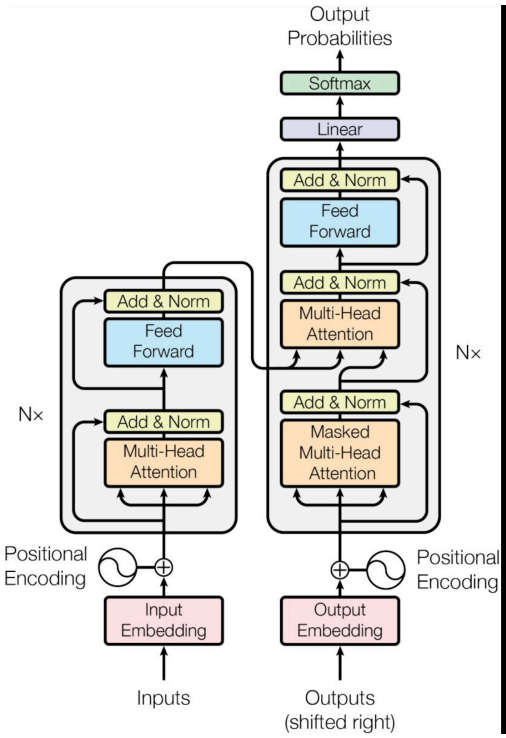
Transformer-based LM

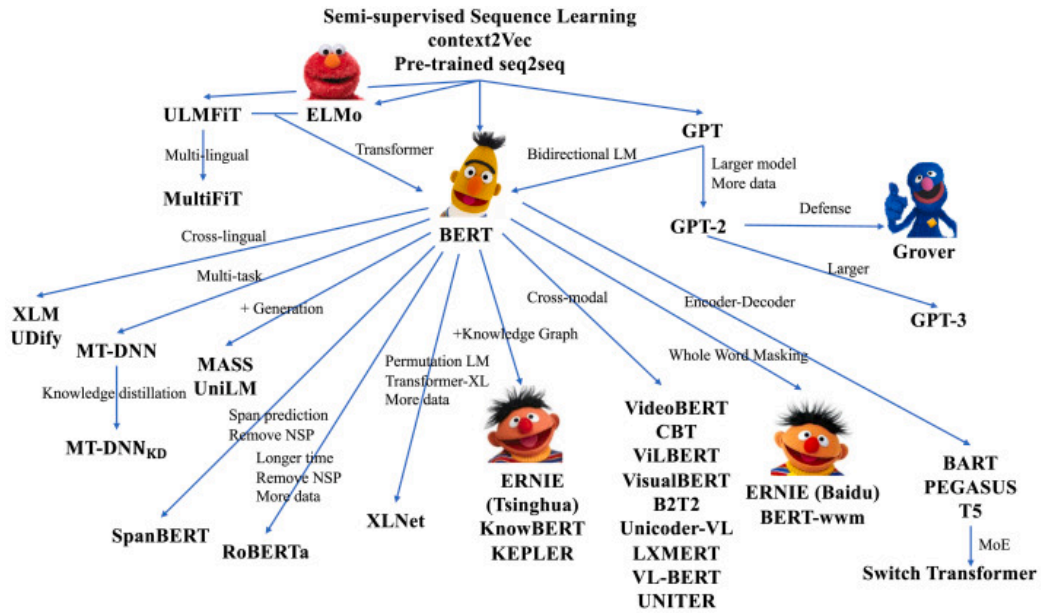
Seepch and Language Processing (3rd ed. draft) by Jurafsky and Martin (https://web.stanford.edu/~jurafsky/slp3/)

The Illustrated Transformers (https://jalammar.github.io/illustrated-transformer/)

Round Up : Transformer



Pre-training Language Models based on Transformer



Self-supervised training algorithm

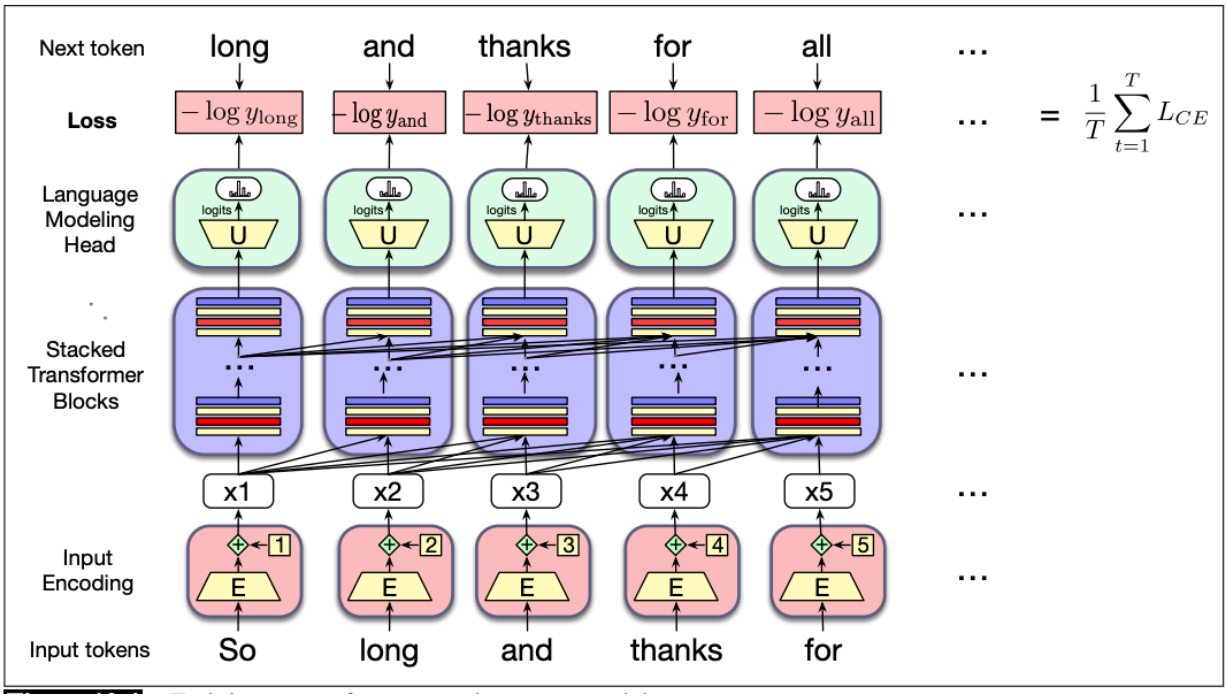
- we take a corpus of text as training material and at each time step t ask the model to predict the next word.
- We call such a model **self-supervised** because we don't have to add any special gold labels to the data; the natural sequence of words is its **own supervision!**

- The cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$$

- In the case of language modeling, the correct distribution yt comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, **the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word (all other words get multiplied by zero).**
- the cross-entropy can be simplied as the negative log probability the model assigns to the next word in the training sequence, using one-hot vector.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$



Training a transformer as a langage model by SLP3

Training Corpora for langauge models

- Training corpora are so large, they are likely to contain many natural examples that can be helpful for NLP tasks, such as question and answer pairs (for example from FAQ lists),

translations of sentences between various languages, documents together with their summaries, and so on

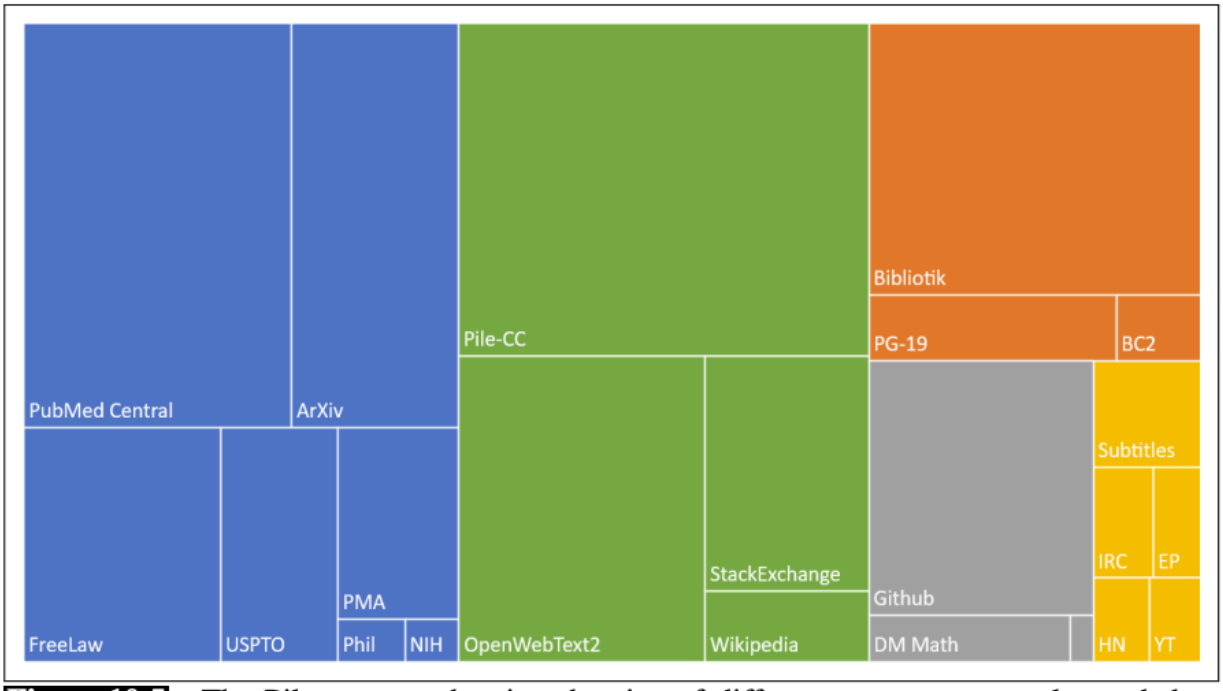


Figure 10.5 The Pile corpus, showing the size of different components, color coded as academic (articles from PubMed and ArXiv, patents from the USPTA; internet (webtext including a subset of the common crawl as well as Wikipedia), prose (a large corpus of books), dialogue (including movie subtitles and chat data), and misc.. Figure from Gao et al. (2020).

- Using large datasets scraped from the web to train language models poses ethical and legal questions:
- **Copyright**: Much of the text in these large datasets (like the collections of fiction and non-fiction books) is copyrighted. In some countries, like the United States, the fair use doctrine may allow copyrighted content to be used for transformative uses, but it's not clear if that remains true if the language models are used to generate text that competes with the market for the text they are trained on
- **Data consent**: Owners of websites can indicate that they don't want their sites to be crawled by web crawlers (either via a robots.txt file, or via Terms of Service).
- Recently there has been a sharp increase in the number of websites that have indicated that they don't want large language model builders crawling their sites for training data.
- Because it's not clear what legal status these indications have in different countries, or whether these restrictions are retroactive, what effect this will have on large pretraining datasets is unclear.
- **Privacy**: Large web datasets also have privacy issues since they contain private information like phone numbers and IP addresses. While filters are used to try to remove websites likely to contain large amounts of personal information, such filtering isn't sufficient.

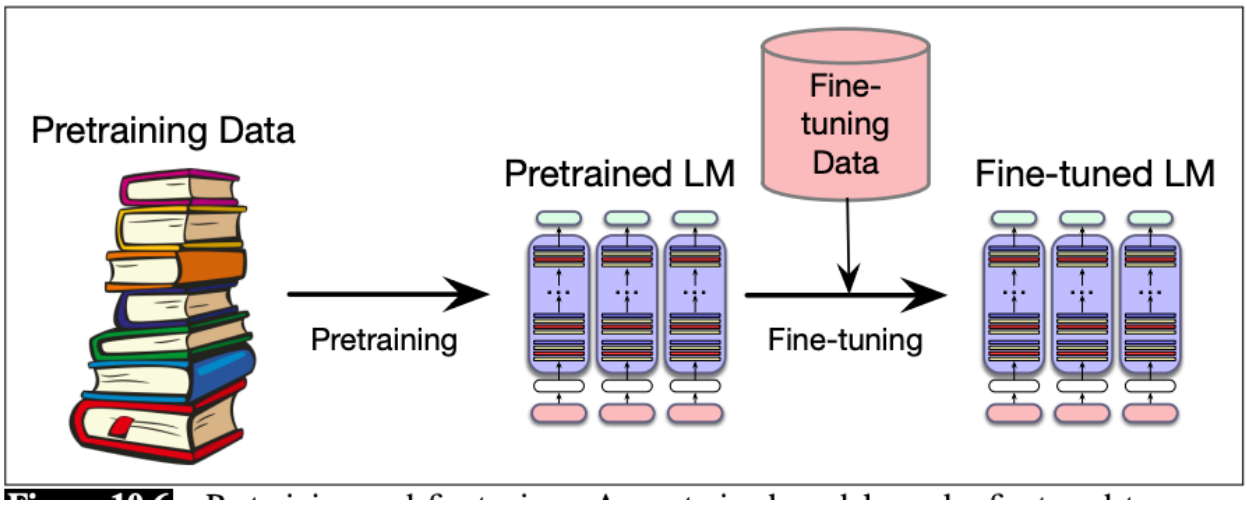
Finetuning

- Although the enormous pretraining data for a large language model includes text from many domains, **it's often the case that we want to apply it in a new domain or task that might not have appeared sufficiently in the pre-training data.**
- For example, we might want a language model that's specialized to **legal or medical text.**

- Or we might have a **multilingual language model** that knows many languages but might benefit from some more data in our particular language of interest.
- Or we want a language model that is specialized to a particular task.

Finetuning

- This process of taking a fully pretrained model and running additional training passes on some new data is



Pretraining and Finetuning

Four kinds of finetuning

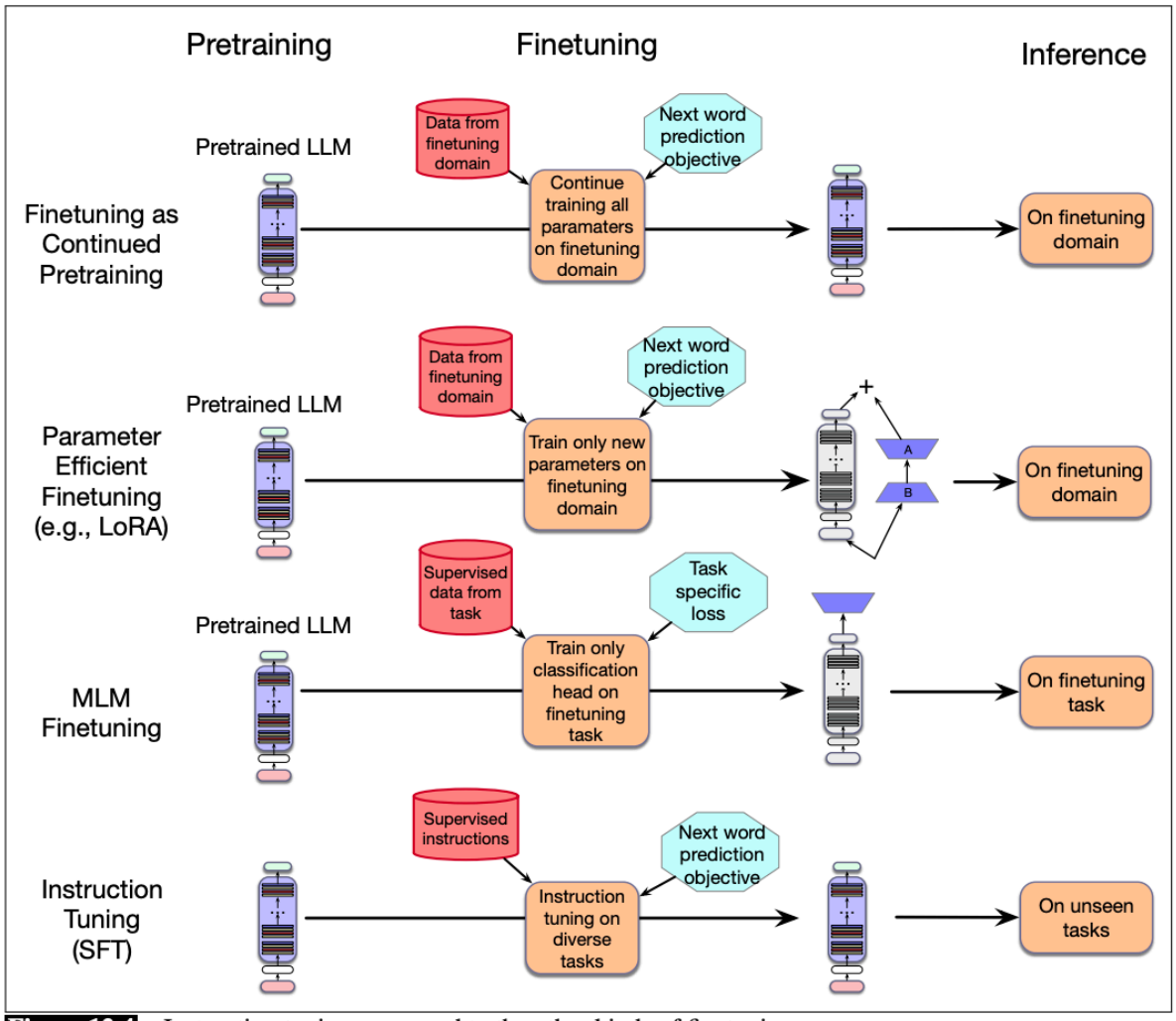


Figure 12.4 Instruction tuning compared to the other kinds of finetuning.

Parameter Efficient Fine Tuning

- Fine-tuning can be very difficult with very large language models, because there are enormous numbers of parameters to train; each pass of batch gradient descent has to backpropagate through many many huge layers.
- This makes finetuning huge language models extremely expensive in processing power, in memory, and in time.
- For this reason, there are alternative methods that allow a model to be finetuned without changing all the parameters. Such methods are called **parameter-efficient fine tuning or sometimes PEFT**, because we efficiently select a subset of parameters parameterefficient when fine tuning.

- Here we describe one such model, called LoRA, for Low-Rank Adaptation.
- The intuition of LoRA is that transformers have many dense layers which perform matrix multiplication (for example the WQ, WK, WV, WO layers in the attention computation).
- Instead of updating these layers during finetuning, with LoRA we freeze these layers and instead update a low-rank approximation that has fewer parameters.
- Consider a matrix W of dimensionality [N ×d] that needs to be updated during finetuning via gradient descent. Normally this matrix would get updates ΔW of dimensionality [N ×d], for updating the N ×d parameters after gradient descent.
- In LoRA, we freeze W and update instead a low-rank decomposition of W.
- We create two matrices A and B, where A has size [N ×r] and B has size [r×d], and we choose r to be quite small, r << min(d,N).

- During finetuning we update A and B instead of W. That is, we replace W + ΔW with W + BA.
- For replacing the forward pass h = xW, the new forward pass is instead:

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB}$$

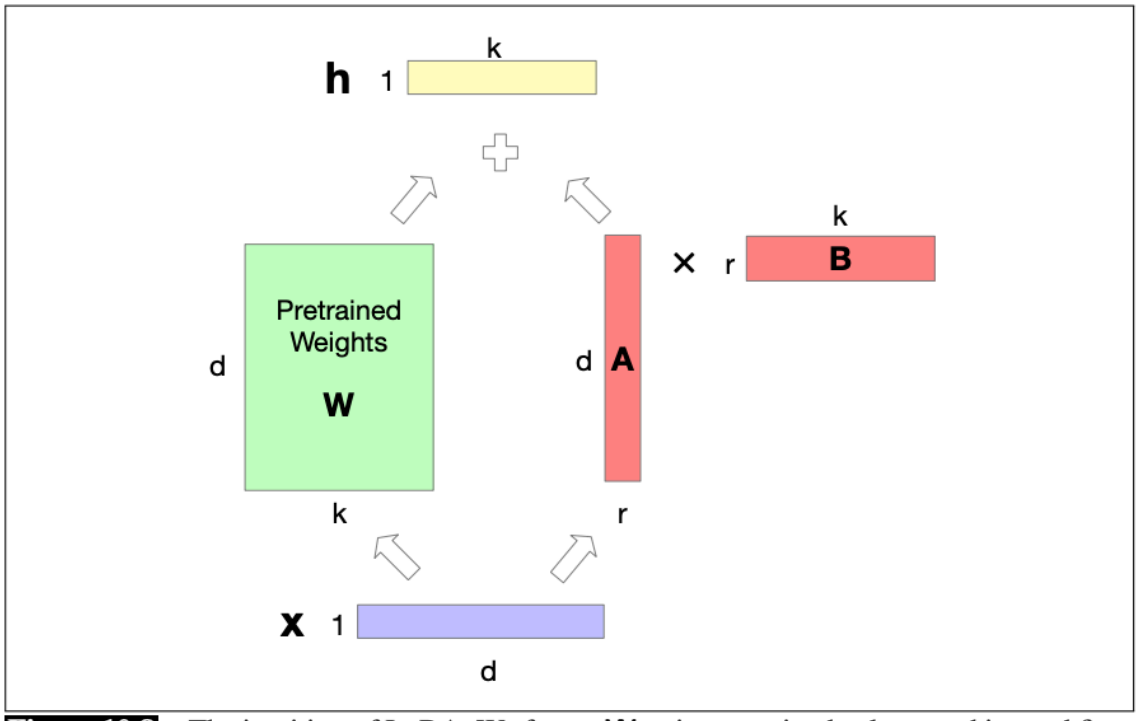


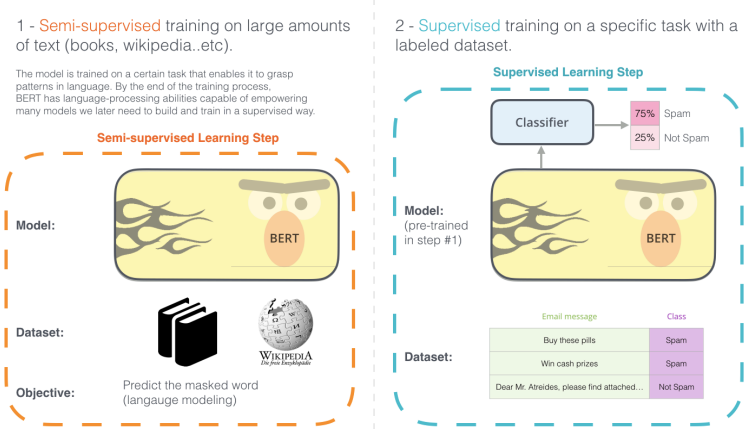
Figure 10.8 The intuition of LoRA. We freeze **W** to its pretrained values, and instead fine-tune by training a pair of matrices **A** and **B**, updating those instead of **W**, and just sum **W** and the updated **AB**.

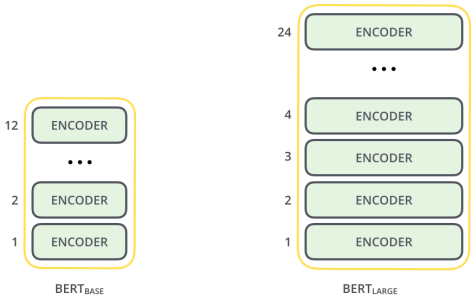
- LoRA has a number of advantages.
- It dramatically reduces hardware requirements, since gradients don't have to be calculated for most parameters. - - The weight updates can be simply added in to the pretrained weights, since BA is the same size as W).
- That means it doesn't add any time during inference.
- And it also means it's possible to build LoRA modules for different domains and just swap them in and out by adding them in or subtracting them from W.
- In its original version LoRA was applied just to the matrices in the attention computation (the WQ, WK, WV, and WO layers).
- Many variants of LoRA exist

Masked Language Model : BERT (Bidirectional Encoder Representations of Transformer)

BERT

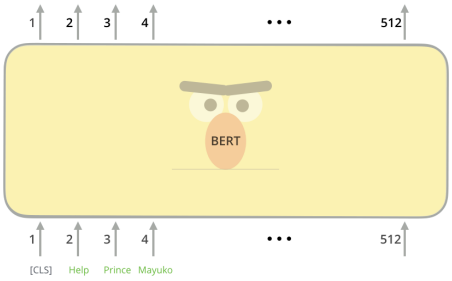
BERT is basically a trained Transformer Encoder stack.



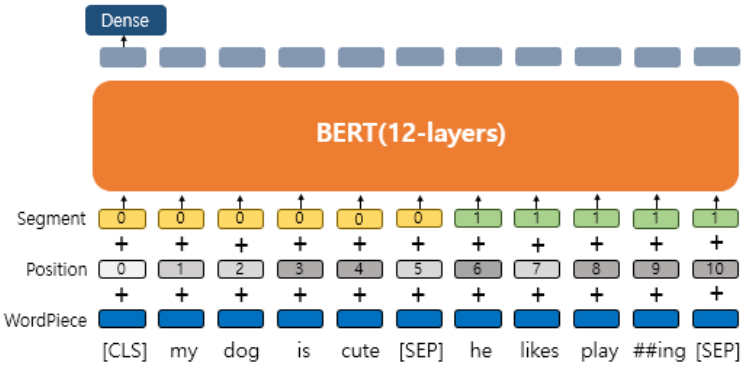


Both BERT model sizes have a large number of encoder layers (which the paper calls Transformer Blocks) – twelve for the Base version, and twenty four for the Large version. These also have larger feedforward-networks **(768 and 1024 hidden units respectively)**, and more attention heads **(12 and 16 respectively)** than the default configuration in the reference implementation of the Transformer in the initial paper **(6 encoder layers, 512 hidden units, and 8 attention heads).**

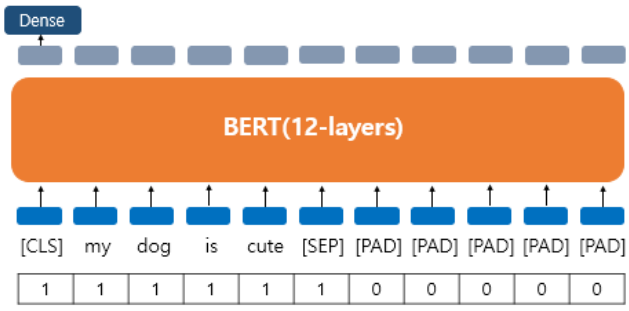
Model Inputs



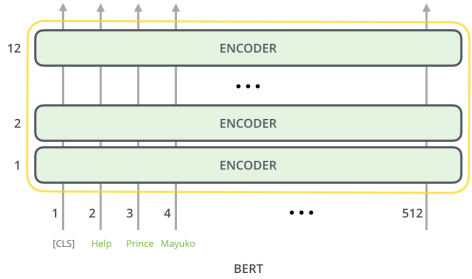
The first input token is supplied with a special [CLS] token for reasons that will become apparent later on. CLS here stands for Classification.



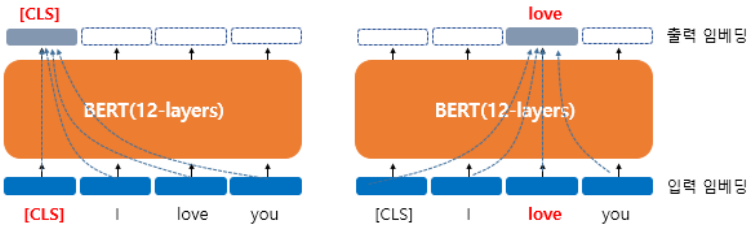
- Attention Mask



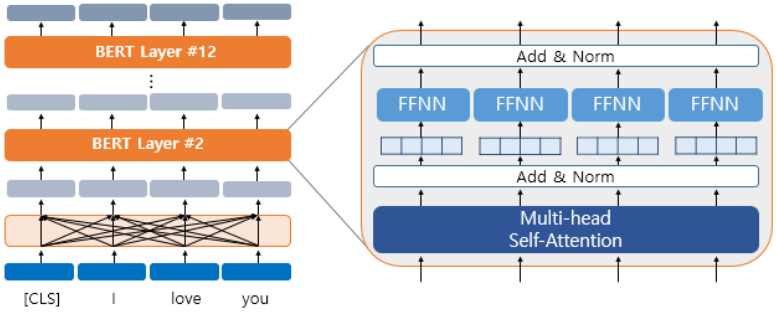
Just like the vanilla encoder of the transformer, BERT takes a sequence of words as input which keep flowing up the stack. Each layer applies self-attention, and passes its results through a feed-forward network, and then hands it off to the next encoder.



In terms of architecture, this has been identical to the Transformer up until this point (aside from size, which are just configurations we can set). It is at the output that we first start seeing how things diverge.



- Transformer-based



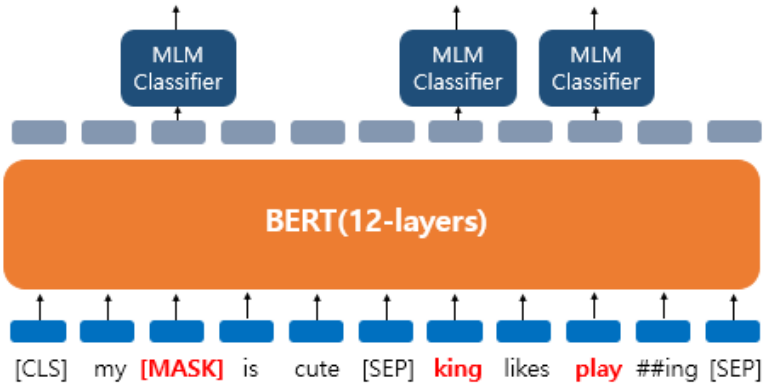
A - Masked Language Modeling (MLM)

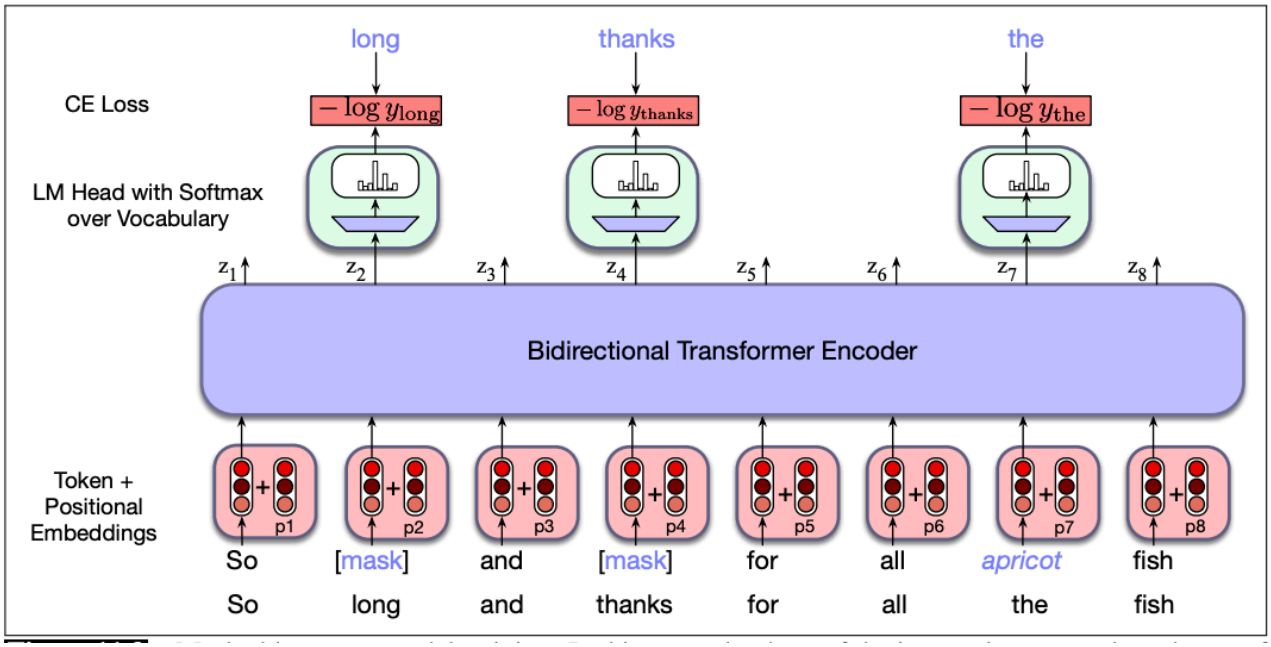
*"The masked language model randomly masks some of the tokens from the input, and the objective is to **predict the original vocabulary id of the masked word based only on its context**. Unlike **left-to-right language model pre-training**, the MLM objective allows the representation to fuse the left and the right context, which allows us to pre-train **a deep bidirectional Transformer."**

The Google AI researchers masked **15% of the words in each sequence at random.** The task? To predict these masked words. A caveat here — the masked words were not always replaced by the masked tokens [MASK] because the [MASK] token would never appear during fine-tuning.

So, the researchers used the below technique:

- 80% of the time the words were replaced with the masked token [MASK]
- 10% of the time the words were replaced with random words
- 10% of the time the words were left unchanged





Masked Language Model Training From [SLP3] (https://web.stanford.edu/~jurafsky/slp3/11.pdf)

- For a given vector of input tokens in a sentence or batch be *x*, let the set of tokens that aremasked be *M*, the version of that sentence with some tokens replaced by masks be xmask, and the sequence of output vectors be h.
- For a given input token xi, such as the word *long* above, the loss is the probability of the correct word long, given xmask (as summarized in the single output vector h
- The language modeling head takes the output vector h from the final transformer layer L for each masked token i, multiplies it by the unembedding layer E to produce the logits u, and then uses softmax to turn the logits into probabilities y over the vocabulary:

$$\mathbf{u}_i = \mathbf{h}_i^L \mathbf{E}^T$$
$$\mathbf{y}_i = \text{softmax}(\mathbf{u}_i)$$

$$L_{MLM}(x_i) = -\log P(x_i|\mathbf{h}_i^L)$$

- The gradients that form the basis for the weight updates are based on the average loss over the sampled learning items from a single training sequence (or batch of sequences).

$$L_{MLM} = -\frac{1}{|M|}\sum_{i \in M} \log P(x_i|\mathbf{h}_i^L)$$

- Note that only the tokens in M play a role in learning; the other words play no role in the loss function, so in that sense BERT and its descendents are inefficient; only 15% of the input samples in the training data are actually used for training weights.

B- Next Sentence Prediction

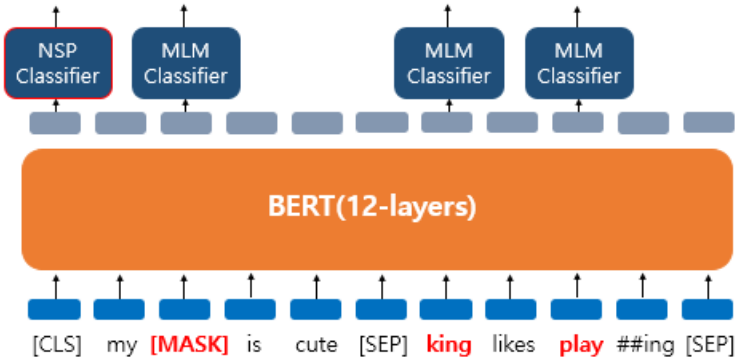
Generally, language models do not capture the relationship between consecutive sentences. BERT was pre-trained on this task as well.

For language model pre-training, BERT uses pairs of sentences as its training data. The selection of sentences for each pair is quite interesting. Let's try to understand it with the help of an example.

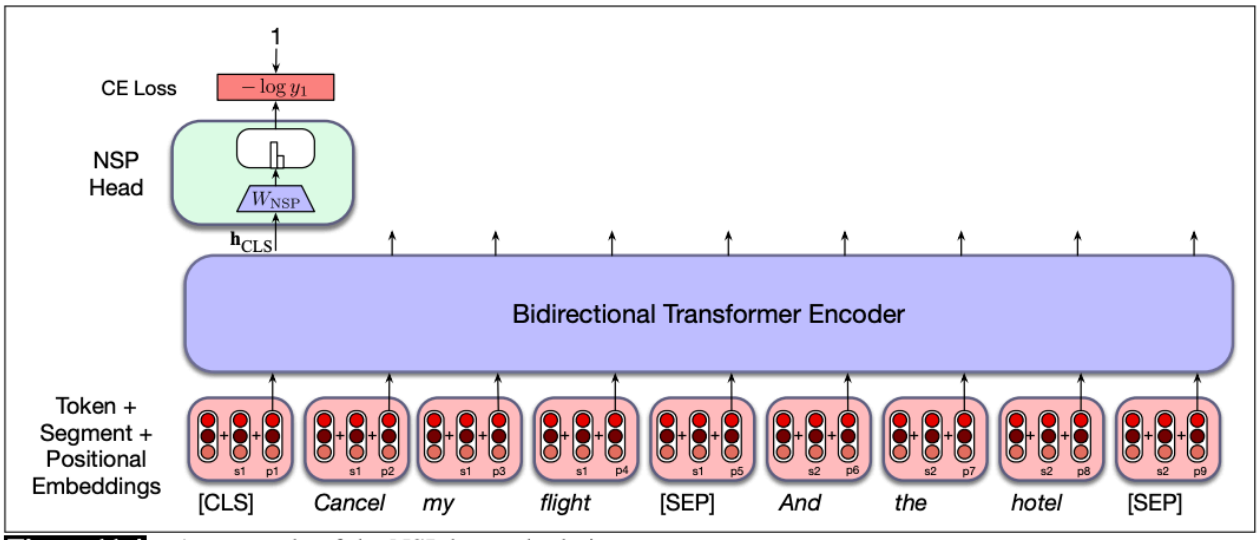
Imagine we have a text dataset of 100,000 sentences and we want to pre-train a BERT language model using this dataset. So, there will be 50,000 training examples or pairs of sentences as the training data.

- For 50% of the pairs, the second sentence would actually be the next sentence to the first sentence
- For the remaining 50% of the pairs, the second sentence would be a random sentence from the corpus
- The labels for the first case would be 'IsNext' and 'NotNext' for the second case

- 이어지는 문장의 경우
  Sentence A : The man went to the store.
  Sentence B : He bought a gallon of milk.
  Label = IsNextSentence
- 이어지는 문장이 아닌 경우 경우
  Sentence A : The man went to the store.
  Sentence B : dogs are so cute.
  Label = NotNextSentence



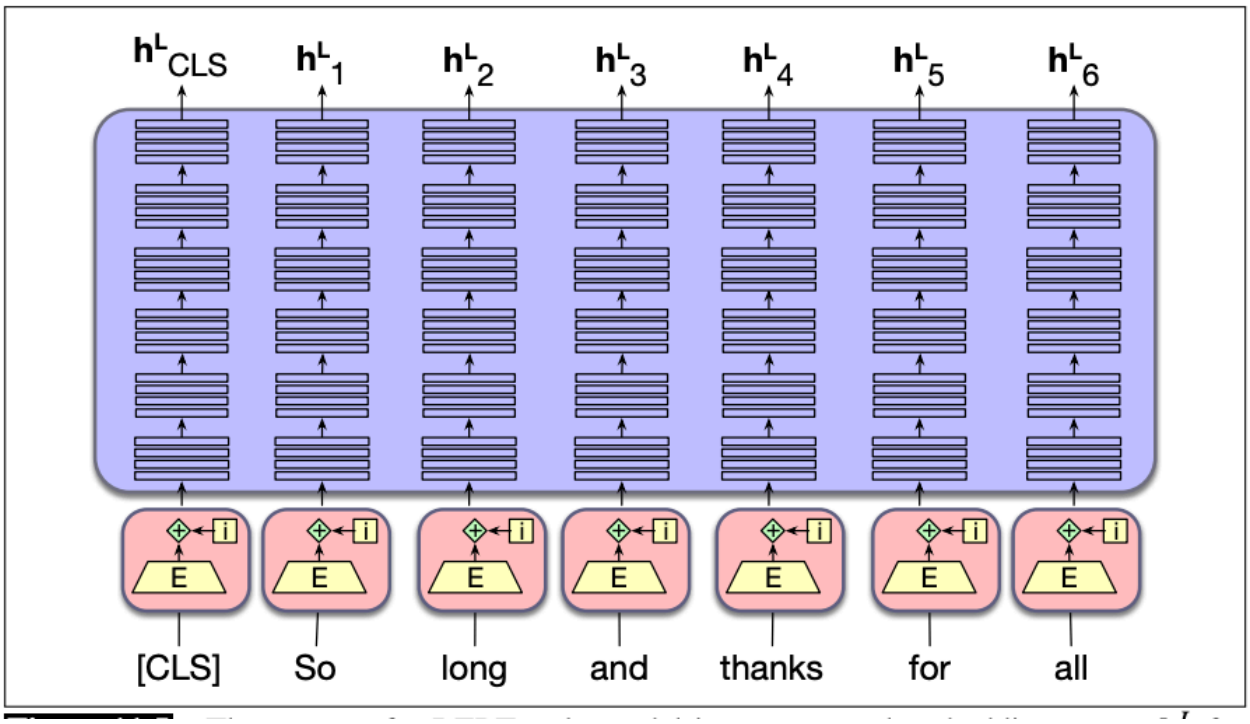
- [SEP]
  - next sentence prediction
  - QA
  - NLI(Natural Language Inference)



NSP Loss Calculation from SLP3

Contextual Embeddings

- Given a pretrained language model and a novel input sentence, we can think of the sequence of model outputs as constituting contextual embeddings for each token in **contextual embeddings** for each token in the input.
- These contextual embeddings are vectors representing some aspect of the meaning of a token in context, and can be used for any task requiring the meaning of tokens or words.

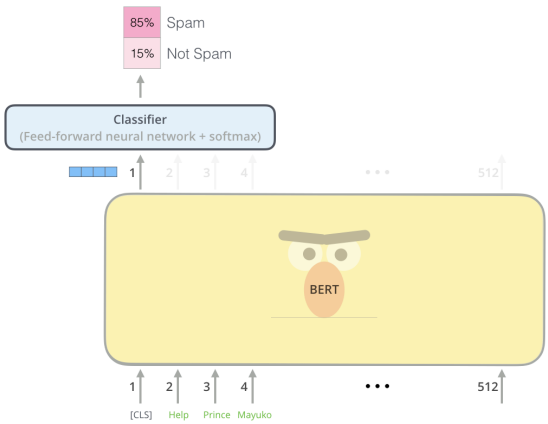


FineTuning for Sequence Classification

- The power of pretrained language models lies in their ability to extract generalizations from large amounts of text—generalizations that are useful for myriad downstream applications. There are two ways to make practical use of the generalizations to solve downstream tasks.
- The most common way is to use natural language to **prompt** the model, putting it in a state where it contextually generates what we want.
- An alternative way to use pretrained language models **finetuning** for downstream applications: a version of the finetuning paradigm
- In the kind of finetuning used for masked language models, we add applicationspecific circuitry (often called a special **head**) on top of pretrained models, taking their output as its input. The finetuning process consists of using labeled data about the application to train these additional application-specific parameters.
- Typically, this training will either freeze or make only **minimal adjustments** to the pretrained language model parameters.
- The most common kinds of finetuning : sequence classification, sentence-pair classification, and sequence labeling.

Sequence Class

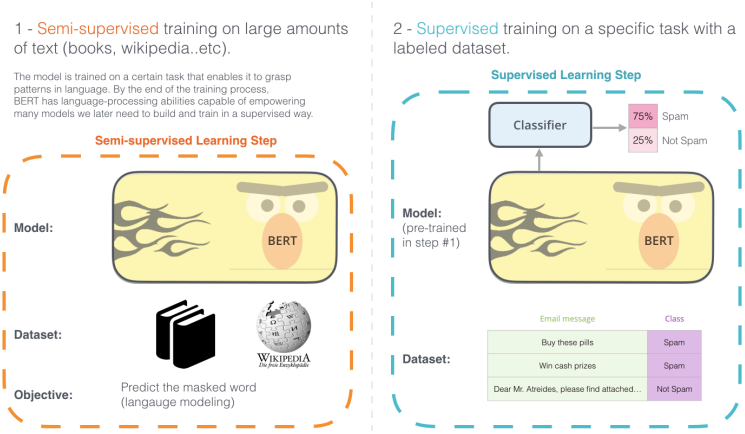
That vector can now be used as the input for a classifier of our choosing. The paper achieves great results by just using a single-layer neural network as the classifier.

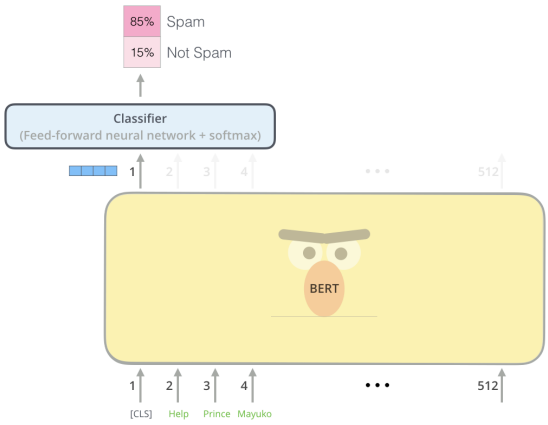


If you have more labels (for example if you're an email service that tags emails with "spam", "not spam", "social", and "promotion"), you just tweak the classifier network to have more output neurons that then pass through softmax.

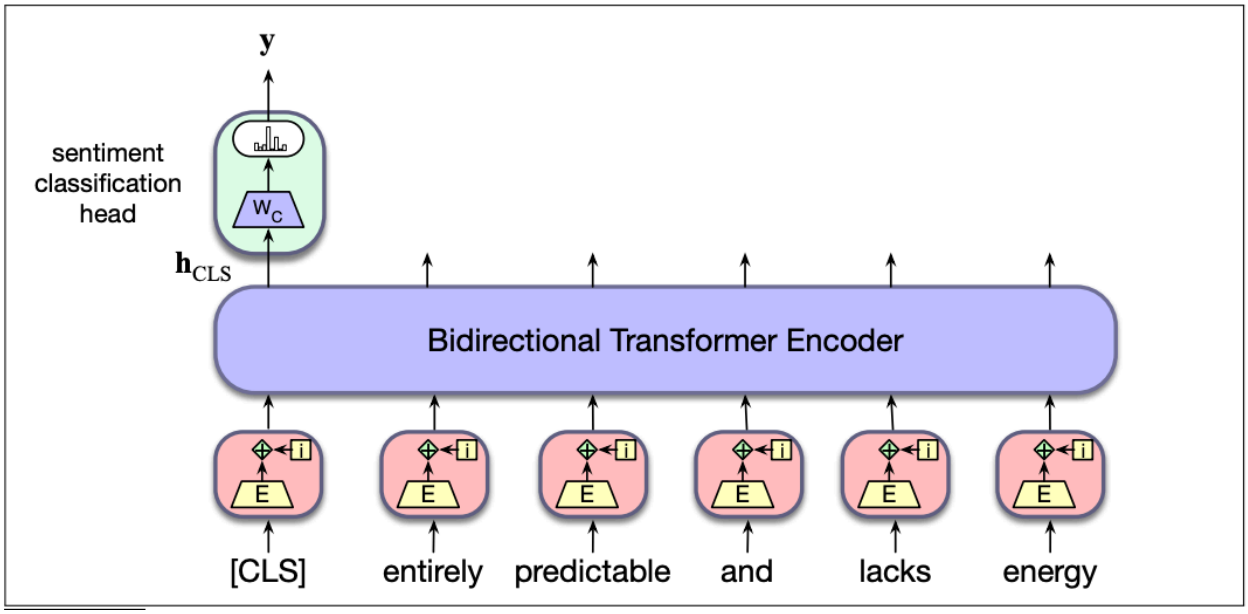
Sequence Classification

BERT is basically a trained Transformer Encoder stack.





If you have more labels (for example if you're an email service that tags emails with "spam", "not spam", "social", and "promotion"), you just tweak the classifier network to have more output neurons that then pass through softmax.



BERT Contextual Embedding

In [ ]:
```
%pip install transformers
```

In [1]:
```
from transformers import BertModel, BertTokenizer
import torch
```

```
2025-02-17 11:46:20.070933: I tensorflow/core/util/port.cc:113] oneDNN custom operati
ons are on. You may see slightly different numerical results due to floating-point ro
und-off errors from different computation orders. To turn them off, set the environme
nt variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-02-17 11:46:20.073094: I external/local_tsl/tsl/cuda/cudart_stub.cc:31] Could no
t find cuda drivers on your machine, GPU will not be used.
2025-02-17 11:46:20.103831: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.c
c:9261] Unable to register cuDNN factory: Attempting to register factory for plugin c
uDNN when one has already been registered
2025-02-17 11:46:20.103867: E external/local_xla/xla/stream_executor/cuda/cuda_fft.c
c:607] Unable to register cuFFT factory: Attempting to register factory for plugin cu
FFT when one has already been registered
2025-02-17 11:46:20.104869: E external/local_xla/xla/stream_executor/cuda/cuda_blas.c
c:1515] Unable to register cuBLAS factory: Attempting to register factory for plugin
cuBLAS when one has already been registered
2025-02-17 11:46:20.110036: I external/local_tsl/tsl/cuda/cudart_stub.cc:31] Could no
t find cuda drivers on your machine, GPU will not be used.
2025-02-17 11:46:20.110673: I tensorflow/core/platform/cpu_feature_guard.cc:182] This
TensorFlow binary is optimized to use available CPU instructions in performance-criti
cal operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operatio
ns, rebuild TensorFlow with the appropriate compiler flags.
2025-02-17 11:46:21.041457: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] T
F-TRT Warning: Could not find TensorRT
```

In [2]:
```
model = BertModel.from_pretrained('bert-base-uncased')
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initiali
zing BertModel: ['cls.predictions.transform.dense.bias', 'cls.predictions.transform.L
ayerNorm.bias', 'cls.predictions.transform.LayerNorm.weight', 'cls.seq_relationship.b
ias', 'cls.predictions.transform.dense.weight', 'cls.predictions.decoder.weight', 'cl
s.predictions.bias', 'cls.seq_relationship.weight']
- This IS expected if you are initializing BertModel from the checkpoint of a model t
rained on another task or with another architecture (e.g. initializing a BertForSeque
nceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a mod
el that you expect to be exactly identical (initializing a BertForSequenceClassificat
ion model from a BertForSequenceClassification model).
```

In [3]:
```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

In [4]:
```
## 입력 문장
sentence = 'I love Paris'
```

In [5]:
```
## 문장을 tokenization
token = tokenizer.tokenize(sentence)
print(token)
```

```
['i', 'love', 'paris']
```

In [6]:
```
#시작부분에 [CLS] 토큰을, 문장 끝에 [SEP] 토큰을 추가한다.
token = ['[CLS]'] + token + ['[SEP]']
print(token)
```

```
['[CLS]', 'i', 'love', 'paris', '[SEP]']
```

In [7]:
```
## 수동으로 간단히 [PAD]추가
token = token + ['[PAD]'] + ['[PAD]']
token
```

Out[7]:
```
['[CLS]', 'i', 'love', 'paris', '[SEP]', '[PAD]', '[PAD]']
```

In [8]:
```
#수동으로 attention mask
attention_mask = [1 if i!='[PAD]' else 0 for i in token]
attention_mask
```

Out[8]:
```
[1, 1, 1, 1, 1, 0, 0]
```

In [9]:
```
# token을 token id로 변환
token_ids = tokenizer.convert_tokens_to_ids(token)
token_ids
```

Out[9]:
```
[101, 1045, 2293, 3000, 102, 0, 0]
```

In [10]:
```
# token id를 텐서로 변환
token_ids = torch.tensor(token_ids).unsqueeze(0)
attention_mask = torch.tensor(attention_mask).unsqueeze(0)
```

In [19]:
```
#token_ids, attention_mask를 모델에 입력하고 임베딩을 획득
hidden_rep, cls_head = model(token_ids, attention_mask = attention_mask, return_dict
```

In [21]:
```
hidden_rep.shape
```

Out[21]:
```
torch.Size([1, 7, 768])
```

In [ ]:
```
hidden_rep[0,2,]
```

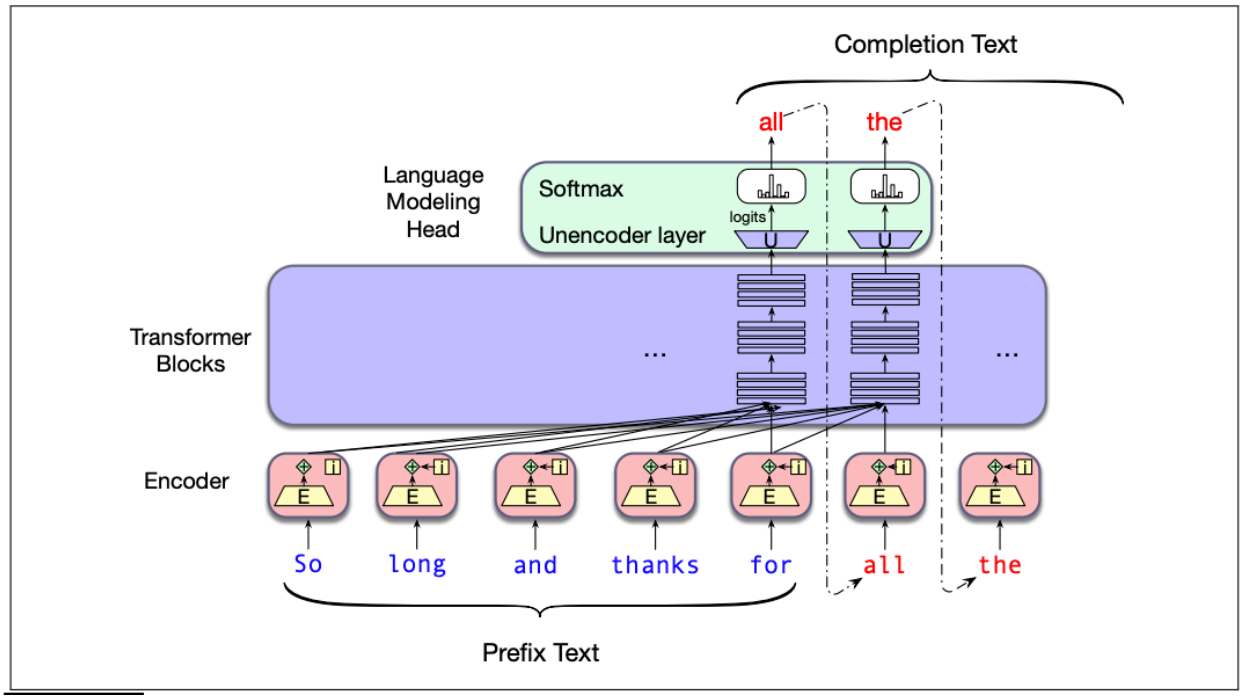
Autoregressive / Causal Language Model

- iteratively predict words left-to-right from earlier words
- We'll first introduce training, seeing how language models are self-trained by iteratively being taught to guess the next word in the text from the prior words

Conditional Generation

- Conditional conditional generation is the task of generating text conditioned on an input piece of text.
- That is, we give the LLM an input piece of text, generally called a **prompt**, and then have the LLM continue generating text token by token, conditioned on the prompt.

- The fact that transformers have such long contexts (many thousands of tokens) makes them very powerful for conditional generation, because they can look back so far into the prompting text.




```
Left-to-right(autoregressive) text completion with transformer-based large language models from [SLP3] (https://web.stanford.edu/~jurafsky/slp3/10.pdf)
```

why should we care about predicting upcoming words or tokens?

- The insight of large language modeling is that many practical NLP tasks can be cast as **word prediction**, and that a powerful-enough language model can solve them with a high degree of accuracy

1. Sentiment Analysis

- The sentiment of the sentence ‘‘I like Jackie Chan" is:

P(positive|The sentiment of the sentence ‘‘I like Jackie Chan" is:) P(negative|The sentiment of the sentence ‘‘I like Jackie Chan" is:)

1. Question-Answering

Q: Who wrote the book ‘‘The Origin of Species"? A:

P(w|Q: Who wrote the book ‘‘The Origin of Species"? A:) P(w|Q: Who wrote the book ‘‘The Origin of Species"? A: Charles)

1. Text Summarization

Original Article

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says.

But not if you live in New England or surrounding states. "We will not ship snow to any states in the northeast!" says Waring's website, ShipSnowYo.com. "We're in the business of expunging snow!"

His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

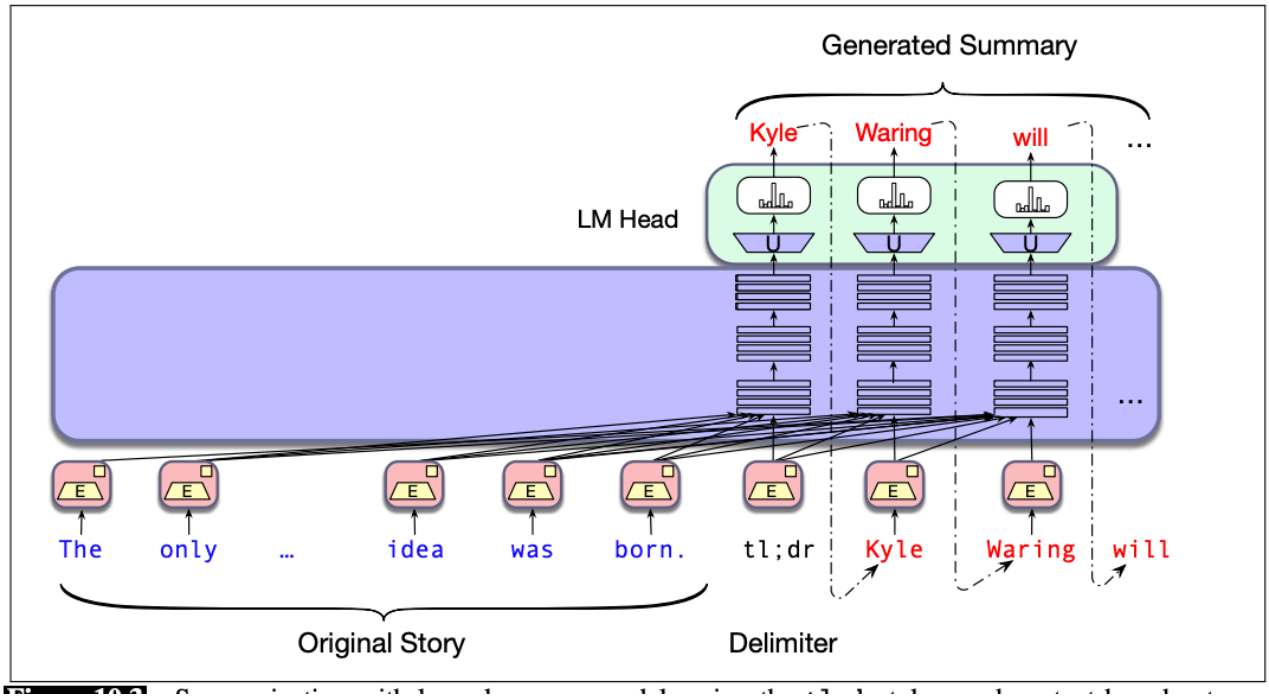
According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. [...]

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

Excerpt from a sample article and its summary from the CNN/Daily Mail summarizaiton Corpus from SLP3

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t})$$



Summarization with large language models using tl;dr token and context-based autoregressive generation by SLP3

Sampling for LLM Generation

- The core of the generation process for large language models is the task of **choosing the single word to generate next based on the context and based on the probabilities that the model assigns to possible words.**
- This task of choosing a word to generate based on the model's probabilities is called **decoding**.

- Decoding from a language model in a left-to-right manner (or right-to-left for languages like Arabic in which we read from right to left), and thus repeatedly choosing the next word conditioned on our previous choices is called **autoregressive generation or causal LM generation**
- The most common method for decoding in large language models is **sampling**.
- Sampling from a model's distribution over words means **to choose random words according to their probability assigned by the model.**

$$
\begin{aligned}
&\mathrm{i} \leftarrow 1 \\
&w_i \sim \mathrm{p}(w) \\
&\textbf{while } w_i \mathrel{!}= \mathrm{EOS} \\
&\quad \mathrm{i} \leftarrow \mathrm{i} + 1 \\
&\quad w_i \sim \mathrm{p}(w_i \mid w_{<i})
\end{aligned}
$$

Random Sampling

- The problem is that even though random sampling is mostly going to generate sensible, high-probable words, **there are many odd, low probability words in the tail of the distribution, and even though each one is low probability, if you add up all the rare words, they constitute a large enough portion of the distribution that they get chosen often enough to result in generating weird sentences.**
- Trading off **two important factors in generation: quality and diversity**
- Methods that emphasize the most probable words tend to produce generations that are rated by people as more accurate, more coherent, and more factual, but also more **boring and more repetitive.**
- Methods that **give a bit more weight to the middle-probability words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality**

Top-k sampling

- Top-k sampling is a **simple generalization of greedy decoding.**
- Instead of choosing the single most probable word to generate, we first truncate the distribution to the top k most likely words, **renormalize to produce a legitimate probability distribution, and then randomly sample from within these k words according to their renormalized probabilities**
- When k = 1, top-k sampling is identical to greedy decoding.
- Setting k to a larger number than 1 leads us to sometimes select a word which is **not necessarily the most probable, but is still probable enough**, and whose choice results in generating **more diverse but still high-enough-quality text**

Nucleus or top-p sampling

- One problem with top-k sampling is that k is fixed, but **the shape of the probability distribution over words differs in different contexts.**
- If we set k = 10, sometimes the top 10 words will be very likely and include most of the probability mass, but other times **the probability distribution will be flatter and the top 10**

words will only include a small part of the probability mass

- **Top-p sampling or nucleus sampling** is to **keep not the top k words, but the top p percent of the probability mass.**
- The goal is the same; to truncate the distribution to remove the very unlikely words.
- But **by measuring probability rather than the number of words**, the hope is that the measure will be more robust in very different contexts, dynamically increasing and decreasing the pool of word candidates.
- Given a distribution P(wt|w<t), the top-p vocabulary V(p) is the smallest set of words such that

$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) \geq p.$$

Temperature sampling

- In temperature sampling, we don't truncate the distribution, but instead reshape temperature sampling it.
- The intuition for temperature sampling comes from thermodynamics, **where a system at a high temperature is very flexible and can explore many possible states, while a system at a lower temperature is likely to explore a subset of lower energy (better) states.**
- In **low-temperature sampling, we smoothly increase the probability of the most probable words and decrease the probability of the rare words.**
- We implement this intuition **by simply dividing the logit by a temperature parameter τ before we normalize it by passing it through the softmax.**
- In low-temperature sampling, $\tau \in (0,1)$.
- Thus instead of computing the probability distribution over the vocabulary directly from the logit, we instead first divide the logits by τ, computing the probability vector y as

$$y = \text{softmax}(u/\tau)$$

- **When τ is close to 1 the distribution doesn't change much.

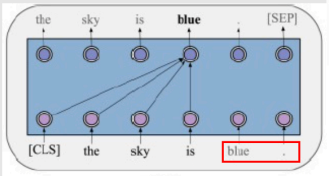
- But the lower τ is, the larger the scores being passed to the softmax (dividing by a smaller fraction $\tau \leq 1$ results in making each score larger).
- Recall that one of the useful properties of a softmax is that it tends to push high values toward 1 and low values toward 0.
- Thus **when larger numbers are passed to a softmax the result is a distribution with increased probabilities of the most high-probability words and decreased probabilities of the low probability words, making the distribution more greedy.**
- As τ approaches 0 the probability of the most likely word approaches 1

Roundup: BERT VS GPT

GPT

Generative Pre-trained Transformer

- Given large-scale corpora without labels, GPT optimizes a standard **autoregressive language modeling**, that is, maximizing the conditional probabilities of all the words by taking their previous words as contexts.
- The adaptation procedure of GPT to specific tasks is fine-tuning, by using the pre-trained parameters of GPT as a start point of downstream tasks.

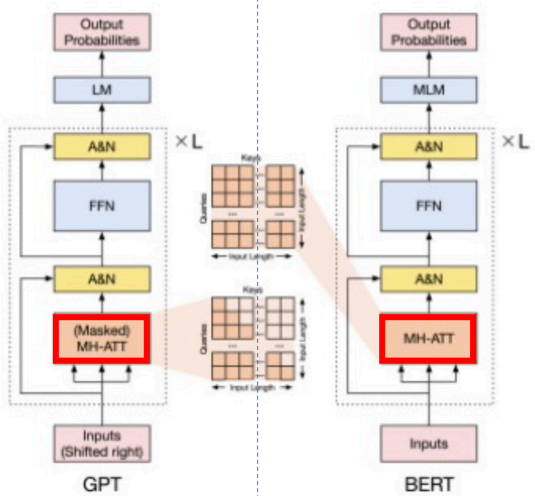


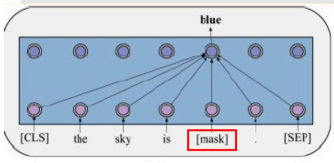
GPT

BERT

Bidirectional Encoder Representations from **Transformers**

- In the pre-training phase, BERT applies **autoencoding language modeling** rather than autoregressive language modeling used in GPT. More specifically, inspired by cloze (Taylor, 1953), the objective masked language modeling (MLM) is designed.
- By modifying inputs and outputs with the data of downstream tasks, BERT could be fine-tuned for any NLP tasks.





BERT

In [ ]: