Natural Language Processing (NLP)

Masked Language Modeling

Encoder-Only Transformers

By:

Dr. Zohair Ahmed

Lecture Overview: From GPT to BERT

- **Context: Previous lecture**
  - We studied **decoder-only models**, focusing on **GPT-1** and **decoding strategies** for text generation.
- **Today's focus**
  - Introduce **BERT** – *Bidirectional Encoder Representations from Transformers*.
  - Understand **masked language modeling (MLM)** and **encoder-only transformers**.
- **Learning goals**
  - Why BERT was introduced (motivation vs GPT).
  - How BERT's architecture & training differ from GPT.
  - How BERT is used in **downstream NLP tasks**.
- **Timeline :** RNNs → Transformers → GPT (decoder-only) ↔ BERT (encoder-only, MLM).

BERT: Historical Context & Impact

- **What is BERT?**
  - A **Transformer-based, encoder-only** model.
  - Full form: **Bidirectional Encoder Representations from Transformers**.
- **Historical placement**
  - Released around **2017**, as a **current of early GPT** models.
- **Impact on NLP**
  - Extremely popular for **3–4 years** before massive LLMs dominated.

Decoder-Only vs Encoder-Only

- **Decoder-only (GPT-style)**
  - Uses **causal language modeling**: predict next word using **only left context**.
  - Natural for **text generation**: continuation, story writing, chat.
- **Encoder-only (BERT-style)**
  - Uses **masked language modeling**: predict **masked tokens** using **both left and right context**.
  - Natural for **understanding tasks**: classification, tagging, QA, NER.
- **Key distinction**
  - GPT: **unidirectional** (left → right).
  - BERT: **bidirectional** (left & right simultaneously).

Decoder-Only vs Encoder-Only

- **Decoder-only (GPT-style)**
  - Uses **causal language modeling**: predict next word using **only left context**.
  - Natural for **text generation**: continuation, story writing, chat.
- **Encoder-only (BERT-style)**
  - Uses **masked language modeling**: predict **masked tokens** using **both left and right context**.
  - Natural for **understanding tasks**: classification, tagging, QA, NER.
- **Key distinction**
  - GPT: **unidirectional** (left → right).
  - BERT: **bidirectional** (left & right simultaneously).

Motivation: Why Bidirectional Context?

- **Problem with strict left-to-right models**
  - Many tasks require **future words** to interpret a token correctly.
- **Named Entity Recognition (NER) example**
  - **Sentence:** I went from Lahore to Karachi.
  - To decide if “Lahore” is a location, “**to** Karachi” helps.
  - Future context improves entity classification.
- **Syntactic/semantic relations**
  - Words relate to **both prior and subsequent tokens**:
  - Subject, verb, and object all mutually depend on each other.
- For *understanding* text, we want **bidirectional representations** → BERT.

Example: Why Both Sides Matter

- **Fill-in-the-blank scenario**
  - Sentence: ___ has shipped its new OS to users.
  - The phrase **“has shipped its new OS”** strongly suggests the blank is a **company**.
- **Key observation**
  - Understanding **the blank** requires:
  - Left context: maybe earlier mentions of products.
  - Right context: “has shipped its new OS”.
- **Goal**
  - Train a model that **looks both ways** in a sentence and builds rich **contextual embeddings**.

Recap: Causal Language Modeling (GPT)

- **Setup**
  - Input sequence: I like to
  - Model predicts **next word**: play / watch / go / read / ...
- **Training objective**
  - Maximize probability $P(x_t|x_1,\ldots,x_{t-1})$.
- **Attention mask**
  - Use **causal mask** so each position only sees **tokens to its left**.
- **Strength**
  - Excellent fit for **text generation** (auto-regressive decoding).

From Causal LM to Masked LM

- **Goal shift**
  - From predicting the **next token** to predicting **masked tokens anywhere**.
- **Masked Language Modeling (MLM)**
  - Mask some tokens in the input.
  - Predict them using **all other tokens** (left + right).
- **Why this helps**
  - Forces the model to encode **deep contextual relationships** across the entire sentence, not just history.

BERT Architecture: Encoder-Only Transformer

- **Base structure**
  - Stack of **Transformer encoder blocks**:
  - Multi-head self-attention.
  - Feed-forward layers.
  - Residual connections + LayerNorm.
- **Input–output**
  - Input: sequence of tokens (with some replaced by **[MASK]**).
  - Output: contextual embeddings for **every position**.
- **No decoder, no cross-attention**
  - Only **self-attention over the input sequence**.
  - Same full attention mask for all (no causal masking).

Formal Objective: Predicting Masked Tokens

- **Notation**
  - Input tokens: $x_1, x_2, \dots, x_T$
  - Some positions are masked (e.g., at index i).
- **MLM objective**
  - Predict the true word at masked positions:
  - $Maximize\ P(y_i = correct_{\mathrm{word}} | x_{\mathrm{i}}, \dots, x_T)$.
  - Unlike causal LM, **uses full sequence**.
- **Independence assumption**
  - If tokens at positions i and j are masked:
  - Predict them **independently**.
  - When predicting at i, position j is still $[MASK]$, and vice versa.

Example: Multiple Masks in a Sentence

- **Illustrative example**
  - True sentence: I like to read a book.
  - Masked input: I [MASK] to read a [MASK].
- **Training behavior**
  - The encoder consumes: I, [MASK], to, read, a, [MASK].
  - At the output:
  - At first masked position, predict distribution over vocabulary; maximize P(like).
  - At second masked position, maximize P(book).
- **Key point**
  - Model learns to infer both **“like”** and **“book”** from the surrounding unmasked words **simultaneously**.

Why Not Sequentially Predict Masks?

- **Natural question**
  - Why not:
  - Predict the 2nd token,
  - Fill it in,
  - Then use that prediction to help predict the 8th token?
- **Design choice in BERT**
  - All masks are predicted **in parallel**.
  - Each masked position **only sees original non-masked tokens**.
- **Rationale**
  - Makes the task **harder** and more robust:
  - Model must rely purely on **true context**, not on its own previous guesses.
  - Encourages stronger **language understanding**.

Setting Up the Masking Task

- **Basic procedure**
  - Take a large text corpus.
  - Randomly select some tokens to **mask**.
  - Replace each selected token with a special [MASK] token (plus variants).
  - Train model to **recover the original words** at those positions.
- **Analogy**
  - Similar to **image inpainting**:
  - Black out patches in an image and train the model to reconstruct them.

Why Not Mask Using Attention Masks?

- **Idea that doesn't work**
  - In causal LM we used **attention masks** to hide future tokens.
  - Could we zero out attention to masked tokens to "remove" them?
- **Problem**
  - If we zero their attention:
    a. It is as if those tokens **don't exist**.
    b. Model cannot know **where the blanks are**.
- **Requirement for MLM**
  - We **want** the model to:
  - See that certain positions are **special blanks**.
  - Attend to them and derive contextual cues.
- **Solution**
  - Do **not** mask attention.
  - Instead, **replace tokens with a visible** [MASK] **token in the sequence**.

Input Representation with Mask Tokens

- **Vocabulary extension**
  - Introduce a special token, e.g., "[MASK]", into the vocabulary.
- **Example**
  - Original: I really enjoyed the talk.
  - Masked: I [MASK] enjoyed the [MASK].
- **Model flow**
  - Encoder reads this sequence with [MASK] present.
  - Produces contextual vectors for **every token**, including [MASK].
  - Final layer outputs at masked positions are passed through a classifier to predict the original word.

MLM Loss Function

- **Training objective**
  - Let M be the set of masked positions.
  - For each $\boldsymbol{i} \in \boldsymbol{M}$:
  - Predict distribution over vocabulary.
  - **Loss**: sum (or average) of **cross-entropy** over all masked positions.
- **Interpretation**
  - Maximizing **log-likelihood** of correct words at those positions.
  - Only masked positions contribute to the MLM loss; unmasked tokens act as context.

How Many Tokens Do We Mask?

- **Masking proportion**
  - Typically, **15% of tokens** are selected for the MLM task.
- **Trade-offs**
  - If masking too high (e.g., 85%):
  - Too little context remains → task becomes **too hard;** model may not converge.
  - If masking too low (e.g., 1 token out of 20):
  - Task becomes **too easy**, model learns less.
- **Empirical choice**
  - **15%** found to be a good balance in the original BERT.

Detailed Masking Strategy: 80 / 10 / 10 Rule

- **Given tokens selected for masking (15% of all tokens):**
  - **80%** of those → replace with [MASK].
    a. Example: the → [MASK].
  - **10%** of those → replace with **random word**.
    a. Example: cat → and (any random vocab token).
  - **10%** of those → **keep the original word unchanged**.
    a. Example: I stays I.
- **Counts example (for 200 tokens)**
  - 15% of 200 = 30 tokens selected.
  - Of these 30:
    a. 24 → [MASK].
    b. 3 → random replacements.
    c. 3 → left unchanged.

Why Use Random Words and Unchanged Tokens?

- **Problem if we only used** [MASK]
  - Pre-training sees [MASK] very often.
  - But **downstream tasks never have** [MASK] in inputs.
  - Model could overfit to unrealistic patterns.
- **Solution: random & unchanged cases**
  - **Random replacements**
    a. Forces the model to be robust to **noise**: some tokens are wrong.
  - **Unchanged tokens**
    a. Model sometimes must **predict a token that is already present**:
      – Input token = output token.
      – Mimics real-world settings where **no masks** appear at test time.
- **Outcome**
  - Better **generalization** from pre-training to real **downstream tasks**.

BERT Model Configuration

- **Encoder-only Transformer**
  - Built as a **multi-layer stack** of identical encoder blocks.
- **Model family**
  - **Base model**: around **12 layers** (Transformer blocks).
  - Another released model: **deeper** (e.g., ~16 layers).
- **Within each layer**
  - Multiple **attention heads** (e.g., 12 heads mentioned).
  - Each head learns different types of **relations** between tokens.

Real Inputs vs Pre-Training Inputs

- **Pre-training**
  - Inputs contain many [MASK] tokens and occasional **random/noisy** tokens.
- **Downstream / inference**
  - Inputs are **normal sentences** (no [MASK]).
  - Pre-training: It was [MASK] yesterday and the ground is wet.
  - Inference: It was raining yesterday and the ground is wet.
- **Why this matters**
  - Without the 80/10/10 strategy, there would be a big **mismatch** between training and usage.
  - Including unchanged and random tokens helps BERT handle **clean, unmasked text** better.

Next Sentence Prediction (NSP)

Next Sentence Prediction (NSP)

- **Second pre-training objective in BERT**
  - In addition to **Masked Language Modeling (MLM)**, BERT introduced **Next Sentence Prediction (NSP)**.
- **Core idea**
  - Train BERT to decide whether **sentence B** is the *actual next sentence* that follows **sentence A** in a corpus.
- **Why introduce NSP?**
  - To give BERT a notion of **inter-sentence relationships**.
  - Initially thought to help with **pair-of-text tasks** like QA, entailment, etc.

Next Sentence Prediction (NSP)

- **Second pre-training objective in BERT**
  - In addition to **Masked Language Modeling (MLM)**, BERT introduced **Next Sentence Prediction (NSP)**.
- **Core idea**
  - Train BERT to decide whether **sentence B** is the *actual next sentence* that follows **sentence A** in a corpus.
- **Why introduce NSP?**
  - To give BERT a notion of **inter-sentence relationships**.
  - Initially thought to help with **pair-of-text tasks** like QA, entailment, etc.

Input Format for NSP: Special Tokens

- **Special tokens used**
  - [CLS] – classification token (placed at the **start** of the sequence).
  - [SEP] – separator token (placed **between** sentence A and B, and often at the end).
- **Sequence construction for A and B**
  - Input sequence:
    [CLS] $tokens\_of\_sentence\_A$ [SEP] $tokens\_of\_sentence\_B$ [SEP]
- **Sequence length**
  - If original combined length is T, new length is **T + 2** (for [CLS] and first [SEP]), plus possibly another [SEP].

Role of the [CLS] Token

- **[CLS] as a special representation**
  - [CLS] is treated as a **special token** (or start-of-sequence token).
- **Passing through the Transformer**
  - [CLS] goes through all encoder layers along with other tokens.
  - At the **final layer**, its output embedding is used as a **summary representation** of the entire input (A + B).
- **For NSP**
  - The final [CLS] vector is fed into a **binary classifier**:
  - Output: probability that **B is the true next sentence of A**.

Role of the [CLS] Token

- **[CLS] as a special representation**
  - [CLS] is treated as a **special token** (or start-of-sequence token).
- **Passing through the Transformer**
  - [CLS] goes through all encoder layers along with other tokens.
  - At the **final layer**, its output embedding is used as a **summary representation** of the entire input (A + B).
- **For NSP**
  - The final [CLS] vector is fed into a **binary classifier**:
  - Output: probability that **B is the true next sentence of A**.

Constructing Training Data for NSP

- **Positive examples (label = 1)**
  - Use **naturally occurring consecutive sentences** in a large corpus like Wikipedia.
  - Example:
    a. Sentence A: a sentence.
    b. Sentence B: the **actual next sentence** that follows A.
- **Negative examples (label = 0)**
  - Take sentence A.
  - Replace B with a **random sentence** from the corpus:
  - Either from the **same article** or from **another article**.
- **Result**
  - Large amounts of labeled (A, B) pairs created **automatically** without manual annotation.

NSP and Two-Sentence Tasks

- **Tasks involving pairs of texts**
  - **Question Answering (QA)**:
    a. Sentence/segment A: **passage**.
    b. Sentence/segment B: **question**.
    c. Model must reason over **both** to find answer.
  - **Textual Entailment / Natural Language Inference (NLI)**:
    a. A: **premise** (original sentence).
    b. B: **hypothesis/claim**.
    c. Task: decide if B is **entailed**, **contradicted**, or **neutral** with respect to A.

- **Early findings**
  - NSP was reported as **helpful** for such **two-sequence problems**, giving BERT a better sense of **sentence-to-sentence relationships**.

Later Findings – NSP's Limited Benefit

- **Subsequent research**
  - Within roughly a year (or even less), several works examined the impact of NSP.
- **Conclusion**
  - NSP objective was found to provide **little or no benefit** for many downstream tasks.
- **Practical outcome**
  - Many later BERT variants **removed the NSP objective**.
  - They trained **only with MLM**, and performance did not degrade, sometimes even improved.
- **Modern trend**
  - Most newer BERT-like models **do not use NSP** as part of pre-training.

Input Embeddings: Three Components

- BERT uses *learned positional embeddings* that serve the same purpose as Transformer positional encodings but the **mechanism is different** from the original sinusoidal encoding.
- **1. Token (word) embeddings**
  - Matrix $W_{embedding}$ of shape $(\boldsymbol{vocabulary}_{\boldsymbol{size}} \times \boldsymbol{d}_{\boldsymbol{model}})$.
  - Each token ID → a **d-dimensional vector**.
- **2. Segment embeddings**
  - To distinguish **sentence A** vs **sentence B** in the same input:
  - Segment 0 embedding: for tokens in A.
  - Segment 1 embedding: for tokens in B.
  - Stored in a small $(\mathbf{2} \times \boldsymbol{d}_{\boldsymbol{model}})$ matrix.
- **3. Positional embeddings**
  - For positions from 0 to $\underset{sequence_{length}}{\max}$ (e.g., 512).
  - Provide information about **order** of tokens.

Encoder Stack and Dual Objectives

- **Transformer encoder layers**
  - Inputs: unified embeddings for [CLS], A tokens, [SEP], B tokens, [SEP].
  - Pass through **multiple encoder layers** (as discussed in the MLM lecture).
- **Outputs**
  - Final contextual representation for **each position** (including [CLS] and all tokens).
- **Two objectives optimized jointly**
  - **Masked Language Modeling (MLM)**:
    a. Predict original words at masked token positions.
  - **Next Sentence Prediction (NSP)**:
    a. Binary classification on top of the final [CLS] vector.
- **Total loss**
  - Total Loss = MLM Loss + NSP Loss
  - Both losses are optimized **together during pre-training**.

Pre-Training Corpora for BERT

- **Compared to GPT (earlier lecture)**
  - BERT used a **larger corpus** than the GPT model previously discussed.
- **Corpora used**
  - **Book Corpus**: about **800 million words**.
  - **Wikipedia**: about **2.5 billion (2,500 million) words**.
- **Perspective**
  - By **modern standards**, these are **small**.
  - At the time, they were considered **quite large** for language model pre-training.

Two Versions – BERT Base vs BERT Large

- **BERT Base**
  - **Layers (encoder blocks):** 12
  - **Hidden size (**d_model**):** 768
  - **Attention heads per layer:** 12
  - **Feed-forward inner dimension:** 3072 (≈ 4 × 768)
- **BERT Large**
  - **Layers:** 24
  - **Hidden size (**d_model**):** 1024
  - **Attention heads per layer:** 16
  - **Feed-forward inner dimension:** 4096 (≈ 4 × 1024)

Embedding Layer Components (Recap)

- For the **BERT Base** model (d_model = 768):
- **Three types of embeddings**
  - **Token embeddings**
    a. Vocabulary size: ~30,000
    b. Embedding dimension: 768
    c. Matrix shape: **30,000 × 768**
  - **Segment embeddings**
    a. Two segments: sentence A and sentence B.
    b. 2 embeddings, each of size 768.
    c. Matrix shape: **2 × 768**
  - **Position embeddings**
    a. For positions up to max sequence length **512**.
    b. Each position has a 768-dimensional vector.
    c. Matrix shape: **512 × 768**

Parameter Count – Embedding Layer (Base Model)

- **Token embeddings**
  - Approximate vocabulary size: 30,000.
  - Each word → 768-dimensional vector.
  - Parameters: 30,000 × 768 ≈ 23,000,000 (**23M**).
- **Segment embeddings**
  - Only **2** segment embeddings.
  - Parameters: 2 × 768 ≈ 1,536 → **negligible** compared to others.
- **Position embeddings**
  - 512 positions × 768 dimensions.
  - Parameters: 512 × 768 ≈ 0.4M (**0.4 million**).
- **Total embedding parameters**
  - Roughly 23M + 0.4M ≈ 23.4M parameters.

Multi-Head Self-Attention Setup (Base Model)

- **Input dimension**
  - Each token embedding is **768-dimensional**.
- **Heads**
  - **12 attention heads** per layer.
- **Per-head dimension**
  - 12 heads × 64 dims/head = 768:
    a. So each head works in a **64-dimensional** space.
- **Process per input:**
  - Take 768-dim input.
  - Map to 12 × 64-dim subspaces (one per head).
  - Concatenate 12 outputs back to **768 dim**.
  - Apply a final linear layer to mix them.

Feed-Forward Network (FFN) Parameters per Layer

- **Structure in each layer**
  - Two linear transformations:
    a. 768 → 3072
    b. 3072 → 768
- **Parameter counts (ignoring biases)**
  - First linear: 768 × 3072
  - Second linear: 3072 × 768
  - Together on the order of **a few million** parameters.
- This FFN + attention + other pieces contributes about **7 million** parameters per layer (rounded figure).

Total Parameter Count: BERT Base

- **Embedding layer**
  - Roughly **23.4M** parameters.
- **Transformer blocks**
  - About **84M** parameters (12 layers × ~7M).
- **Total**
  - 23.4M + 84M ≈ 107.4M parameters.
- **Paper vs lecture calculation**
  - Paper reports **~110M** parameters.
  - Difference due to:
    a. Ignoring biases and other small components in rough calculations.
    b. Exact vocabulary size is actually **30,522**, not 30,000.
- **Takeaway**
  - BERT Base has roughly **110 million parameters**, comparable to the GPT model discussed earlier (≈117M).

Tokenization

Tokenization

- We need to turn raw text into **tokens** that models can process.
- Tokenization choice affects **vocabulary size**, **sequence length**, **parameters**, and **compute cost**.

Character-Level Tokenization

- **Definition**
  - Every **character** is treated as a token.
- **Example (English)**
  - Vocabulary: a–z, digits 0–9, punctuation, special symbols, etc.
  - Total vocabulary size: **< 100 tokens** (order of 100).
- **Properties**
  - **Very small vocabulary**.
  - **No out-of-vocabulary (OOV)** issue at word level:
  - Any word can be represented as a sequence of characters.

Word-Level Tokenization

- **Definition**
  - Every **word** (as humans usually define it) is a token.
  - Use **whitespace splitting**: each span between spaces is a word.
- **Example**
  - Sentence: "I really enjoyed enjoying this talk."
  - Tokens: ["I", "really", "enjoyed", "enjoying", "this", "talk."]
- **Resulting vocabulary**
  - Could easily reach **hundreds of thousands** of unique words:
  - Example: **500k** word vocabulary.
-

Problems with Large Word Vocabularies

- **1. Out-of-Vocabulary (OOV) and new words**
  - You will see **new words** at test time:
    a. Typos, neologisms, inflected forms, domain-specific terms.
  - Pure word-level tokenization has no natural way to handle them (OOV).
- **2. Parameter explosion (embeddings)**
  - Embedding matrix shape: $vocab_{size} \times d_{model}$.
  - Example: 500k × 1024:
    a. Every new word adds **1024 new parameters**.
    b. Total parameters grow very large.
- **3. Expensive output softmax**
  - Model predicts a **distribution over the entire vocabulary**.
  - With 500k vocab:
    a. Softmax over **500,000** entries each time.
    b. Denominator is sum over 500k values → **computationally expensive**.

Problems with Very Small Character Vocabularies

- **Sequence length explosion**
  - Consider a typical sentence (e.g., news, Wikipedia):
    a. **~20 words** on average.
    b. Each word ~**5–6 characters** on average.
  - Token count with character-level tokens:
    a. 20 words × 5–6 chars ≈ 100–120 tokens.
- **Impact on context window**
  - With maximum length **512 tokens**:
    a. Previously (word-level) could hold ~**512 words**.
    b. With character-level, can hold only **~100 words**.
  - So, a 512-token limit now covers **much less text**.
- **Inefficiency**
  - Feels like “feeding very little information per token”:
    a. Many tokens just to express a simple sentence.
    b. More tokens → more **attention computation** and **memory usage**.

Character-Level: Why Not Good Enough?

- **Pros**
  - Tiny vocabulary (~100).
  - No OOV issues at character level.
- **Cons**
  - **Long sequences** even for short sentences:
    a. More attention operations, slower training/inference.
  - 512-token limit now captures **much shorter context**.
  - Overly fine-grained: each token carries **very limited semantic information**.
- **Conclusion**
  - Character-level is too **inefficient** for typical NLP tasks with Transformers.

Word-Level: Why Not Good Enough?

- **Pros**
  - Intuitive, simple to implement (split on spaces).
  - Tokens correspond to what humans think of as **words**.
- **Cons**
  - **Very large vocabulary** (e.g., 500k):
    a. Large **embedding matrix** (many parameters).
    b. Very expensive **softmax** over all words.
  - Poor handling of:
  - **Rare words**, **morphology**, **typos**, **compounds**, **new words**.
- **Conclusion**
  - Word-level is too **heavy** (parameters & compute) and brittle with OOV.

Need for a Middle Ground – Subword Tokenization

- **Observation**
  - Character-level is **too small** a unit.
  - Word-level is **too large** a unit.
- **Idea**
  - Use **subwords**: pieces between characters and full words.
  - Examples:
    a. "don't" → "do", "n't"
    b. "can't" → "ca", "n't" or "can", "n't" depending on scheme.
- **Goal**
  - Find a sweet spot where:
    a. Vocabulary is **moderate**.
    b. Most words can be expressed as **few subword tokens**.
    c. New/rare words can still be decomposed into **familiar pieces**.

Intuition for Subwords

- **Example: “don’t”**
  - Instead of single token "don't" or characters ['d','o','n',"'",'t']:
    a. Represent as subwords: "do" and "n't".
  - Reasonable because:
    a. "do" is a valid word and appears in many forms: do, does, doing, don’t.
    b. "n't" appears in can't, won't, doesn't, etc.
- **Benefits**
  - Common subword pieces like "do" and "n't" appear **frequently in the corpus**.
  - We can reuse these subwords across many words:
    a. Reduces **vocabulary**.
    b. Preserves useful **morphological information**.
-

Tokenization Categories

- **Three main tokenization levels**
  - **Character-level**
    a. Each character is a token.
    b. Simple, small vocab, but long sequences.
  - **Word-level**
    a. Split on spaces, each word is a token.
    b. Intuitive, but large vocab and OOV issues.
  - **Subword-level**
    a. Words are broken into **meaningful pieces**.
    b. Middle ground: moderate vocab, reasonable sequence lengths.

- **Within the sub word-level category**,
  - **Byte Pair Encoding (BPE)**
  - **Word Piece**
  - **Sentence Piece**
- **Practical takeaway**
  - Modern large NLP models almost always use **subword tokenization**.