HW4P2

Automatic Speech Recognition with an Encoder-Decoder Transformer

11-785: Introduction to Deep Learning (Fall 2025)

Schedule

Release

7 November 2025

- This homework is out of 100 points and will consist of a checkpoint + final submission (both are required to achieve full score on this assignment).
- ☐ Join the kaggle competition (HERE)

Checkpoint Deadline (10 pts)

21 November 2025, 11:59 PM EST

- ☐ Complete HW4P2 **Starter** notebook and submit to **Kaggle**. Achieve low cutoff of **50%**
- ☐ Complete the HW4P2 **Checkpoint** notebook, achieve **FULL** score on MCQ, submit to **Autolab** and obtain **FULL** score.
- **ALL** above requirements must be met to achieve 10/10 points. Otherwise, 0/10 will be awarded. (Click here for submission instructions)

On-Time Deadline (90 pts)

5 December 2025, 11:59 PM EST

- Cutoff scores will be released on Autolab after the checkpoint deadline.
- ☐ Submit final scores to **Kaggle by 5 December 2025, 11:59 PM EST**
- ☐ Clean up your code notebook, complete the Final Submission notebook, and submit the **output zip file to Autolab by 7 December 2025, 11:59 PM EST** (**REQUIRED**, otherwise 0/90 will be assigned for this section) Click here for submission instructions

Slack Submission

11 December 2025, 11:59 PM EST

- ☐ Join and submit your final predictions to SLACK kaggle (HERE).
- ☐ Submit to autolab with the same On-Time Deadline submission instructions.
- You are in charge of keeping track of the slack days used, and you will be ineligible for bonus points for reaching high-cutoff for this homework.

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to help your friends debug
  - You are allowed to look at your friends code
  - You are not allowed to type code for your friend
  - You are not allowed to look at your friends code while typing your solution
  - You are not allowed to copy and paste solutions off the internet
  - You are not allowed to import pre-built or pre-trained models
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared to all submitted code in this semester and in previous semesters using MOSS.

  We encourage you to meet regularly with your study group to discuss and work on the homework. You will not only learn more, you will also be more efficient that way. However, as noted above, the actual code used to obtain the final submission must be entirely your own.

Homework Objectives

- **If you complete this homework successfully, you would ideally have learned**
  - Understand the fundamentals of Automatic Speech Recognition (ASR), including feature extraction and sequence-to-sequence modeling.
  - Implement positional encoding to incorporate sequence order in transformer models for speech processing.
  - Develop and apply padding masks and attention masks to handle variable-length speech inputs and enforce causal constraints in decoding.
  - Understand the role of multihead self-attention in the encoder to capture temporal dependencies in speech representations and cross-attention in the decoder to attend to encoded features for transcription generation.
  - Implement key transformer components such as self-attention layers, cross-attention layers, feedforward layers, and full encoder and decoder layers.
  - Train a transformer-based ASR model using an encoder-decoder architecture with CTC or seq2seq objectives.
  - Explore decoder pre-training techniques, such as initializing the ASR decoder with a pre-trained language model to leverage prior linguistic knowledge.
  - Understand progressive training strategies.
  - Decode speech inputs using greedy and beam strategies to generate transcriptions and evaluate model performance using ASR metrics such as CER (Character Error Rate).
  - Tune parameters and hyperparameters to find an optimal solution to the ASR problem.

IMPORTANT NOTE

This write-up focuses on modeling written language. Since written language can be represented at different levels—characters. subwords or words; we will use the term **tokens** to refer to these units. What a token may correspond to will depend on the modeling choice.

The write-up may also include several pseudocode listings. These are intended to convey the logic behind various approaches, but are *not* written in Python and may not be directly translatable. They serve as a conceptual guide; you will need to determine how to implement the logic in Python.

Finally, this document is detailed and covers foundational concepts beyond the specific homework tasks. It includes explanations and pseudocode that may not be directly related to the assignment but are designed to enhance your understanding. The actual homework problems are explicitly listed in certain sections. Please read the document carefully to identify these sections and focus on the relevant material.

Contents

1 Setup & Workflow

1.1 About the Handout

In this assignment, we are introducing a new format for assignment delivery, designed to enhance your development workflow. The key motivations for this change are the following.

- **Test Suite Integration**: Your code will be tested in a manner similar to `HWP1's`.
- **Local Development**: You will be able to perform most of your initial development locally, reducing the need for compute resources.
- **Hands-on Experience**: This assignment provides an opportunity to build an end-to-end deep learning pipeline from scratch. We will be substantially reducing the number of abstractions compared to previous assignments.

Notebook Execution Requirements For our provided notebooks to work, your notebook's current working directory must be the same as the handout directory. This is important because the relative imports in the notebooks depend on the current working directory. This can be achieved by:

1. Physically moving the notebook into the handout directory.
2. Changing the notebook's current working directory to the handout directory using the `os.chdir()` function.

Your current working directory should contain the following files:

```
handout_directory/
├── README.md
├── requirements.txt
├── hw4lib/
├── mytorch/
├── tests/
└── hw4_data_subset/
```

1.1.1 Dataset Structure

We have provided a subset of the dataset for you to use. This subset allows you to implement and test your code locally while maintaining the same structure as the original dataset. The dataset is organized as follows:

```
hw4_data_subset/
├── hw4p1_data/ ..... For causal language modeling
│   ├── train/
│   ├── valid/
│   └── test/
└── hw4p2_data/ ..... For end-to-end speech recognition
    ├── dev-clean/
    │   ├── fbank/
    │   └── text/
    ├── test-clean/
    │   └── fbank/
    └── train-clean-100/
        ├── fbank/
        └── text/
```

1.1.2 Main Library (`hw4lib/`)

For `HW4P1` and `HW4P2`, you will incrementally implement components of `hw4lib` to build and train two models:

- **HW4P1**: A *Decoder-only Transformer* for causal language modeling.
- **HW4P2**: An *Encoder-Decoder Transformer* for end-to-end speech recognition.

Many components you implement will be reusable across both parts, reinforcing modular design and efficient implementation. You should see the following files in the `hw4lib/` directory (`__init__.py` files are omitted):

```
hw4lib/
├── data/
│   ├── tokenizer_jsons/
│   ├── asr_dataset.py
│   ├── lm_dataset.py
│   └── tokenizer.py
├── decoding/
│   └── sequence_generator.py
├── model/
│   ├── masks.py
│   ├── positional_encoding.py
│   ├── speech_embedding.py
│   ├── sublayers.py
│   ├── decoder_layers.py
│   ├── encoder_layers.py
│   └── transformers.py
├── trainers/
│   ├── base_trainer.py
│   ├── asr_trainer.py
│   └── lm_trainer.py
└── utils/
    ├── create_lr_scheduler.py
    └── create_optimizer.py
```

1.1.3 MyTorch Library Components (`mytorch/`)

In `HW4P1` and `HW4P2`, you will build and train Transformer models using PyTorch's `nn.MultiHeadAttention`. To deepen your understanding of its internals, you will also implement a custom `MultiHeadAttention` module from scratch in your `mytorch` library, designed to closely match the PyTorch interface. You should see the following files in the `mytorch/` directory:

```
mytorch/
└── nn/
    ├── activation.py
    ├── linear.py
    ├── scaled_dot_product_attention.py
    └── multi_head_attention.py
```

1.1.4 Test Suite (`tests/`)

In `HW4P1` and `HW4P2`, you will be provided with a test suite to verify your implementation. The `tests/` directory should contain the following files:

```
tests/
├── testing_framework.py
├── test_mytorch*.py
├── test_dataset*.py
├── test_mask*.py
├── test_positional_encoding.py
├── test_sublayers*.py
├── test_encoderlayers*.py
├── test_decoderlayers*.py
├── test_transformers*.py
├── test_hw4p1.py
└── test_decoding.py
```

1.2 Uploading Your Handout

Regardless of the environment you work in, you must have a reliable way of uploading your handout to your working directory for the project to function correctly. Below are some recommended methods:

1.2.1 Recommended: Using a Private GitHub Repository

The most reliable way to manage your handout is by storing it in a **private** GitHub repository. Follow these steps:

1. Set up a Personal Access Token (PAT) for your repository.
2. Clone your repository using the following command:

```
!git clone https://<your_personal_access_token>@github.com/<username>/<repository>.git
```

1.2.2 Using Google Drive

Another approach is to store your handout in Google Drive and retrieve it in Colab using one of the following methods:

- **Mount Google Drive**: If using Colab, Use the built-in Colab feature to mount your drive and manually move your files to the working directory.
- **Use gdown**: If your file is publicly accessible with edit permissions, you can download it using gdown.

1.2.3 Manual Upload

For a quick solution, you can manually upload your handout into the filesystem.

WARNING: YOU MUST ENSURE CONSISTENCY. You will be implementing components of `hw4lib` inside the project directory throughout both `HW4P1` and `HW4P2`. It is **your responsibility** to ensure that changes remain consistent across both parts.

1.2.4 Example: Our Preferred Approach

Below is a sample script demonstrating a preferred way to clone and manage your repository securely:

```
import os

# Store your Personal Access Token securely in an environment variable
os.environ['GITHUB_TOKEN'] = "your_personal_access_token"

GITHUB_USERNAME = "your-username"
REPO_NAME      = "IDL-HW4"
TOKEN = os.environ.get("GITHUB_TOKEN")
repo_url       = f"https://{TOKEN}@github.com/{GITHUB_USERNAME}/{REPO_NAME}.git"

# Clone the repository
!git clone {repo_url}

# To pull your latest changes (Ensure you are in the correct directory)
!git pull
```

1.3 Setup Instructions

Setup instructions for **Local**, **PSC**, **Colab**, and **Kaggle** environments are available in the **README.md** file within the handout, as well as in the **HW4P1_nb.ipynb** and **HW4P2_nb.ipynb** notebooks.

2 Introduction

Welcome to HW4P2! In this assignment, you will reuse some of the components you built in **HW4P1** and implement additional components to build an **encoder-decoder transformer** model from scratch, train it on a speech-to-text task, and use it for automatic speech recognition. This will be a monumental task, but we promise it will be a rewarding one. We will break down this task into several smaller tasks, each of which will build on the previous. We will begin with an overview of the training objective, evaluation, and inference before breaking down the implementation into several smaller tasks.

WARNING: This assignment is designed to be significantly more challenging than previous assignments. We recommend starting early and reaching out to Piazza if you get stuck.

2.1 Training Objective

We aim to train an **encoder-decoder transformer** to model the probability of a sequence of text tokens given an input sequence of acoustic features:

$$P(y_1, ..., y_M | x_1, ..., x_N) = \prod_{t=1}^{M} P(y_t | y_1, ..., y_{t-1}, x_1, ..., x_N) \tag{1}$$

where $x_1, ..., x_N$ represents a sequence of acoustic feature frames extracted from speech, and $y_1, ..., y_M$ represents the corresponding text tokens. The conditional probability $P(y_t | y_1, ..., y_{t-1}, x_1, ..., x_N)$ is modeled using our transformer.

2.1.1 Training with Supervision

We train the model using a large corpus of **paired speech and text**. Given an input speech utterance represented as a sequence of feature frames $x_1, ..., x_N$, the target is a sequence of tokens $y_1, ..., y_M$ representing the transcription.

Example For an audio clip containing the spoken phrase:

> "I swam across the river"

the target text sequence would be:

- **Input:** Speech features like filterbanks (HW4P2) or MFCCs (HW1P2, HW3P2) extracted from the speech signal.
- **Target:** ["I", "swam", "across", "the", "river"]

We can use a decoder model to generate the token sequence corresponding to the speech transcript, token by token, as done previously in **HW4P1**. The main difference is that this output needs to be conditioned on the entire input sequence corresponding to the speech features! An encoder transformer can be used to process and map the entire input sequence into a suitable internal representation before decoding starts.

2.1.2 Preventing Information Leakage

To ensure that the decoder **does not access future tokens during training**, we apply the same techiques as in **HW4P1**:

1. **Start and End Tokens:**
   - Prepend a special **start-of-sequence** token (⟨SOS⟩) to the **target sequence** to signal the beginning of decoding.
   - Append an **end-of-sequence** token (⟨EOS⟩) to mark the end of the sequence and allow the model to learn when to stop decoding.
2. **Padding and Pad Masks:**
   - Since GPUs require **fixed-length tensors**, shorter sequences are **padded** using a special padding token (⟨PAD⟩).

- A **pad mask** ensures that these padded positions do not contribute to model predictions, preventing unwanted bias.

3. **Causal Masking in the Decoder:**
   - The decoder uses a **causal mask** (also called a look-ahead mask) to prevent each position from attending to future tokens, ensuring autoregressive generation.
   - This masking enforces left-to-right decoding, where each token is predicted based only on previous tokens in the sequence.

Handling Variable-Length Inputs in the Encoder: Unlike the decoder, the encoder processes the entire input sequence without causal restrictions. However, padding is still necessary to fit variable-length sequences into a batch, requiring a **pad mask** to prevent padded positions from affecting the encoder's learned representations.

2.2 Training Process

2.2.1 Feature Extraction

Each speech signal is converted into a sequence of F-dimensional feature vectors using a feature extraction pipeline:

$$x_1, ..., x_N \in \mathbb{R}^F \tag{2}$$

where:

- F is the dimensionality of the extracted features (e.g., Mel spectrogram, MFCCs).
- Each frame x_i is a feature vector representing a short segment of the speech signal.

We have already taken care of this for you! In this context, you will be working with sequences of 80-dimensional filterbank features as your speech representations.

2.2.2 Transformer Processing

The input sequence of **F-dimensional features** is optionally processed using shallow convolutional layers to capture local dependencies and recurrent layers to model temporal patterns to produce a sequence of **D-dimensional features**. Positional encodings are then added to this sequence to preserve sequence order before passing the features through a stack of **encoder layers**, where **self-attention** and **feedforward transformations** are applied.

The encoder outputs a sequence of **encoder hidden states** with dimensionality of D_e, which serve as input to the decoder. The decoder, functioning as an autoregressive model under causal masking, generates **decoder hidden states** of D_d dimension. For each token input, the decoder employs masked self-attention to generate its outputs and uses cross-attention to condition on the encoder's representations, refining its initial predictions from masked self-attention by deeply integrating encoder hidden states, for every token step.

2.2.3 Output Logits

The model produces a sequence of hidden states $\tilde{y}_1, ..., \tilde{y}_M$, each of dimensionality D_d. To obtain logits over the vocabulary (K), we apply a **final linear transformation**:

$$Y = XW^{(p)} \tag{3}$$

where:

- $W^{(p)} \in \mathbb{R}^{D_d \times K}$ is the **projection matrix**.
- $X \in \mathbb{R}^{M \times D_d}$ is the output of the hidden state of the decoder.
- $Y \in \mathbb{R}^{M \times K}$ are **logits**, representing unnormalized scores over the vocabulary.

2.2.4 Softmax and Cross-Entropy Loss

We can apply the **softmax function** to obtain probabilities:

$$P(y_t | y_1, ..., y_{t-1}, x_1, ..., x_N) = \text{softmax}(Y_t) \tag{4}$$

However, **we do not explicitly compute softmax in the code**. Instead, PyTorch's `nn.CrossEntropyLoss`:

- Takes the **raw logits** Y and the **ground-truth token sequence**.
- **Internally applies softmax** before computing the loss.

This training approach **maximizes the likelihood of the training data** and enables the model to learn meaningful speech-to-text mappings.

2.3 Evaluation: Measuring ASR Model Performance

To evaluate the quality of our **encoder-decoder transformer**, we need a metric that captures how well the model predicts the transcription given the input speech features.

2.3.1 Character Error Rate (CER)

A common metric for ASR evaluation is **Character Error Rate (CER)**, which measures the difference between the model's predicted transcript and the ground truth:

$$CER = \frac{S + D + I}{N} \tag{5}$$

where:

- S is the number of substitutions (incorrect characters).
- D is the number of deletions (missing characters).
- I is the number of insertions (extra characters).
- N is the total number of characters in the reference transcript.

A lower CER indicates a better ASR model. This metric is what we will be using to evaluate your ASR model. Your goal is to train your transformer to minimize this error.

2.4 Decoding: Sequence Generation in ASR

After training the model, we can use it to transcribe speech into text. During inference, the transformer decoder outputs a probability distribution over possible tokens at each time step. To construct the final transcription, a decoding strategy must be employed to select tokens based on these probabilities. Below are common decoding strategies used in ASR:

2.4.1 Greedy Decoding

Greedy decoding selects the most probable token at each time step without considering future probabilities. While efficient with a complexity of $O(KN)$, it is prone to errors since it does not optimize the overall sequence probability. This often leads to suboptimal transcriptions, especially in ambiguous contexts.

2.4.2 Beam Search Decoding

Beam search maintains B candidate hypotheses at each step, selecting the top B most probable sequences based on cumulative probability. This approach balances efficiency and accuracy by pruning lower-probability sequences while exploring multiple possibilities. With a complexity of $O(BKN)$, it provides better transcriptions than greedy decoding but is more computationally demanding.

2.4.3 Language Model Rescoring

To further improve transcription accuracy, an external language model (LM) (eg. your **HW4P1** model!) can be used to rescore beam search candidates. The LM assigns additional probabilities based on linguistic context, helping resolve ambiguities and improving fluency. This is particularly beneficial for ASR in noisy conditions or domain-specific applications.

3 Tasks

Your model performance in HW4P2 will be graded on Kaggle. Download the Autolab starter code. The following sections describe the tasks you will need to complete for this assignment. Let us begin with an overview.

3.1 Task 1: Automatic Speech Recognition with a Transformer Encoder-Decoder

- Familiarize yourself with the `tokenize`, `encode` and `decode` methods of `H4Tokenizer` class.
- Implement the `ASRDataset` class to load and pre-process the data.
- Implement the Transformer Sublayers: `CrossAttentionLayer`
- Implement Transformer layers: `CrossAttentionDecoderLayer` and `SelfAttentionEncoderLayer`.
- Implement the `EncoderDecoderTransformer` class.
- **Optional/Recommended:** Implement `Beam` decoding.
- Implement parts of `ASRTrainer`.
- Train the model on the dataset.
- Transcribe speech features in the test set and submit to Kaggle.

Restrictions:

- You may **only** use the data provided as part of this homework; using external data is strictly prohibited.
- The validation set **must not** be used for training.
- You are required to use at least **greedy decoding** for speech transcription. **Beam search** is optional but recommended.
- You may **only** use the standard **pre-norm Transformer** architecture; other transformer variants (eg. Conformer, E-Branchformer, etc.) are not permitted.
- The total parameter count of your model must not exceed **30M**.

NOTE: All implementations have detailed specifications, implementation details, and hints in their respective source files. Make sure to read all the comments and docstrings in their entirety to understand the implementation details!

4 Task 1: Automatic Speech Recognition with a Transformer Encoder-Decoder

In both **HW4P1** and **HW4P2**, you will incrementally implement components of **hw4lib** to build and train two models:

1. A Decoder-only Transformer for causal language modeling
2. Encoder-Decoder Transformer for end-to-end speech recognition.

The following sections will walk you through the steps needed to complete the latter.

NOTE: The following sections assume that you have already implemented the necessary components of **hw4lib** required for **HW4P1**.

REMINDER: All implementations have detailed specifications, implementation details, and hints in their respective source files. Make sure to read all the comments and docstrings in their entirety to understand the implementation details!

4.1 Data-Processing Components

To begin, we need essential tools for converting raw speech-text pairs into a format suitable for training an ASR model. A typical preprocessing pipeline performs the following steps:

1. Load paired speech features (e.g., 80-dimensional filterbanks) and text data into memory.
2. Normalize the extracted features for consistency across different recordings.
3. Tokenize the text transcripts (e.g., at the character or subword level).
4. Construct a vocabulary dictionary mapping each token to a numerical index.
5. Convert text transcripts into sequences of numerical indices.
6. Align speech features and text sequences, ensuring compatibility for model training.

4.1.1 About the `H4Tokenizer` (`hw4lib/data/tokenizer.py`)

Tokens are the smallest units of text your model will process. Each time step corresponds to one token, but what constitutes a "token" depends on the tokenization strategy you choose. For example, the sentence "Baby needs a new pair of shoes" can be represented as:

- A sequence of 7 word-level tokens, drawn from a large vocabulary (typically tens or hundreds of thousands of words).
- Or as a sequence of 30 character-level tokens, using a much smaller vocabulary (e.g., 256 ASCII characters).

For this assignment, you will convert each text transcript into:

- A sequence of tokens.
- A corresponding sequence of numerical indices, where each index represents the token's position in the vocabulary.

These numerical sequences will serve as inputs to and outputs from your model. During inference or generation, you will also need to reverse this process—converting indices back to tokens, and then reconstructing the original text.

We have provided the `H4Tokenizer` class in `hw4lib/data/tokenizer.py` to handle tokenization for both `HW4P1` and `HW4P2`.

You will be working with two broad categories of tokenization strategies:

- **Character-level tokenization:** Each character in the language is treated as a token, resulting in a small vocabulary but longer token sequences for each sentence.
- **Subword tokenization:** Splits words into smaller reusable subword units. The subword method uses Byte

Pair Encoding (BPE) to learn and apply subword merges. This approach allows for a compact vocabulary that still captures uncommon or rare words by representing them as a combination of subwords.

Selecting an appropriate tokenization strategy is crucial, as it affects the model's vocabulary size, memory efficiency, and handling of rare or out-of-vocabulary words. As part of this assignment, you will explore how different tokenization strategies affect model performance in both `HW4P1` and `HW4P2`.

`H4Tokenizer` supports the following tokenization strategies:

- Character-level tokenization
- Subword tokenization with a vocabulary size of 1,000
- Subword tokenization with a vocabulary size of 5,000
- Subword tokenization with a vocabulary size of 10,000

Before proceeding, familiarize yourself with the `H4Tokenizer` class and its key methods:

- `tokenize`
- `encode`
- `decode`

Their documentation can be found in their soure files. You will use these methods both when preparing datasets and during model decoding.

4.1.2 Dataset Implementation (`hw4lib/data/asr_dataset.py`)

For `HW4P2`, you will be working with a dataset located in the `hw4p2_data` subdirectory inside the `hw4_data` directory. The structure is organized as follows:

```
hw4_data/
└── hw4p2_data/
    ├── train-clean-100/
    │   ├── fbank/
    │   └── text/
    ├── dev-clean/
    │   ├── fbank/
    │   └── text/
    └── test-clean/
        └── fbank/
```

The `train-clean-100`, `dev-clean`, and `test-clean` directories contain the dataset splits. Each split consists of:

- Speech features stored in `.npy` format within the `fbank` subdirectory.
- Text transcripts stored in `.npy` format within the `text` subdirectory.

Note that the `test-clean` directory contains only speech features. Your model will generate transcriptions for these samples, which will be evaluated on Kaggle.

To work with this dataset, you will use the `ASRDataset` class provided in `hw4lib/data/asr_dataset.py`. This class is designed to:

- Load the `.npy` feature files containing filterbank representations.
- Load the corresponding transcript files (except for the test partition).
- Tokenize the transcripts using a provided tokenizer.
- Apply normalization strategies to the features:
  - **Global MVN**: Normalization using global mean and variance computed from training data.
  - **Cepstral**: Per-utterance mean and variance normalization.
  - **None**: No normalization applied.

- Apply **SpecAugment** data augmentation (only for training), including:
  - Time masking (masking random time steps).
  - Frequency masking (masking random frequency bands).
- Prepare two versions of each tokenized transcript:
  - A **shifted** version with a Start-of-Sequence (SOS) token prepended.
  - A **golden** version with an End-of-Sequence (EOS) token appended.
- Track dataset statistics such as total characters, tokens, and sequence lengths.
- Provide a custom `collate_fn` function for batching data correctly by:
  - Padding features and transcripts to batch-uniform lengths.
  - Providing lengths for packed sequence processing and mask generation.
  - Ensuring proper tensor types.

Your task is to complete parts of the `__init__` method and fully implement the `__len__`, `__getitem__`, and `collate_fn` methods **according to the provided specifications in the source file**. Your implementation should ensure sequences are properly aligned, padded, and formatted for training and evaluating an automatic speech recognition model.

Run the command in the **Dataset Implementation** section of `HW4P2_nb.ipynb` to test your implementation incrementally.

4.2 Model Implementations

The following sections will guide you through the incremental process of building a Encoder-Decoder Transformer model. Specifically, we will implement the pre-norm variant of the encoder-decoder transformer architecture. Before we begin, let's define some key terminology that will be used throughout the architecture construction process:

Terminology:

- **Sublayer:** A fundamental building block within a Transformer layer. Examples include self-attention, cross-attention and feedforward layers.
- **Layer:** A complete unit in a Transformer model, consisting of multiple sublayers (e.g., a Cross-Attention Decoder Layer that includes self-attention, cross-attention and feedforward layers).
- **Transformer:** A deep neural network architecture composed of multiple layers along with additional components like embedding layers, positional encoding layers, etc.

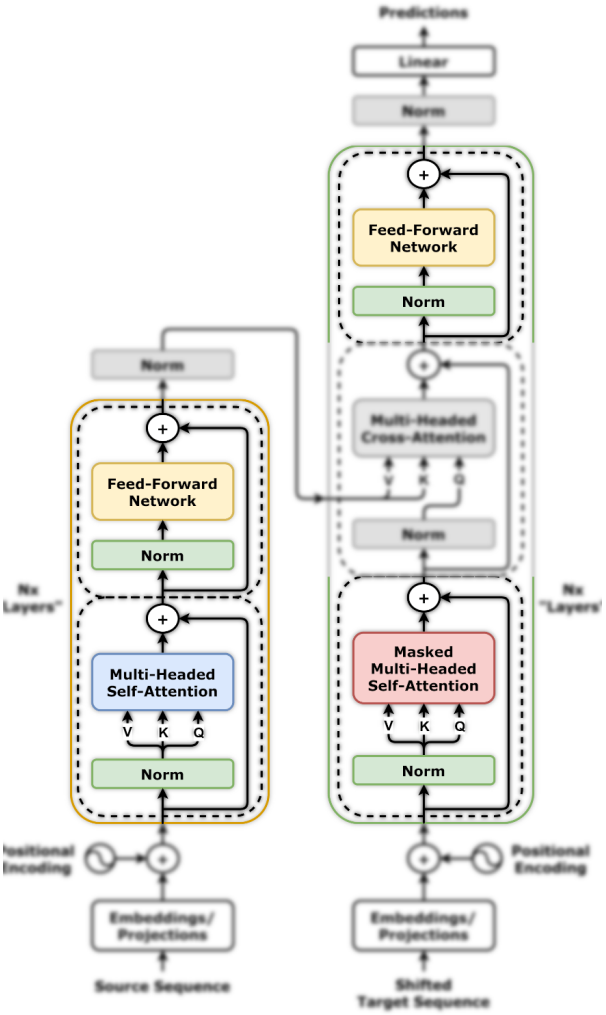
NOTE: As you incrementally implement each component, you can test your implementations using the commands provided in the `Model Implementations` section of `HW4P2_nb.ipynb`.

4.2.1 Transformer Sublayers (`hw4lib/model/sublayers.py`)

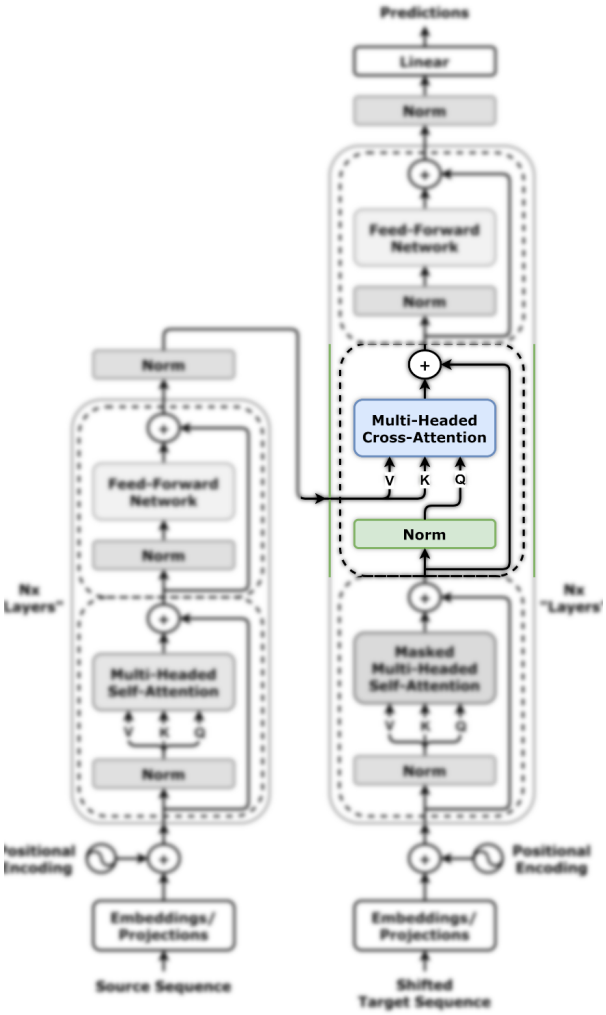
Great news! Most of the required sublayers for this assignment have already been implemented. Figure 1a (highlighted in color) illustrates the components that can be directly reused from your work in **HW4P1**.

In the following section, you will implement the final missing sublayer required for the pre-norm variant of the encoder-decoder Transformer. We recommend that you familiarize yourself with PyTorch's `nn.MultiheadAttention` class before implementing the sublayer.

CrossAttentionLayer: Implement the `CrossAttentionLayer` class in `hw4lib/model/sublayers.py`. Refer to Figure 1b (just the colored portion) for the layer architecture. This class will contain the logic for the cross-attention mechanism. **Follow the specifications and doc strings in the source file carefully for our tests to work.**



(a) Sublayers built in **HW4P1**



(b) Cross-Attention Sublayer architecture

Figure 1: Transformer Sublayers

4.2.2 Transformer Cross-Attention Decoder Layer (`hw4lib/model/decoder_layers.py`)

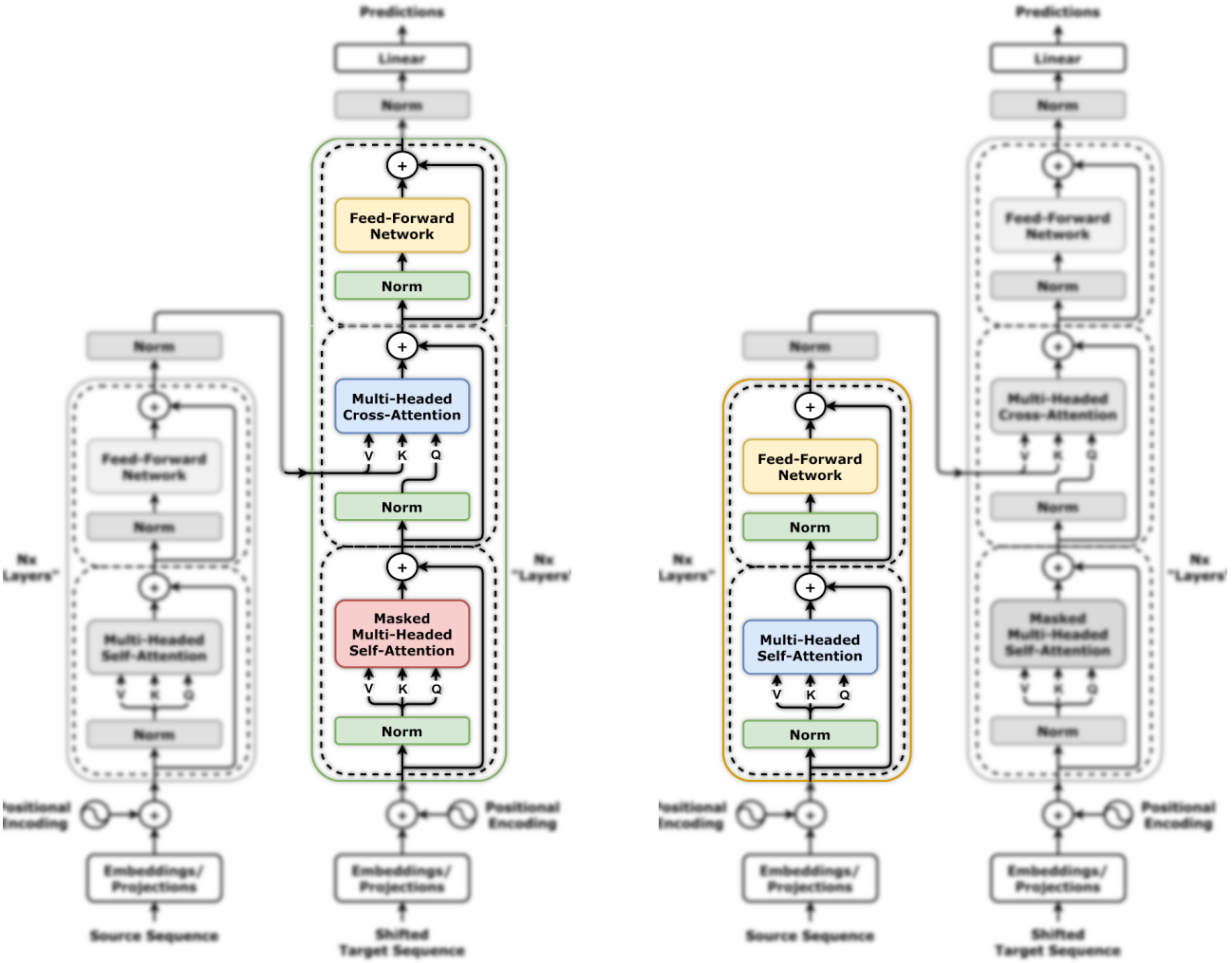
Implement the `CrossAttentionDecoderLayer` class in `hw4lib/model/decoder_layers.py`. Refer to Figure 2a (just the colored portion) for the layer architecture. This class just contains the SelfAttentionLayer, CrossAttentionLayer and FeedForwardLayer as submodules. **Follow the specifications and doc strings in the source file carefully for our tests to work.**

4.2.3 Transformer Self-Attention Encoder Layer (`hw4lib/model/encoder_layers.py`)

Implement the `SelfAttentionEncoderLayer` class in `hw4lib/model/encoder_layers.py`. Refer to Figure 2b (just the colored portion) for the layer architecture. This class just contains the SelfAttentionLayer and FeedForwardLayer as submodules. The implementation will be nearly identical to the `SelfAttentionDecoderLayer` class you implemented in **HW4P1**, with one key distinction: there are no causal constraints on attending to the input sequence. **Follow the specifications and doc strings in the source file carefully for our tests to work.**

4.2.4 Encoder-Decoder Transformer (`hw4lib/model/transformers.py`)

In this section, we integrate all the components to construct the Encoder-Decoder Transformer. Specifically, implement the `EncoderDecoderTransformer` class in `hw4lib/model/transformers.py`. Refer to Figure 3 (focus on the colored portion) for the layer architecture. This class will contain the following submodules:



(a) Cross-Attention Decoder Layer Architecture

(b) Self-Attention Encoder Layer architecture

Figure 2: Transformer Layers

- A `SpeechEmbedding` layer to process input speech features and apply time reduction. (See the provided `SpeechEmbedding` class (`hw4lib/model/speech_embedding.py`) for reference.
- A `PositionalEncoding` layer to inject positional information into the input and target embeddings.
- A user-specified number of `SelfAttentionEncoderLayer`'s to encode the input sequence.
- A user-specified number of `CrossAttentionDecoderLayer`'s to decode using both the target sequence and encoder output.
- A `nn.Embedding` layer to convert each token in the target sequence into a sequence of `d_model`-dimensional vectors.
- A final `nn.Linear` layer to obtain logits over the vocabulary space.
- A `nn.LayerNorm` for normalization after both encoding and decoding.
- A `nn.Dropout` layer for regularization.
- Although not part of the standard transformer architecture, we ask that you incorporate an additional linear layer, referred to as the **CTC Head**, to project the encoder's hidden state representations into the vocabulary space, facilitating auxiliary training with the CTC objective function. This has been shown to improve alignment between input and output sequences, improving model performance and convergence speed.

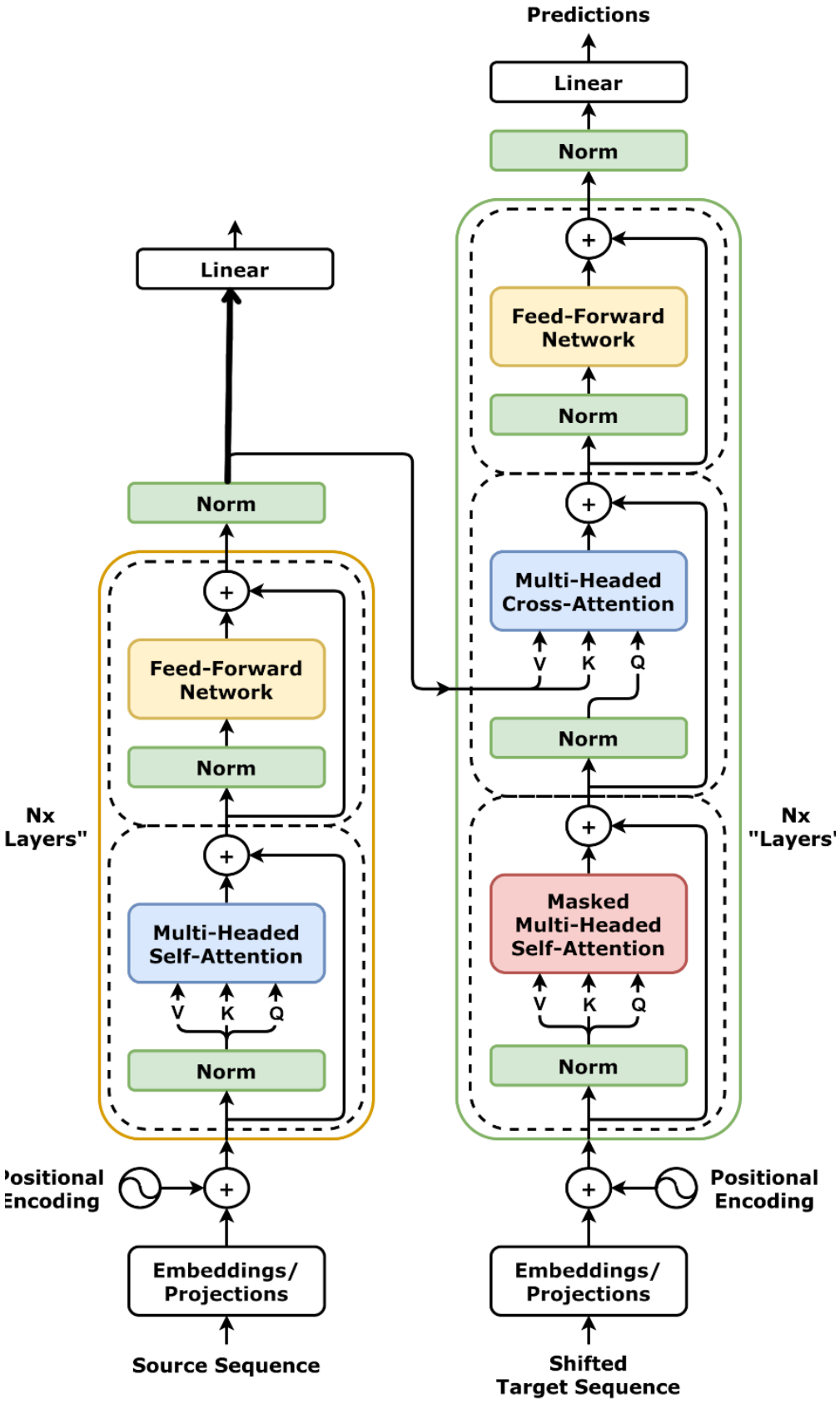


Figure 3: Full Transformer

Ensure that you follow the specifications and docstrings in the source file carefully to ensure compatibility with our tests.

4.3 Optional/Recommended: Beam Search Decoding Implementation

Implement the `generate_beam` method in the `SequenceGenerator` class. The class definition can be found in `hw4lib/decoding/sequence_generator.py`. This method generates token sequences using a beam search decoding strategy. You can test your implementation using the command in the `Decoding Implementation` section of the `HW4P2_nb.ipynb` notebook.

The following pseudocode provides a guide to implementing the `generate_beam` method:

Pseudocode 1: Beam Search Decoding

```
function generate_beam(x, beam_width, temperature, repeat_penalty):

    # Initialize scores and flags
    initialize scores as zeros of shape (batch_size, beam_width)
    initialize finished flags as False for each sequence (batch_size, beam_width)

    # Compute initial logits and probabilities
    logits = score_fn(x) # (batch_size, vocab_size)

    # Process logits
    logits = apply_repeat_penalty(logits, x, repeat_penalty)
    logits = logits / temperature
    log_probs = log_softmax applied to logits

    # Select top beam_width tokens
    # (batch_size, vocab_size) -> (batch_size, beam_width)
    next_tokens = top beam_width tokens from log_probs
    scores = scores corresponding to the top beam_width tokens

    # (batch_size, seq_len) -> (batch_size, beam_width, seq_len)
    expand x along beam dimension

    # (*, seq_len) -> (*, seq_len+1)
    append next_tokens to x

    update finished flags where EOS token is encountered

    for t in range(1, max_length):
        if all sequences are finished:
            break

        # Compute logits for next tokens
        next_token_scores = []
        for each beam:
            compute logits and append to next_token_scores

        # (batch_size, beam_width, vocab_size)
        next_token_scores = stack next_token_scores along beam dimension

        # Process logits
        next_token_scores = apply_repeat_penalty(next_token_scores, x, repeat_penalty)
        next_token_scores = next_token_scores / temperature
        next_token_scores = log_softmax applied to next_token_scores

        # Compute cumulative scores
        cum_scores = scores + next_token_scores

        # (batch_size, beam_width * vocab_size)
```

```
        flatten cum_scores for beam selection

        # Select top beam_width candidates
        # (batch_size, beam_width)
        indices = indices with top beam_width scores from cum_scores
        scores = scores corresponding to selected indices

        # Re-map to get beam indices and token ids
        # (batch_size, beam_width)
        beam_indices = indices // vocab_size
        next_tokens = indices % vocab_size

        update finished flags where EOS token is encountered

        reorder x based on beam_indices
        # (batch_size, beam_width, seq_len) -> (batch_size, beam_width, seq_len+1)
        append next_tokens to x

    sort sequences in the beam hypotheses in descending order of scores
    return x, scores
```

In this pseudocode:

- `x`: Input tensor of shape $(\text{batch\_size}, \text{seq\_len})$ representing token sequences of the same length.
- `beam_width`: Number of beams to keep at each decoding step.
- `temperature`: Scalar value used to scale logits before selecting the next token.
- `repeat_penalty`: Scalar factor applied to penalize repeated tokens during decoding.
- `score_fn`: Function that takes the current sequences `x` of shape $(\text{batch\_size}, \text{seq\_len})$ and returns logits of shape $(\text{batch\_size}, \text{vocab\_size})$.
- `apply_repeat_penalty`: Function that modifies logits to penalize repeated tokens; input and output shapes are $(\text{batch\_size}, \text{vocab\_size})$.

NOTE: Implementing this function efficiently is crucial since beam search requires multiple sequence evaluations per step. We recommend using vectorized operations instead of `for` loops. Useful PyTorch functions include `torch.topk`, `torch.gather`, `torch.cat`, `torch.sort`, `torch.log_softmax`, and logical tensor operations.

4.4 Training, Evaluation & Generation

Fantastic work! You've reached the final step—training your encoder-decoder transformer for autoregressive language modeling. Just one last task remains!

4.4.1 ASRTrainer

(`hw4lib/trainers/asr_trainer.py`)

The `ASRTrainer` class is a partially implemented module responsible for managing the training, validation, and generation loops of the ASR model. Completing this implementation will provide deeper insight into how various components interact within the overall system.

Your task is to complete the necessary in-fill sections while ensuring the integrity of the existing code. You only need to modify the parts explicitly marked with `TODO` comments.

WARNING: The classes provided for training your model are given to help you organize your training code. You shouldn't need to change the rest of the notebook, as these classes should run the training, save models/predictions and also generate plots. If you do choose to diverge from our given code, Any additional changes should be made only with a clear understanding of their impact..

Implementation Tasks:

- **Initialize the loss criterion's in `__init__`.**
- **Complete the training loop in `_train_epoch`.**
- **Integrate your greedy decoding implementation in `recognize`.**
- **Implement the validation loop in `_validate_epoch`.**
- **Finalize the full training pipeline in `train`.**

Upon completing these tasks, your `ASRTrainer` class will be fully functional, supporting training, evaluation, and speech transcription. Experiment freely!

Important: You must optimize for **Character Error Rate (CER)** by comparing the transcribed speech features with the ground truth transcript on the test set.

5 Submission

Once you have completed all implementations, trained your model, and achieved a satisfactory **Character Error Rate (CER)** on the validation set, you are ready to submit your results on Kaggle. To do so, you must generate a `submission.csv` file with the following structure:

id	transcription
0	Hello world
1	This is an example
2	ASR models transcribe speech
3	Optimization reduces CER

Table 1: Example format of `submission.csv`.

Submission Steps: We've reserved a cell in the notebook that will do this `.csv` generation for you:

- Run the provided cell once you are satisfied with your current state to generate the `.csv` file.
- Submit the `.csv` file to the competition on **Kaggle**.

Early Cutoff: To qualify for the **Early Cutoff Bonus**, you must **achieve a CER of** $\leq 50\%$ **on the Public subset of the test set.** Our reference implementation, using the hyperparameters provided in `HW4P2_nb.ipynb`, meets this criterion in less than **10 epochs**.

The **HIGH**, **MEDIUM**, and **LOW** cutoff thresholds will be announced on **Piazza** after the early submission deadline has passed.

6 Tips & Tricks

Implementing and training a Transformer model for Automatic Speech Recognition (ASR) is both challenging and rewarding. Below are strategies to enhance your model's performance and efficiency.

6.1 Managing Trade-offs through Feature Representation

Balancing training efficiency and model performance is crucial. Training a Transformer can be resource intensive, but careful choices in feature selection, tokenization, and embedding can mitigate this.

6.1.1 Impact of Tokenization

The choice of tokenization strategy affects sequence length, vocabulary size, and how well the text aligns with the speech features.

- **Character-Level Tokenization**: Represents text as individual characters. This leads to longer sequences, but reduces vocabulary size and eliminates out-of-vocabulary (OOV) issues. However, since the sequence length becomes much larger, decoding with algorithms like beam search can be significantly slower.

- **Subword Tokenization (e.g., BPE)**: Breaks words into frequently occurring sub-units, reducing sequence length while maintaining flexibility for rare words, but requires a large vocabulary. In general, **higher vocabulary sizes lead to shorter sequence lengths**, as fewer tokens are required to represent the same transcription. However, this can come at the cost of increased model complexity

Choosing the right tokenization method impacts model efficiency and transcription accuracy.

6.1.2 Speech Feature Representation

You are provided with 80-dimensional filter bank features extracted from the raw audio. However, experimenting with the number of features used can influence performance. Using fewer features can reduce memory and computation costs. However, dropping too much information might result in loss of critical information needed for accurate transcription.

6.1.3 Sequence Downsampling for Efficient Training

The sequence of filter banks is processed by an embedding layer (The **SpeechEmbedding** class) consisting of stacked CNNs and/or LSTMs. The raw speech feature sequence can be long, making training slow and memory-intensive. Downsampling techniques can help reduce sequence length while preserving relevant information:

- **CNN-based Downsampling**: Using convolutional layers with strides greater than 1 reduces the sequence length early in the model, lowering memory requirements. - **Pooling Layers**: Applying time-axis pooling, such as max or average pooling, can further compress the sequence without losing essential information. - **Subsampling in LSTMs**: Incorporating frame-skipping mechanisms or subsampling every few frames in the LSTM layers could also reduce computational complexity.

Properly applying these techniques can significantly improve training efficiency and inference speed while maintaining transcription accuracy. However, again too much downsampling can lead to loss of critical information and poor performance.

6.2 Integrating your Language Model into Transformer-Based ASR Systems

Incorporating language models (LM) into transformer-based automatic speech recognition (ASR) systems can significantly enhance transcription accuracy by providing contextual linguistic understanding. This section explores various some ways you can use the language model you trained in **HW4P1** in **HW4P2**. fusion strategies and introduces a training approach that leverages pretrained decoders to effectively integrate LMs into ASR architectures.

6.2.1 Shallow Fusion for Language Model Integration

In the **Shallow Fusion** approach, the ASR model and an external language model (LM), such as the one trained in **HW4P1**, are combined at the decoding stage. During each step of the beam search, a log-linear interpolation is

performed between the ASR model's logits and the language model's logits to enhance the transcription accuracy. While this method is conceptually simple, it can be computationally expensive due to the addition need for LM inference.

This is trivially supported by the `ASRTrainer` class provided in the handout through its **recognize** method, although the implementation is not provided in the notebook. You will need to read through and understand the inference pipeline and then write the necessary code to integrate shallow fusion with your ASR model.

Beyond shallow fusion, there are other fusion-based language model integration methods, such as Deep Fusion, Cold Fusion, and Component Fusion. While the pipeline we have provided can be modified to support these methods, they tend to be more complex to implement. As such, we leave it up to you to explore these approaches independently.

Please note that implementing these advanced fusion techniques is not required to achieve the **HIGH** cutoff. If you are interested in exploring them, we recommend doing so after reaching the **HIGH** cutoff using the existing methods.

6.2.2 Training Strategy 1: Pretrained Decoder Initialized

Another way you can use your **HW4P1** model is to use it to initialize your **HW4P2** model! This approach has the potential to speed up convergence and improve model performance by leveraging prior knowledge from a pre-trained language model. It is particularly beneficial in scenarios where you have limited paired speech-text data for the ASR task but plenty of unpaired text-data to have a well-trained decoder model. The unpaired text data we have provided for **HW4P1** is in the domain and is perfectly suited for this task! In this mode of training, you can leverage a pre-trained Decoder-Only Transformer model to initialize the Encoder-Decoder Transformer for your Automatic Speech Recognition (ASR) task. The steps involved are as follows:

1. **Initialize with Pretrained Weights:** The Encoder-Decoder Transformer is initialized with the shared weights from a pre-trained Decoder-only transformer. This includes components such as the self-attention layers, feed-forward networks (FFNs), and other relevant layers.
2. **Freeze Pretrained Weights:** Initially, you will freeze the pre-trained weights of the decoder and train only the encoder and cross-attention layers on the ASR task. This ensures that the pre-trained decoder weights are not modified during the early stages of training.
3. **Unfreeze and Fine-tune:** After training the encoder and cross-attention layers, you can unfreeze all the weights (including those of the pre-trained decoder) and proceed to fine-tune the entire model on the ASR task. Optionally, you can apply a lower learning rate for the pre-trained decoder weights to prevent them from changing too rapidly.

Again, this is trivially supported by the framework provided in the handout, although the implementation is not directly available in the notebook. You will need to read through and understand the framework, then write the necessary code to implement this feature yourself. To help you with this, here are a couple of useful resources:

1. The `from_pretrained_decoder` method from the `EncoderDecoderTransformer` class. This method can be used for initializing the model with a pretrained decoder and selectively freezing layers.
2. The `create_optimizer` function in `hw4lib/utils/create_optimizer`. This function is useful for setting layer-specific learning rates, which is particularly helpful when applying different learning rates to the frozen and trainable layers.

NOTE: The Decoder-Only Transformer checkpoint you use must be compatible with the Encoder-Decoder Transformer! The shared weights should have the same dimensionality, and the same tokenization strategy should be used across both models.

6.3 Training Strategy 2: Progressive Trainer

Progressive Training is an advanced but effective strategy that gradually increases the model's capacity during training. This approach offers several key advantages, particularly for large, complex models like Transformers. The core idea behind progressive training is that a model trained on a simpler version of the task can generalize more effectively to more complex tasks, thus resembling a form of transfer learning. By progressively increasing the difficulty, this strategy helps reduce the risk of overfitting early in the training process.

One of the primary benefits of progressive training is its ability to speed up the initial stages of training by using fewer parameters. This enables faster convergence, allowing the model to learn the basic structure of the problem before dealing with more complex aspects. Additionally, progressive training stabilizes the learning process by gradually introducing more complexity, which minimizes the risk of unstable gradients. This also leads to better resource utilization, as only a small portion of the model or data is trained initially, saving both memory and computational resources and facilitating quicker experimentation.

It is essential to define the concept of "difficulty" when using this strategy. In the context of training an ASR model, difficulty can be gradually increased by scaling up the dataset, increasing model complexity, or adjusting regularization techniques such as augmentation, dropout, or label smoothing. Another way to define difficulty could be heuristic-based—for example, shorter sentences may be easier to transcribe than longer ones. In this case, you could split your dataset by sentence length and initially train the model on shorter sequences before gradually introducing longer ones.

Additionally, it is important to determine the schedule for increasing difficulty. Simple strategies may use a fixed schedule, such as progressively doubling the number of encoder-decoder layers every 5 epochs while simultaneously increasing regularization. Alternatively, more sophisticated approaches may incorporate self-paced learning, where the difficulty level increases proportionally to the model's performance on the current training data. This adaptive method can provide more flexibility and allow the model to progress at its own pace, ensuring it is always challenged without being overwhelmed.

We recommend the following schedule: You can start with a smaller model, initially using just 1 encoder and 1 decoder layer. The idea is to progressively increase the model's capacity by adding more layers after each pretraining iteration. Here's the step-by-step process:

1. **Start Small:** Begin training with a minimal architecture, typically with only 1 encoder layer and 1 decoder layer.
2. **Progressive Layer Addition:** After each pretraining iteration, increase the number of layers in the encoder and decoder, making the model progressively deeper. This allows the model to gradually learn complex features without overwhelming it early on.
3. **Freezing Layers:** Optionally, you can freeze the previously added layers to prevent them from being updated in the early stages of training. This can help the new layers to better learn the task-specific features before fine-tuning the entire model.
4. **Regularization:** Initially, keep regularization techniques such as `dropout` and `label smoothing` low or disabled. As the model complexity increases, you can gradually enable and tune these techniques to prevent overfitting.
5. **Unfreeze All Layers:** Once the model has trained sufficiently with the progressively added layers, unfreeze all layers and train the full model. At this point, all layers should be updated during training, with appropriate regularization applied.

This approach allows you to start with a simpler model, gradually making it more complex. The progressive addition of layers helps the model learn effectively without requiring all parameters to be learned at once, which can lead to faster convergence and potentially better generalization.

The schedule above is supported by the `ProgressiveTrainer` class provided in the handout. The `Progressive Trainer` class inherits from the `ASRTrainer` class and implements the curriculum learning approach described above where model complexity and regularization are gradually increased through defined training stages. However, the implementation is not directly available to you in the notebook. You will need to read through and understand the framework, then write the necessary code to implement this yourself. To help you with this, here are a couple of useful resources:

1. The `ProgressiveTrainer` class can be used to define your schedule and train via in a one-shot manner. You can also use it to train the full model.
2. The `create_optimizer` function in `hw4lib/utils/create_optimizer`. This function is useful for setting layer-specific learning rates, which is particularly helpful when applying different learning rates to the frozen and trainable layers.

NOTE: This strategy can be combined with the **Pretrained Decoder Initialized strategy (discussed pre-**

viously)!

6.4 Gradient Accumulation

Gradient Accumulation is a technique that allows you to train models with large batch sizes even when limited by memory constraints. This method is particularly useful when training large models like Transformers, which can require substantial memory, especially with long sequences and large batch sizes. Instead of updating the model weights after each mini-batch, gradient accumulation accumulates gradients over multiple mini-batches before performing a weight update. This effectively simulates the effect of a larger batch size without requiring the memory capacity to hold all the gradients at once. This is already implemented in the framework that we have provided. You will be able to use this in your pipeline through your **config**.

6.5 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:

1. **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.
2. **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).
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.

6.6 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, reduce the dimensionality of the speech features, down-sample the length of the speech embedding sequence, switch to a different tokenization strategy, or simplify your model architecture.

6.7 Monitoring Training

6.7.1 Visualizing & Interpreting Attention Plots

Visualizing attention plots, especially decoder masked self-attention and decoder cross-attention, is a valuable tool to monitor and interpret the training process of an encoder-decoder transformer model in Automatic Speech Recognition (ASR). These attention maps allow us to understand how the model attends to different parts of the input sequence and previous tokens during the generation of each output token. If the attention mechanism is poorly aligned, it may indicate problems in the model's learning process, such as insufficient training or issues with the architecture. The framework provided will dump these plots periodically during training. We highly recommend monitoring this periodically!

Decoder Masked Self-Attention The decoder masked self-attention is primarily responsible for generating the output sequence based on previously generated tokens. In the case of ASR, the self-attention mechanism is autoregressive, meaning it can only attend to the prior generated tokens (and not future ones). As a result, attention plots for this mechanism typically show a diagonal pattern, where each token primarily attends to itself and its preceding tokens.

Expected Shape and Trends:

- **Diagonal Dominance:** The attention plots will often show a diagonal pattern, which indicates that, at each decoding step, the model mainly focuses on previously generated tokens.
- **Shifted Attention:** As the model progresses through training, attention may gradually shift to earlier tokens, reflecting the model's growing ability to maintain context over longer sequences.
- **Noise Reduction:** At the early stages of training, attention may appear more scattered. However, as the model trains, attention tends to become more focused, indicating improved understanding of prior context.

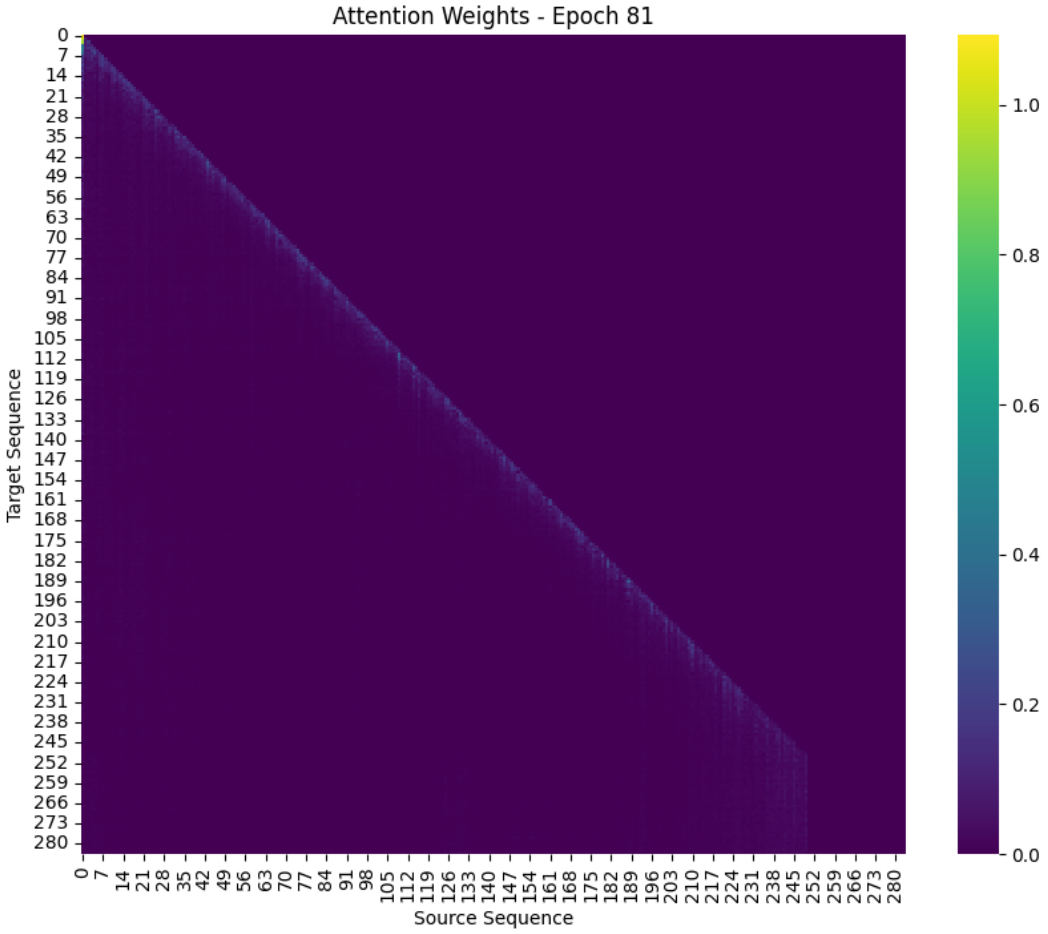


Figure 4: Example of a well-aligned Decoder masked self-attention heatmap.

Decoder Cross-Attention The decoder cross-attention mechanism is crucial in ASR tasks, as it allows the decoder to pay attention to the encoded speech features during the generation of each output token. In contrast to self-attention, which focuses on previously generated tokens, cross-attention allows the model to attend to relevant portions of the input speech sequence.

Expected Shape and Trends:

- **Initial Noisiness:** Initially, attention in the cross-attention plots may appear noisy and all over the place as the model learns which input sequences it should focus on for the corresponding output sequence.
- **Alignment with Speech Segments:** In the cross-attention layer, attention should progressively align with specific segments of the audio features that correspond to the current token being generated in the transcription. Given that in ASR, the input (speech) and output (text) are order-synchronous, you should expect the attention maps to exhibit a diagonal-like structure. This structure will become increasingly pronounced as the model trains, reflecting a growing ability to attend to the correct segments of the input audio at each decoding step.

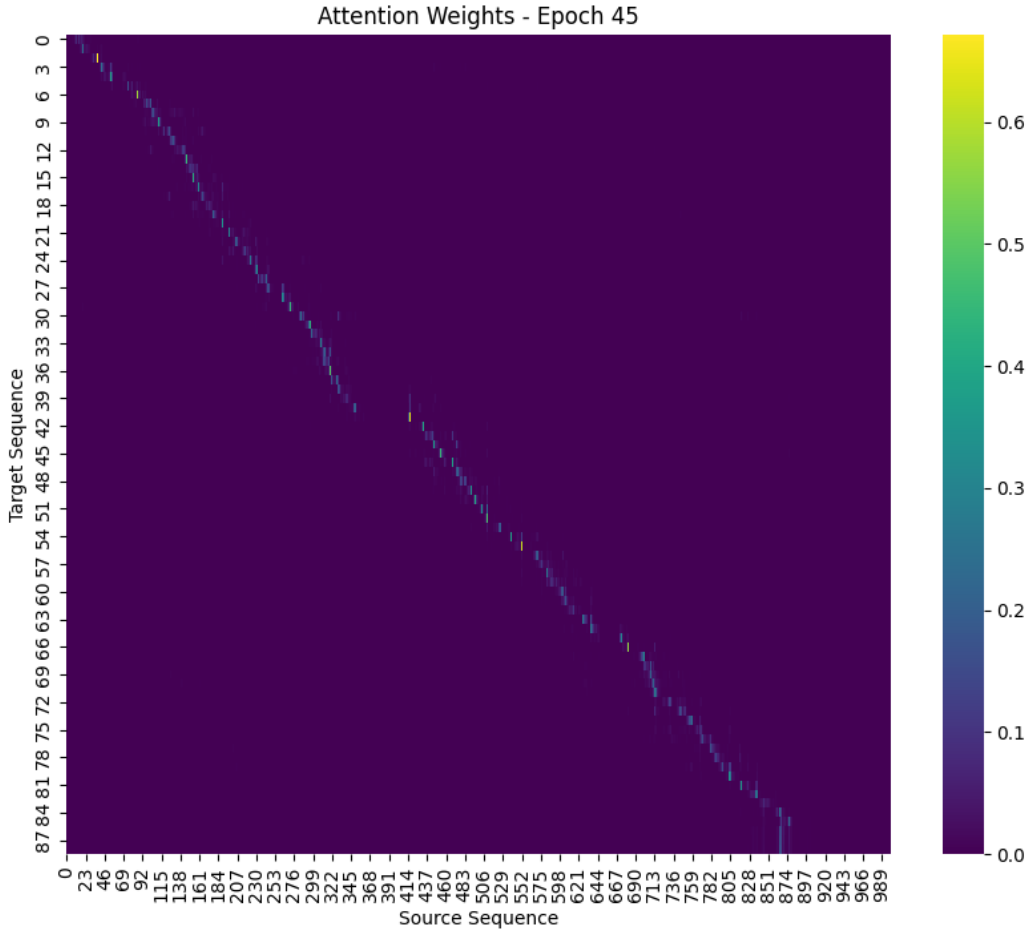


Figure 5: Example of a well-aligned Decoder cross-attention heatmap.

6.7.2 Generated Text Comparison

Along with the attention plots, we also recommend periodically inspecting the generated text and how it compares with the ground truth during training. Early on, the generated text may be noisy or contain errors, but as training progresses, you should expect the text to become more accurate and coherent. Tracking improvements in the generated output, in combination with attention plot analysis, can provide a deeper understanding of the model's performance and highlight areas that may need further optimization.

6.8 Miscellaneous

Training a Transformer model to reach optimal performance can be time-consuming, likely taking longer than previous tasks you have worked on. We recommend saving and tracking checkpoints regularly during training to avoid losing progress in case of interruptions. This also allows you to resume training from the last saved state. The framework provided to you implements rigorous checkpointing both locally **and in Weights and Biases (WanDB)**, but it is your responsibility to track and manage these checkpoints effectively.

When debugging common issues, carefully examine the loss curves, check the gradients for abnormalities, and ensure that the data pipeline is working correctly. If the model is not converging, it might be tempting to add more complexity. However, do not overlook the basics; consider adjusting the learning rate, experimenting with different optimization algorithms, or training without regularization to assess the model's capacity before introducing more complexity.

This homework is a true litmus test for your understanding of concepts in Deep Learning. It is not easy to implement and the competition will 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!