HW4P1

Autoregressive Language Modeling with a Causal Transformer Decoder

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

OUT: **November 7, 2025, 11:59 PM ET**
Early Bonus: **November 21, 2025, 11:59 PM ET**
DUE: **December 5, 2025, 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 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**
  - Implement self-attention from scratch, including scaled dot-product attention and multi-head attention.
  - Understand the fundamentals of Language Modeling, including next-token prediction and full-sequence generation.
  - Implement positional encoding to incorporate sequence order in transformer models.
  - Develop and apply causal and padding masks to enforce autoregressive behavior and handle variable-length inputs.
  - Explore multihead attention and its ability to capture multiple dependencies in the input sequence.
  - Implement key transformer components such as self-attention layers, feedforward layers, and full decoder layers.
  - Train a causal transformer decoder model for autoregressive text generation.
  - Generate text samples using the trained language model to evaluate its qualitative performance.

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

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
│   ├── test/
│   ├── train/
│   └── valid/
└── 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/
│   ├── asr_dataset.py
│   ├── lm_dataset.py
│   ├── tokenizer.py
│   └── tokenizer_jsons/
├── decoding/
│   └── sequence_generator.py
├── model/
│   ├── decoder_layers.py
│   ├── encoder_layers.py
│   ├── masks.py
│   ├── positional_encoding.py
│   ├── speech_embedding.py
│   ├── sublayers.py
│   └── transformers.py
├── trainers/
│   ├── asr_trainer.py
│   ├── base_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
    ├── multi_head_attention.py
    └── scaled_dot_product_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/
├── test_dataset*.py
├── test_decoding.py
├── test_decoderlayers*.py
├── test_encoderlayers*.py
├── test_hw4p1.py
├── test_mask*.py
├── test_mytorch*.py
├── test_positional_encoding.py
├── test_sublayers*.py
├── test_transformers*.py
└── testing_framework.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 Preamble

This homework deals with the problem of **language modelling**, i.e. modelling the probability distribution of symbol sequences in a language, and **generating** sequences from this distribution.

In this context, a **language** is defined as the set of all valid sequences of symbols from a vocabulary. For example, if your symbols were words, and the vocabulary were the set of all words in a dictionary, language would be the set of all valid word sequences, comprising words from the dictionary. If your symbols were characters, your vocabulary would be the set of all possible characters (including spaces and punctuation), and the language would comprise all possible character sequences.

More generally, we work with **tokens**, which are strings of characters (see 6.1.1 for the tokenizers used).

Representing sequences of tokens as $x_0, x_1, \ldots$, the probability distribution of the language is the distribution

$$P(x_0, x_1, \ldots)$$

which specifies the probability that the given sequence $x_0, x_1, \ldots$ will occur in a random selection of a token sequence from the language.

The problem of language modelling is that of **characterizing** this probability distribution $P(x_0, x_1, \ldots)$, such that the probability of any given sequence may be computed, and that of **generation** of token sequences, i.e. drawing samples (which are represented as token sequences) from this distribution.

The problem of language modelling has a long and hallowed history, with early formal grammars from the 50s–70s giving way to statistical models such as n-gram models from the 80s onwards. The state of the art, currently, are neural models, typically built using transformers, which far outperform all prior models by a tremendous margin.

In this homework we will learn to build a neural language model using transformers.

The set of all valid token sequences $x_0, x_1, \ldots$ is infinitely large, and hence characterizing $P(x_0, x_1, \ldots)$ over all valid sequences directly is impossible. Instead, our model breaks this problem down using Bayes' rule:

$$P(x_0, x_1, \ldots) = P(x_0)\, P(x_1 \mid x_0)\, P(x_2 \mid x_0, x_1) \cdots = \prod_t P(x_t \mid x_0, \ldots, x_{t-1}).$$

Note that this decomposes the problem of computing the probability of entire symbol sequences into that of sequentially computing the probability of the **next** token x_t, given the sequence of all previous tokens $x_0, \ldots, x_{t-1}$ in the sequence. It turns out that **this** problem - computing the probability of the next token given the past tokens, also referred to as **next-token prediction** - is something that transformer-based models can indeed do efficiently (how we deal with the very first token x_0, which has no prior tokens, will be discussed later).

The **problem** of **generating** text too can now be converted simply to that of sequentially drawing tokens from the distribution $P(x_t \mid x_0, \ldots, x_{t-1})$. The problem of **completion** of a given sequence $x_0, \ldots, x_k$ (by generating $x_{k+1}, x_{k+2}, \ldots$) too can be cast as simply attempting to draw the most likely sequence of next tokens given our initial sequence.

We will see how to do all this in this homework.

3 Introduction

Welcome to HW4P1! In this assignment, you will implement a **decoder-only transformer** from scratch, train it for autoregressive language modeling, and use it to generate text. The architecture follows the design of GPT-2, a foundation for modern language models such as GPT-4, GPT-4o, and LLaMA-3. Your goal is to build the core architecture and training pipeline of a decoder-only transformer.

The work is divided into incremental steps, each building on the last. Many of the components you build for this homework will be reused in HW4P2. We begin with the training objective, evaluation, and inference, then proceed through the implementation in well-defined stages. The process will be challenging, but we promise it will be a rewarding one.

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.

3.1 Training Objective

We aim to train a **decoder-only transformer** to model the probability of a sequence of tokens autoregressively:

$$P(x_1, ..., x_N) = \prod_{t=1}^{N} P(x_t | x_1, ..., x_{t-1}) \tag{1}$$

where $x_1, ..., x_N$ represents a sequence of tokens and each conditional probability $P(x_t | x_1, ..., x_{t-1})$ is modeled using our transformer.

3.1.1 Training with Self-Supervision

We train the model using a large corpus of **unlabeled text**. Given a token sequence $x_1, ..., x_N$, the target is x_{N+1}, the next token in the sequence. Instead of processing each training sample independently, we train on **entire sequences in parallel**, where each token serves as both an input and a target.

Example For the sentence:

"⟨SOS⟩ I swam across the river ⟨EOS⟩"

- **Input:** ["⟨SOS⟩", "I", "swam", "across", "the"] → **Target:** "river"
- **Input:** ["I", "swam", "across", "the", "river"] → **Target:** "⟨EOS⟩"

This **parallel processing improves efficiency** but requires mechanisms to prevent **cheating** (i.e., directly copying the next token).

3.1.2 Preventing Information Leakage

To ensure that the model **does not access future tokens during training**, we apply:

1. **Start and End Tokens:**
   - Prepend a special **start-of-sequence** token ⟨SOS⟩ to the **input sequence**.
   - Append a **end-of-sequence** token ⟨EOS⟩ to the **target sequence**.
   - This effectively shifts the input sequence **rightward**, aligning it with the target. Using our example above:
     - **Input:** ["⟨SOS⟩", "I", "swam", "across", "the", "river"]
     - **Target:** ["I", "swam", "across", "the", "river", "⟨EOS⟩"]
     - This ensures that each token at positions 0 to i in the input corresponds to the token at position i in the target, with the input sequence shifted by one position relative to the target.
2. **Causal Masking:**
   - In the transformer's **self-attention layers**, a **causal mask** is applied to ensure that each token **only attends to past tokens**.
   - This prevents future token information from leaking into the model.
3. **Padding and Pad Masks:**
   - Since GPUs require **fixed length tensors**, shorter sequences are **padded** using a special ⟨PAD⟩ token.
   - A **pad mask** ensures that these padded positions do not influence the predictions.

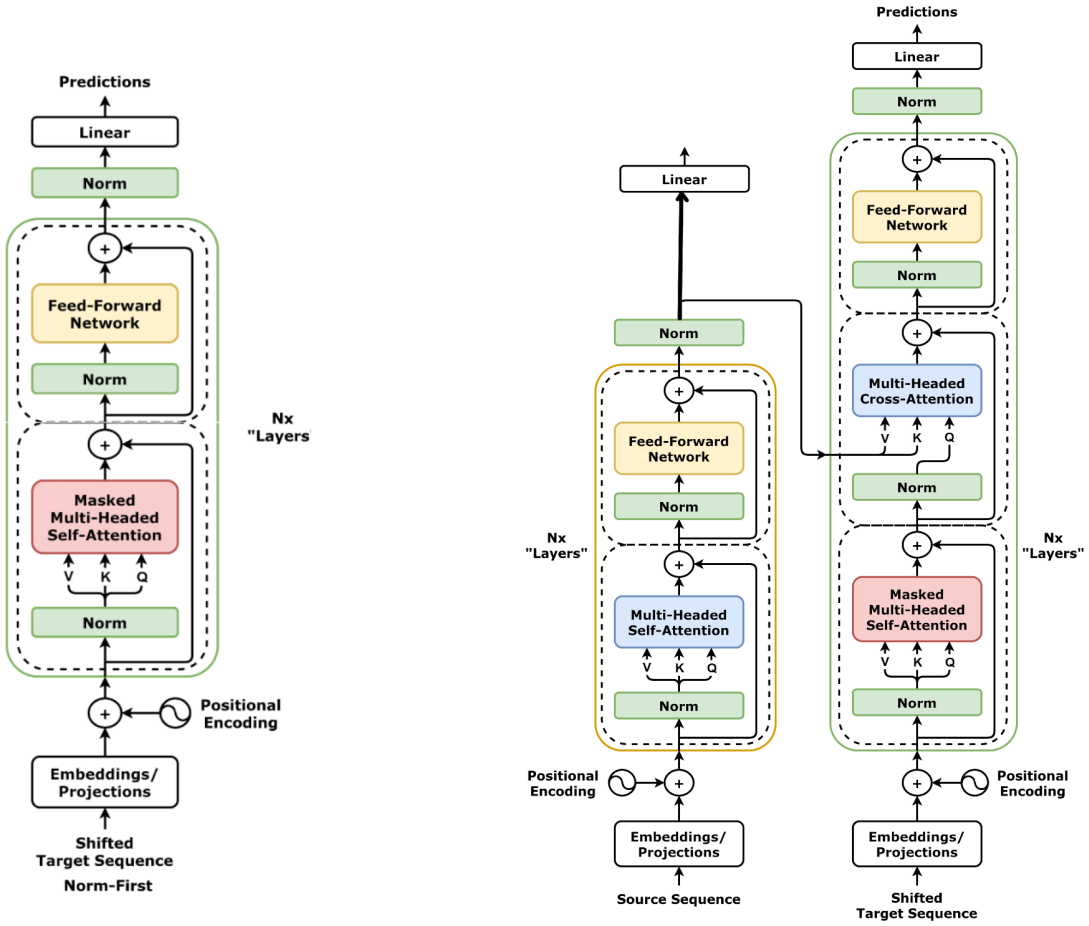
3.1.3 Decoder-Only vs Encoder-Decoder Architectures

Before we proceed with implementing our decoder-only transformer, it's important to understand how it differs from the encoder-decoder architecture you'll implement in HW4P2.

1. Decoder-Only Transformer (This Assignment - HW4P1):

- Single stack of decoder layers
- Uses causal masking to prevent attending to future tokens
- Generates text autoregressively (predicting next token)
- Examples: GPT series, LLaMA

2. Encoder-Decoder Transformer (Next Assignment - HW4P2):
   - Separate encoder and decoder stacks
   - Encoder processes entire input sequence (no masking)
   - Decoder attends to encoder output via cross-attention
   - Used for sequence-to-sequence tasks
   - Examples: Original Transformer, T5, BART



(a) Decoder-Only Transformer (GPT-style, HW4P1)

(b) Encoder-Decoder Transformer (Original Transformer, HW4P2)

Figure 1: Comparison of Transformer architectures. **(a)** Decoder-only model: a single stack of layers with masked self-attention for autoregressive generation. **(b)** Encoder-decoder model: separate encoder and decoder stacks, where the encoder processes the full input sequence and the decoder uses both masked self-attention and cross-attention to the encoder output. The encoder-decoder architecture is designed for sequence-to-sequence tasks.

3. Key Architectural Differences:
   a. Attention Patterns:

- Decoder-only: Only self-attention with causal masking
- Encoder-decoder: Self-attention (encoder), self-attention (decoder), and cross-attention (decoder attends to encoder output)

b. Use Cases:

- Decoder-only: Language modeling, text generation, completion tasks
- Encoder-decoder: Translation, summarization, speech recognition (HW4P2)

c. Input Processing:

- Decoder-only: Processes input and generates output using the same stack
- Encoder-decoder: Separate processing for input (encoder) and output generation (decoder)

3.2 Training Process

Notation Key. Throughout this section, we use the following symbols:

Symbol	Description
N	Sequence length (number of tokens in the input/output)
K	Vocabulary size (number of possible tokens)
D	Embedding dimension (size of token/vector representations)
B	Beam width (number of hypotheses maintained in beam search)
T	Temperature parameter for sampling (softmax scaling)

3.2.1 Token Embedding

Each token in the input sequence is converted into a sequence of D-dimensional embedding vectors using the model's embedding layer:

$$x_1, ..., x_N \in \{1, ..., K\} \mapsto E(x_1), ..., E(x_N) \in \mathbb{R}^D \tag{2}$$

where:

- K is the vocabulary size.
- $E(x_i)$ represents the learned embedding for token x_i.
- Implemented using PyTorch's `nn.Embedding` which maps token indices to dense vectors.

3.2.2 Transformer Processing

The sequence of **D-dimensional embeddings** is injected with **positional encodings** and passed through **decoder layers**, applying **self-attention, feedforward transformations and masking**.

3.2.3 Output Logits

The model produces a sequence of hidden states $h_1, ..., h_N$, each of dimensionality D. To obtain logits over the vocabulary (K), we apply a **final linear transformation**:

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

where:

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

3.2.4 Softmax and Cross-Entropy Loss

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

$$P(x_t|x_1, ..., x_{t-1}) = \text{softmax}(Y_t) \tag{4}$$

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 log-softmax and NLL** before computing the loss (numerically stable).

3.3 Generation: Sampling Strategies

3.3.1 Greedy Search

Greedy search selects the token with the highest probability at each step. While efficient with a complexity of $O(KN)$, it is deterministic and can lead to suboptimal sequences, as it does not maximize the joint distribution over all tokens.

3.3.2 Beam Search

Beam search maintains B hypotheses and explores multiple token sequences by considering the top B most probable tokens at each step. The most probable sequence is chosen after pruning. Its complexity is $O(BKN)$, but it can be computationally expensive for large models.

3.3.3 Top-$\tilde{K}$ Sampling

Top-$\tilde{K}$ ($\tilde{K}$ not to be confused with K, which is the vocabulary size) sampling samples from the top $\tilde{K}$ most probable tokens at each step, allowing for more diversity compared to greedy search. It helps prevent the generation of non-sensical sequences while controlling the exploration of less likely tokens.

3.3.4 Nucleus Sampling (Top-p Sampling)

Nucleus sampling selects the smallest set of tokens whose cumulative probability mass exceeds a threshold, allowing the number of candidates to adapt dynamically at each step. This ensures that sampling focuses on the most probable region of the distribution while still allowing diversity, unlike Top-$\tilde{K}$ which uses a fixed-size candidate set.

3.3.5 Temperature Sampling

Temperature scaling softens the probabilities by introducing a temperature parameter T in the softmax function. A temperature of $T = 0$ results in greedy search, while $T = 1$ recovers the standard softmax. As T increases, the distribution becomes more uniform, encouraging more exploration of less probable tokens.

Each of these strategies can be used depending on the trade-off between diversity and determinism, and the goal of the generated sequences.

4 Tasks

All problems in HW4P1 will be graded in Autolab. 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.

4.1 Task 1: MyTorch Implementations

- Modify your `Linear` implementation from HW1P1 to support arbitrary dimensions.
- Modify your `Softmax` implementation from HW1P1 to support arbitrary dimensions.
- Implement the `ScaledDotProductAttention` class.
- Implement the `MultiHeadAttention` class.

4.2 Task 2: Language Modeling with a Causal Transformer Decoder

- Familiarize yourself with the `tokenize`, `encode` and `decode` methods of `H4Tokenizer` class.
- Implement the `LMDataset` class to load and pre-process the data.
- Implement the `CausalMask` and `PadMask` functions to handle masking.
- Implement the `PositionalEncoding` class to handle positional encodings.
- Implement the Transformer Sublayers: `SelfAttentionLayer` and `FeedForwardLayer`
- Implement Transformer layer: `SelfAttentionDecoderLayer`.
- Implement the `DecoderOnlyTransformer` class.
- Implement `Greedy` decoding.
- Implement parts of `LMTrainer`.
- Train the model on the dataset.
- Achieve a per-character perplexity of less than **3.5** on the test set.
- Generate some sample text using the model.

Restrictions:

- You may **only** use the data provided as part of this homework; external data is strictly prohibited.
- The validation and test set **must not** be used for training.
- You must use at least **greedy decoding** for **text generation**. **Beam search** is optional but not-required.

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!

5 Task 1: MyTorch Implementations

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 as part of your `mytorch` library, designed to closely match the PyTorch interface.

We recommend developing the components incrementally and using the command provided in the `MyTorch Implementations` cell of `HW4P1_nb.ipynb` to test your implementation.

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!

5.1 Linear Layer (`mytorch/nn/linear.py`)

In HW1P1, you implemented a `Linear` layer that took a 2D array of shape (N, C) and computed the linear transformation of each element over the C classes. In this assignment, in order to be able to use the `Linear` layer in the `MultiHeadAttention` class, you will implement a more generic `Linear` class in `mytorch/nn/linear.py`, where the number of dimensions of the input can be arbitrary. The implementation will be largely the same as HW1P1 but with a few key differences. Feel free to use your HW1P1 implementation as a reference.

5.1.1 Forward Pass

Your input tensor **A** will be of shape $(*, \text{in\_features})$ where $*$ is a variable number of dimensions. We recommend taking the following approach:

1. Store the original shape of the input tensor in the `input_shape` attribute.
2. Flatten the input tensor to 2D of shape $(\text{batch\_size}, \text{in\_features})$ where $\text{batch\_size} = \prod(*)$.
3. Perform the affine transformation.

$$Z = AW^T + b \tag{5}$$

4. Then unflatten the output back to the original dimensions.

5.1.2 Backward Pass

The gradient of the loss with respect to the output **Z** will be of shape $(*, \text{out\_features})$. We recommend taking the following approach:

1. Reshape the gradient to 2D of shape $(\text{batch\_size}, \text{out\_features})$ where $\text{batch\_size} = \prod(*)$.
2. Reshape the input tensor you stored in the forward pass to 2D of shape $(\text{batch\_size}, \text{in\_features})$ where $\text{batch\_size} = \prod(*)$.
3. Compute the gradient of the loss with respect to the input using the chain rule as you did in HW1P1. You can refer to the following equations:

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Z} \cdot W \tag{6}$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot A \tag{7}$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{B} \frac{\partial L}{\partial Z_i} \tag{8}$$

   where B is the batch size.

4. And then reshape the gradient with respect to the input back to the original shape.

5.2 Softmax (`mytorch/nn/activation.py`)

In HW1P1, you implemented a `Softmax` activation function that took a 2D array of shape (N, C) and computed the softmax of each element over the C classes. In this assignment, in order to be able to use the `Softmax` class in the `ScaledDotProductAttention` class, you will implement a more general `Softmax` class in `mytorch/nn/activation.py`, where the number of dimensions of the input can be arbitrary and the softmax can be computed over any dimension. The implementation will largely be the same as HW1P1 but with a few key differences. Feel free to use your HW1P1 implementation as a reference.

5.2.1 Forward Pass

Given a single input vector $\mathbf{Z}$ with shape $(C,)$, whose m-th element is denoted by z_m, the `softmax` function will return a vector $\mathbf{A}$ with shape $(C,)$, where the m-th element a_m is given by:

$$a_m = \frac{\exp(z_m)}{\sum_{k=1}^{C} \exp(z_k)} \tag{9}$$

Similar calculations would apply for any N-dimensional input tensor $\mathbf{Z}$. You must return a tensor with the same shape as $\mathbf{Z}$ but with the softmax probabilities **along the dimension specified by the `dim` parameter** defined in the `Softmax` class constructor. Make sure you do this in a numerically stable way.

5.2.2 Backward Pass

For an N-dimensional input tensor, the backward pass follows similar principles to the 2D case in HW1P1. For any slice along the specified dimension `dim`, we need to compute how changes in the input affect the output probabilities.

For a slice of the input tensor along dimension `dim`, let's call the input values $\mathbf{z}$ and output probabilities $\mathbf{a}$. The Jacobian $\mathbf{J}$ for this slice has elements given by:

$$J_{mn} = \begin{cases} a_m(1 - a_m) & \text{if } m = n \\ -a_m a_n & \text{if } m \neq n \end{cases} \tag{10}$$

where a_m refers to the m-th element of the probability vector $\mathbf{a}$.

The gradient for this slice is then computed as:

$$\frac{\partial L}{\partial \mathbf{z}} = \frac{\partial L}{\partial \mathbf{a}} \cdot \mathbf{J} \tag{11}$$

This calculation must be performed for each slice along the specified dimension while keeping all other dimensions fixed. The final gradient tensor will have the same shape as the input tensor.

We recommend the following approach while implementing the backward pass, which we will illustrate with an example:

If the input tensor Z has shape (N, C, H, W) and softmax was applied along dimension C, then:

1. Identify the dimension along which softmax was applied in the forward pass:

$$\text{dim} = 1$$

2. Move dimension 1 to the last position to obtain a tensor of shape (N, H, W, C).
3. Flatten the remaining dimensions to obtain a 2D tensor of shape $(N \cdot H \cdot W, C)$.
4. Perform operations on this 2D tensor as done in HW1P1.
5. Reshape the tensor back to the 4D shape (N, H, W, C).
6. Finally, move the last dimension back to its original position to revert back to the original shape (N, C, H, W).

Hint: The function `np.moveaxis` might be useful in this implementation.

5.3 Scaled Dot-Product Attention (`mytorch/nn/scaled_dot_product_attention.py`)

Implement the scaled-dot product attention in `mytorch/nn/scaled_dot_product_attention.py` in a setting similar to what you will deal with in `HW4P1` and `HW4P2`.

Hint: Let the shapes be your guide.

5.3.1 Forward Pass

Implement the `forward` method for the `ScaledDotProductAttention` class as shown in Figure 2

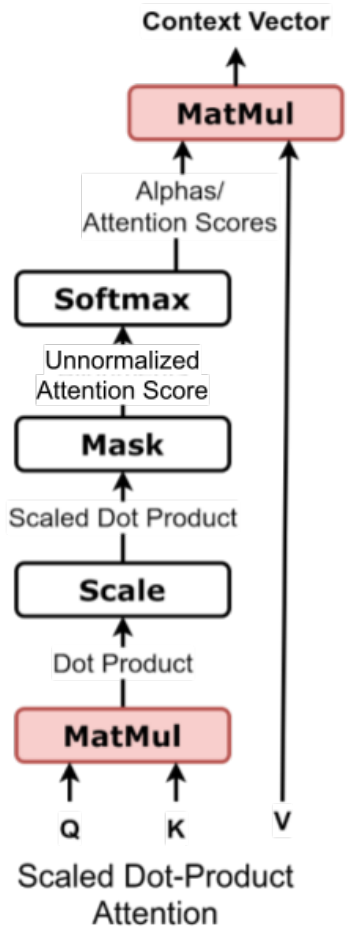


Figure 2: Scaled Dot-Product Attention

Changed the above figure to reference "Unnormalized attention score" rather than "Alignment".

The scaled dot-product attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{12}$$

where, **Q**, **K**, and **V** are the query, key, and value matrices respectively and d_k is the embedding dimension of the query and key matrices.

In our setting, we have:

- A query matrix **Q** of shape $(N, ..., H, L, E)$
- A key matrix **K** of shape $(N, ..., H, S, E)$

- A value matrix $\mathbf{V}$ of shape $(N, ..., H, S, Ev)$
- Optionally, we will have a boolean mask matrix $\mathbf{M}$ of shape $(N, ..., H, L, S)$

where:

- N is the batch size
- H is the number of attention heads for the key and value
- L is the length of the target sequence
- S is the length of the source sequence
- E is the embedding dimension
- Ev is the value dimension
- ... can be any arbitrary number of dimensions

Where, The output tensor has the shape:

$$(N, \ldots, H, L, Ev)$$

Masking selectively disables certain keys (e.g., padding tokens or future positions) during attention computation. It ensures that attention focuses only on valid or causally allowed positions before normalization. If a mask is provided, use it to add **-self.eps** to the attention scores for positions that should not be attended to, i.e., where:

$$\textbf{mask} == \text{True}$$

Leave the rest of the attention scores unchanged.

NOTE: The code refers to the input of the softmax function as the *attention scores*. Remember to store these attention scores for the backward pass.

5.3.2 Backward Pass

Implement the `backward` method for the `ScaledDotProductAttention` class. Given the gradient of some arbitrary loss with respect to the output, compute the gradients with respect to Q, K, and V matrices using the chain rule.

The backward pass follows these steps:

1. Gradient with respect to V:
$$\frac{\partial L}{\partial V} = A^T \cdot \frac{\partial L}{\partial O} \tag{13}$$
2. Gradient with respect to attention scores A:
$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial O} \cdot V^T \tag{14}$$
3. Gradient with respect to the scaled dot-product:
$$\frac{\partial L}{\partial S} = \text{softmax\_backward}\left(\frac{\partial L}{\partial A}\right) \tag{15}$$
4. Gradient with respect to Q:
$$\frac{\partial L}{\partial Q} = \left(\frac{\partial L}{\partial S} \cdot \frac{1}{\sqrt{d_k}}\right) \cdot K \tag{16}$$
5. Gradient with respect to K:
$$\frac{\partial L}{\partial K} = \left(\frac{\partial L}{\partial S} \cdot \frac{1}{\sqrt{d_k}}\right)^T \cdot Q \tag{17}$$

Where:

- A: Attention scores computed in the forward pass
- $\frac{\partial L}{\partial O}$: Gradient of loss with respect to output
- $\frac{\partial L}{\partial S}$: Gradient of loss with respect to scaled dot-product
- d_k: Embedding dimension of query and key matrices

5.4 Multi-Head Attention (`mytorch/nn/multi_head_attention.py`)

Multi-head attention allows the model to attend to different positions in the input sequence in parallel, with each head learning to capture distinct linguistic features such as syntactic structure, word sense disambiguation, or long-range dependencies. This enables richer representations that improve the model's ability to generate coherent and contextually appropriate text.

For example, consider the input sequence:

$$[\text{The},\ \text{cat},\ \text{sat},\ \text{down}]$$

When the model is processing the token "sat," different attention heads may capture different relationships:

$$\begin{aligned} \text{Head 1 (subject focus):} &\quad [0.1,\ \mathbf{0.7},\ 0.2,\ 0.0] \\ \text{Head 2 (modifier focus):} &\quad [0.0,\ 0.1,\ 0.2,\ \mathbf{0.7}] \end{aligned}$$

Token	**The**	**cat**	**sat**	**down**
Head 1 (from "sat")	0.1	**0.7**	0.2	0.0
Head 2 (from "sat")	0.0	0.1	0.2	**0.7**

Head 1 allows "sat" to connect strongly with its subject "cat," while Head 2 connects it with the modifier "down." Together, these heads form a richer representation of the sentence.

You will implement the `MultiHeadAttention` class that processes queries, keys, and values through multiple parallel attention heads, using your generic `Linear`, `Softmax`, and `ScaledDotProductAttention` classes.

5.4.1 Forward Pass

Implement the `forward` method for the `MultiHeadAttention` class as shown in Figure 3

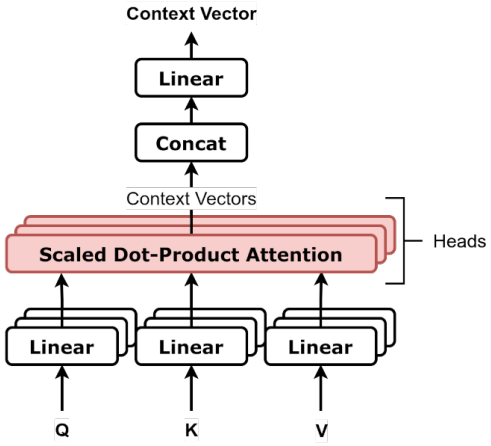


Figure 3: Multi-Head Attention

The multi-head attention mechanism takes three to five inputs:

- Query matrix Q of shape (N, L, E)

- Key matrix K of shape (N, S, E)
- Value matrix V of shape (N, S, E)
- Optional key padding mask of shape (N, S)
- Optional attention mask of shape (L, S)

where:

- N is the batch size
- L is the length of the target sequence
- S is the length of the source sequence
- E is the embedding dimension

The forward pass follows these steps:

1. Project the query, key, and value inputs into the same embedding dimension using the `Linear` layers.

$$Q' = W_q \cdot Q + b_q \tag{18}$$
$$K' = W_k \cdot K + b_k \tag{19}$$
$$V' = W_v \cdot V + b_v \tag{20}$$

2. Implement the `_split_heads` method (see comments) and use it to split the projected query, key, and value matrices into multiple heads. H is the number of attention heads specified in the `MultiHeadAttention` class constructor.

$$Q(N, L, E) \to Q'(N, H, L, E/H) \tag{21}$$
$$K(N, S, E) \to K'(N, H, S, E/H) \tag{22}$$
$$V(N, S, E) \to V'(N, H, S, E/H) \tag{23}$$

3. Implement the `_merge_masks` method (see comments) and use it to merge the key padding mask (N, S) and attention mask (L, S) into a single mask of shape (N, H, L, S).
4. Apply the scaled dot-product attention mechanism to each head:

$$O' = \text{ScaledDotProductAttention}(Q', K', V', \text{mask}) \tag{24}$$

5. Implement the `_concat_heads` method (see comments) and use it to concatenate the attention outputs from all heads.

$$O'(N, H, L, E/H) \to O''(N, L, E) \tag{25}$$

6. Project the concatenated attention outputs back to the original embedding dimension to get the final output:

$$O = W_o \cdot O'' + b_o \tag{26}$$

5.4.2 Backward Pass

Implement the `backward` method for the `MultiHeadAttention` class. Given the gradient of some arbitrary loss with respect to the output, compute the gradients with respect to Q, K, and V.

This should be pretty straighforward. The steps are:

1. Backpropagate through output projection using the gradient of the loss wrt the output.

2. Split gradients into multiple heads using the `_split_heads` method.
3. Backpropagate gradients through scaled dot-product attention for each head.
4. Merge heads of gradients for Q, K, V using the `_concat_heads` method.
5. Backpropagate gradients through input projections.

6 Task 2: Language Modeling using a Causal Transformer 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 former.

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!

6.1 Data-Processing Components

To get going, we will need some basic tools for converting raw text into sequences of the appropriate form. Typical preprocessing pipelines execute the following steps:

1. Load text as strings into memory.
2. Split the strings into tokens (e.g., words or characters).
3. Build a vocabulary dictionary to associate each vocabulary element with a numerical index.
4. Convert the text into sequences of numerical indices.

6.1.1 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.

The input sequence (with ⟨SOS⟩) will be fed into the model, while the output sequence (ending with ⟨EOS⟩) serves as the prediction target. 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.

6.1.2 Dataset Implementation (`hw4lib/data/lm_dataset.py`)

For `HW4P1`, you will be working with a dataset located in the `hw4p1_data` subdirectory inside the `hw4_data` directory. The structure is organized as follows:

```
hw4_data/
└── hw4p1_data/
    ├── train/
    ├── valid/
    └── test/
```

The `train`, `valid`, and `test` folders contain the dataset splits, with each split consisting of text files stored in `.npy` format.

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

- Load the `.npy` text files,
- Tokenize the sequences using a provided tokenizer,
- Prepare two versions of each tokenized sequence:
  - 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 transcripts to batch-uniform lengths.
  - Providing lengths for 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 autoregressive language model.

Run the command in the **Dataset Implementation** section of `HW4P1_nb.ipynb` to test your implementation incrementally.

6.2 Model Implementations

The following sections will guide you through the incremental process of building a Decoder-Only Transformer model. Specifically, we will implement the pre-norm variant of the decoder-only 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 and feedforward layers.
- **Layer:** A complete unit in a Transformer model, consisting of multiple sublayers (e.g., a Self-Attention Decoder Layer that includes self-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 `HW4P1_nb.ipynb`. There are also cells to enable you to visualize some of your implementations.

6.2.1 Masks (`hw4lib/model/masks.py`)

Before implementing the decoder-only transformer, we need to create helper functions for masking, which will be implemented in `hw4lib/model/masks.py`.

NOTE: While it's possible to implement these functions naively using a `for` loop, we recommend using vectorized operations to speed up the implementation, as these functions will be called repeatedly during training. Some useful PyTorch functions for this task include `torch.ones_like`, `torch.arange`, `torch.expand`, `torch.repeat`, `torch.tril`, and various boolean operations (, `>=`, `==`, `!=`, etc.) on tensors. It is possible to implement these functions using PyTorch's built-in functions with just 3-5 lines of code each.

Causal Mask: Implement the `CausalMask` function in `hw4lib/model/masks.py`. This function should take a padded batch of input sequences with shape $(N, T, \dots)$ or (N, T) and return a mask of shape (T, T), where T is the sequence length. The mask should be `True` for positions that should be masked (i.e., positions that should not attend to future tokens), and `False` for positions that are allowed.

For example, if we have a batch of two sequences with variable lengths, padded to the same length (with `0` as the padding token):

$$\text{input} = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Then the output mask will be:

$$\text{mask} = \begin{bmatrix} \text{False} & \text{True} & \text{True} & \text{True} & \text{True} \\ \text{False} & \text{False} & \text{True} & \text{True} & \text{True} \\ \text{False} & \text{False} & \text{False} & \text{True} & \text{True} \\ \text{False} & \text{False} & \text{False} & \text{False} & \text{True} \\ \text{False} & \text{False} & \text{False} & \text{False} & \text{False} \end{bmatrix}$$

NOTE: As with the previous example, while this illustrates the (N, T) case, your function should also support the $(N, T, \dots)$ case.

Pad Mask: Implement the `PadMask` function in `hw4lib/model/masks.py`. This function should take a padded batch of input sequences with shape $(N, T, \dots)$ or (N, T) and a tensor of input lengths with shape $(N,)$, and return a mask of shape (N, T) where N is the batch size and T is the sequence length. The mask should be `True` for padding positions and `False` for non-padding positions.

For example, if we have a batch of two sequences of variable lengths, padded to the same length within the batch (with `0` as the padding token):

$$\text{input} = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

And the lengths tensor:

$$\text{lengths} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Then the output mask will be:

$$\text{mask} = \begin{bmatrix} \text{False} & \text{False} & \text{False} & \text{True} & \text{True} \\ \text{False} & \text{False} & \text{True} & \text{True} & \text{True} \end{bmatrix}$$

NOTE: While this example illustrates the (N, T) case, your function implementation should also support the $(N, T, \dots)$ case.

6.2.2 Positional Encoding (`hw4lib/model/positional_encoding.py`)

Transformers lack an inherent understanding of token order, so positional encodings are added to the input embeddings. These encodings inform the model about the sequential nature of the data and help differentiate between tokens based on their positions.

In this section, you will implement a fixed positional encoding scheme based on sine and cosine functions, as introduced in the original Transformer paper (*Vaswani et al., 2017*).

Given an input representation consisting of d-dimensional embeddings for T tokens, positional encoding generates a matrix $\mathbf{P} \in \mathbb{R}^{T \times d}$ of the same shape. The elements of $\mathbf{P}$ are computed as follows:

$$\mathbf{P}_{t,2i} = \sin\left(\frac{t}{10000^{\frac{2i}{d}}}\right), \quad \mathbf{P}_{t,2i+1} = \cos\left(\frac{t}{10000^{\frac{2i}{d}}}\right)$$

Implement the following methods of the `PositionalEncoding` class in `hw4lib/model/positional_encoding.py`:

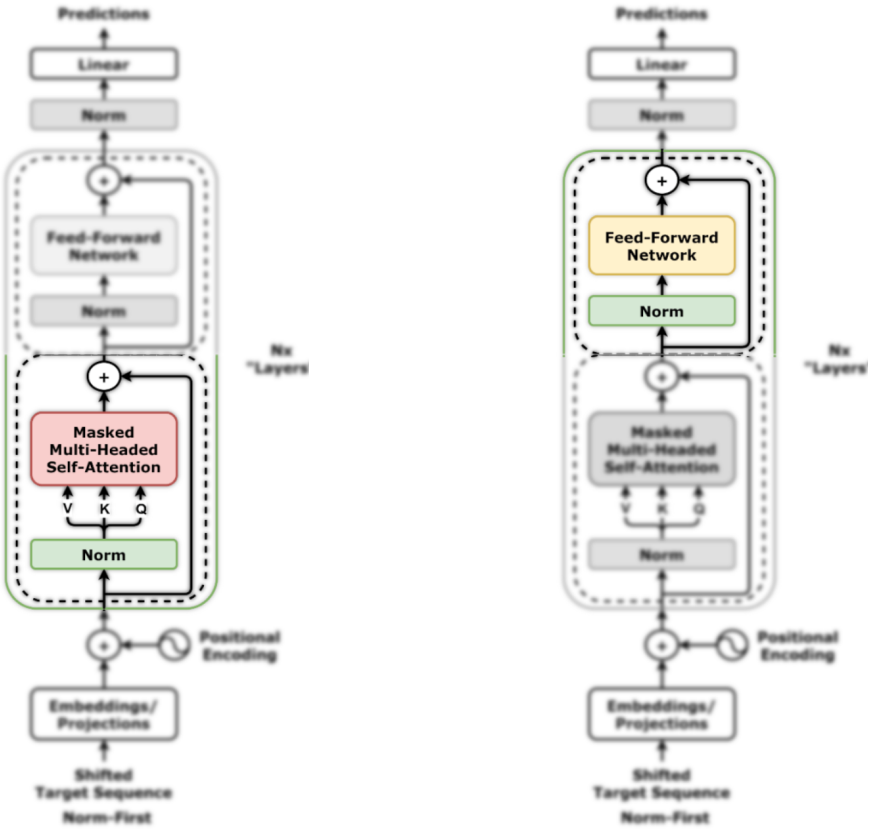
- `create_pe_table`: Constructs the positional encoding matrix $\mathbf{P}$ of shape $(\text{max\_len}, d\_model)$ where:
  - `max_len`: The maximum length of input sequences.
  - `d_model`: The dimensionality of each token embedding.
- `forward`: Adds the positional encoding matrix $\mathbf{P}$ to the input embeddings before feeding them into the model.

NOTE: Remember that the positional encoding matrix should be registered as a buffer (e.g., using `self.register_buffer()`) to ensure it is not treated as a learnable parameter.

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

In the following sections, you will implement the sublayers of the layer's present in the pre-norm variant of the decoder-only transformer. We recommend that you familiarize yourself with PyTorch's `nn.MultiheadAttention` class before implementing the sublayers.

SelfAttentionLayer: Implement the `SelfAttentionLayer` class in `hw4lib/model/sublayers.py`. Refer to Figure 4a (just the colored portion) for the layer architecture. This class will contain the logic for the self-attention mechanism. **Follow the specifications and doc strings in the source file carefully for our tests to work.**



(a) Self-Attention Sublayer Architecture

(b) FeedForward Sublayer Architecture

Figure 4: Transformer Sublayers

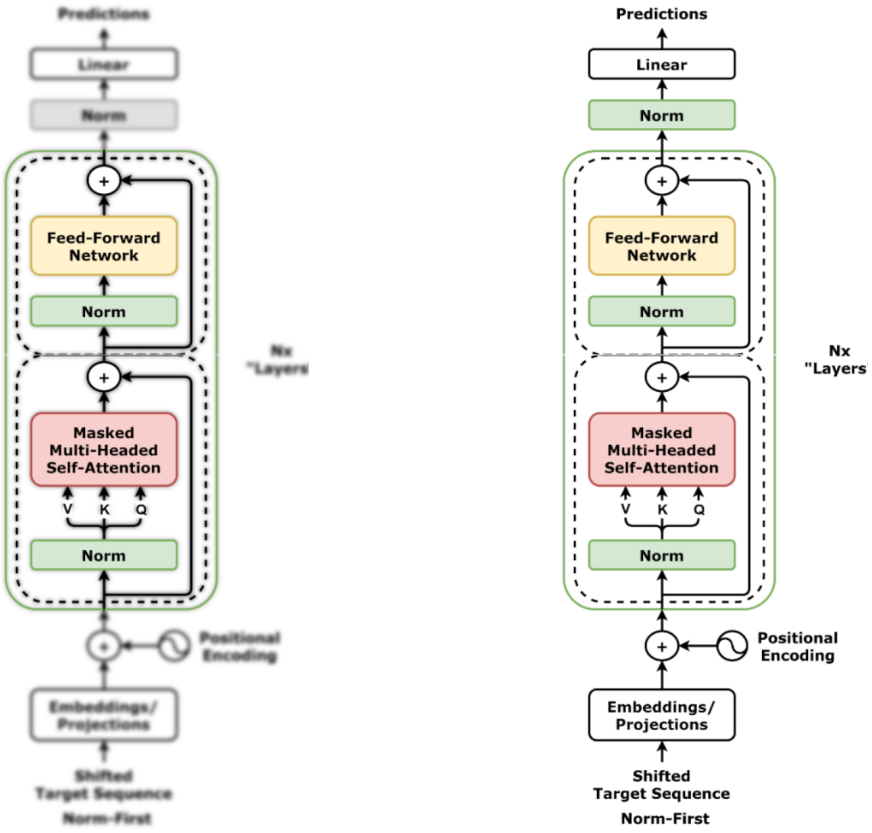
FeedForwardLayer: Implement the `FeedForwardLayer` class in `hw4lib/model/sublayers.py`. Refer to Figure 4b (just the colored portion) for the layer architecture. **Follow the specifications and doc strings in the source file carefully for our tests to work.**

NOTE: Implement the `FeedForwardNetwork` part by setting the `ffn` attribute to a `nn.Sequential` module consisting of two linear layers with a GELU activation and dropout in between, where the input of dimension `d_model` is first projected to a higher-dimensional space `d_ff`, non-linearly transformed, regularized via dropout, and then projected back to `d_model`. The `FeedForwardLayer` class should inherit from `nn.Module` and implement the `forward` method. **You must follow this specification exactly to be compatible with out test suite.**

6.2.4 Transformer Self-Attention Decoder Layer (`hw4lib/model/decoder_layers.py`)

Implement the `SelfAttentionDecoderLayer` class in `hw4lib/model/decoder_layers.py`. Refer to Figure 5a (just the colored portion) for the layer architecture. This class just contains the SelfAttentionLayer and FeedForwardLayer as submodules. **Follow the specifications and doc strings in the source file carefully for our tests to work.**

6.2.5 Decoder-Only Transformer (`hw4lib/model/transformers.py`)



(a) Self-Attention Decoder Layer Architecture

(b) Decoder-Only Transformer Architecture

Figure 5: Decoder Layer & Full Transformer

Finally, integrate all the components you've implemented to construct the Decoder-Only Transformer. Specifically, implement the `DecoderOnlyTransformer` class in `hw4lib/model/transformers.py`. Refer to Figure 5b (focus on the colored portion) for the layer architecture. This class will contain the following submodules:

- A `PositionalEncoding` layer to inject positional information into the input embeddings.

- A `nn.Embedding` layer to convert each token in the input sequence into a sequence of `d_model`-dimensional vectors.
- A user-specified number of `SelfAttentionDecoderLayer`'s.
- A final `nn.Linear` layer to obtain logits over the vocabulary space.
- A final `nn.LayerNorm` for normalization.
- A `nn.Dropout` layer for regularization.

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

6.3 Decoding Implementation

Implement the `generate_greedy` method in the `SequenceGenerator` class. You can find the class definition in `hw4lib/decoding/sequence_generator.py`. This method is designed to generate token sequences using a greedy decoding approach. You can run the command in the `Decoding Implementation` section of the `HW4P1_nb.ipynb` notebook to test your implementation.

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

Pseudocode 1: Greedy Decoding

```
function generate_greedy(x, temperature, repeat_penalty):

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

    for t in range(max_length - current_sequence_length):
        if all sequences are finished:
            break

        logits = score_fn(x)
        logits = apply_repeat_penalty(logits, x, repeat_penalty)
        logits = logits / temperature
        log_probs = log_softmax applied to raw logits

        next_tokens = tokens with highest log_probs for each sequence
        token_scores = log_probs corresponding to next_tokens

        update scores only for unfinished sequences

        append next_tokens to x

        update finished flags to True if EOS token is generated for that sequence

    return x, scores
```

In this pseudocode:

- `x`: Input tensor of shape $(\text{batch\_size}, \text{seq\_len})$ representing token sequences of the same length with no padding.
- `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 $(\text{batch\_size}, \text{vocab\_size})$ for the next token.
- `apply_repeat_penalty`: Function that modifies logits to penalize repeated tokens; input and output shapes

are (batch_size, vocab_size).

NOTE: While it is possible to implement these functions naively using a `for` loop, we recommend using vectorized operations to speed up the implementation, as these functions will be called repeatedly during inference. Some useful PyTorch functions for this task include `torch.zeros`, `torch.zeros_like`, `torch.all`, `torch.where`, `torch.gather`, `torch.cat`, `torch.log_softmax`, and various boolean operations `(|, ==, !=, etc.)` on tensors. It is possible to implement each line of the pseudocode using just 1-2 lines of code.

6.4 Training, Evaluation & Generation

Fantastic work! Just one last implementation task remains before you'll be able to start training your decoder-only transformer for autoregressive language modeling.

6.4.1 LMTrainer

(`hw4lib/trainers/lm_trainer.py`)

The `LMTrainer` class is a partially implemented module responsible for managing the training, validation, and generation loops of the language 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 in `__init__`.**
- **Complete the training loop in `_train_epoch`.**
- **Integrate your greedy decoding implementation in `generate`.**
- **Implement the validation loop in `_validate_epoch`.**
- **Finalize the full training pipeline in `train`.**

Upon completing these tasks, your `LMTrainer` class will be fully functional, enabling training, evaluation, and text generation. Experiment away! **You must achieve a per-character perplexity ≤ 3.5 in order to get points for Task 2.**

7 Submission

Once you have completed all implementations, trained your model, and achieved the required perplexity on the test set, you are ready to submit your assignment. To do so, you will need to create a `handin.tar` file with the following directory structure:

```
handin/
├── mytorch/ ..................... Your implemented modules
├── test_metrics.json ..... Results from evaluation
├── test_generated_results.json ..... Sample text generations
└── model_arch.txt ..... Model architecture summary
```

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

- Run the provided cell once you are satisfied with your current implementation and state to generate the `handin.tar` file.
- Upload the `handin.tar` file to the `HW4P1` assignment on Autolab.

Point Distribution:

- Linear: 5 points
- Softmax: 5 points
- ScaledDotProductAttention: 10 points
- MultiHeadAttention: 10 points
- LanguageModel: 70 points

Good luck !