Text/sequence Classification with (L)LM

Summer Methods Workshop

Taegyoon Kim | DHCSS, KAIST

Jul 31, 2025

Supervised Learning

We will focus on text classification with supervised learning

- Goal
  - To classify documents into pre-defined categories
  - E.g., sentiment of comments (e.g., positive-negative), stance on issues (e.g., for-against-neutral), etc.
- We need
  - Labeled data set (for training and/or testing)
  - Model that maps texts to labels
  - Evaluation approaches: performance metrics, cross-validation, etc.

Supervised Learning

Evolution of text classification

- Dictionary methods
  - Based on counting/weighting of relevant keywords
  - Readily available and fast ↔ sub-optimal performance
  - However, if you want to quick, preliminary analysis for concepts supported by an existing dictionary, it can be helpful
  - E.g., LIWC, VADER, Moral Foundations Dictionaries, etc.

Supervised Learning

Evolution of text classification (cont'd)

- Traditional ML algorithms
  - This approach provides the groundwork for many foundational concepts in text classification
  - Classifiers (models) are trained to learn the relationships between texts and labels (i.e., classes)
  - So this requires training (labeled) data (pairs of a text and a label)
  - (On average) more training data, higher performance
  - E.g., logistic regression, random forest, SVM, deep neural networks

Supervised Learning

Evolution of text classification (cont'd)

- Fine-tuning representation models (e.g., BERT family)
  - These models are pre-trained with massive amounts of text data
  - Given a text, representation models encode it into a vector (an array of numbers) that captures its meaning
  - We can fine-tune such a model for classification with potentially higher performance
  - They tend to achieve higher performance than the traditional ML approaches
  - It might still require (potentially large) labeled training data for high performance

Supervised Learning

Evolution of text classification (cont'd)

- Prompting generative models (e.g., GPT family)
  - Like representation models, these models are pre-trained on vast amounts of text data, often even more extensively
  - Generative models are designed to generate text outputs given input text prompts
  - We can prompt such a model with unlabeled texts to generate labels
  - Still requires labeled data: not for training but for testing performance
  - (It is also possible to "fine-tune" these models)

Overview of Process

Overall process

- Step 1: building a labeled data set
- Step 2: training model(s)
- Step 3: evaluating performance

Overview of Process

Overall process

- **Step 1: building a labeled data set**
- Step 2: training model(s)
- **Step 3: evaluating performance**

* Steps 1 and 3 apply to all of the classification approaches

Overview of Process

Step 1: build a labeled data set

- Label texts following systematic labeling guidelines and check inter-coder reliability
- This (C) will serve as "ground-truth" or "gold standards"
- C is used to training (building a model) and validation (evaluating performance)

Overview of Process

Step 1: build a labeled data set

Doc number	Text	y
1	This is great!	0
2	%@% off!	1
…		
9999	This is sick	0
10000	Love BTS <3	0

Overview of Process

Step 1: build a labeled data set

Doc number	Text	y
1	This is great!	0
2	%@% off!	1
…		
9999	This is sick	0
10000	Love BTS <3	0

Overview of Process

Step 2: train models

- Does not apply to prompting generative models
- Randomly split C into a training set (C_{train}) and a test set (C_{test})
  - Typically, C_{train} : C_{test} = 7:3 or 8:2
  - E.g., identifying YouTube comments containing hate speech
    - C: 10,000 comments labeled for the presence of hate speech
    - $C_{training}$: 8,000 comments for training
    - C_{test} : 2,000 comments for test
- Generate X_{train} (feature matrix) from C_{train}
  - E.g., count vectors, TF-IDF, or embeddings

Overview of Process

Step 2: train models

Index	y	Feature 1	Feature 2	…	Feature V-1	Feature V
1	0	3	1.4	…	1.7	6
2	1	-0.8	6.4	…	5.7	-1.6
…						
7999	0	-2.8	0.9	…	3.3	-0.6
8000	0	3.7	1.4	…	5.7	-5.8

Overview of Process

Step 2: train models

- Choose a model F (e.g., random forest) and learn model parameters β (e.g., an array of coefficients)
  - The model provides a mapping between X_{train} and y_{train}
- Loss (cost) function: measures how much model predictions ($\hat{y}_{train}$) differ from the true labels (y_{train})
  - $\hat{y}_{train} = F(\hat{\beta} * X_{train})$
  - β is estimated in a way that minimizes the difference between the two
- As a result, we get a classifier: $\hat{y} = F(\hat{\beta} * X)$

Overview of Process

Step 2: train models

Index	y	$\hat{y}$	Feature 1	Feature 2	…	Feature V-1	Feature V
1	0	1	3	1.4	…	1.7	6
2	1	1	-0.8	6.4	…	5.7	-1.6
…							
7999	0	0	-2.8	0.9	…	3.3	-0.6
8000	0	0	3.7	1.4	…	5.7	-5.8

Overview of Process

Step 3: evaluate performance

- We held out another labeled set C_{test} (n = 2,000) (**why?**)
- Use the classifier $F(\hat{\beta} * X)$ to generate predictions $\hat{y}_{test}$
- Compare the predictions $\hat{y}_{test}$ and the true labels y_{test}
- Performance metrics include accuracy, precision, recall, etc.
- (Then use the classifier for unlabeled data)

Overview of Process

Step 3: evaluate performance

Index	y	$\hat{y}$	Feature 1	Feature 2	…	Feature V-1	Feature V
1	1	1	3.12	1.99	…	5.77	0.36
2	1	0	-0.8	1.14	…	9.71	-1.66
…							
1999	0	0	-2.11	0.95	…	1.23	-0.62
2000	0	0	3.71	1.48	…	1.7	-5.84

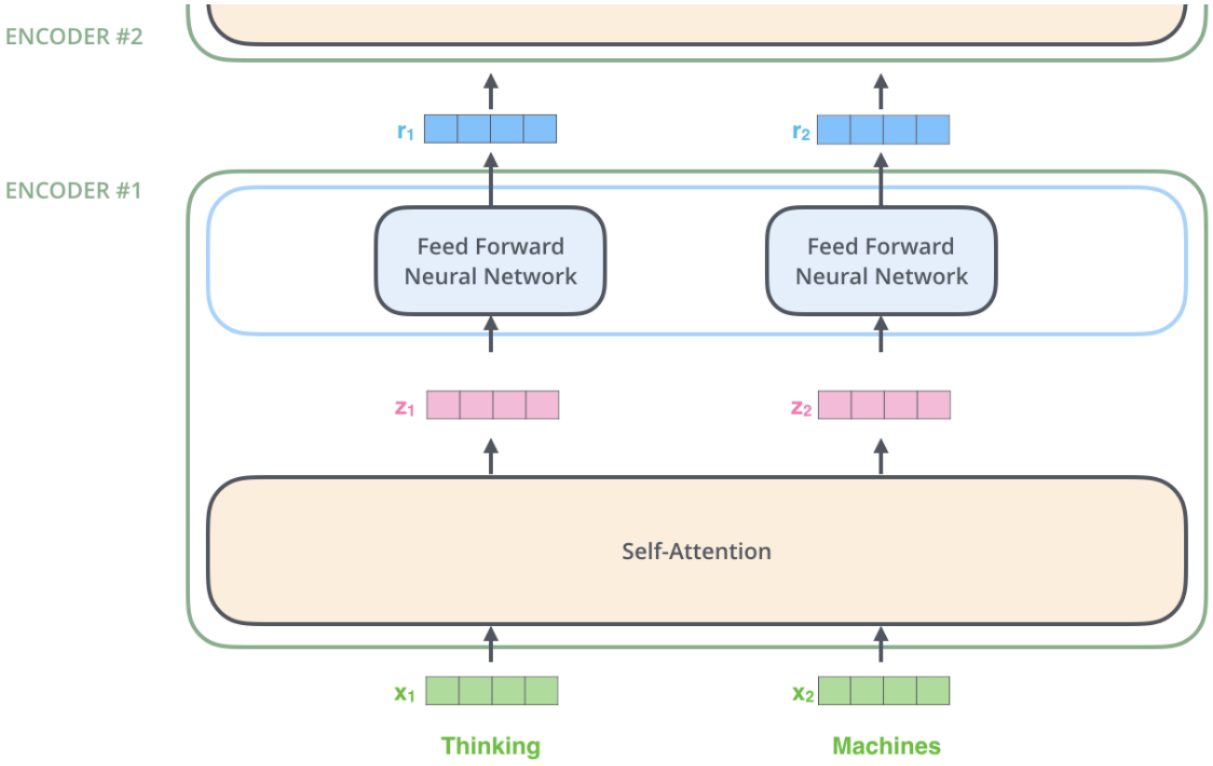
Overview of Transformer

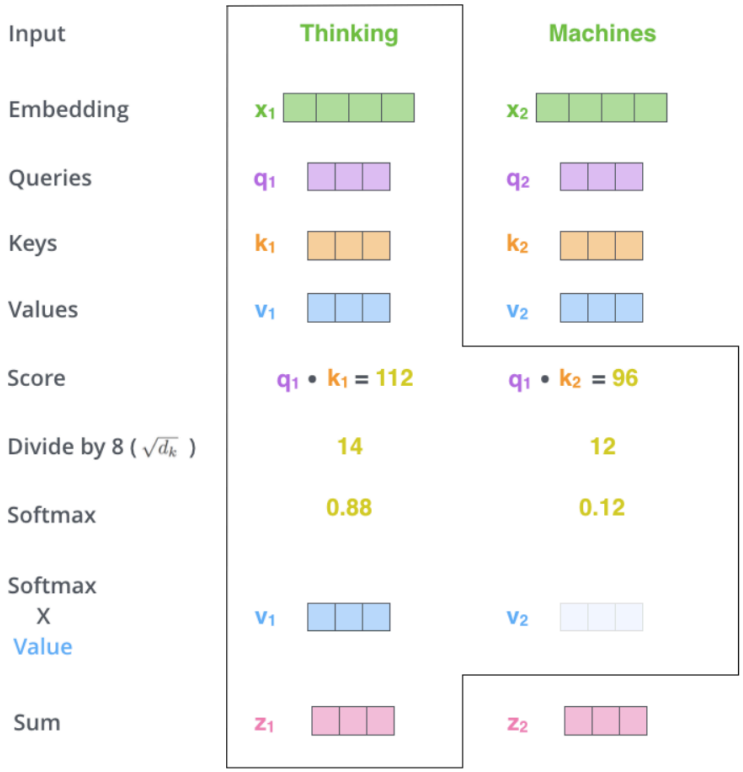
What is Transformer

- One of the key motivations is to capture contextual meanings of language
- Transformer is a type of neural network architecture with self-attention mechanisms (Vaswani et al. (2017))
- Transformers consist of a stack of (encoder/decoder) layers mainly consisting of self-attention and feed forward neural networks
- With self-attention, each word (or token) in a sentence (or sequence) is represented as a vector that reflects its relationships with all other words in the sequence

Overview of Transformer

Transformer encoder





Overview of BERT

Bi-directional **E**ncoder **R**epresentations from **T**ransformers

- A form of transformer trained as a language model
  - Trained based on two tasks: masked token prediction, next sentence prediction
  - Introduced by Devlin et al. (2018)
- Pre-trained on huge data sets (BookCorpus and Wikipedia)
- Two versions of the model introduced in the paper
  - BERT BASE (12 layers)
  - BERT LARGE (24 layers)

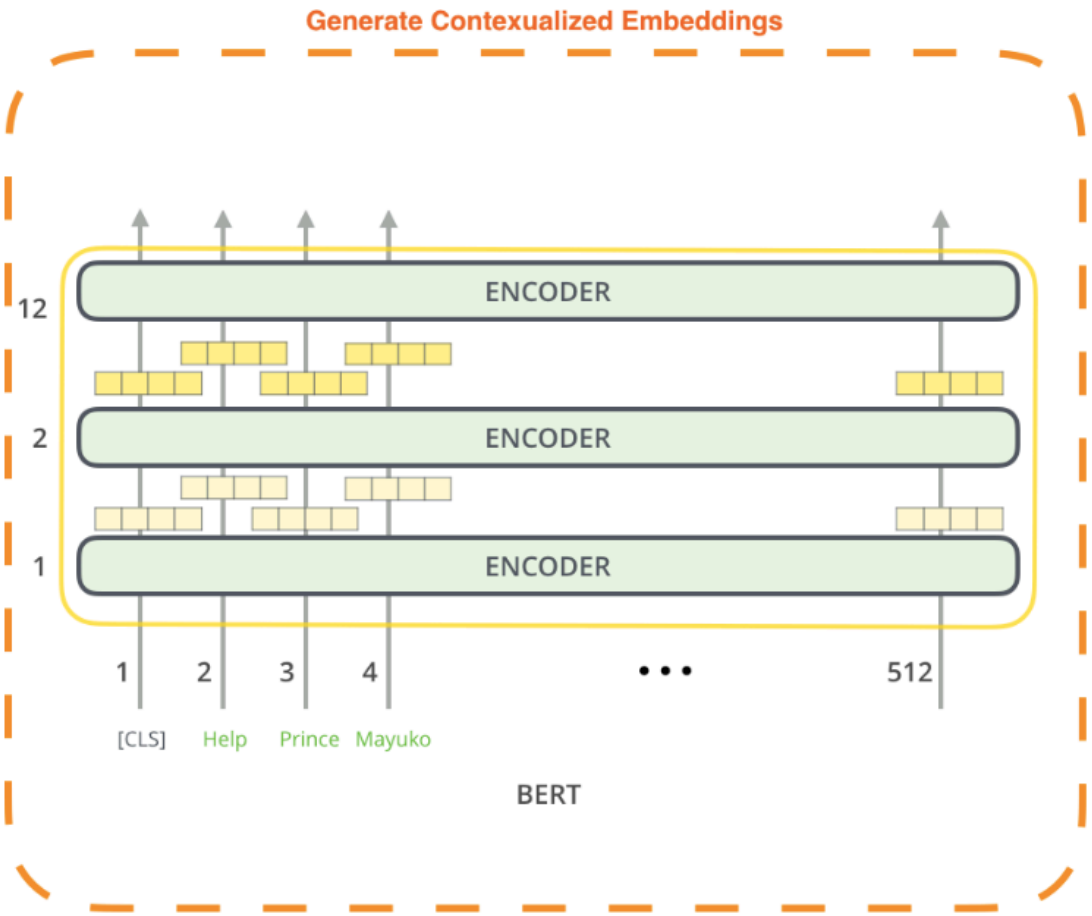
Overview of BERT

Peak at under the hood

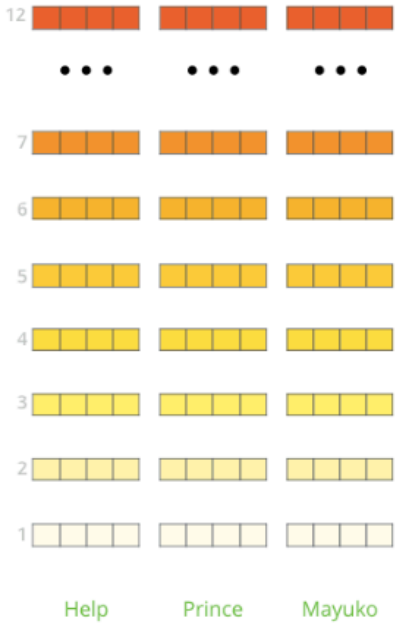
- BERT generates its own embeddings (from scratch) as part of its training process
- Stack of Transformer encoder layers involving “multi-head” self-attention
- Each layer passes its results through a feed-forward network, and then hands it off to the next encoder in the stack
- Each position outputs a vector of size 768 (BASE) or 1024 (LARGE)

Overview of BERT

Extracting embeddings



The output of each encoder layer along each token's path can be used as a feature representing that token.



But which one should we use?

Training BERT

Overview

- BERT is trained using two unsupervised objectives:
  - Masked Language Modeling for learning word-level representations
  - Next Sentence Prediction for learning sentence-level relationships
- MLM teaches BERT to understand context by predicting missing words in a sentence
- NSP teaches BERT to model discourse coherence by predicting if one sentence follows another
- Together, these tasks help BERT learn deep, contextual understanding of language

Training BERT

Task 1: Masked Language Model

- The MLM training objective is to predict the original inputs for each of the masked tokens
- Comparison with left-to-right/causal models
  - Please turn your homework _____.
  - MLM: Please turn _____ homework in.
- A form of denoising

Training BERT

Task 1: Masked Language Model

- 15% of the input tokens in a training sequence are sampled for learning
- Out of these:
  - 80% are replaced with `[MASK]`
  - 10% are replaced with another token from the vocabulary
    - randomly sampled based on token uni-gram probabilities
  - 10% are left unchanged

Training BERT

Task 1: Masked Language Model

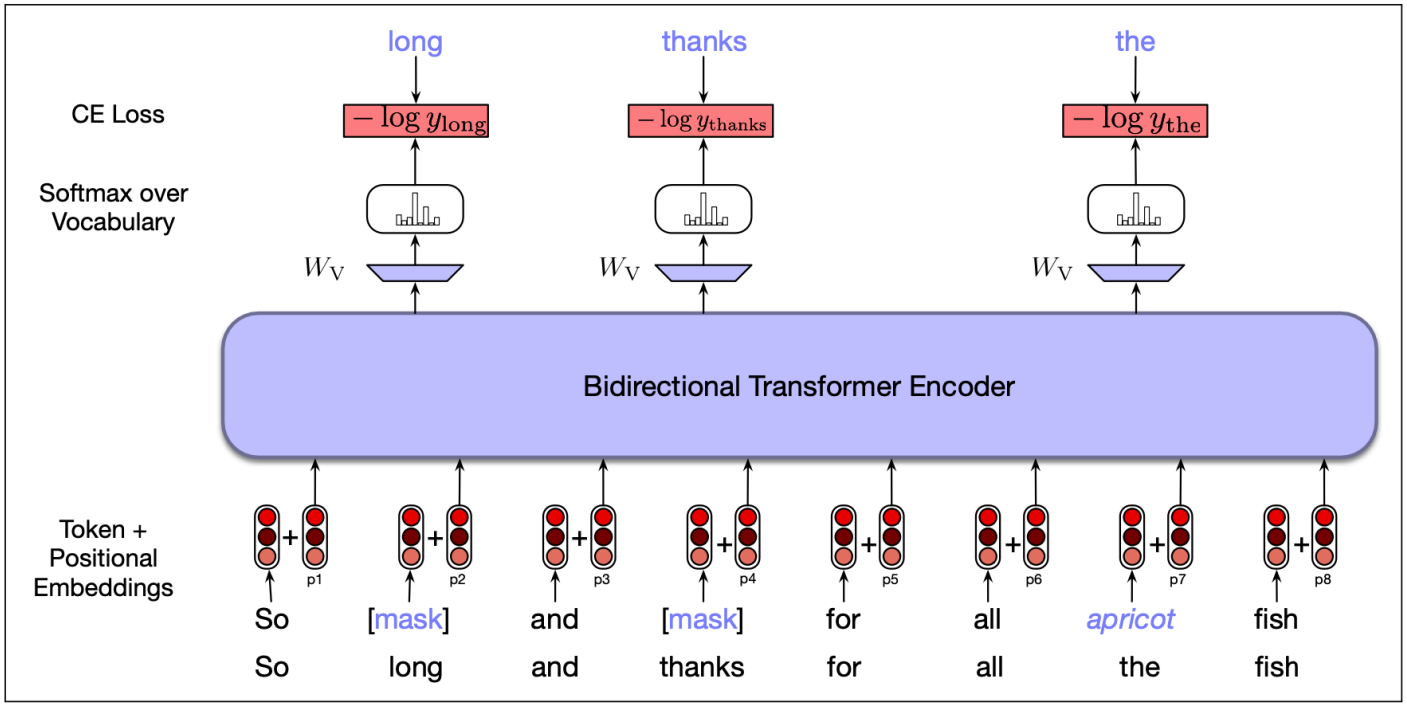


Figure 11.5 Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word. The probabilities assigned by the model to these three items are used as the training loss. (In this and subsequent figures we display the input as words rather than subword tokens; the reader should keep in mind that BERT and similar models actually use subword tokens instead.)

Training BERT

Task 2: Next Sentence Prediction

- MLM focuses on word-level representations
- However, many NLP applications rely on sentence-level understanding
- With NSP (Next Sentence Prediction) task
  - The model is trained to predict whether one sentence follows another
    - 50% of training pairs are actual consecutive sentences
    - 50% are random mismatches

Training BERT

Task 2: Next Sentence Prediction

- Input uses special tokens:
  - [CLS] at the beginning
    - [SEP] between and after the two sentences
    - Sentences A and B are marked with sentence IDs
- E.g., "[CLS] cancel my flight [SEP] And the hotel [SEP]"
- [CLS] token's final vector is used for NSP prediction
- Classification head (≈ a feed-forward neural network + softmax) outputs a binary label (next or not)

Training BERT

Task 2: Next Sentence Prediction

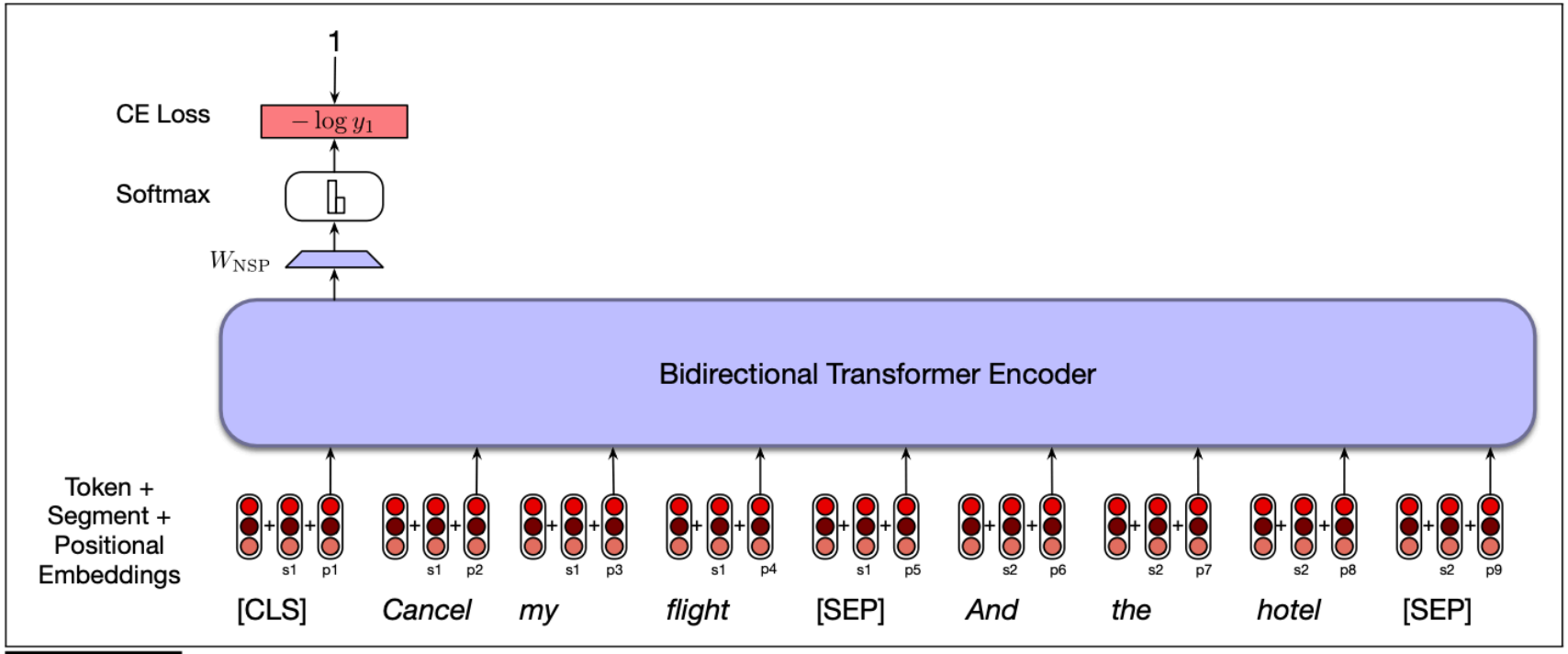


Figure 11.7 An example of the NSP loss calculation.

Fine-tuning BERT

What is fine-tuning, and why?

- Pre-trained language models (like BERT) capture general, transferable knowledge from massive text corpora
- This knowledge can be adapted for a wide variety of downstream tasks (e.g., classification, QA, NER)
- Fine-tuning adds task-specific layers and updates the model using supervised data from the target task
- Typically this training will
  - Either freeze
  - Or make only minimal adjustments to the pertained language model parameters

→ Stronger performance than training a new model from scratch, especially with limited data

Fine-tuning BERT

Overall process of fine-tuning BERT

- `[CLS]` token represents the entire input sequence
- It is a special token added to the start of input during fine-tuning (also during pre-training)
- An vector is learned for this token (a.k.a. “sentence embedding” since it refers to the entire sequence)
- The output vector in the final layer for `[CLS]` serves as representation of the entire input sequence (a summary representation)

Fine-tuning BERT

Overall process of fine-tuning BERT

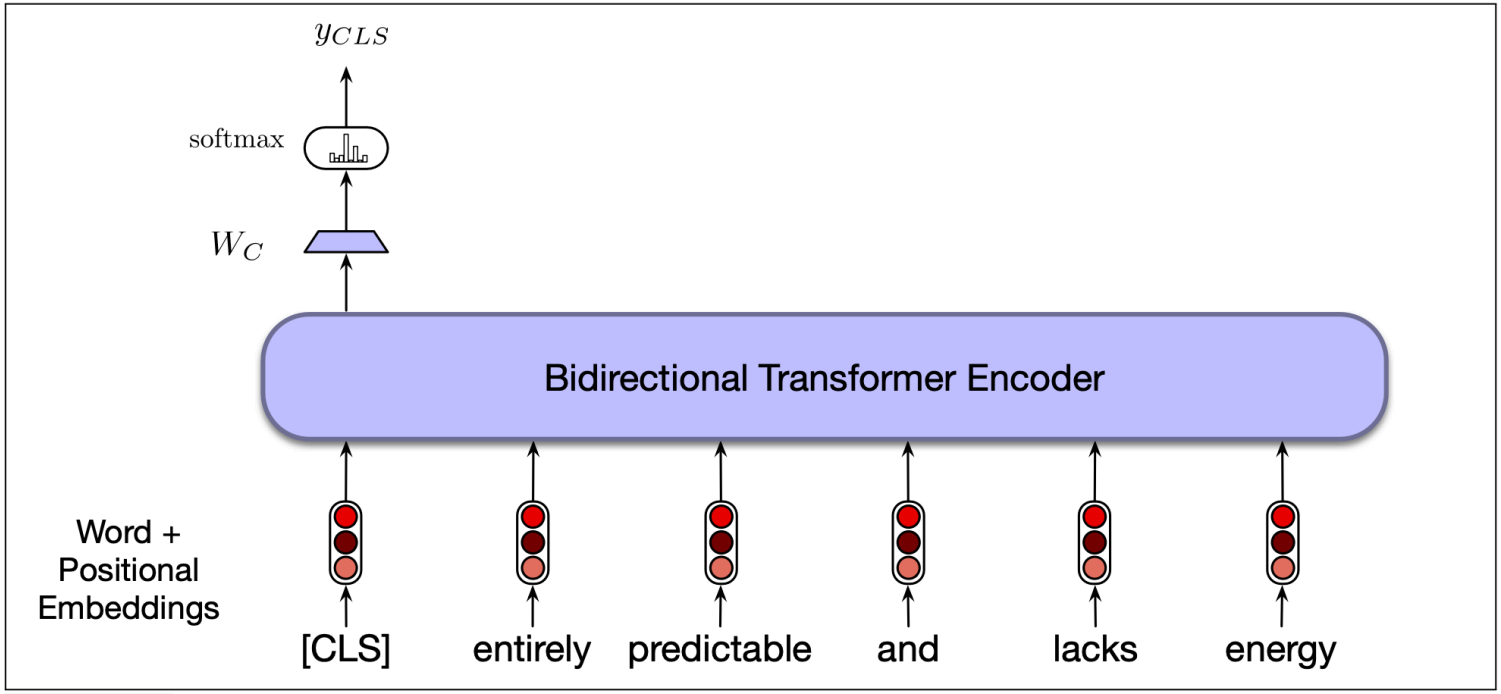


Figure 11.8 Sequence classification with a bidirectional transformer encoder. The output vector for the [CLS] token serves as input to a simple classifier.

Generative LLMs

What do we mean by LLMs?

- Overall, LLMs tend to refer to generative models
  - GPT, LLaMA, Claude, Gemini, etc.
- Various terminology: autoregrsesive, left-to-right, or causal models
- Autoregressive models
  - Use their earlier predictions to make later predictions (e.g., the model's first generated token is used to generate the second token)
- They are primarily decoder-only models
  ↔ encoder-only (representation) models (e.g., BERT)
  ↔ encoder-decoder (seq-to-seq) models (e.g., T5, BART)

Generative LLMs

Autoregressive models do not generate all at once

- They generate one token at a time (just like ChatGPT)
- Each token generation step is one forward pass through the model
  - The input tokens go into the model and flow through the computations to produce an output
- After each token generation, the generated token is appended to the end of the original input from the previous run
- So the model is run in a loop to sequentially expand the generated text until completion

Generative LLMs

Autoregressive models do not generate all at once

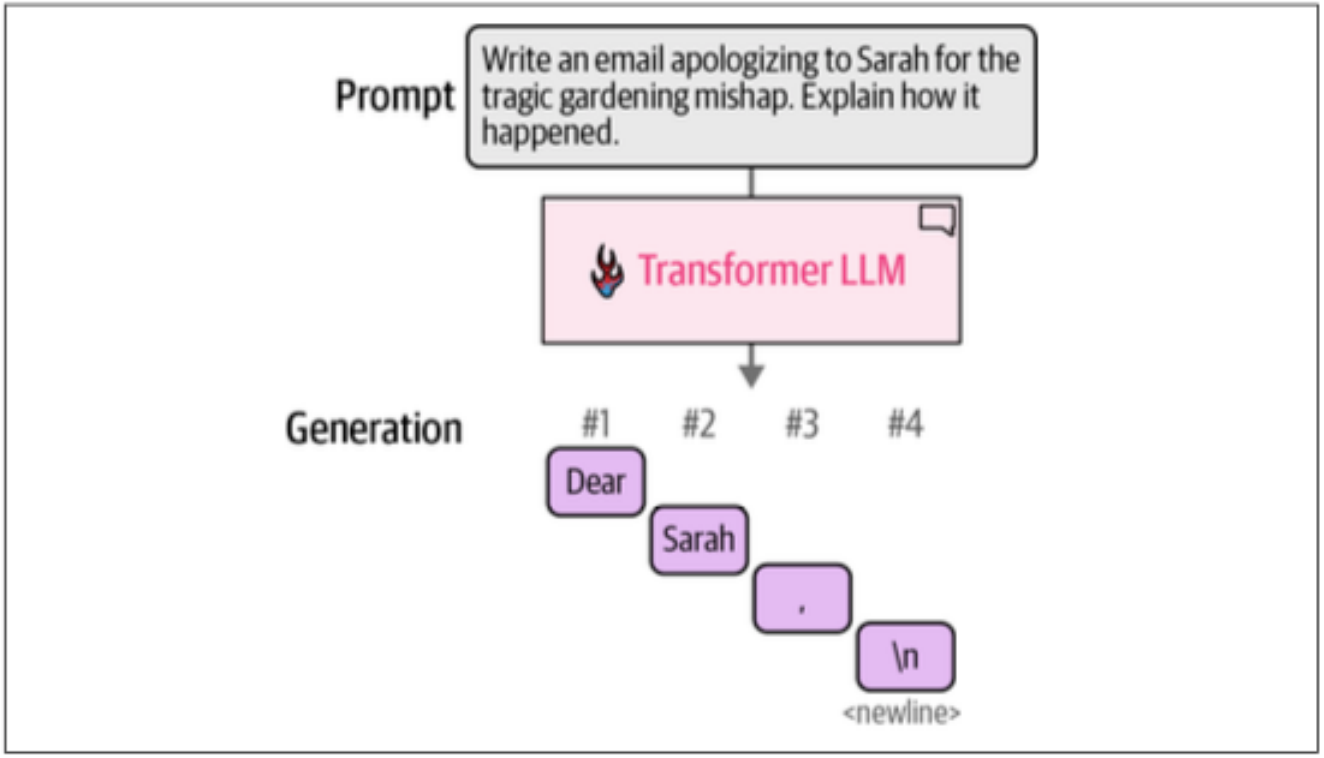


Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

Generative LLMs

Autoregressive models do not generate all at once

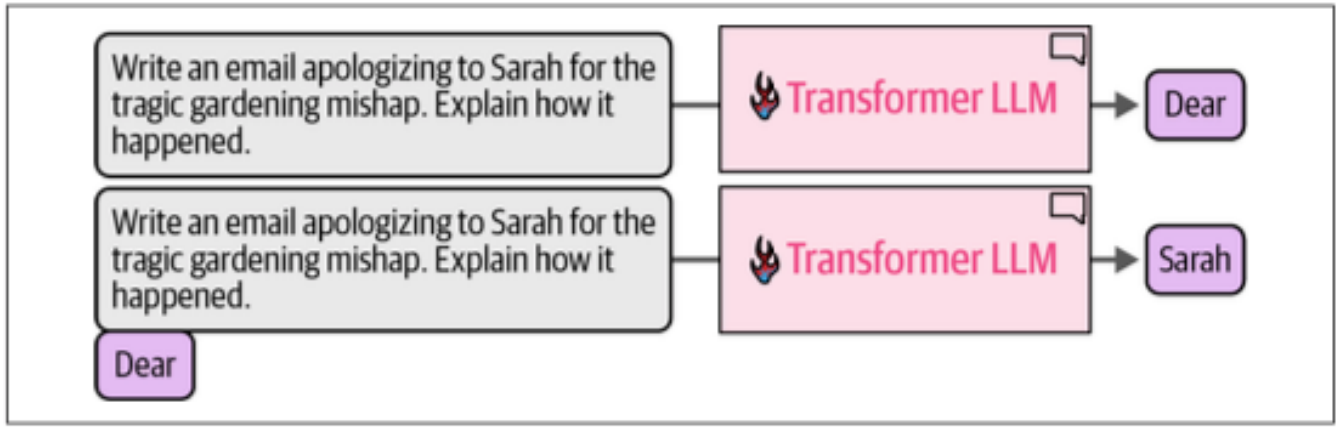


Figure 3-3. An output token is appended to the prompt, then this new text is presented to the model again for another forward pass to generate the next token.

Generative LLMs

How is the next token predicted?

- Only the final token's output vector (= vector representation) is passed into the final layer (language model head) to predict the next token
- This head produces a probability distribution over the vocabulary
- Then, why compute representations for all tokens if we discard all but the last?
  - The final output vector incorporates contextual information from all preceding tokens
  - This allows the vector to reflect the overall meaning of the input

Generative LLMs

How is the next token predicted?

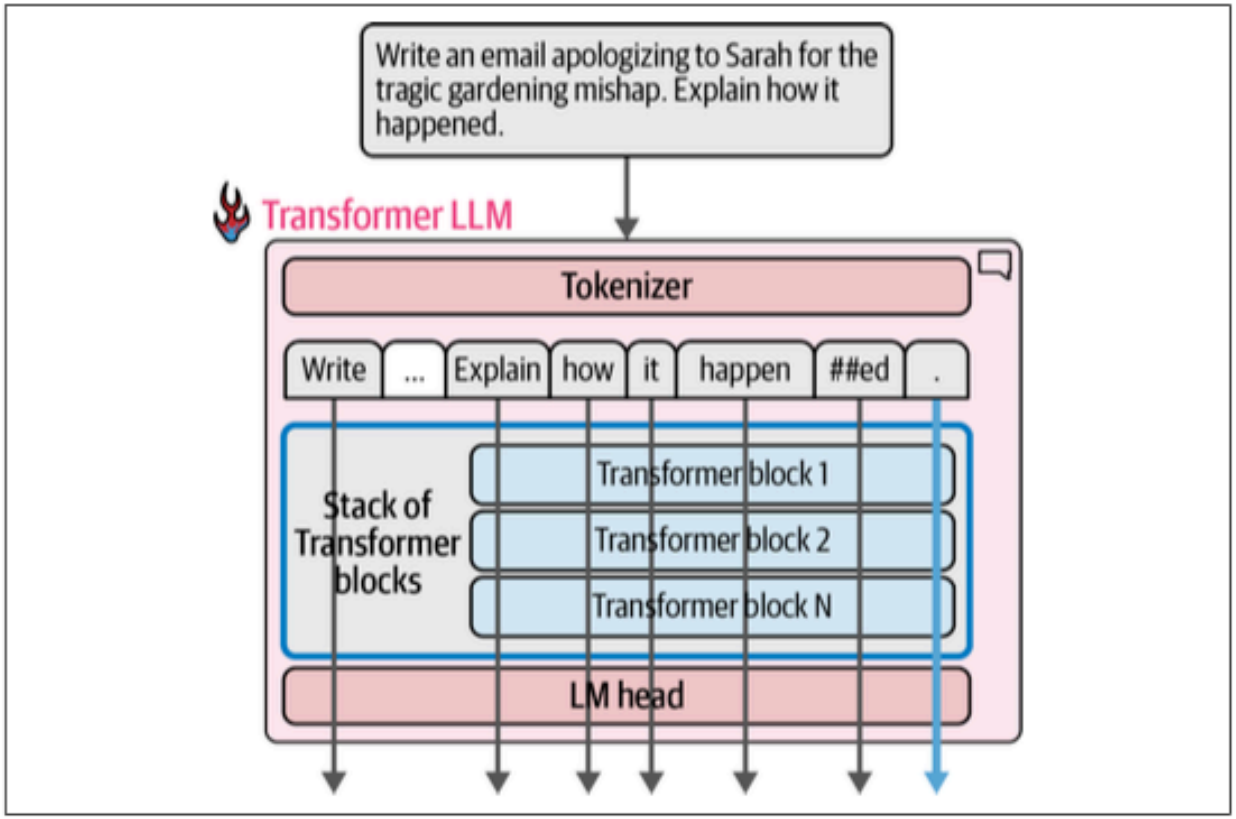


Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

Generative LLMs

How is the next token predicted?

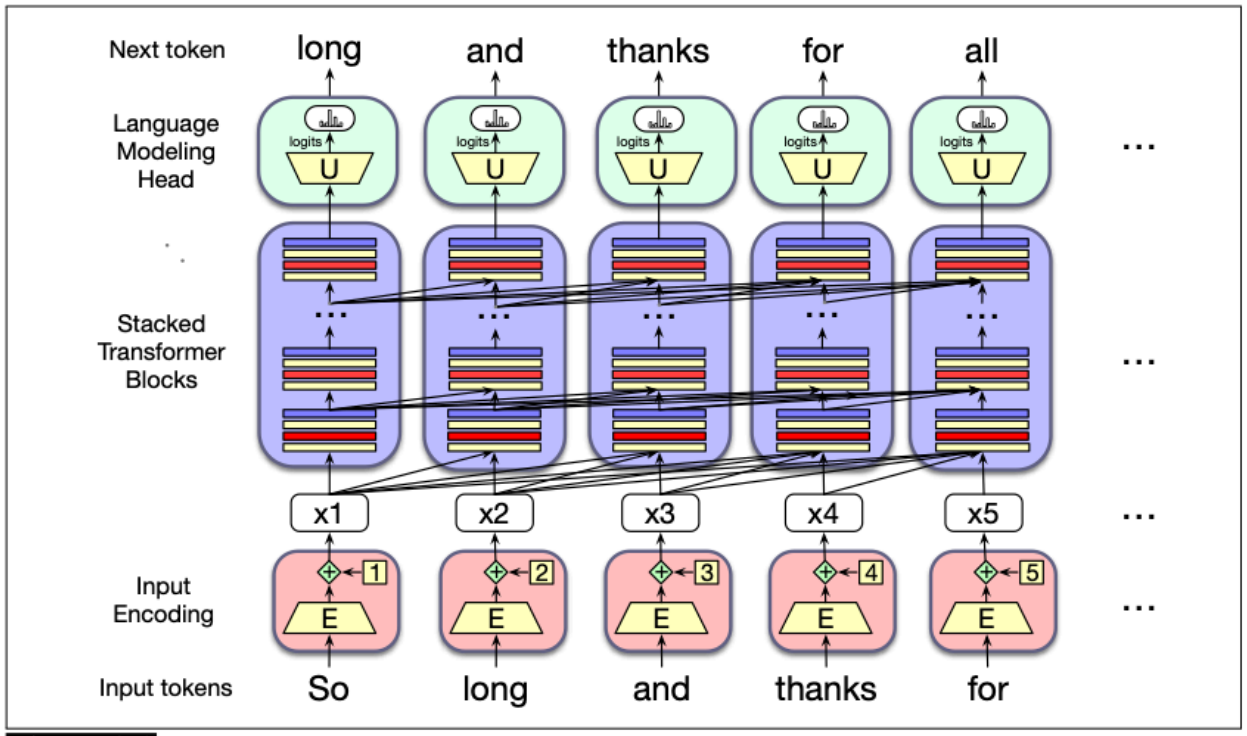


Figure 9.1 The architecture of a (left-to-right) transformer, showing how each input token get encoded, passed through a set of stacked transformer blocks, and then a language model head that predicts the next token.

Attention in Generative Models

Simplified illustration

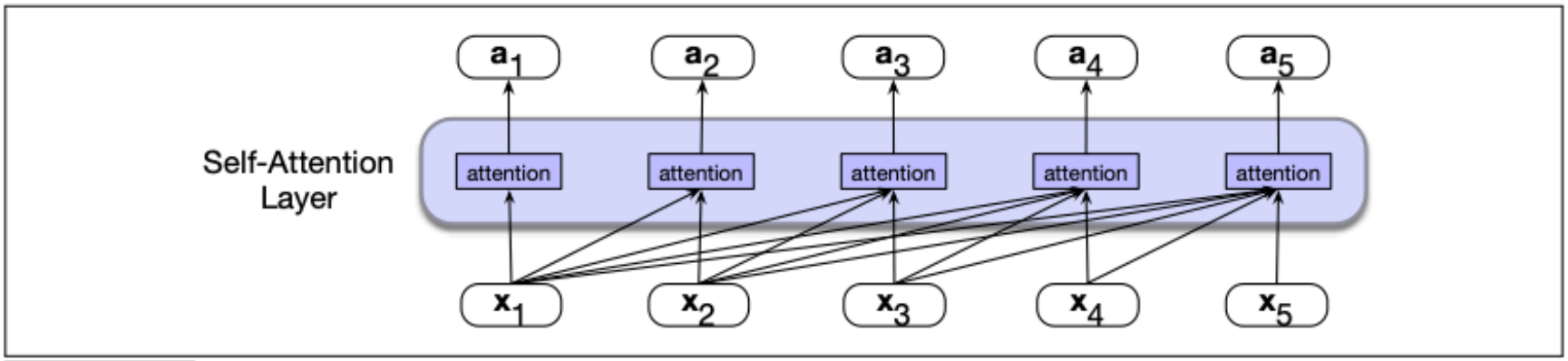


Figure 9.3 Information flow in causal self-attention. When processing each input $\mathbf{x}_i$, the model attends to all the inputs up to, and including $\mathbf{x}_i$.

Attention in Generative Models

Simplified illustration

$$a_i = \sum_{j \leq i} \alpha_{ij} \cdot x_j$$

Where:

- a_i is the updated representation for token i
- x_j is the input representation of token j
- α_{ij} is the attention weight from token i to token j

Attention in Generative Models

Self-attention

- The weights are computed using similarity scores between tokens
- The dot product is used to measure how similar tokens i and j are
- Normalize these scores across all tokens using softmax
- This results in a probability distribution over all tokens j, indicating how much attention token i pays to each

$$\alpha_{ij} = softmax(\text{score}(x_i, x_j)) \forall j \leq i$$

Attention in Generative Models

This is a simplified view, and actual attention involves learned projection matrices

- In practice, each attention head uses three distinct learned weight matrices:
  - W^Q (query), W^K (key), and W^V (value)
- Each input vector x_i is projected into three new vectors:
  - Query: $q_i = x_i W^Q$
  - Key: $k_i = x_i W^K$
  - Value: $v_i = x_i W^V$
- The attention weights are calculated using the dot product between the current token's query q_i and all previous keys k_j (for j $\leq$ i)

- These scores are normalized via softmax to produce weights α_{ij}, which are used to compute a weighted sum of the value vectors:

$$a_i = \sum_{j \leq i} \alpha_{ij} \cdot v_j$$

- For a more detailed explanation, see pp. 5–7 in Ch. 9 of [JM]

Language Modeling Head

Once we have a vector representation for the last token

- This should be mapped to a probability distribution over $|V|$
- The task of the language modeling head is to take the output of the final transformer layer from the last position and use it to predict the upcoming word at the next position
- It takes the output of the last token at the last layers and produces a probability distribution over $|V|$
- Details can be found pp. 16–18 [JM]

Language Modeling Head

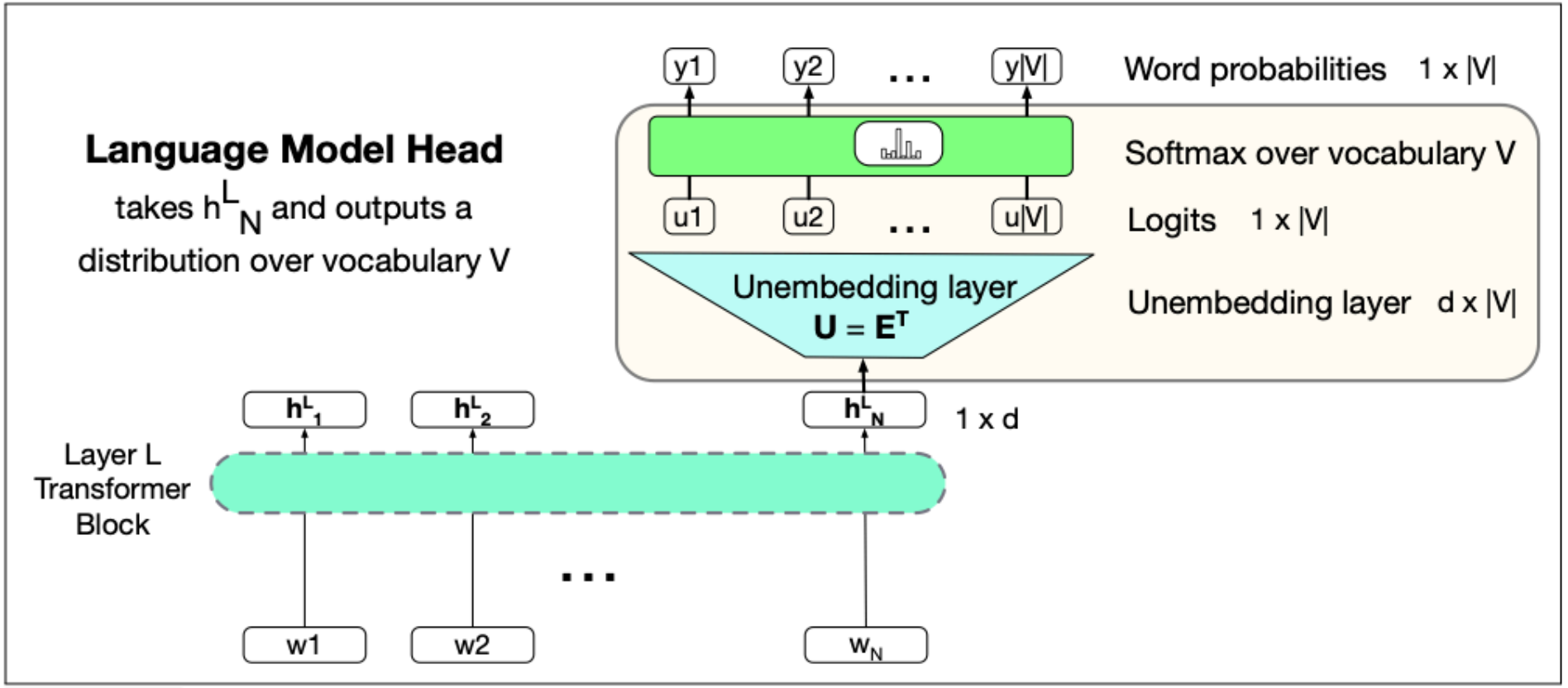


Figure 9.14 The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer ($\mathbf{h}_N^L$) to a probability distribution over words in the vocabulary V.

Decoding

The task of choosing a word to generate based on the model-generated probabilities

- Greedy decoding
  - Always selects the token with the highest probability at each step
    - Simple and fast, but can lead to bland outputs
    - Deterministic: given the same prompt, always produces the same output
- Sampling-based decoding
  - Selects the next token by sampling from the probability distribution
  - Allows for more diverse and creative text generation
  - Stochastic: same prompt may produce different outputs each time

Decoding

Varieties of sampling-based decoding

- Pure/random sampling
  - Sample directly from the probability distribution
  - *How can this go wrong?*
- Top-k sampling
  - We truncate the distribution to the most likely tokens, re-normalize to produce a legitimate probability (= adding up to 1), and then randomly sample form within these k words
  - When k=1, top-k sampling = greedy decoding
  - When $k > 1$, it leads to choosing a token that is not necessarily the most probable

Decoding

Varieties of sampling-based decoding (cont'd)

- Top-p sampling
  - Depending on the distribution, the most probable tokens from top-k sampling will only include a fraction in the probability distribution
  - Top-p sampling (a.k.a. nucleus sampling) keeps the top p percent of the probability mass
  - Likely more robust with various shapes of of the probability distribution
  - Higher p yields more randomness

Decoding

Varieties of sampling-based decoding (cont'd)

- Temperature sampling
  - Originates from thermodynamics (a system at a high temperature is flexible)
  - Instead of truncating the distribution, we reshape it
  - For more randomness, we make the distribution flatter
  - Higher temperature parameter, τ, indicates more randomness ($\tau = 0$: greedy decoding)

Generative LLMs for Text Classification

Promises

- No need for labeled training data (zero-shot/few-shot prompting)
- High performance in various tasks (classification, sentiment, ideology)
- Little programming or machine learning expertise required
- Multi-lingual and adaptable across domains

Generative LLMs for Text Classification

Pitfalls

- Transparency and replicability issues with closed models (e.g., GPT-4)
- Computational power (open-source models) or API usage fees (closed-source, proprietary models)
- Lack of validation framework: ↔ Traditional ML tools (e.g., train-test splits, CV) do not directly apply
- Small prompt changes can produce drastically different results

Generative LLMs for Text Classification

Recommended pipeline (Törnberg et al. 2024)

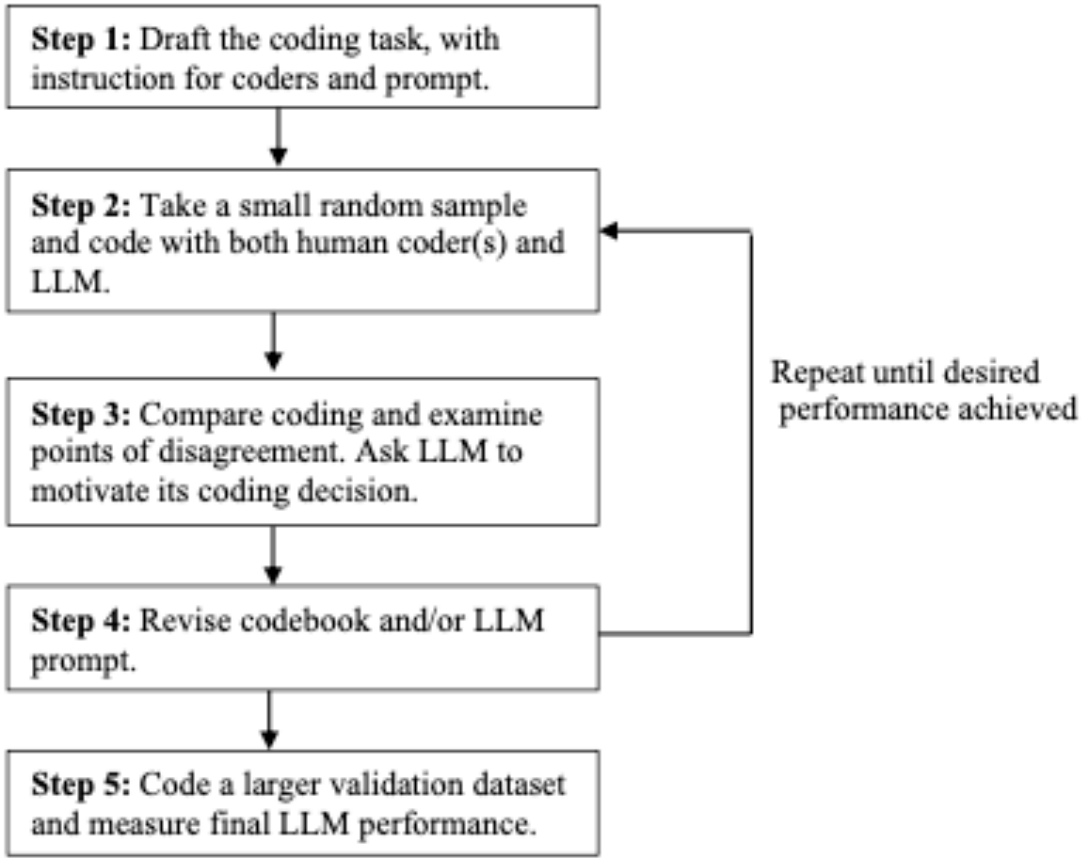


Figure 1: Example of a systematic coding procedure.

Tutorial Materials

- Stance detection on abortion with BERT fine-tuning: here
- Introduction to text classification with generative models: link