INF265 Deep Learning – Project 3

Transformer Models

Odin Hoff Gardå

April 2, 2025

Introduction

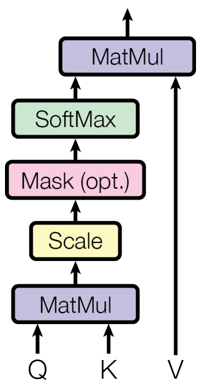
This project is about *transformers*, a type of deep learning model that has revolutionized the field of natural language processing (NLP). The transformer model was introduced in the paper ”Attention is all you need” [5]. The transformer model has several advantages over traditional RNNs and LSTMs, such as better parallelization and the ability to capture long-range dependencies in the data. The original transformer model consists of an encoder-decoder architecture with a self-attention mechanism that allows the model to focus on different parts of the input data when making predictions. The encoder processes the input data and the decoder generates the output data based on the encoded sequence. In this project, we will focus on two other popular versions of the transformer model: an encoder-only model and a decoder-only model. See table 1 for a comparison of different transformer architectures and their common use cases.

Architecture	**Common Uses**	**Example Models**
Encoder-Decoder	Translation, text summarization, speech-to-text, image captioning	Transformer, T5, BART
Encoder-Only	Text classification, sentiment analysis, keyword extraction, extractive summarization	BERT, RoBERTa
Decoder-Only	Text generation (chatbots, stories, articles), code generation, abstractive summarization, translation	GPT, LLaMA

Table 1: Comparison of different Transformer architectures.

Scaled Dot-Product and Multi-Head Attention

The key component of the transformer model is the attention mechanism, which allows the model to focus on different parts of the input data when making predictions. We give the formula for the scaled dot-product attention mechanism and the multi-head attention mechanism below.



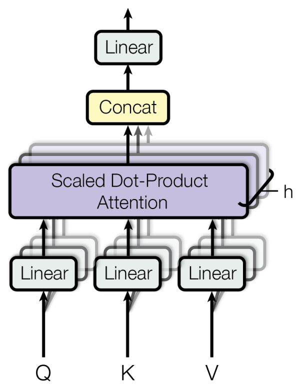


Figure 1: **Left:** The scaled dot-product attention mechanism. **Right:** Multi-head attention mechanism with h heads. The figures are taken from [5].

As shown in fig. 1, given Q (query), K (key) and V (value) matrices, the scaled dot-product attention is computed as

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

Since we are only doing self-attention[1] in this project, we obtain Q, K and V by multiplying the input sequence x with weight matrices W^Q, W^K and W^V, respectively. Multi-head attention is then computed as $\mathrm{MultiHead}(x) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W^O$ where $\mathrm{head}_i = \mathrm{Attention}(xW_i^Q, xW_i^K, xW_i^V)$. The matrices W_i^Q, W_i^K and W_i^V are the weight matrices for the i-th head, and W^O is the output weight matrix. However, instead of having $3h$ separate weight matrices W_i^Q, W_i^K and W_i^V for the different heads, one can use three matrices W^Q, W^K and W^V and perform the computation in parallel before splitting the output into h heads. This is a more efficient way to compute multi-head attention and is how you should implement the multi-head attention mechanism in the first part of the project.

See section A.3 for a list of online resources to learn more about transformers and the attention mechanism.

Formal Requirements

Projects are a compulsory part of the course and should be done in pairs. If you have a good reason for working alone, please contact Nello Blaser before 18th of April. This project contributes 15% to the final grade.

Submission. The deadline is *Friday, 25th of April 23:59* and you should submit your project on MittUiB[2].

Report. Your report should explain your approach to solving the tasks, present your results, and discuss any challenges you encountered. Examples and visualizations are appreciated. Add a short paragraph explaining the division of work between you and your partner (both members will receive the same grade). See section A.2 for more information on what to include in the report.

Submission format. You are expected to deliver two files: a report in PDF format and your code in a ZIP file (please do not put the report in the ZIP file). Please keep the file size reasonable.

Code of conduct. If you use code from other sources, you must provide proper attribution in your code and report. This includes code and text from language models, previous exercises, websites, public repositories, etc. Put simply, be transparent about what you have used and where it comes from.

Late submission policy. All late submissions will recieve a 2 point deduction. In addition, there is a 2-point deduction for every 12-hour period after the deadline. For example, if you submit 00:01 on Saturday, you will receive a 4-point deduction. If you submit 12:01 on Saturday, you will receive a 6-point deduction, and so on. It is not possible to re-take the project.

Project Outline

The project consists of two parts:

Part 1: Encoder-only Model for Sentiment Classification

In this part, you will implement the masked attention mechanism from scratch in PyTorch, and use it to build an encoder-only model for sentiment classification. The model will take as input a sequence of words (tokenized text) and output a sentiment label (positive or negative). You will train the model on a dataset of IMDb movie reviews from [2] and evaluate its performance on a test set. The main focus of this part is to understand how the masked attention mechanism works and how it can be used for text classification.

[1]Cross-attention is used in the encoder-decoder architecture.
[2]Deliver here: `https://mitt.uib.no/courses/51049/assignments/97789`.

Part 2: Decoder-only Model for Text Generation

In this part, you will build a decoder-only model for text generation using the transformer architecture and train it on a subset of the GooAQ dataset [1] consisting of question-answer pairs. Since the model is computationally expensive to train, you will use Google Colab to train the model on a GPU. Furthermore, a simple chatbot web application will be provided so that you can interact with the model and see how it performs in practice. The main focus of this part is to understand how a transformer model can be used for text generation.

⚠ **Start Early!**

This project is time-consuming, so please start early working on the project. The first part is technical as you need to understand the multi-head attention mechanism and how to implement it. The second part is time-consuming since the model is computationally expensive to train and Google Colab has a limited amount of time you can use the GPU each day.

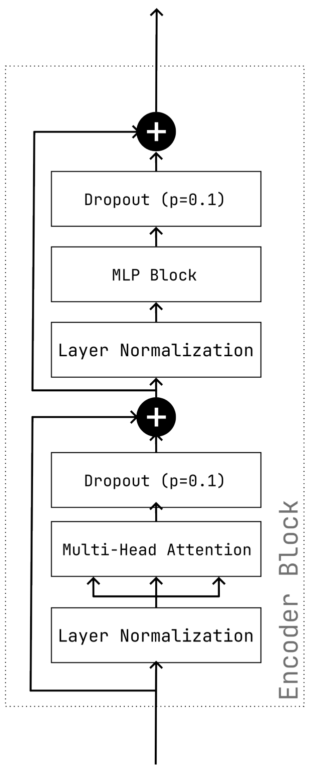
1 Encoder-only Model for Movie Review Sentiment Classification

You will implement an encoder-only model for sentiment classification using the transformer architecture. We are using a word-level tokenizer, splitting the text into words and converting them to integers. The model will take as input a sequence of words (tokenized text) and a mask indicating the padding tokens, and output a number $\hat{p} \in [0, 1]$ representing the sentiment of the text (positive or negative). We train the model with the AdamW optimizer using binary cross-entropy loss.

There are three special tokens in play: `[PAD]`, `[CLS]` and `[UNK]`. The `[PAD]` token is used for padding, the `[CLS]` token is used to indicate the start of the sequence, and the `[UNK]` token is used to represent unknown words. The `[UNK]` token is output by the tokenizer when a word is not in the vocabulary. The input sequence is prepended with the `[CLS]` token and padded with the `[PAD]` token as needed. In the last part of the model, we use the final embedding of the `[CLS]` token to make the final prediction using a simple linear layer with sigmoid activation.

The main steps of our encoder-only model are as follows:

1. The text is tokenized into a sequence of integers using a word-level tokenizer.
2. The tokenized sequence is prepended with the special token `[CLS]` and padded or truncated as needed.
3. Positional encodings (see [5, Section 3.5]) are added to the tokenized sequence to provide the model with information about the position of the tokens in the sequence.
4. The tokenized sequence is passed through an embedding layer converting the tokens to vectors.
5. The embedded sequence together with a padding mask is passed through a stack of encoder blocks (see fig. 2).
6. The final embedding of the `[CLS]` token is passed through a linear layer with sigmoid activation to make the final prediction.



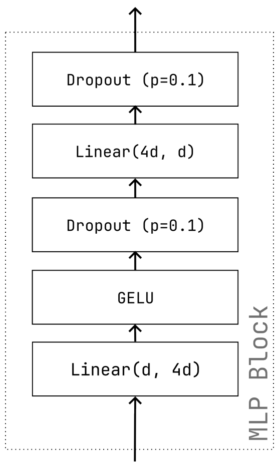


Figure 2: **Left:** Encoder block with multi-head attention and a feed-forward network (MLP). Note the dropout layers, layer normalization layers and the skip connections. **Right:** The MLP block in the encoder block with linear layers and a GELU activation function. We use a fixed hidden size of $4d$ were d is the input dimension.

1.1 Tasks

Everything for this task should be implemented in the notebook `imdb_sentiment_encoder.ipynb`. The tasks for this part of the project are as follows:

1. Complete the code for the `IMDBDataset` class in the provided notebook (1.4). The `__getitem__` method should return a tuple of the tokenized review and the sentiment label. You need to prepend the special tokens `[CLS]` and perform padding and/or truncation of the tokenized review as needed.
2. Implement the function `create_mask` (2.1) generating the padding mask for the input sequence. The output mask is a boolean tensor with the same shape as the input sequence, where **`True`** indicates a padding token.
3. Implement the multi-head attention mechanism *from scratch* in PyTorch by completing the code for the `MultiheadAttention` class (2.2). This is the most difficult part.
4. Implement the encoder block by completing the code for the `EncoderBlock` class (2.3). Use fig. 2 as a reference for the architecture of the encoder block.
5. Implement positional encoding by completing the code for the `PositionalEncoding` class (2.4). See [5, Section 3.5] for the formula for positional encodings. Compute the positional encoding values in the `__init__` method, store them using `self.register_buffer`, and apply the positional encodings in the `forward` method. You should not need any loops to implement this.
6. Train the encoder-only model on the IMDb movie reviews dataset for at least three epochs. Each epoch should take around 5 to 15 minutes training on CPU. If training loss has not decreased halfway through the first epoch, you should stop the training and try to debug the model. You should expect a validation accuracy around 80% to 85% after three epochs if you use the default hyperparameters.
7. Evaluate the model on the test set to estimate the performance of your model on unseen reviews. The code is provided in the notebook (2.6). You can modify the code to print the confusion matrix and misclassified reviews to better understand the model's performance.

8. Go to `https://imdb.com` (or another movie review website), and pick some reviews of your favorite movies. Use the model to predict the sentiment of the reviews and compare the predictions with your own sentiment.

> ⚠ You have to implement the masked attention mechanism from scratch in PyTorch. Off-the-shelf PyTorch classes including `MultiheadAttention` and `TransformerEncoder` from `torch.nn` are not allowed to be used in this part of the project. If in doubt, ask us.

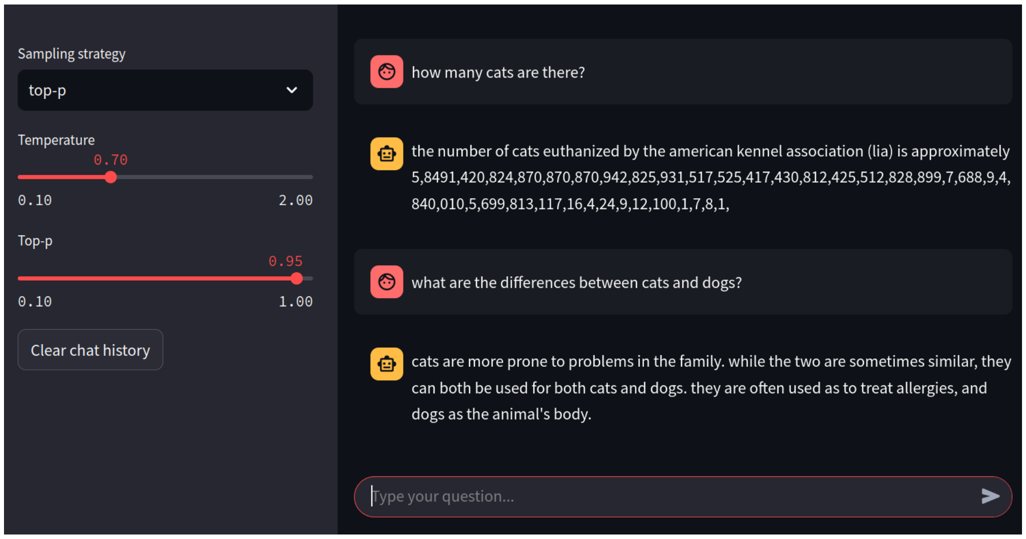
1.2 Tips

- Spend some time to understand how the multi-head attention mechanism works before implementing it.
- Use `torch.nn.Linear` to define the linear transformations for the query, key, and value matrices in the `MultiheadAttention` class.
- Do *not* use loops when implementing the masked attention mechanism. Instead, use matrix operations to compute the attention scores. This is an exercise in efficient tensor manipulation. Useful methods include `torch.matmul`, `Tensor.transpose`, `Tensor.unsqueeze`, `Tensor.view` and `Tensor.reshape`.
- You only need three linear layers for the query, key, and value matrices in the `MultiheadAttention` class, respectively (and one more for the output linear layer). See the following SE post: `https://datascience.stackexchange.com/q/80660` if you are unsure how to implement this.
- For masking, you can use the `Tensor.masked_fill` method to set the attention scores to `float("-inf")`, i.e., $-\infty$ for padding tokens before applying the softmax function.
- Since training is computationally expensive, you do not need to do any hyperparameter tuning. You can use the provided hyperparameters in the notebook. If it works as expected, move on to the second part.

2 Decoder-only Model for Text Generation

In this task, you will build a decoder-only model for text generation using the transformer architecture. This is similar to the (early) GPT models [3, 4], where the model generates text one token at a time based on the input sequence using different sampling strategies. This idea of using previous predictions to generate the next is called *auto-regression*. As in the previous task, we still use a padding mask so that the model does not attend to padding tokens in the input sequence. However, we also use a causal mask to ensure that the model only attends to previous tokens when making predictions.

We will train the model on a subset of the GooAQ dataset [1] consisting of $859,765$ question-answer pairs. The dataset is available from Hugging Face Datasets as `odinhg/gooaq-subset`. After finishing the training, you can interact with the model using a provided chatbot web application built with the `streamlit` library. The performance will not be amazing, but you will hopefully get more or less grammatically correct answers (see fig. 3 for an example).



how many cats are there?

the number of cats euthanized by the american kennel association (lia) is approximately 5,8491,420,824,870,870,870,942,825,931,517,525,417,430,812,425,512,828,899,7,688,9,4, 840,010,5,699,813,117,16,4,24,9,12,100,1,7,8,1,

what are the differences between cats and dogs?

cats are more prone to problems in the family. while the two are sometimes similar, they can both be used for both cats and dogs. they are often used as to treat allergies, and dogs as the animal's body.

Figure 3: The chatbot providing helpful answers to our questions.

Since training the model is computationally expensive, you will use Google Colab to train the model on a GPU. You will be provided with a notebook that you can use to train the model. See section A.1 for detailed instructions.

Code Template Overview

A code template is provided for this part as well.

Files:

- `tokenizer.py`: Code for defining and training the BPE tokenizer.
- `dataset.py`: The dataset class providing tokenized sequences, target sequence (labels) and padding mask.
- `model.py`: The decoder block and the decoder-only model.
- `config.py`: Configuration settings. Simply use `from config import config` to import the configuration settings and access the settings using `config.setting_name`, e.g., `config.batch_size`.
- `train.py`: Training function for training the model.
- `inference.py`: Inference logic for generating text using the trained model.
- `chatbot.py`: Chatbot application for interacting with the trained model.
- `gpu_training_colab_notebook.ipynb`: Notebook for training the model on Google Colab.
- `utils.py`: Miscellaneous utility functions.

Tips:

- Make sure to understand the provided code and how the different parts of the model are connected.
- Most script files can be run and contain some simple tests for sanity checking your implementation by inspecting the output. Feel free to add your own test cases to ensure that your implementation is working as expected before training the full model.
- In `config.py`, uncomment the last part to use a tiny model for testing and debugging locally on CPU. Remember to comment it out before training the full model on Google Colab.

The Tokenizer

For this model, we use a byte-pair encoding (BPE) tokenizer to tokenize the text. The BPE tokenizer is a subword tokenizer that splits words into subword units. We use the following special tokens: `[UNK]`, `[SEP]`, `[END]`, and `[PAD]`. The `[UNK]` token is used to represent unknown words, the `[SEP]` token is used to separate the question and answer sequences, the `[END]` token is used to indicate the end of the sequence, and the `[PAD]` token is used for padding.

The code is already impelemented in the `tokenizer.py` file. You can run the code to train the BPE tokenizer on the GooAQ dataset. The tokenizer will be saved to disk with the filename specified in the configuration file `config.py`. There is no need to modify the code in the `tokenizer.py` file, but you should try to understand how the BPE tokenizer works and how to use it to tokenize text. A brief explanation of the BPE tokenizer should be included in your report.

2.1 Tasks

2.1.1 Dataset

In the file `dataset.py`, implement the `__getitem__` method of the class `QADataset`. The method should return a dictionary with the keys `"source_sequence"`, `"target_sequence"`, and `"key_padding_mask"` corresponding to the source sequence, target sequence, and key padding mask, respectively. The source sequence is the tokenized question-answer text (with the `[SEP]` token separating the question and answer and the `[END]` token at the end with padding as needed), and the target sequence is the source sequence shifted one position to the right. The key padding mask is a boolean tensor indicating the padding tokens in the source sequence. Note that we need to set the labels in the target sequence corresponding to padding tokens to -100 so that the `CrossEntropyLoss` ignores them (this is the default value for the `ignore_index` argument of `CrossEntropyLoss`).

In other words, the main steps of the `__getitem__` method are as follows:

1. Tokenize the question and answer.
2. Concatenate the tokens with the `[SEP]` between.
3. Add padding/truncate as needed and add the `[END]` token to the end.
4. Create the source and target sequence.
5. Create the padding mask as a boolean tensor with `True` where the padding tokens are in the source sequence and `False` elsewhere.
6. Set the target sequence to -100 in all position corresponding to `[PAD]` tokens to make the loss function ignore these positions.

Tips: You can start by creating a sequence of length `max_length + 1` and then extract the source and target sequences from this sequence. Make sure to test your implementation thoroughly before moving on to the next task as it is easy to make subtle mistakes here!

2.1.2 Model

For this part, you will implement a GPT-2 style decoder block that will be used in the decoder-only transformer model. See fig. 4 for the architecture of the decoder block and its MLP block. The decoder-only model consists of a stack of decoder blocks followed by a linear layer. Taking softmax of the model output gives a probability distribution over the vocabulary for each token in the sequence which can be used to choose the next token when generating text.

Decoder Block

Implement the decoder block class `DecoderBlock` in the file `model.py`. Use the `torch.nn.MultiheadAttention` class for the masked multi-head attention mechanism in the decoder block.

Tips: Remember to set `batch_first=True` in the constructor of the `MultiheadAttention` class. When calling the forward method, use the `key_padding_mask` argument to mask the padding tokens in the input sequence and the `attn_mask` argument to apply the causal mask. Furthermore, set `need_weights=False` and `is_causal=True` when calling the forward method.

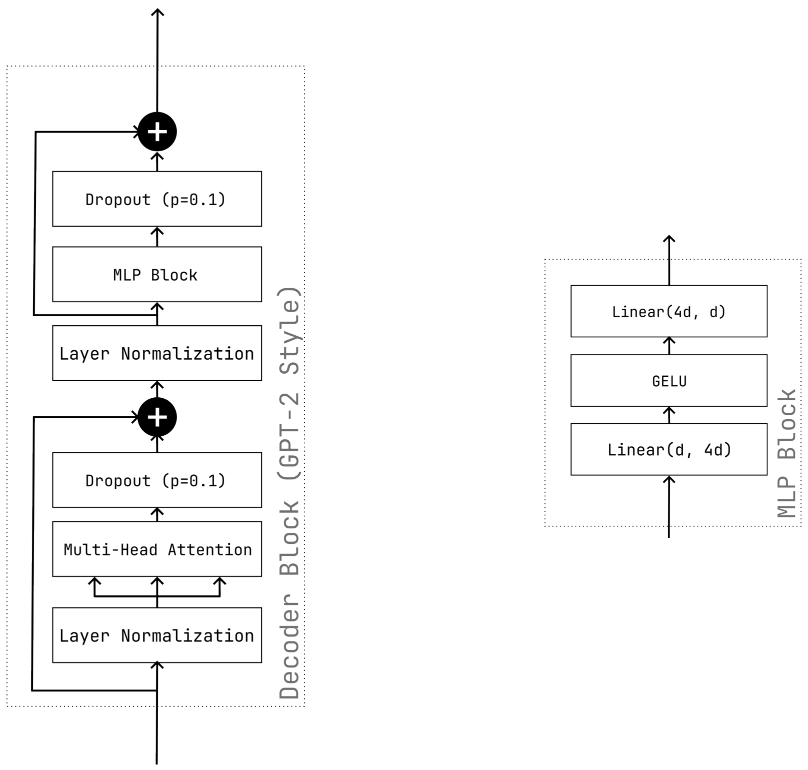


Figure 4: **Left:** Decoder block with masked multi-head attention and feed-forward network. Note the dropout layers, layer normalization layers and the skip connections. **Right:** The MLP block in the decoder block with linear layers and a GELU activation function. We use a fixed hidden size of $4d$ were d is the input dimension.

Positional Encoding

Implement the positional encoding class `PositionalEncoding` in the file `model.py`. You can reuse your code from the encoder-only model for this task, but make sure to move the positional encoding values to the same device as the input sequence in the `forward` method.

Causal Mask

Implement the method `generate_causal_mask` of the class `TransformerModel` in the file `model.py`. The method takes a positive integer `seq_length` as input and returns a tensor of shape `(seq_length, seq_length)` with the causal mask. For example, if `seq_length=3`, the output tensor should be of shape `(3, 3)` and have the following values:

```
tensor([[False,  True,  True,  True],
        [False, False,  True,  True],
        [False, False, False,  True],
        [False, False, False, False]])
```

You can use `torch.ones` and `torch.triu` with `diagonal=1` to generate the causal mask.

Decoder-only Transformer Model

The decoder-only model is implemented in the class `TransformerModel` in the file `model.py`. The model consists of an embedding layer, a stack of decoder blocks, and a linear classification layer. This class in already implemented, but make sure to understand how the model works and how the decoder blocks are stacked together. You should explain the architecture of the model in your report (see section A.2).

2.1.3 Training

Train your model on a GPU using Google Colab. See section A.1 for detailed instructions. The training function is defined in the the file `train.py`.

The training script uses automatic mixed precision (AMP) and `torch.compile` to speed up training. The AdamW optimizer is used with a learning rate set in the config file. Every 500 steps, the model and optimizer states are saved to disk so that you can resume training if needed. If you want to re-train everything from scratch, make sure to delete the `temp` folder first. With the default configuration (which you probably should stick to), expect your model to have around 36 million parameters and take around 30 minutes to train one epoch on a T4 GPU. Training loss should decrease quickly to around $5 - 6$. For simplicity, we do not use a validation set in this task.

After training, you simply download the model checkpoint and the tokenizer file from Google Drive and put them in your local `temp` folder.

2.1.4 Inference

In this task, we work in the file `inference.py` and implement the logic for generating text using the trained model. You will implement two different sampling strategies: greedy sampling and top-p (nucleus) sampling. Given a question, we tokenize it and append the `[SEP]` token to the end of the sequence. We then pass the tokenized question through the model and look at the output logits for the last token. Depending on the sampling strategy, we choose the next token in the sequence and append it to the input sequence. We continue this process until we reach the maximum sequence length or the `[END]` token is predicted.

The sampling code is already implemented in the function `sample_sequence` in the file `inference.py`. Before implementing the sampling strategies, make sure you understand the `sample_sequence` function.

1. Greedy Sampling. In greedy sampling, we choose the next token in the sequence by selecting the token with the highest probability (or logit). Implement the greedy sampling strategy in the `greedy_sampling` function in the file `inference.py`.

2. Top-p (Nucleus) Sampling. In top-p sampling, we choose the next token in the sequence by sampling from the top p most likely tokens. We also use a temperature parameter $\tau > 0$ to control the the sharpness of the distribution generated by the softmax function. A higher temperature will make the distribution more uniform, while a lower temperature will make the distribution more peaked. The softmax with temperature is computed as

$$\mathrm{softmax}(z)_i = \frac{\exp(z_i/\tau)}{\sum_{j=1}^{n} \exp(z_j/\tau)},$$

and can be implemented in PyTorch with `torch.nn.functional.softmax` and dividing the logits by τ before applying the softmax function.

Let us demonstrate top-p sampling with a simple example where we assume our vocabulary size to be 8, $p = 0.95$ and $\tau = 0.7$. Suppose we pass a tokenized sequence through the model and get the following output logits for the last token: $[-1, 2, -3, 3, -3, 0, 2, 1]$. We first apply the softmax with temperature to the logits to get the corresponding probabilities $[0.002, 0.154, 0.000, 0.643, 0.000, 0.009, 0.154, 0.037]$. We then sort the probabilities in descending order and calculate the cumulative distribution function (CDF) to find the tokens needed to reach the top $p = 0.95$ probability mass. In this case, we have $0.643 + 0.154 + 0.154 = 0.951$, so we choose the tokens corresponding to the indices $[3, 1, 6]$ and randomly sample from this set of tokens to get the next token in the sequence.

Implement the top-p sampling strategy in the `top_p_sampling` function in the file `inference.py`.

In your report, you should discuss the differences between the two sampling strategies and how they affect the generated text.

2.1.5 Experimenting with the Chatbot

Install the `streamlit` library by running `conda install -c conda-forge streamlit` in your conda environment. Start the chatbot application by running `streamlit run chatbot.py` in the terminal. The chatbot application should open in your browser. Note that the first prediction might take some time since the model is loaded into memory. You can choose sampling strategy and parameters in the sidebar.

A Appendices

A.1 How to train your model on Google Colab

To train your model on Google Colab, carefully follow the instructions below. If you do not have a Google account, you need to create one before continuing.

> You only have a limited amount of time to train the model on Google Colab each day. The runtime will be disconnected after a certain amount of time, and you will lose all your progress. The provided training script saves the model to Google Drive regularly so that you can resume training if this happens. Make sure to have the notebook open as you might have to complete a CAPTCHA to stay connected every now and then. If you are not training, make sure to disconnect the runtime to not use up your available time.

A.1.1 Upload Required Files to Google Drive

The notebook will mount your Google Drive and import the required files from there. It will also save the model checkpoint and the tokenizer file to your Google Drive.

1. Go to `https://drive.google.com/` and sign in with your Google account.
2. Go to `My Drive` and create folder with the name `inf265_project_3` (the name is important).
3. Upload the files `tokenizer.py`, `dataset.py`, `model.py`, `config.py`, `utils.py` and `train.py` from your computer to the folder `inf265_project_3` in Google Drive.

A.1.2 Open the Notebook in Google Colab and Connect to a GPU

1. Go to `https://colab.research.google.com/` and sign in with your Google account.
2. From the `File` menu, select `Upload notebook` and upload `gpu_training_colab_notebook.ipynb`.
3. Click on the `Runtime` menu and select `Change runtime type`.
4. Under `Runtime type`, select `Python 3` and `T4 GPU` under `Hardware accelerator`.
5. Click `Save` and then click `Connect` to connect to the runtime.

Once you are connected, you can run the cells in the notebook to train the tokenizer and your model.

A.1.3 Download the Model Checkpoint and Tokenizer File

After training, go to the folder `inf265_project_3` in your Google Drive and download the model checkpoint and the tokenizer file from the `temp` folder. Put them in the `temp` folder in your local project directory. You now have a trained model and tokenizer that you can use for inference.

A.2 Writing the Report

Here, we list some points you can include in your report for this project. You do not have to follow this structure exactly, but it can be a good starting point for writing your report. The list is not necessarily exhaustive, and you can include additional information if you find it relevant. Using ChatGPT (or similar services) to help you understand concepts is fine, but make sure what you write is actually correct and use your own words. Keep the report concise and to the point.

Part 1: Encoder-only Model for Movie Review Sentiment Classification

- An explanation of the pre-processing and tokenization of the IMDb movie reviews.
- A brief description of the (encoder-only) transformer model and the attention mechanism.
- An overview of your approach to solving the tasks.
- Results: Training and validation loss and accuracy after each epoch, and the test accuracy.

- A brief discussion of the results and any challenges you encountered.
- The predictions of the model on the IMDb movie reviews and your own sentiment of the reviews. Any interesting observations?

Part 2: Decoder-only Model for Text Generation

- Explain your implementation of the decoder block.
- Give a brief description of the decoder-only transformer model. Discuss the architecture of the model, including the role of causal (attention) and padding masking, positional encodings and the embedding layer.
- A few sentences on how the BPE tokenizer works (does not need to be detailed, but should give an idea of how it works, and compare to word-level tokenization).
- Results: Training loss and any interesting observations during training.
- A brief discussion of the results and any challenges you encountered.
- The predictions of the model using the chatbot application. Any interesting answers? How does the sampling strategy affect the answers? Any sign of overfitting? ”Hallucinations”?
- How would you improve the model given more time and resources? For example, more data, more compute, more time for training, etc.

A.3 Resources

Here are some resources that you might find helpful for this project.

Description	**URL**
Transformers (how LLMs work) explained visually (3Blue1Brown, YouTube)	`https://youtu.be/wjZofJX0v4M`
Attention in transformers, step-by-step (3Blue1Brown, YouTube)	`https://youtu.be/eMlx5fFNoYc`
Decoder-Only Transformers: The Workhorse of Generative LLMs (Blog post)	`https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse`
How does the (decoder-only) transformer architecture work? (AI StackExchange)	`https://ai.stackexchange.com/a/40180`
Explaining the role of masking in attention layers (Stack Overflow)	`https://stackoverflow.com/a/59713254`
Attention is all you need (original transformers paper)	`https://arxiv.org/pdf/1706.03762`
The Illustrated GPT-2 (Blog post)	`https://jalammar.github.io/illustrated-gpt2/`

References

[1] Daniel Khashabi et al. “GooAQ: Open question answering with diverse answer types”. In: *arXiv preprint arXiv:2104.08727* (2021).

[2] Andrew Maas et al. “Learning word vectors for sentiment analysis”. In: *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 2011, pp. 142–150.

[3] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).

[4] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[5] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).