THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RELAXED SYSTEM LAB

Transformer Architecture and Training Objectives

COMP4901Y

Binhang Yuan

Overview

RELAXED SYSTEM LAB

- What is a language model?
- Tokenization:
  - How do we represent language to machines?
- Model architecture:
  - Transformer architecture, which is the main innovation that enabled large language models.
- Training objectives:
  - How are large language models (LLM) trained?

Language Model

What Is a Language Model?

- The classic definition of a ***language model (LM)*** is <u>a probability distribution over sequences of tokens</u>.
- Suppose we have a vocabulary $\mathcal{V}$ of a set of tokens.
- A language model P assigns each sequence of tokens $x_1, x_2, \dots, x_L \in \mathcal{V}$ to a probability (a number between 0 and 1): $p(x_1, x_2, \dots, x_L) \in [0,1]$.
- The probability intuitively tells us how "good" a sequence of tokens is.
  - For example, if the vocabulary is $\mathcal{V} = \{\text{ate}, \text{ball}, \text{cheese}, \text{mouse}, \text{the}\}$, the language model might assign:

$$p(\text{the}, \text{mouse}, \text{ate}, \text{the}, \text{cheese}) = 0.02$$
$$p(\text{the}, \text{cheese}, \text{ate}, \text{the}, \text{mouse}) = 0.01$$
$$p(\text{mouse}, \text{the}, \text{the}, \text{chesse}, \text{ate}) = 0.0001$$

Language Model Generation

RELAXED SYSTEM LAB

- A language model P takes a sequence and returns a probability to assess its goodness.
- We can also generate a sequence given a language model.
- The purest way to do this is to sample a sequence $x_{1:L}$ from the language model P with probability equal to $p(x_{1:L})$ denoted:

$$x_{1:L} \sim P$$

Autoregressive Language Models

- A common way to write the joint distribution $p(x_{1:L})$ of a sequence to $x_{1:L}$ is using the *chain rule of probablity*:

$$p(x_{1:L}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \dots p(x_L|x_{1:L-1}) = \prod_{i=1}^{L} p(x_i|x_{1:i-1})$$

- In particular, $p(x_i|x_{1:i-1})$ is a conditional probability distribution of the next token x_i given the previous tokens $x_{1:i-1}$.
- An autoregressive language model is one where each conditional distribution $p(x_i|x_{1:i-1})$ can be computed efficiently (e.g., using a feedforward neural network).

Tokenization

Tokenization

- Recall: language model P is a probability distribution over a sequence of tokens where each token comes from some vocabulary $\mathcal{V}$, e.g.,:

  [I, love, cats, and, dogs]

- Natural language doesn't come as a sequence of tokens, but as just a string (concretely, sequence of Unicode characters):

  I love cats and dogs

- A *tokenizer* converts any string into a sequence of tokens:

  I love cats and dogs $\Longrightarrow$ [I, love, cats, and, dogs]

Split by Space

RELAXED
SYSTEM LAB

- The simplest solution is to do: text.split(' ')
- This doesn't work for languages such as Chinese, where sentences are written without spaces between words:
  - 我今天去了商店: [I went to the store today.]
- Then there are languages like German that have long compound words:
  - Abwasserbehandlungsanlange: [Wastewater treatment plant]
- Even in English, there are hyphenated words (e.g., father-in-law) and contractions (e.g., don't), which should get split up.

What Makes a Good Tokenization?

- We don’t want too many tokens:
  - The extreme case characters or bytes;
  - The sequence becomes difficult to model.
- We don’t want too few tokens:
  - There won’t be parameter sharing between words (e.g., should mother-in-law and father-in-law be completely different)?
  - This is especially problematic for morphologically rich languages (e.g., Arabic, Turkish, etc.).
- Each token should be a linguistically or statistically meaningful unit.

Some Encoding Methods

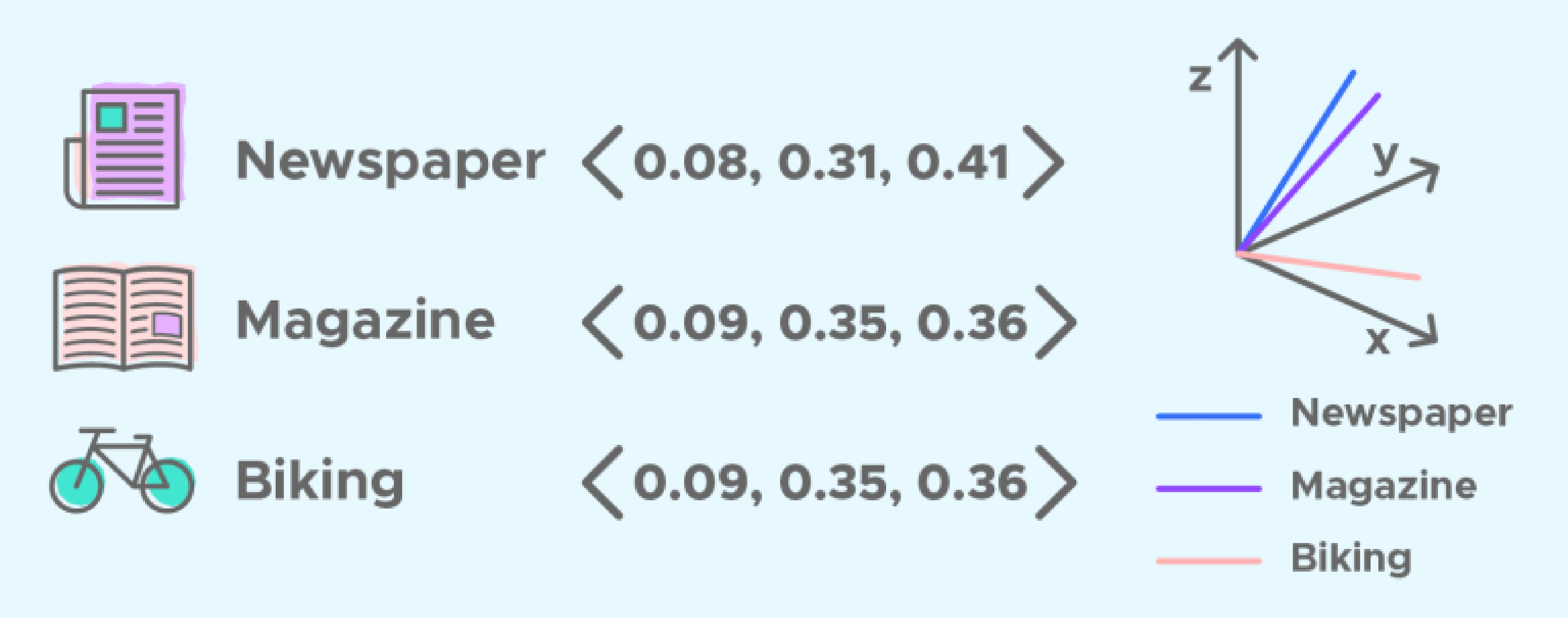
RELAXED SYSTEM LAB

- Byte pair encoding (BPE)
    - Start with each character as its own token and combine tokens that co-occur a lot.
    - https://arxiv.org/pdf/1508.07909.pdf
- Unigram model (SentencePiece):
    - Rather than just splitting by frequency, a more "principled" approach is to define an objective function that captures what a good tokenization looks like.
    - https://arxiv.org/pdf/1804.10959.pdf

Representation: Word as Vectors

RELAXED SYSTEM LAB

- Tokens can be represented as number index:
  [I, love, cats, and, dogs] $\Longrightarrow$ [328, 793, 3989, 537, 3255, 269]
- But indices are also meaningless.
- Represent words in a vector space
  - Vector distance $\Longrightarrow$ similarity。



LLM Architecture

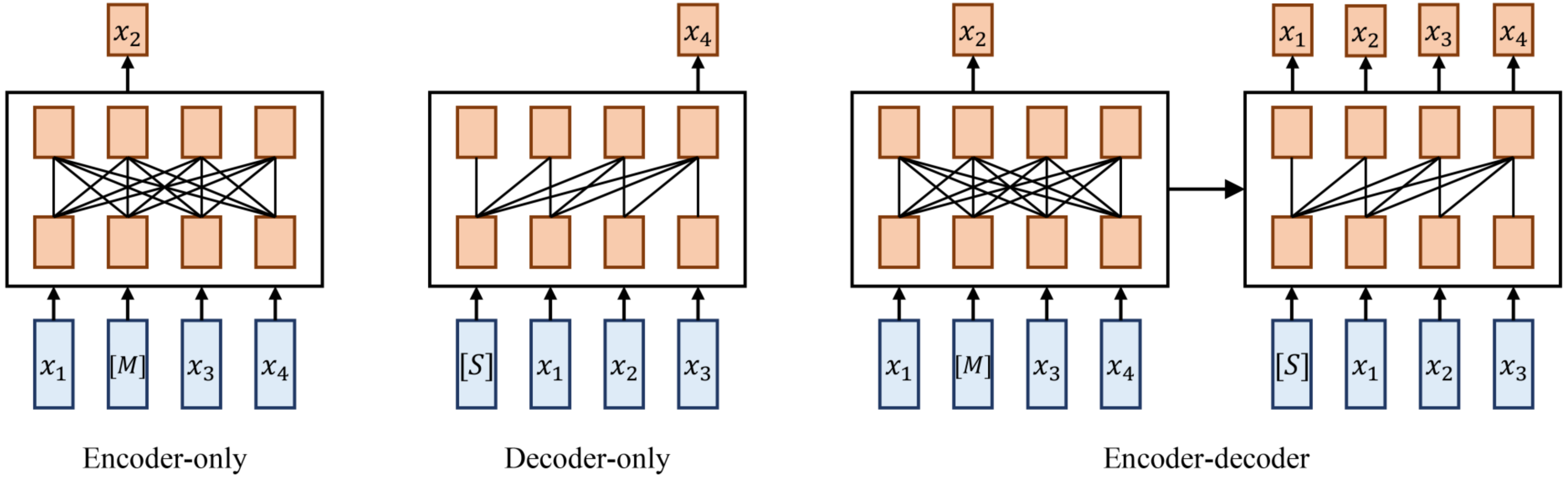
Contextual Embeddings

RELAXED SYSTEM LAB

- Language model:
  - Associate a sequence of tokens with a corresponding sequence of contextual embeddings.
- Embedding function (analogous to a feature map for sequences):
  - $\emptyset: \mathcal{V}^L \rightarrow \mathbb{R}^{L\times D}$
  - A token sequence $x_{1:L}[x_1, x_2, \dots, x_L] \in \mathcal{V}^L$
  - Map to $\emptyset(x_{1:L}) \in \mathbb{R}^{L\times D}$
- For example, if $D = 2$:
  - [I, love, cats, and, dogs] $\Longrightarrow$ [328, 793, 3989, 537, 3255, 269] $\Longrightarrow \begin{bmatrix} (0.2, 0.3) \\ (0.8, 0.7) \\ (0.2\ 0.1) \\ (0.0,0.7) \\ (0.1, 0.0) \\ (0.1, 0.4) \end{bmatrix}$

Types of language models

- Encoder-only models (BERT, RoBERTa, etc.)
- Encoder-decoder models (BART, T5, etc.)
- **Decoder-only models** (GPT-3, Llama-2 etc.)



Encoder-only Models

- Encoder-only models produce contextual embeddings but cannot be used directly to generate text:

$$x_{1:L} \Rightarrow \emptyset(x_{1:L})$$

- These contextual embeddings are generally used for classification tasks (sometimes boldly called natural language understanding tasks).
  - Example: sentiment classification: [[CLS],the,movie,was,great] ⇒ positive.
- Pros:
  - Contextual embedding for x_i can depend bidirectionally on both the left context ($x_{1:i-1}$) and the right context ($x_{i+1:L}$).
- Cons:
  - Cannot naturally generate completions.
  - Requires more ad-hoc training objectives (masked language modeling).

Decoder-only Models

RELAXED SYSTEM LAB

- Decoder-only models are our standard autoregressive language models.
- Given a prompt $x_{1:i}$ produces both contextual embeddings and a distribution over next tokens x_{i+1}, and recursively, over the entire completion $x_{i+1:L}$:

$$x_{1:i} \Rightarrow \emptyset(x_{1:i}), p(x_{i+1}|x_{1:i})$$

- Example: text autocomplete
  - [[CLS],the,movie,was]⇒great
- Pro:
  - Can naturally generate completions.
  - Simple training objective (maximum likelihood).
- Con:
  - Contextual embedding for x_i can only depend **unidirectionally** on both the left context ($x_{1:i-1}$).

Encoder-decoder Models

- Encoder-decoder models can be the best of both worlds: they can use bidirectional contextual embeddings for the input $x_{1:L}$ and can generate the output $y_{1:L}$:

$$x_{1:L} \Rightarrow \emptyset(x_{1:L}), p(y_{1:L} | \emptyset(x_{1:L}))$$

- Example: table-to-text generation
  - [name,:,Clowns,|,eatType,:,coffee,shop]⇒[Clowns,is,a,coffee,shop].
- Pro:
  - Can naturally generate outputs.
- Con:
  - Requires more ad-hoc training objectives.

EmbedToken

RELAXED SYSTEM LAB

- Convert sequences of tokens into sequences of vectors.
- EmbedToken does exactly this by looking up each token in an embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times D}$, a parameter that will be learned from data
- $\text{EmbedToken}(x_{1:L}: \mathcal{V}^L) \to \mathbb{R}^{L \times D}$:
  - Turns each token x_i in the sequence $x_{1:L}$ into a vector;
  - Return $\left[E_{x_1}, E_{x_2}, \ldots, E_{x_L}\right]$.
- These are *context-independent* word embeddings.
- Next the TransformerBlock(s) takes these context-independent embeddings and maps them into contextual embeddings.

TransformerBlock

- TransformerBlock(s) takes these context-independent embeddings and maps them into contextual embeddings.
- TransformerBlocks$(x_{1:L}: R^{L\times D}) \rightarrow \mathbb{R}^{L\times D}$:
  - Process each element x_i in the sequence $x_{1:L}$ with respect to other elements.
- TransformerBlock(s) are the building blocks of decoder-only (GPT-2, GPT-3), encoder-only (BERT, RoBERTa), and decoder-encoder (BART, T5) models.

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

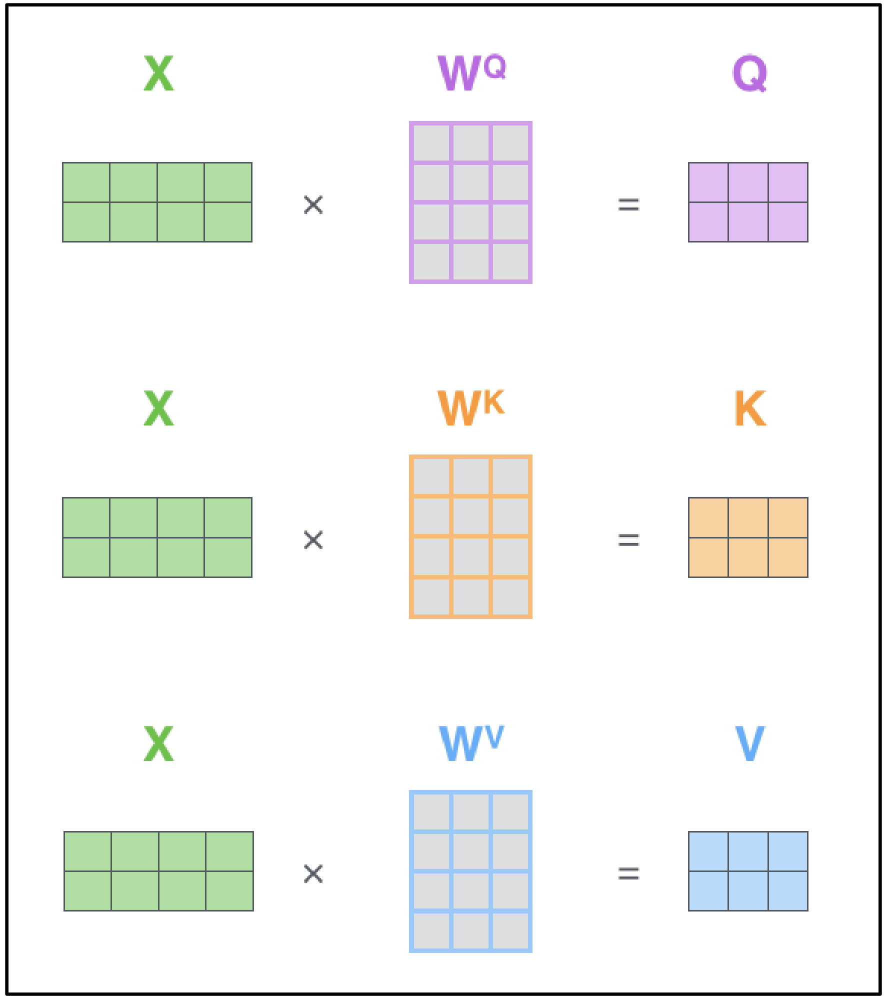
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Attention Mechanism-1

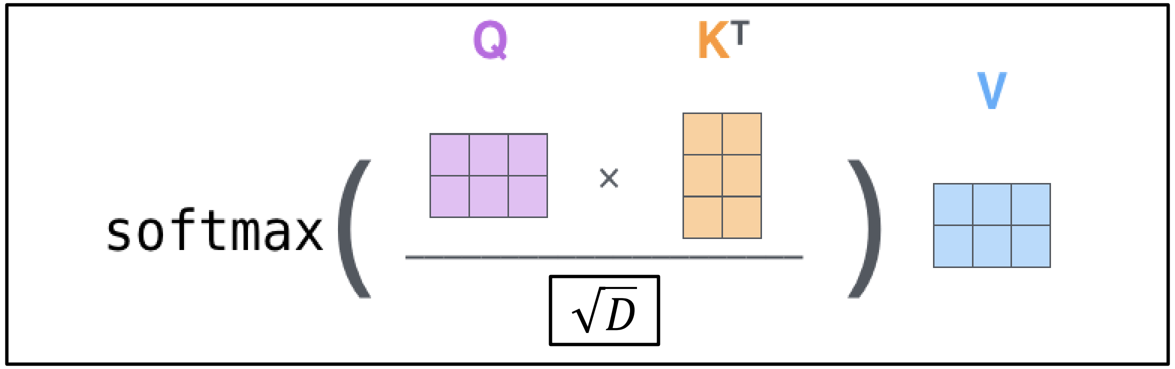
- **First step**: in each transformer block, for each token, we create a query vector, a key vector, and a value vector by multiplying the embedding by three weight matrices.
- Formally, for each token x_i:
  - Query: $q_i = x_i \times W^Q$
  - key: $k_i = x_i \times W^K$
  - Value: $v_i = x_i \times W^V$
- In the tensor representation for sequence $x_{1:L}$:
  - Query: $Q = q_{1:L} = x_{1:L} \times W^Q$
  - key: $K = k_{1:L} = x_{1:L} \times W^K$
  - Value: $V = v_{1:L} = x_{1:L} \times W^V$



RELAXED SYSTEM LAB

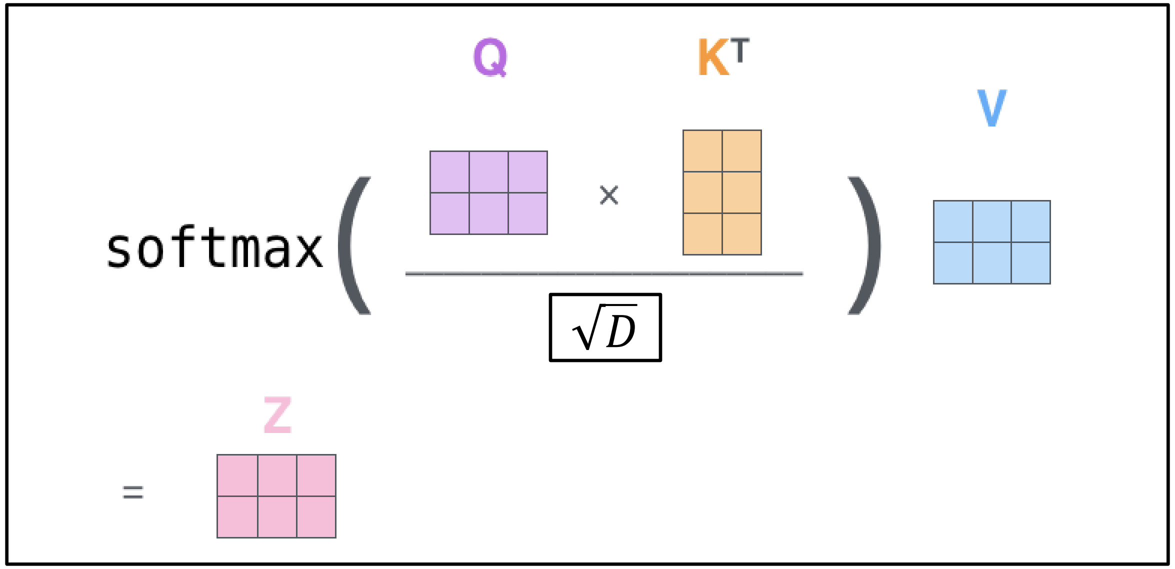
Attention Mechanism-2

- **<u>Second step</u>**: Calculate a score determining how much focus to place on other parts of the input sentence as we encode a token at a certain position.
- Calculated by:
  - Taking the dot product of the query vector with the key vector of the respective word we're scoring;
  - Divide the scores by the square root of the dimension of the key vectors;
  - Conduct a softmax operation.
- score = $\text{softmax}(\frac{QK^T}{\sqrt{D}})$



Attention Mechanism-3

- **<u>Third step</u>**: combine the value and the score.
  - $Z = \text{att} = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V$
- **<u>Multi-head Attention</u>**: there can be multiple aspects (e.g., syntax, semantics) we would want to match on.
- To accommodate this, we can simultaneously have **multiple attention heads (e.g. H heads)** and simply combine their outputs, e.g:
  - $Z = [\text{att}_1, \text{att}_2, \ldots, \text{att}_\text{H}]$
- The attention output will be:
  - $\text{Out} = ZW^O$



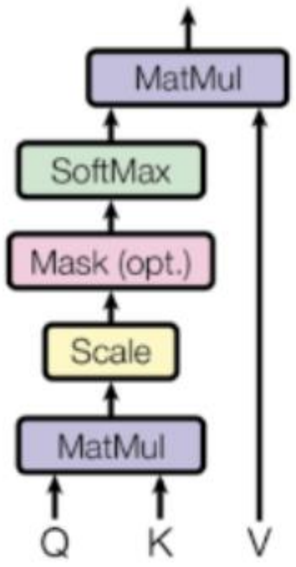
Feedforward Layer

- After the attention layer, the output is put to a feed-forward neural network, then sends out the output upwards to the next encoder.
  - $x'_{1:L} = \mathrm{relu}(ZW^1)W^2$
  - W^1, W^1 are two weight matrices;
  - $x'_{1:L}$ is the output embedding for the current layer and the input of the next layer.
- Summarize a common weight dimension in one TransformerBlock:
  - Attention layer: $W^Q, W^K, W^V, W^O \in \mathbb{R}^{D\times D}$
  - Feedforward layer: $W^1 \in \mathbb{R}^{D\times 4D}$, $W^2 \in \mathbb{R}^{4D\times D}$

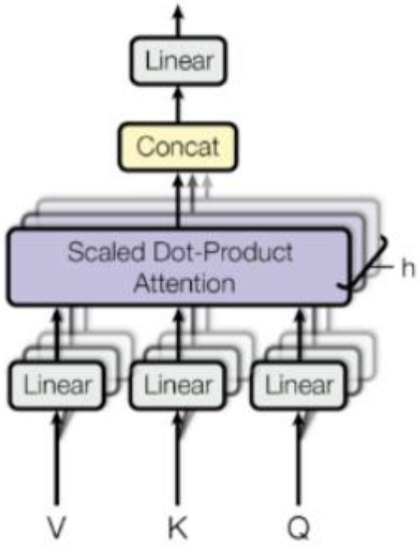
Other Components

- Residual connections:
  - Instead of simply return $\text{TransformerBlock}(x_{1:L})$
  - Return: $x_{1:L} + \text{TransformerBlock}(x_{1:L})$
- Layer normalization:
  - $\text{LayerNorm}(x_{1:L}) = \alpha \frac{x_{1:L} - \mu}{\sigma} + \beta$
- Positional embeddings:
  - So far, the embedding of a token doesn't depend on where it occurs in the sequence, which is not sensible.
  - $\begin{cases} \text{PosEmb}(\mathrm{i}, 2j) = \sin(\frac{\mathrm{i}}{10000^{2j/D}}) \\ \text{PosEmb}(\mathrm{i}, 2j+1) = \cos(\frac{\mathrm{i}}{10000^{2j/D}}) \end{cases}$

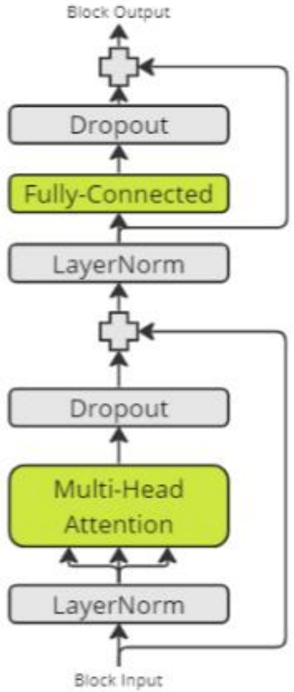
Put Them Together



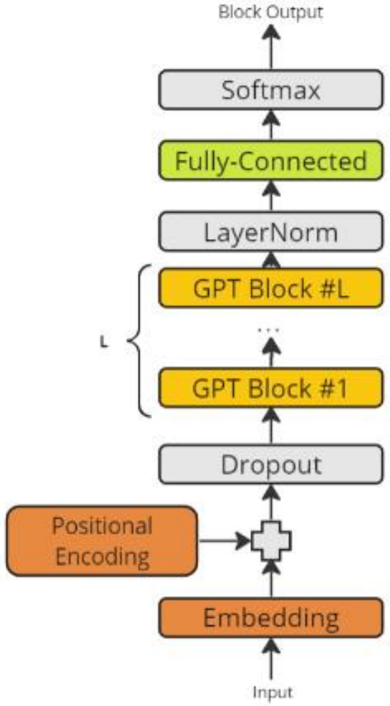
Scale Causal Attention



Multi-Head Attention



Transformer Block



GPT Model

LLM Training Objectives

Decoder-only Model Training Objectives

- Recall that an autoregressive language model defines a conditional distribution: $p(x_i|x_{1:i-1})$
- Define it as follows:
  - Map $x_{1:i-1}$ to contextual embedding $\emptyset(x_{1:i-1})$;
  - Apply an embedding matrix $E \in \mathbb{R}^{D\times|\mathcal{V}|}$ to obtain scores for each token $E\emptyset(x_{1:i-1})_{i-1}$;
  - Exponentiate and normalize it to produce the distribution over x_i.
- Put them together:

$$p(x_{i+1}|x_{1:i}) = \mathrm{softmax}(E\emptyset(x_{1:i})_i)$$

Decoder-only Model Training Objectives

RELAXED SYSTEM LAB

- Maximum likelihood. Let θ be all the parameters of large language models.
- Let $\mathcal{D}$ be the training data consisting of a set of sequences. We can then follow the maximum likelihood principle and define the following negative log-likelihood objective function:

$$\mathcal{O}(\theta) = \sum_{x_{1:L \in \mathcal{D}}} -\log p_\theta(x_{1:L}) = \sum_{x_{1:L \in \mathcal{D}}} \sum_{i=1}^{L} -\log p_\theta(x_i | x_{1:i-1})$$

- Then we can use the SGD optimizers we have talked to optimize this loss function.

References

- https://scholar.harvard.edu/sites/scholar.harvard.edu/files/binxuw/files/mlfs_tutorial_nlp_transformer_ssl_updated.pdf
- https://jalammar.github.io/illustrated-transformer/
- https://stanford-cs324.github.io/winter2022/lectures/introduction/
- https://stanford-cs324.github.io/winter2022/lectures/modeling/
- https://stanford-cs324.github.io/winter2022/lectures/training/