CS324 - Large Language Models

Modeling, Training

HUMANE Lab

김건수

2025.01.17

Overview

- Modeling
  - Tokenization: how a string is split into tokens
  - Models: focus on the **Transformer** architecture, which is the key innovation that enabled large language models(LLMs).
- Training
  - Objective functions
  - BERT, T5, BART
  - Optimization algorithms

Tokenization

- Language models: Predict probabilities over token sequences.
- Natural language: Exists as strings of characters(e.g., the mouse ate the cheese)
- Tokenizer: Converts strings into token sequences.(e.g., [the, mouse, ate, the, cheese])

→Importance: Tokenization significantly affects model performances.

- Tokenization methods:
  - Split by spaces
  - Byte pair encoding
  - Unigram model(SentencePiece)

Tokenization - Split by spaces

- Simple tokenization: Splitting by spaces is insufficient for many languages
  - ex: Chinese, German, English(e.g., don’t)
- Challenges with space-based splitting:
  - Doesn’t account for linguistic nuances or complex word structures.
  - Requires more sophisticated tokenization methods.
- Good tokenization criteria:
  - Avoid too many tokens(e.g., characters or bytes).
  - Avoid too few tokens to ensure parameter sharing (e.g., similar structures like mother-in-law and father-in-law).
  - Tokens should represent meaningful linguistic or statistical units.

Tokenization – Byte pair encoding(BPE)

- Key Idea
  - Derived from data compression to tokenize text.
  - Iteratively merges frequently co-occurring tokens.
- Methods
  - Input: Training corpus(character sequences).
  - Initialize vocabulary with characters.
  - Merge most frequent token pairs until desired vocabulary size.

Tokenization – Byte pair encoding(BPE)

- Example
  - Input : [t, h, e, ⎵, c, a, r], [t, h, e, ⎵, c, a, t], [t, h, e, ⎵, r, a, t]
  - Merges: t, h → th, th, e → the, c, a → ca
  - Final Vocabulary: [a, c, e, h, t, r, ca, th, the]

- Unicode Challenge
  - Vast Unicode space(144,697 characters).
  - Solution: Run BPE on bytes to handle unseen characters.
  - Example: “今天” → [x62, x11, 4e, ca].

Tokenization – Unigram model(SentencePiece)

- Key Idea:
  - Tokenization T divides a sequences $x_{1:L}$ into token spans.
  - Likelihood: $p(x_{1:L}) = \Pi_{(i,j)\in T} P(x_{i:j})$

- Example:
  - Training string: a,ba,bc
  - Tokenization: T = {(1,2), (3,4), (5,5)}, V = {ab, c}
  - Likelihood: $p(x_{1:L}) = \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} = \frac{4}{9}$

Tokenization – Unigram model(SentencePiece)

- Algorithm Steps:
  1. Start with a large seed vocabulary V.
  - Iteratively optimize:
    - Optimize p(x) and T using the EM algorithm.
    - Compute loss(x) for each token $x \in V$:
    - Measure likelihood reduction if x is removed.
    - Sort tokens by loss and keep the top 80%.
- Advantages:
  - More structured and statistically principled compared to frequency-based methods.
  - Allows for dynamic refinement of the token vocabulary.

Models

- Language Models:
  - Represent sequences as probabilities ($p(x_1, \ldots, x_L)$.
  - Versatile but inefficient for some tasks when modeling full sequences.
- Contextual Embeddings:
  - Map token sequences to embeddings influenced by surrounding context.
  - Example: [the, mouse, ate, the, cheese] → $\phi[(1, 0.1), (0,1), (1,1), (1, -0.1), (0, -1)]$.
- Core Idea:
  - Embedding adapt based on the token’s position and nearby words.
  - Captures meaning dynamically for more task-specific efficiency.

Models – Types of language models : Encoder-only

- Encoder-only models:
  - Examples: BERT, RoBERTa, etc.
  - Generate contextual embeddings.
  - Used primarily for classification tasks (e.g., sentiment analysis, natural language inference).
- Advantages:
  - Contextual embeddings account for bidirectional context(left and right).
- Limitations:
  - Can not generate text or completions naturally.
  - Require ad-hoc training objectives(e.g., masked language modeling).

Models – Types of language models : Decoder-only

- Decoder-only models:
  - Examples: GPT-2, GPT-3, etc.
  - Operate autoregressively: given a prompt $x_{1:i}$, they provide
    - Contextual embeddings $\phi(x_{1:i})$
    - Next-token distribution $p(x_{i+1}|x_{1:i})$
  - Enables text generation (e.g., autocomplete).
- Advantages:
  - Naturally generate completions.
  - Simple training objective: maximum likelihood.
- Limitation:
  - Contextual embeddings for each token only use unidirectional (left-to-right) context.

Models – Types of language models : Encoder-Decoder

- Encoder-Decoder models:
  - Examples: BART, T5, etc.
  - Encoder produces bidirectional contextual embeddings from input $x_{1:L}(\phi(x_{1:L}))$.
  - Decoder generates output $y_{1:L}$ conditioned on those embeddings $(p(y_{1:L}|\phi(x_{1:L})))$.
- Advantages:
  - Bidirectional context in the encoder for richer embeddings.
  - Natural text generation from the decoder.
- Limitation
  - Requires more complex training objectives(e.g., denoising or other recontruction tasks).

Models - Architecture

- Token-to-Vector Conversion
  - EmbedToken: Maps each token x_i to a vector using an embedding matrix $E \in \mathbb{R}^{[V] \times d}$.
  - Output: A sequence of context-independent embeddings $[Ex_1, \ldots, Ex_L]$.
- Abstract SequenceModel
  - Transforms context-independent embeddings into contextual embeddings.
  - Examples of specific implements include:
    - FeedForwardSequenceModel, SequenceRNN, TransformerBlock

Models - Architecture

- FeedForwardSequenceModel
  - Uses a fixed-length context(like an n-gram model).
  - For each position i in [1, …, L]:
    1. Collect the last n embeddings: $(x_{i-n+1}, \dots, x_i)$.
    2. Apply a feedforward network to produce h_i.
  - Returns the updated sequence $[h_1, \dots, h_L]$.

Models – Recurrent Neural Networks

- SequenceRNN:
  - Processes sequences from x_1 to x_L recursively to compute vectors h_1 to h_L.
  - Updates hidden states: $h_i = RNN(h_{i-1}, x_i)$.
- RNN Module Functionality
  - Takes the current state h and a new observation x, returning an updated state.
  - Implementations include SimpleRNN, LSTM, and GRU.
- SimpleRNN
  - Updates the hidden state h by applying a linear transformation followed by a non-linear function(e. g., logistic or ReLU).

Models – Recurrent Neural Networks

- Bidirectional RNN
  - Enhances sequence processing by running RNNs in both forward and reverse directions.
  - Combine forward and backward states to capture dependencies from both past and future context.
- Challenges and Advancements
  - Simple RNNs are difficult to train due to vanishing gradients.
  - LSTM and GRU were developed to better capture long-term dependencies and mitigate training iss ues.
  - Despite potential, practical dependencies on distant tokens may not be robust
  - LSTMs significantly advanced deep learning’s impact in NLP.

Models – Transformer

- Overview & Context
  - Transformers power major NLP models (GPT, BERT, T5).
  - Core mechanism: attention, originally used in machine translation.
- Attention Mechanism
  - Concept: A “soft” lookup for matching a query y against key-value pairs from each token x_i.

Models – Transformer

- Implementation:
  1. Transform x_i into keys ($W_{key}x_i$) and values ($W_{value}x_i$).
  2. Transform y into a query ($W_{query}y$).
  3. Compute scores (dot products of query and keys).
  4. Normalize scores with softmax to form weights α_i.
  5. Weighted sum of the values yields the attention output.
- Multi-Headed Attention: Multiple parallel attention heads capture different “aspects”.

Models – Transformer

- Self-Attention
  - Each token x_i serves as the query y when comparing with every other token in the sequence.
  - Produces a contextualized vector for each position that reflects all other positions in the sequence.
- Feedforward Layer
  - Applies a simple, per-token transformation: $y_i = W_2 \max(0, W_1 x_i + b_1) + b_2$.
  - Complements self-attention by introducing additional nonlinear processing.

Models – Transformer

- Improving Trainability
  - Residual Connections: Add the original input back to the function output $(x + f(x))$ to ensure grad ients can flow even if f becomes difficult to train.
  - Layer Normalization: Normalizes each token's embedding to avoid overly large or small values.
  - AddNorm encapsulates residual addition plus layer norm.
  - Stacking multiple TransformerBlocks leads to large-scale models (e.g., GPT-3).

Models – Transformer

- Positional Embeddings
  - Transformers lack inherent sequence order; positional embeddings inject location information.
  - Common approach uses sinusoidal functions sin for even dimensions, cos for odd).
  - Result: each token embedding is enriched with its positional signal in the sequence.
- Key Takeaways
  - Attention is the centerpiece, enabling global context capture.
  - Residual/LayerNorm ensure deep stacking is feasible.
  - Positional Encodings provide ordering cues in an otherwise order-agnostic architecture.
  - Transformers' flexibility and scalability underpin modern large language models.

Models – GPT-3

- GPT-3 Architecture
  - $GPT-3(x_{1:L}) = TransformerBlock^{96}(EmbedTokenWithPosition(x_{1:L})$
- Variants in Transformer Implementations

1. Layer Normalization: “Post-norm” (original Transformer) vs. “pre-norm” (GPT-2), affecting training s tability.

2. Dropout: Applied throughout to mitigate overfitting.

3. Sparse Transformer (GPT-3):Interleaves sparse and dense layers to reduce parameters.

4. Masking Schemes: Differ among encoder-only, decoder-only, and encoder-decoder models.

Objective functions

- Decoder-Only (GPT-Style)
  - Autoregressive LM: $p(x_i|x_{1:i-1})$
  - Training Objective: Maximum Likelihood (NLL)
- Encoder-Only (BERT-Style)
  - Bidirectional embeddings (no need to generate text).
  - Masked Language Modeling (MLM):
    - Randomly mask ~15% of tokens, predict them from context.
    - Use partial replacement with real words to avoid test-train mismatch.
  - Next Sentence Prediction (NSP):
    - Predict whether the second sentence truly follows the first.

Objective functions

- Encoder-Decoder (T5 / BART-Style)
  - Bidirectional encoder (like BERT) + autoregressive decoder (like GPT).
  - Often trained with denoising tasks (e.g., masked spans, permuted sentences).

BERT

- Architecture: 24 Transformer layers, d_{model}=1024, 16 attention heads(≈355M params)
- Key Tokens:
  - [CLS] for classification embedding.
  - [SEP] to divide sequences.
- Improvements (RoBERTa):
  - Removed NSP, used more data (160GB), trained longer → better benchmarks.

T5 & BART

- T5 (Text-to-Text)
  - Everything cast as text-to-text (including classification).
  - Trained on replacing masked spans → “i.i.d. noise.”
  - Large version ~11B parameters.
- BART
  - RoBERTa-like encoder, GPT-like decoder.
  - Denosing tasks (mask 30% tokens, permute sentences).
  - Strong on both classification & generation.

Optimization

- Core Objective:
  - $\mathrm{O}(\theta) = \sum_{x \in D} -log p_{\theta}(x)$
- Adam Optimizer
  - Momentum + Adaptive Learning Rates.
  - Increases memory from storing (m_t, v_t) .
  - AdaFactor reduces memory usage by storing row/column stats instead of full moments.

Optimization

- Mixed-Precision Training (FP16)
  - Master weights in FP32, everything else in FP16.
  - Loss scaling to avoid underflow.
- Learning Rate Schedules
  - Warmup then decay (common in Transformers).
  - Cosine decay, etc.
- Initialization Tricks
  - Xavier for most layers.
  - Additional scaling (e.g., $1/\sqrt{N}$ for GPT, or $1/\sqrt{d}$ for T5 attention matrices).