WK is a (dim, Dk) matrix
    value = E * WV # WV is a (dim, Dv) matrix
    query = D * WQ # WQ is a (dim, Dq) matrix
    # Reshape Q,K,V to compute attention in parallel for each head
    query = reshape(query, (1, no_of_heads, Dq//no_of_heads))
    key = reshape(key, (sequence_length, no_of_heads, Dk//no_of_heads))
    value = reshape(value, (sequence_length, no_of_heads, Dv//no_of_heads))
    # Transpose to (no_of_heads, 1, Dkv//no_of_heads)
    query = transpose(query)
    # Transpose to (no_of_heads, sequence_length, Dkv//no_of_heads)
    key = transpose(key)
    value = transpose(value)
    mask = ? #Similar to single head attention, but now have one for each head
    context, attention_weights = Attend(query,key,value,mask) #Assuming Attend works on
        minibatches, Attend here just computes the dotproducts instead of projecting the
        matrices again
    context = reshape(context, (1, Dv))
    context = context * W_O #W_O is a (Dv,Dv) matrix
    return context, attention_weights
end
```

E Optimizer and Learning Rate

From our empirical studies, we recommend you use Adam or AdamW and start with a learning rate of $1e^{-3}$, and experiment with the scheduling methods. On the other hand, the LAS paper uses ASGD as the optimizer with a learning rate of 0.2. The learning schedule involved a geometric decay of 0.98 per 3M utterances (i.e., 1/20-th of an epoch). Feel free to try that if it works for you.

F Transformers and Pretraining

In the context of transformer models and their pre-training, a common strategy involves selectively freezing parts of the model while training others to optimize learning for specific tasks. For instance, when working with transformer-based models, you might focus initially on pre-training the encoder component to capture rich contextual representations from the input data. This pre-training could involve tasks such as masked language modeling, where certain words or tokens are hidden from the model, and it must predict them based on the context provided by the surrounding tokens.

Subsequently, to specialize the model for a particular downstream task (e.g., text classification, question-answering), you might then freeze the encoder to preserve the learned contextual embeddings and focus on training the decoder or a specific task-oriented head with a relevant objective. This approach allows the model to leverage the generalized understanding acquired during pre-training while adapting its output layers to perform well on the target task. By freezing parts of the model, computational resources are conserved, and training time is reduced since only a portion of the model's parameters are updated. This technique of staged training—starting with pre-training on a large corpus to learn a broad representation, followed by fine-tuning on task-specific data—has proven effective across a variety of NLP applications.

9.1 Position Encoding

Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models. For long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently. There are several requirements for the positional encodings such as; they should have some representation of time, they should be unique for each position and they should be bounded.
In the original transformer paper, they use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information.

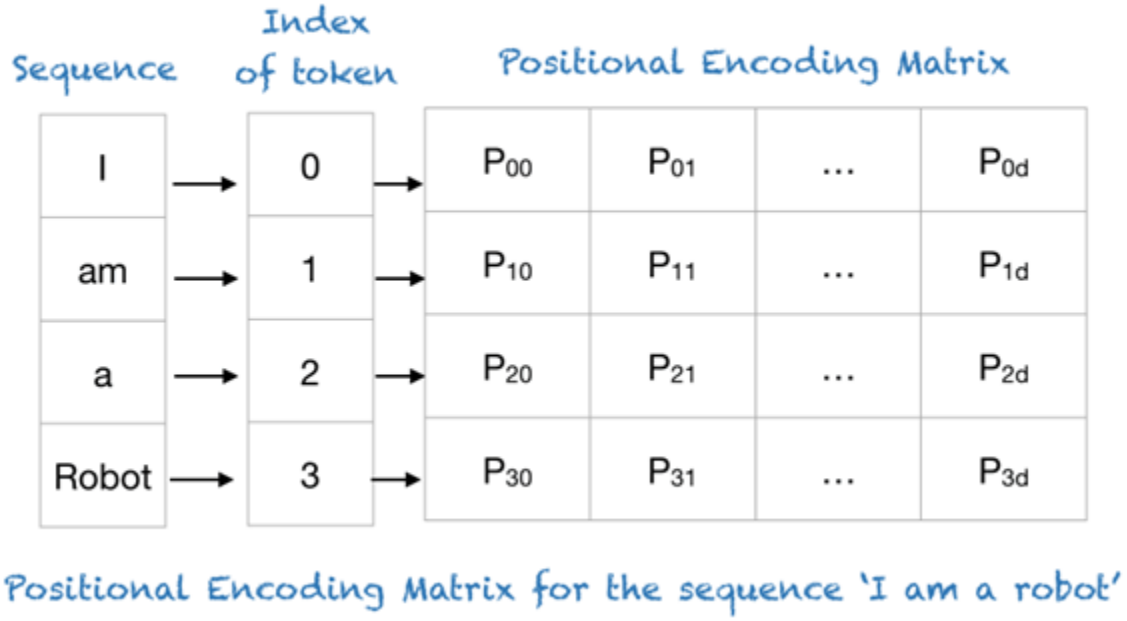


Figure 12: Adopted from machinelearningmastery

9.1.1 Positional Encoding Layer in Transformers

Suppose you have an input sequence of length L and require the position of the kth object within this sequence. The positional encoding is given by sine and cosine functions of varying frequencies:

$$P(k, 2i) = sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = cos\left(\frac{k}{n^{2i/d}}\right)$$

where
k: position of an object in the input sequence, $0 \leq k \leq l/2$
d: dimension of the output embedding
P(k, j): Position function for mapping a position k in the input sequence to index (k, j) of the positional matrix.
n: User-defined scalar, set to 10,000 by the authors of Attention Is All You Need.
i: Used for mapping to column indices , with a single value of maps to both sine and cosine functions. In the above expression, you can see that even positions correspond to a sine function and odd positions correspond to cosine functions.

Listing 6 Positional Encoding

```
function P = PositionEncoding(seq_len, d, n)
    P = # Initialize P(seq_len, d) matrix to hold the positional encoding
    for t =1:seq_len
        for i = 1:d/2
            denominator = # compute the denominator
            P[k, 2i] = # compute the positional embedding using sin
            P[K, 2i+1] = # compute the positional embedding using cos
        end
    end
    return P
```

References

Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition. In 2018 IEEE international conference on acoustics, speech and signal processing (ICASSP), pages 5884–5888. IEEE, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.