1 Encoder-Only Models: BERT

(a)(i) Why MLM is effective for contextual info

In MLM, BERT must predict a masked token using both its left and right context. To succeed, the hidden representation at each position has to encode rich information about surrounding words and longer range dependencies, so the model learns contextualized embeddings rather than fixed type embeddings.

(a)(ii) BERT's MLM strategy + changing masking ratio

BERT MLM data creation:

- 15% of tokens selected.
- Of these: 80% replaced by [MASK], 10% random token, 10% unchanged.
- Predict original token.

Effects of changing the masking ratio:

- (a) Drastically increase masking ratio: More tokens become masked, so each masked token has less surrounding context and predictions become harder; the input also looks very artificial (too many [MASK]), increasing mismatch with fine tuning where [MASK] rarely appears. This can hurt representations.
- (b) Drastically reduce masking ratio: Very few tokens are supervised at each pass $\rightarrow$ less training signal and more trivial predictions (most tokens are visible), so the model may learn weaker contextual representations.

(a)(iii) One disadvantage of MLM

A disadvantage is pretrain finetune mismatch. The special [MASK] token never appears at fine tuning or test time, but is common during pretraining, so the model is trained on artificial inputs that differ from true downstream inputs.

(b) NSP: purpose, benefit, why dropped later

- Purpose: NSP is a binary task: given sentence A and sentence B, predict whether B is the true next sentence after A in the corpus or a randomly sampled sentence. The goal is to teach BERT intersentence coherence and discourse relations.
- Benefits:
  - For extractive QA (SQuAD), NSP helps BERT learn how questions and passages relate, improving the ability to attend to the right part of the passage.
  - For NLI (entailment), NSP like training encourages the model to reason about how one sentence logically follows from or is compatible with another.
- Why some variants drop NSP: Empirically, later models like RoBERTa found NSP either did not help or even slightly hurt performance; better results came from more data and longer MLM training without NSP, so many variants omit it.

(c) Fine tuning BERT for binary sentiment classification

Steps:

- Input format:
  - For each text: [CLS] + tokenized text + [SEP].
  - Optionally use segment IDs (all zeros since single sequence).
- Output layer:
  - Take the final hidden state for [CLS]: $h_{[CLS]}$
  - Add a linear classifier: $\mathbf{o} = W_{\text{cls}} h_{[\text{CLS}]} + b, \quad \mathbf{o} \in \mathbb{R}^2$
- Loss: Use cross entropy loss over the two classes on top of the logits $\mathbf{o}$.
- Training: Fine tune all BERT parameters + classifier jointly on labeled sentiment data.

(d) Fine-tuning BERT for NER

- Input:
  - Sentence tokenized to WordPieces: $[CLS], w_1, ..., w_n, [SEP]$.
  - Optionally use a mask to indicate which subword is the first piece of each word.
- Output layer:
  - For each token position i (usually first subword of each word), take hidden state h_i.
  - Add a token level classifier: $o_i = W_{ner} h_i + b, o_i \in \mathbb{R}^{|Y|}$ where Y is the set of NER tags (e.g., B-PER, I-PER, O, ...).
  - Apply softmax per token.
- Loss:
  - Sum or average cross entropy over all labeled tokens in the sentence; ignore padding or non first subwords.
- Why BERT is effective for NER
  - Provides contextual representations: each token embedding reflects its context, crucial for distinguishing, for instance, "Washington" (person vs location).
  - Bidirectional attention captures both left and right context.
  - Can model long range dependencies and complex patterns like multiword entities.

(e) Extractive QA with BERT

(i) Input for passage P and question Q:

$$\texttt{[CLS]}\ Q\ \texttt{[SEP]}\ P\ \texttt{[SEP]}.$$

- One sequence with both question and passage.
- Use segment IDs (token type embeddings):
  - 0 for [CLS] + question tokens + first [SEP]
  - 1 for passage tokens + second [SEP].

(ii) Predicting start/end tokens:

- BERT outputs hidden states $h_1, ..., h_T$ for all tokens.
- Add two trainable vectors (or equivalently two linear layers):

$$w_{\text{start}}, w_{\text{end}}.$$

- Compute logits:

$$s_i = w_{\text{start}}^\top h_i, \quad e_i = w_{\text{end}}^\top h_i.$$

- Apply softmax over all positions:

$$p_{\text{start}}(i) = \text{softmax}(s)i, \quad p\text{end}(i) = \text{softmax}(e)_i.$$

- Train with cross entropy on the true start index and true end index.

(iii) Handling lengthy passages
One common strategy: sliding window over the passage.

- Break P into overlapping chunks, each of length $\leq$ max sequence length.
- For each chunk, form:

$$[\text{CLS}]\ Q\ [\text{SEP}]\ \text{chunk}(P)\ [\text{SEP}].$$

- Run BERT on each chunk and pick the best span across all chunks.

Trade offs:

- Pros: still uses standard BERT, no truncating big parts of the passage.
- Cons: more compute (multiple forward passes) and may lose some global context across chunk boundaries.

(f) Why bidirectionality is hard for autoregressive generation

BERT is trained with bidirectional attention and MLM, where each token representation can see future tokens. Autoregressive generation, however, must generate token x_t using only $x_{<t}$.

Because BERT doesn't enforce a causal left to right mask:

- There is no direct way to get $p(x_t \mid x_{<t})$ from a single forward pass.
- Need iterative masking and reprediction (like Gibbs style sampling), which is slow and approximate.
- This makes BERT unsuitable as a standard left to right generator compared to decoder only models.

2 Low-Rank Adaptation (LoRA)

Let base weight $W \in \mathbb{R}^{d\times k}$, and LoRA matrices $A \in \mathbb{R}^{d\times r}, B \in \mathbb{R}^{r\times k}$ with $r \ll d, k$.

(a) How LoRA adapts W

LoRA parameterizes the weight update as a low rank matrix:

$$\Delta W = AB$$

and uses an effective weight:

$$W' = W + \Delta W = W + AB.$$

During fine tuning, W is frozen and only A, B are trained.

(b) Space savings during fine tuning

Number of parameters in full W : dk.
Full rank fine tuning:

- Store the full matrix W plus optimizer states.
  - With SGD with momentum: for example, weights + momentum $\rightarrow$ about $2dk$ floats.
  - With Adam: weights + first moment + second moment $\rightarrow$ about $3dk$ floats.

LoRA fine tuning:

- Train only A and B: parameters $dr + rk = r(d + k)$.
- Optimizer states scale with these parameters:
  - SGD: $\approx 2r(d + k)$.

– Adam: $\approx 3r(d+k)$.

So LoRA reduces memory by factor roughly:

$$\frac{r(d+k)}{dk}$$

for parameters (and the same factor for optimizer states), which is much smaller when r $\ll \min(d,k)$. The base W (size dk) is stored once regardless, but no longer store separate full copies or full optimizer states per task.

(c) Space savings at inference/decoding

For a single model:

- Full rank decoding stores W of size dk.
- LoRA decoding stores W plus A, B: total d $k + r(d+k)$.

So for a single task, LoRA does not reduce memory at inference; it may slightly increase it.
However, if there are many task variants:

- Full rank: each task needs its own full $W^{(task)} \rightarrow$ per task cost dk.
- LoRA: share one base W and store per task only $A^{(task)}, B^{(task)} \rightarrow$ per task cost $r(d+k)$.

Then per task storage is much smaller with LoRA.

(d)(i) Workflow for version control and efficiency

One reasonable design:

- Keep a single frozen base model with weights $\{W_i\}_{i=1}^{N}$.
- For each new data batch $m = 1, .., M$:
  - Create a new set of LoRA adapters $\{A_i^{(m)}, B_i^{(m)}\}$ (initialized as in (e)) on top of the current best model (base + accepted adapters).
  - Fine tune these adapters on batch m.
  - Evaluate on a held out validation set.
- If performance improves, accept this version:

  - Either keep $\{A_i^{(m)}, B_i^{(m)}\}$ as the new active adapters, or merge them into the base weights and reset adapters.

- If performance degrades, discard the adapters and revert to the previous best.

Key advantage: Very lightweight rollback (just revert to previous adapters) and highly space efficient (store only low rank deltas per accepted version).

Key disadvantage: Sequential, batch by batch updates may not find the global optimum; each batch's adapter is optimized myopically and evaluation overhead might be high.

(d)(ii) Storage cost over a year

Assume worst case: all M batches produce accepted updates.

- Base model (shared): Ndk parameters.
- Per batch, per weight matrix LoRA parameters: $r(d + k)$.
- For all N matrices and M accepted batches: $Ndk + MNr(d + k)$ parameters total (ignoring optimizer states once training is done).

(d)(iii) LoRA vs full rank version control

If instead we store a full copy of the model for every accepted version:

- After M batches: store $M+1$ full models (initial + M versions): $(M+1)Ndk$ parameters.

With LoRA:

- Store one base model: Ndk.
- Plus M sets of LoRA deltas: $MNr(d + k)$.
- Total: $Ndk + MNr(d + k)$.

Since $r \ll \min(d, k)$, $Ndk + MNr(d + k) \ll (M + 1)Ndk$
for large M, meaning huge savings in model storage across many versions.

(e) Initialization of A and B

Standard LoRA initialization:

- B is initialized to zero.
- A is initialized with a small random distribution (e.g., Gaussian with small variance).

So initially:
$\Delta W = AB \approx 0$
and the effective weight $W' \approx W$, i.e., the model behaves exactly like the pretrained base before any fine tuning.

Advantage: Fine tuning starts from the exact pretrained model, ensuring stable behavior and no sudden performance drop.

Disadvantage: Because updates are initially very small (especially through zero initialized B), early learning may be slower; also, very low rank updates may limit expressiveness if r is too small.

3 Constrained Generation

Let $p_{\mathrm{lm}}(x)$ be the LM distribution and a an attribute.

(a) Bayes expression for $p_{\mathbf{lm}}(x \mid a)$

$$p_{\mathrm{lm}}(x \mid a) \frac{p_{\mathrm{lm}}(a \mid x)\, p_{\mathrm{lm}}(x)}{p_{\mathrm{lm}}(a)}.$$

(b) Why external classifier is an approximation

We approximate $p_{\mathrm{lm}}(a \mid x)$ with an external classifier $p_{\mathrm{clf}}(a \mid x)$.
This is only an approximation because:

- The classifier is trained on finite labeled data and may not match the true conditional distribution.
- It may not share the same data distribution or inductive biases as the LM.
- Calibration errors, modeling assumptions, and optimization issues mean $p_{\mathrm{clf}}(a \mid x) \neq p_{\mathrm{lm}}(a \mid x)$ in general.

So the classifier estimates the term but is not exactly equal to it.

(c) Sampling n sentences then reranking

Procedure: sample $x^{(1)}, ..., x^{(n)} \sim p_{\text{lm}}(x)$, then rerank or select those with high $p_{\text{clf}}(a \mid x)$.

Even if the classifier equals the true $p_{\text{lm}}(a \mid x)$, this does not produce faithful samples from $p_{\text{lm}}(x \mid a)$ because:

- Sampling are effectively from $p_{\text{lm}}(x)$ and then applying a deterministic selection (for instance, choosing the top scoring candidate), which yields something like $x^* = \arg\max_x p_{\text{lm}}(a \mid x) p_{\text{lm}}(x)$, i.e., a MAP sample, not a random draw from the full conditional.
- Finite n cannot explore full conditional support.
- With finite n, the candidate set is tiny relative to the support of $p_{\text{lm}}(x \mid a)$, so many high probability conditional samples are never considered.
- Even using weighting/importance sampling with a small n leads to high variance and approximate, not exact, sampling.

So this reranking approach fails to sample correctly from the conditional distribution; it tends to produce a few "best" examples rather than a representative set.

(d) Tokenwise conditioning: form of $p_{\mathbf{lm}}(x_i \mid x_{<i}, a)$

We know:

$$p_{\text{lm}}(x \mid a) \prod_{i=1}^{n} p_{\text{lm}}(x_i \mid x_{<i}, a).$$

Using Bayes on each step:

$$p_{\text{lm}}(x_i \mid x_{<i}, a) \frac{p_{\text{lm}}(a \mid x_{<i}, x_i)\, p_{\text{lm}}(x_i \mid x_{<i})}{p_{\text{lm}}(a \mid x_{<i})}.$$

Up to normalization w.r.t. x_i, this is:

$$p_{\text{lm}}(x_i \mid x_{<i}, a) \propto p_{\text{lm}}(a \mid x_{\leq i})\, p_{\text{lm}}(x_i \mid x_{<i}).$$

So to approximate it, we need a classifier that estimates:

$$p(a \mid x_{\leq i})$$

for any prefix $x_{\leq i}$.

Difference from classifier in (b):

- The earlier classifier takes a complete sentence x and outputs $p(a \mid x)$.
- This new classifier takes a prefix (partial sentence) $x_{\leq i}$ as input and predicts how likely the completed sentence will have attribute a. It must operate token wise / prefix wise, not just on full finished sequences.

This is the "future discriminator" idea in FUDGE: it scores continuations at each time step.

(e) Training the prefix classifier

We need training data of the form (prefix, a) or $(\text{prefix}, \text{label for } a)$.

A practical scheme:

- Start with a corpus of sentences labeled by attribute a.
  Example: movie reviews labeled as positive/negative.
- For many positions i, produce prefixes $x_{\leq i}$.
- For each labeled sentence (x, a) with tokens $x_1, \ldots, x_n$:
  - For a subset or all positions $i = 1, ..., n$, create a training example: (prefix $x_{\leq i}$, a).
  - Optionally, include negative prefixes from sentences with opposite labels.
- Train a classifier C that, given a prefix $x_{\leq i}$, outputs $p_C(a \mid x_{\leq i})$.
  - This can be implemented as a small LM or encoder that reads the prefix and predicts a with cross entropy loss.

During constrained generation:

- At each time step i, for each candidate next token, we form a hypothetical prefix $x_{\leq i}$ and evaluate $p_C(a \mid x_{\leq i})$.
- Combine that with the base LM next token probabilities as derived in (d) to bias sampling toward attribute consistent continuations.

This way, the classifier learns how prefixes evolve toward the target attribute and supports tokenwise conditional sampling.