# Course Materials for GEN-AI

*Northeastern University*

These materials have been prepared and sourced for the course **GEN-AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

**Instructor: Ramin Mohammadi**
r.mohammadi@northeastern.edu

*Thank you for your understanding and collaboration.*

---

Content sourced from multiple references - see Reference section for details

# Pre-Training

## Introduction to Pre-Training

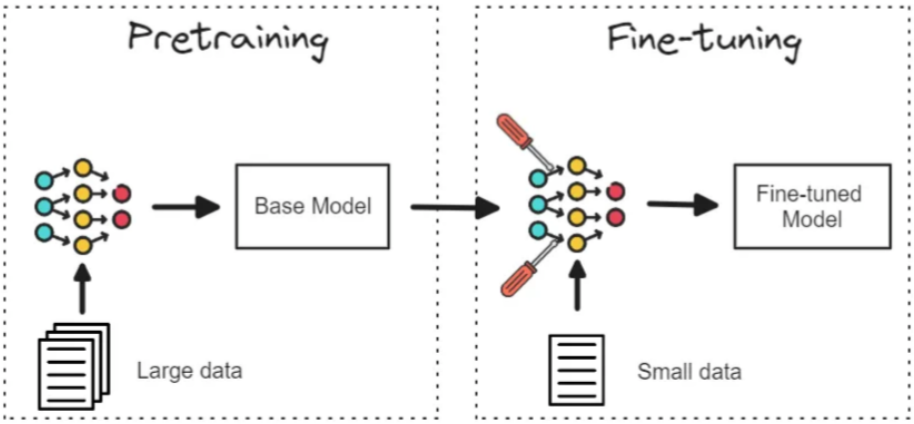



Figure 1: Pre-training vs Fine-tuning

Pre-training has become a fundamental step in modern natural language processing models. The idea is to first train a model on a large, general dataset before fine-tuning it on a smaller, task-specific dataset. During pre-training, the model learns a broad understanding of language structure, syntax, and semantics, typically through unsupervised or self-supervised tasks such as predicting missing words (masked language modeling) or the next sentence. This initial training allows the model to acquire knowledge that can later be adapted to a wide variety of downstream tasks, such as text classification, translation, and question answering. By leveraging pre-trained representations, models achieve better performance and require less task-specific data, making pre-training an essential aspect of deep learning in NLP.

### Transformers for natural language processing

In the previous section we described the transformer. let's describes how it is used in natural language processing (NLP) tasks. A typical NLP pipeline starts with a **tokenizer** that splits the text into words or word fragments. Then each of these tokens is mapped to a learned embedding. These embeddings are passed through a series of transformers.

#### Tokenization

A text processing pipeline begins with a tokenizer, which splits the text into smaller constituent units (tokens) from a vocabulary of possible tokens. In the discussion above, we have implied that these tokens represent words, but there are several difficulties.

- Inevitably, some words (e.g., names) will not be in the vocabulary.
- It's unclear how to handle punctuation, but this is important. If a sentence ends in a question mark, we must encode this information.
- The vocabulary would need different tokens for versions of the same word with different suffixes (e.g., walk, walks, walked, walking), and there is no way to clarify that these variations are related.

One approach would be to use letters and punctuation marks as the vocabulary, but this would mean splitting text into very small parts and requiring the subsequent network to re-learn the relations between them.
In practice, a compromise between letters and full words is used, and the final vocabulary includes both common words and word fragments from which larger and less frequent words can be composed. The vocabulary is computed using a sub-word tokenizer, such as byte pair encoding (Figure 2), that greedily merges commonly occurring sub-strings based on their frequency.

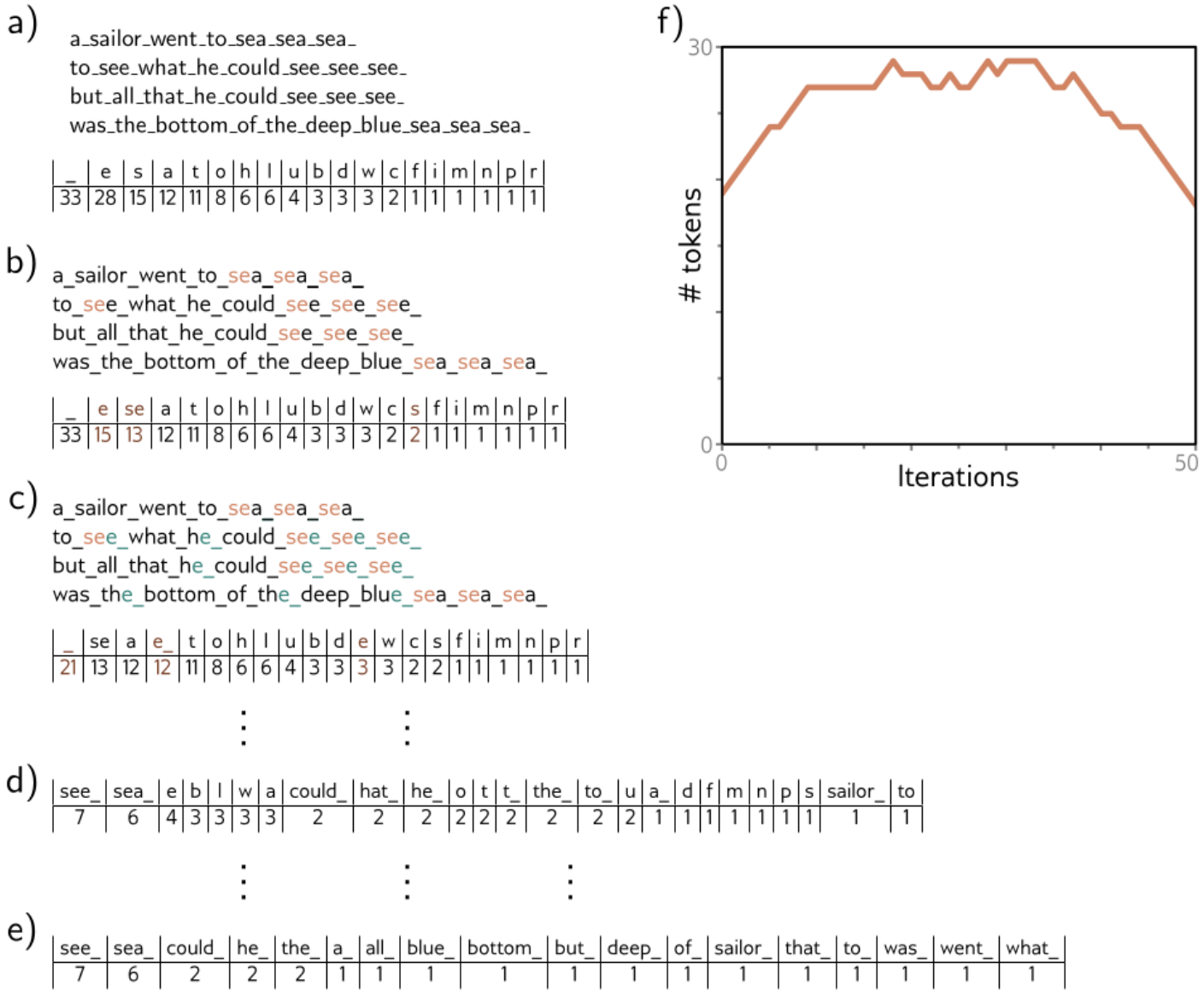



Figure 2: Sub-word tokenization. (a) A passage of text from a nursery rhyme. The tokens are initially just the characters and whitespace (represented by an underscore), and their frequencies are displayed in the table. (b) At each iteration, the sub-word tokenizer looks for the most commonly occurring adjacent pair of characters (in this case, `se`) and merges them. This creates a new token and decreases the counts for the original tokens `s` and `e`. (c) At the second iteration, the algorithm merges `e` and the whitespace character _. Note that the last character of the first token to be merged cannot be whitespace, which prevents merging across words. (d) After 22 iterations, the tokens consist of a mix of letters, word fragments, and commonly occurring words. (e) If we continue this process indefinitely, the tokens eventually represent full words. (f) Over time, the number of tokens increases as we add word fragments to the letters, then decreases again as we merge these fragments. In a real situation, there would be a very large number of words, and the algorithm would terminate when the vocabulary size (number of tokens) reached a predetermined value. Punctuation and capital letters would also be treated as separate input characters.

**Embeddings**

Each token in the vocabulary $V$ is mapped to a unique word embedding, and the embeddings for the entire vocabulary are stored in a matrix $W_e \in \mathbb{R}^{D \times |V|}$. To accomplish this, the $N$ input tokens are first encoded in the matrix $T \in \mathbb{R}^{|V| \times N}$, where the $n$th column corresponds to the $n$th token and is a $|V| \times 1$ one-hot vector (i.e., a vector where every entry is zero except for the entry corresponding to the token, which is set to one). The input embeddings are then computed as $X = W_e T$, and $W_e$ is learned like any other network parameter (see Figure 3).

A typical embedding size $D$ is 1024, and a typical total vocabulary size $|V|$ is 30,000. This means that, even before the main network, there are many parameters in $W_e$ to learn.

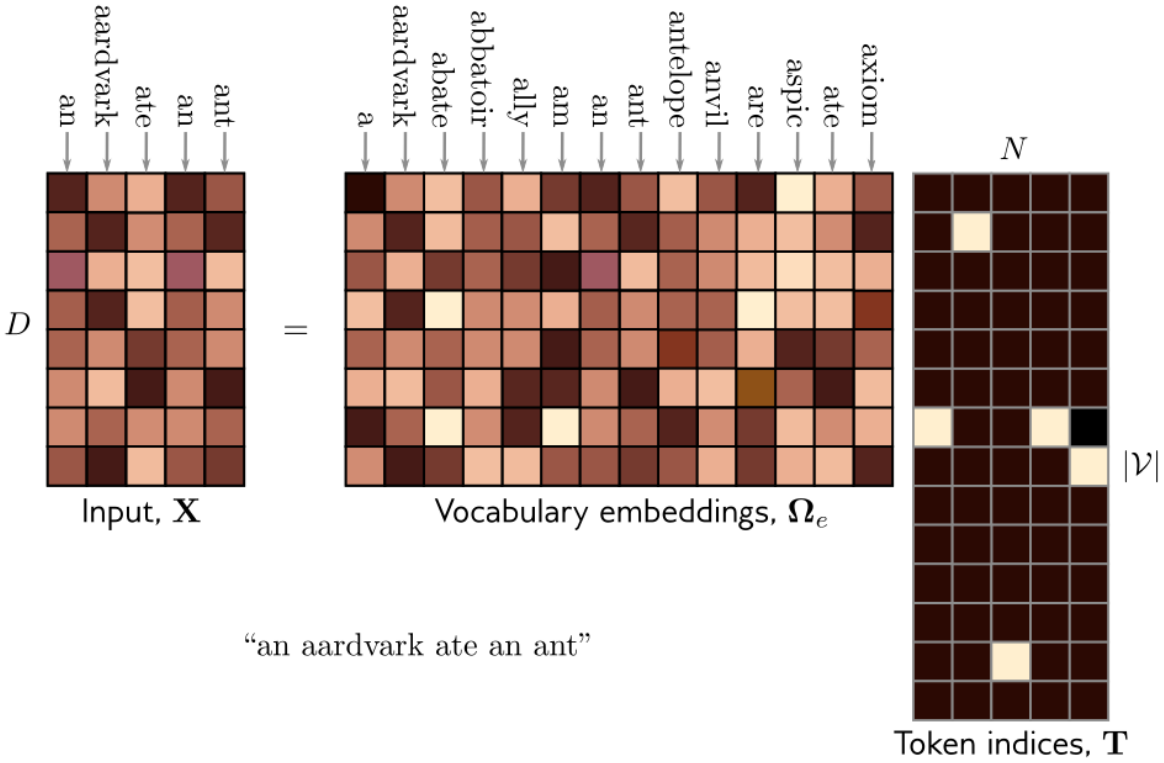



Figure 3: The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains $N$ embeddings of length $D$ and is created by multiplying a matrix $W_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the word or sub-word indices. The vocabulary matrix $W_e$ is considered a parameter of the model and is learned along with the other parameters. Note that the two embeddings for the word *an* in $\mathbf{X}$ are the same.

## pretrained word embeddings

As we have seen before in Embedding chapter, we can use pre-trained embedding in our task/model:

- Start with pretrained word embeddings (no context!)
- Learn how to incorporate context in an LSTM or Transformer while training on the task.

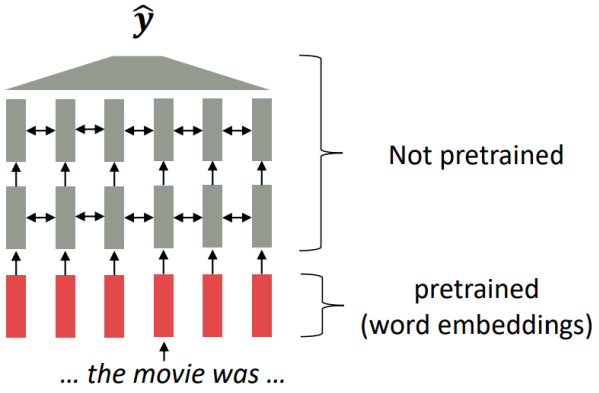



Figure 4: Pre-trained Embedding

**Some issues to think about:**

- The training data we have for our **downstream task** (like question answering) must be sufficient to teach all contextual aspects of language.
- Most of the parameters in our network are randomly initialized!

[Recall, *movie* gets the same word embedding, no matter what sentence it shows up in]

## Pretraining whole models

In modern NLP, rather than using only pre-trained embeddings, we pre-train the entire model. This approach enables the model to learn rich language representations that can be applied to various applications without requiring a large task-specific training dataset.

- All (or almost all) parameters in NLP networks are initialized via **pretraining**.
- Pretraining methods hide parts of the input from the model, and train the model to reconstruct those parts.

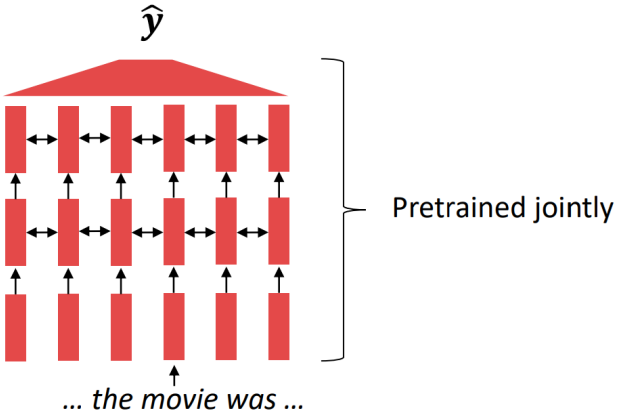



Figure 5: pretraining whole models

This has been exceptionally effective at building strong:

- **representations of language**
- **parameter initializations** for strong NLP models.
- **Probability distributions** over language that we can sample from.

## What Can We Learn from Reconstructing the Input?

By masking words in a sentence and training neural networks (NNs) to predict these missing words, we encourage the model to learn comprehensive information about each word—syntactically, semantically, and more. This approach allows the model the flexibility to learn all the essential dimensions needed to fully represent a word's meaning and context.

## What the Model Learns from Reconstructing the Input

By masking words in a sentence and training a model to predict these missing words, we create a supervised learning setup where the missing word acts as the label. Through gradient updates, the model learns rich associations, such as word co-occurrences, syntax, semantics, and even latent sentiments. Below are examples of how masked word prediction enables the model to learn various aspects of language.

- **Syntax and Article Choice**:

  *I put ___ fork down on the table.*

  Here, the model learns the syntactical structure and article choice, such as selecting "the" or "a" to fit the sentence context.

- **Coreference Resolution**:

  *The woman walked across the street, checking for traffic over ___ shoulder.*

  The model learns to identify that the missing word "her" refers back to "the woman," understanding coreference relationships in sentences.

- **Semantic Category Learning**:

  *I went to the ocean to see the fish, turtles, seals, and _____.*

  The model infers that the missing word should belong to the same semantic category, like "dolphins" or "whales," learning associations within a conceptual family.

- **Sentiment Understanding**:

  *Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was ___.*

  The model learns sentiment from context, understanding that a likely completion could be "terrible" or "disappointing," indicating a negative sentiment.

- **Spatial and Location Understanding**:

  *Iroh went into the kitchen to make some tea. Standing next to Iroh, Zuko pondered his destiny. Zuko left the _____.*

  The model learns spatial awareness, recognizing that the blank likely corresponds to a location related to the previous sentences, such as "kitchen" or "room."

- **Geographical Knowledge**:

  *Northeastern University is located in _____, Massachusetts.*

  Here, the model learns geographic information, predicting that the blank should be filled with "Boston."

- **Sequence Prediction and Pattern Recognition**:

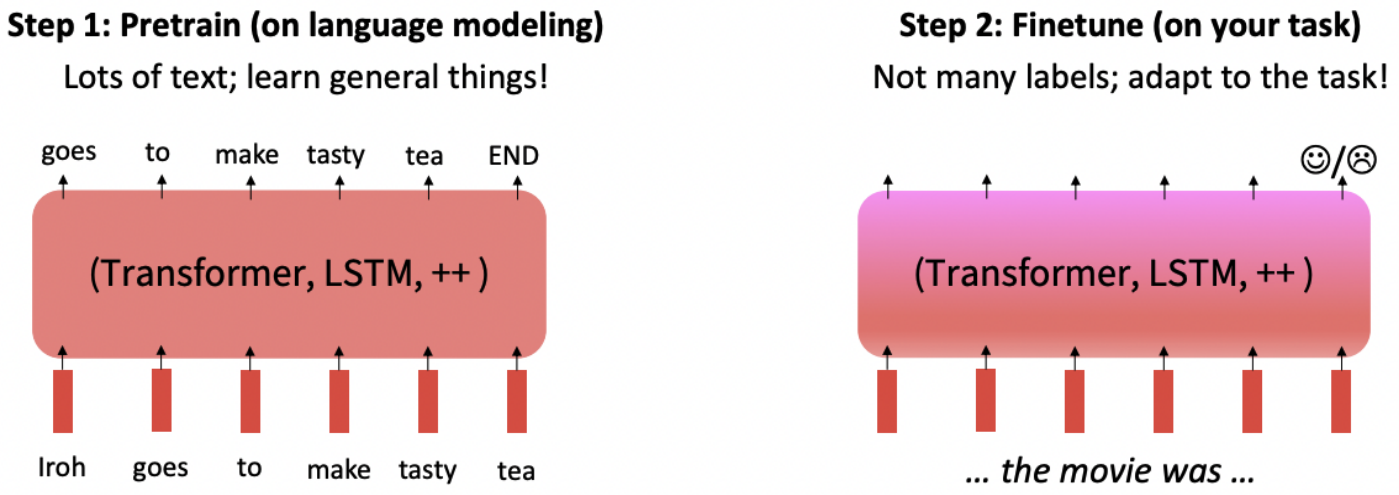



Figure 7: Fine-tuning

*I was thinking about the sequence that goes*
*1, 1, 2, 3, 5, 8, 13, 21, ___*

In this example, the model identifies the Fibonacci sequence, learning to predict the next number, 34, based on the sequence pattern.

Through such examples, the model learns a diverse range of linguistic and contextual patterns, from syntax and coreference to semantic categories, sentiment, spatial reasoning, geographical knowledge, and even numerical patterns. This rich representation of language enables the model to perform well on various downstream tasks.

## Pretraining through language modeling

Recall the **language modeling** task:

- Model $p_\theta(w_t|w_{1:t-1})$, the probability distribution over words given their past contexts.
- There's lots of data for this! (In English.)

**Pretraining through language modeling:**

- Train a neural network to perform language modeling on a large amount of text.
- Save the network parameters.

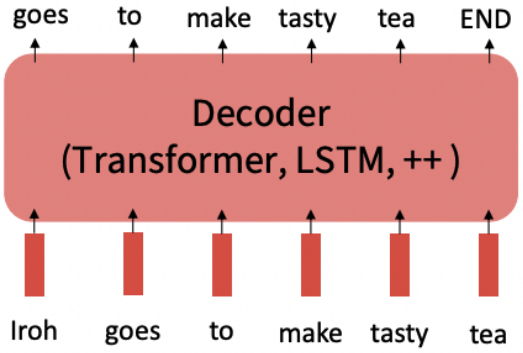



Figure 6: Pre-training of language model

## The Pretraining / Finetuning Paradigm

Pretraining enhances NLP applications by providing a strong parameter initialization for models. With a pre-trained model, instead of learning parameters from scratch, we can use these pre-trained parameters as the starting point for our model (a process known as fine-tuning). This approach significantly reduces the need for large task-specific datasets and enables the model to adapt effectively to new tasks with minimal additional data.

## Stochastic gradient descent and pretrain/finetune

Why should pretraining and finetuning help, from a "training neural nets" perspective?

- Pretraining provides parameters $\hat{\theta}$ by approximating $\min_\theta \mathcal{L}_{\text{pretrain}}(\theta)$.
  - (The pretraining loss.)
- Then, finetuning approximates $\min_\theta \mathcal{L}_{\text{finetune}}(\theta)$, starting at $\hat{\theta}$.
  - (The finetuning loss.)
- The pretraining may matter because stochastic gradient descent sticks (relatively) close to $\hat{\theta}$ during finetuning.
  - So, maybe the finetuning local minima near $\hat{\theta}$ tend to generalize well!
  - And/or, maybe the gradients of finetuning loss near $\hat{\theta}$ propagate nicely!

## Pretraining for Three Types of Architectures

The neural architecture influences the type of pretraining, and natural use cases.

### Encoders

- Gets bidirectional context – can condition on future!
- How do we train them to build strong representations?

### Encoder-Decoders

- Combines advantages of both encoders and decoders.
- What's the best way to pretrain them?

### Decoders

- Language models! What we've seen so far.
- Great for generation; can't condition on future words.

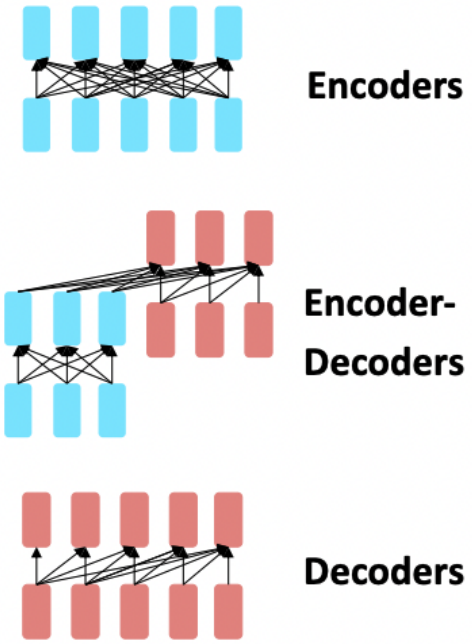



Figure 8: Illustration of different architectures and their pretraining use cases.

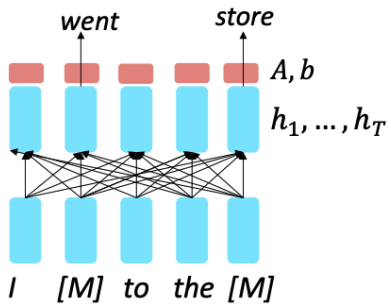
## Pretraining encoders: what pretraining objective to use?

So far, we've looked at language model pretraining. But **encoders get bidirectional context**, so we can't do language modeling!

**Idea**: replace some fraction of words in the input with a special [MASK] token; predict these words.

$$h_1, \ldots, h_T = \text{Encoder}(w_1, \ldots, w_T)$$
$$y_i \sim Ah_i + b$$

Only add loss terms from words that are "masked out." If $\tilde{x}$ is the masked version of sequence $x$, we're learning $p_\theta(x|\tilde{x})$. Called **Masked LM**.




## BERT: Bidirectional Encoder Representations from Transformers

BERT is an encoder model that uses a vocabulary of 30,000 tokens. Input tokens are converted to 1024-dimensional word embeddings and passed through 24 transformers. Each transformer contains a self-attention mechanism with 16 heads. The queries, keys, and values for each head are of dimension 64 (i.e., the matrices $W_{v,h}$, $W_{q,h}$, and $W_{k,h}$ are $1024 \times 64$). The dimension of the single hidden layer in the fully connected network within each transformer is 4096. The total number of parameters is approximately 340 million. When BERT was introduced, this was considered large, but it is now much smaller than state-of-the-art models.
Encoder models like BERT exploit transfer learning. During pre-training, the parameters of the transformer architecture are learned using self-supervision from a large corpus of text. The goal is for the model to acquire general information about the statistics of language. In the fine-tuning stage, the resulting network is adapted to solve a particular task using a smaller body of supervised training data.

### BERT Origin:

**Devlin et al., 2018** proposed the "Masked LM" objective and **released the weights of a pretrained Transformer**, a model they labeled BERT.

- Replacing the input word with `[MASK]` 80% of the time.
- Replacing the input word with a random token 10% of the time.
- Leaving the input word unchanged 10% of the time (but still predicting it).

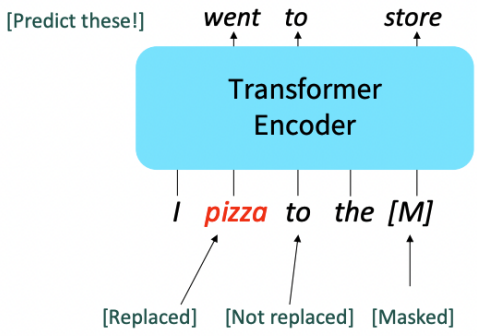



Figure 9: Illustration of different architectures and their pretraining use cases.

**Purpose:** This strategy prevents the model from becoming over-reliant on masked words alone and encourages it to build robust representations for all words. Since there are no masked tokens during fine-tuning, training on various scenarios (masked, replaced, and unchanged tokens) enables the model to better understand context and generalize across tasks.
In the original BERT paper, the pretraining input to BERT was two separate contiguous chunks of text (Segment A and B) and additionally BERT was trained to predict whether one chunk / segment follows the other or is randomly sampled.

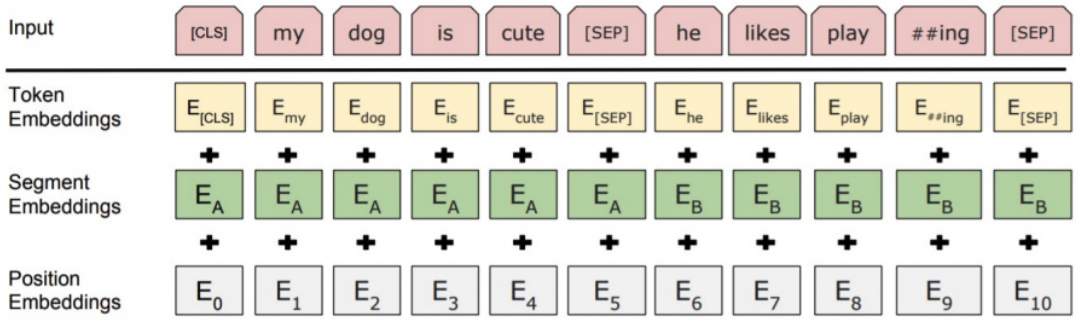



Figure 10: Pretraining input structure for BERT with token embeddings, segment embeddings, and position embeddings

## BERT Pre-training

In the pre-training stage, the network is trained using self-supervision. This allows the use of enormous amounts of data without the need for manual labels. For BERT, the self-supervision task consists of predicting missing words from sentences from a large internet corpus. During training, the maximum input length is 512 tokens, and the batch size is 256. The system is trained for a million steps, corresponding to roughly 50 epochs of the 3.3-billion word corpus.
Predicting missing words forces the transformer network to understand some syntax. For example, it might learn that the adjective `red` is often found before nouns like `house` or `car` but never before a verb like `shout`. It also allows the model to learn superficial `common sense` about the world. For example, after training, the model will assign a higher probability to the missing word `train` in the sentence `The <mask> pulled into the station` than it would to the word `peanut`. However, the degree of "understanding" this type of model can ever have is limited.

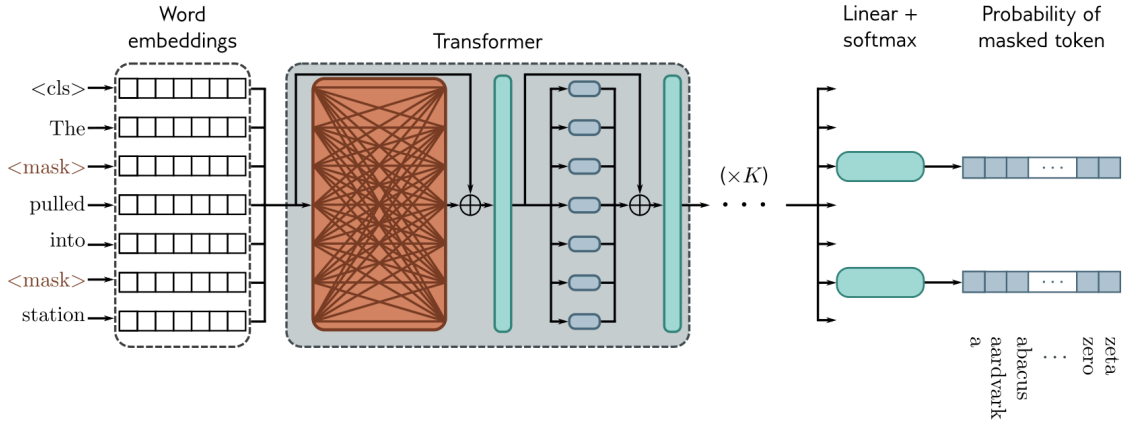



Figure 11: Pre-training for BERT-like Encoder. The input tokens, along with a special `<cls>` token denoting the start of the sequence, are converted to word embeddings. These embeddings are represented as rows rather than columns, so the box labeled "word embeddings" is $\mathbf{X}^T$. These embeddings are then passed through a series of transformers, where orange connections indicate that every token attends to every other token within these layers, producing a set of output embeddings. A small fraction of the input tokens is randomly replaced with a generic `<mask>` token. During pre-training, the objective is to predict the missing word from its corresponding output embedding. To achieve this, the output embeddings are passed through a softmax function, and the multiclass classification loss is used. This task leverages both left and right contexts to predict the missing word but has the drawback of being data-inefficient; for example, seven tokens need to be processed to contribute only two terms to the loss function.

## Details about BERT

- **Two models were released:**

  - BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
  - BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.
- **Trained on:**
  - BooksCorpus (800 million words)
  - English Wikipedia (2,500 million words)
- **Pretraining is expensive and impractical on a single GPU.**
  - BERT was pretrained with 64 TPU chips for a total of 4 days.
  - (TPUs are special tensor operation acceleration hardware)
- **Finetuning is practical and common on a single GPU.**
  - "Pretrain once, finetune many times."

## Fine-tuning

In the fine-tuning stage, the model parameters are adjusted to specialize the network for a particular task. An extra layer is appended onto the transformer network to convert the output vectors to the desired output format. Examples include:

- **Text classification:** In BERT, a special token known as the classification or `<cls>` token is placed at the start of each string during pre-training. For text classification tasks like *sentiment analysis* (in which the passage is labeled as having a positive or negative emotional tone), the vector associated with the `<cls>` token is mapped to a single number and passed through a logistic sigmoid (see Figure 13a). This contributes to a standard binary cross-entropy loss.
- **Word classification:** The goal of *named entity recognition* is to classify each word as an entity type (e.g., person, place, organization, or no-entity). To this end, each input embedding $x_n$ is mapped to an $E \times 1$ vector where the $E$ entries correspond to the $E$ entity types. This is passed through a softmax function to create probabilities for each class, which contribute to a multiclass cross-entropy loss (see Figure 13b).
- **Text span prediction:** In the SQuAD 1.1 question answering task, the question and a passage from Wikipedia containing the answer are concatenated and tokenized. BERT is then used to predict the text span in the passage that contains the answer. Each token maps to two numbers indicating how likely it is that the text span begins and ends at this location. The resulting two sets of numbers are put through two softmax functions. The likelihood of any text span being the answer can be derived by combining the probability of starting and ending at the appropriate places.

BERT was massively popular and hugely versatile; finetuning BERT led to new state-of-the-art results on a broad range of tasks.

| System | MNLI-(m/mm) 392k | QQP 363k | QNLI 108k | SST-2 67k | CoLA 8.5k | STS-B 5.7k | MRPC 3.5k | RTE 2.5k | **Average** - |
|---|---|---|---|---|---|---|---|---|---|
| Pre-OpenAI SOTA | 80.6/80.1 | 66.1 | 82.3 | 93.2 | 35.0 | 81.0 | 86.0 | 61.7 | 74.0 |
| BiLSTM+ELMo+Attn | 76.4/76.1 | 64.8 | 79.8 | 90.4 | 36.0 | 73.3 | 84.9 | 56.8 | 71.0 |
| OpenAI GPT | 82.1/81.4 | 70.3 | 87.4 | 91.3 | 45.4 | 80.0 | 82.3 | 56.0 | 75.1 |
| $\text{BERT}_{\text{BASE}}$ | 84.6/83.4 | 71.2 | 90.5 | 93.5 | 52.1 | 85.8 | 88.9 | 66.4 | 79.6 |
| $\text{BERT}_{\text{LARGE}}$ | **86.7/85.9** | **72.1** | **92.7** | **94.9** | **60.5** | **86.5** | **89.3** | **70.1** | **82.1** |

Figure 12: Bert applications

- **QQP**: Quora Question Pairs (detect paraphrase questions)
- **QNLI**: natural language inference over question answering data
- **SST-2**: sentiment analysis
- **CoLA**: corpus of linguistic acceptability (detect whether sentences are grammatical.)

- **STS-B**: semantic textual similarity
- **MRPC**: Microsoft paraphrase corpus
- **RTE**: a small natural language inference corpus

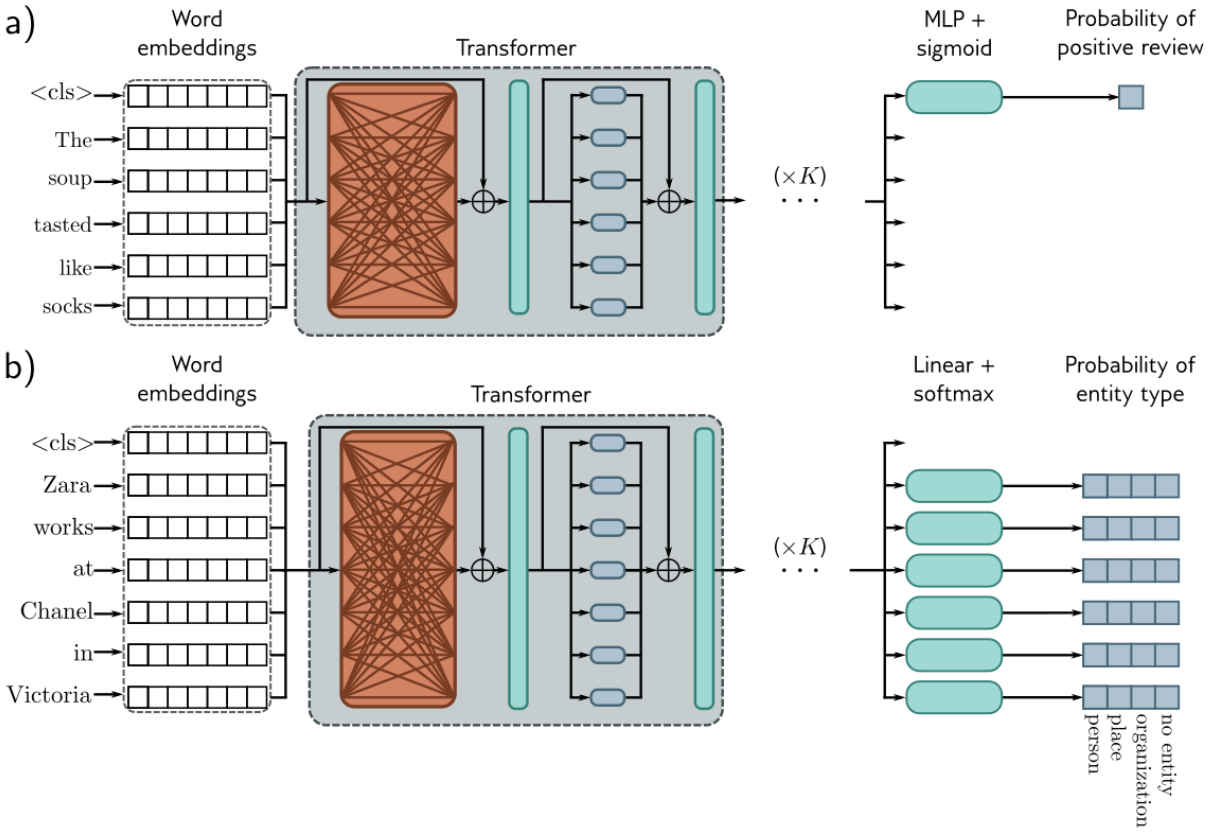



Figure 13: After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required. a) Example text classification task. In this sentiment classification task, the ¡cls¿ token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

## Full Finetuning vs. Parameter-Efficient Finetuning

Finetuning every parameter in a pretrained model works well, but is memory-intensive. But lightweight finetuning methods adapt pretrained models in a constrained way. Leads to less overfitting and/or more efficient finetuning and inference.

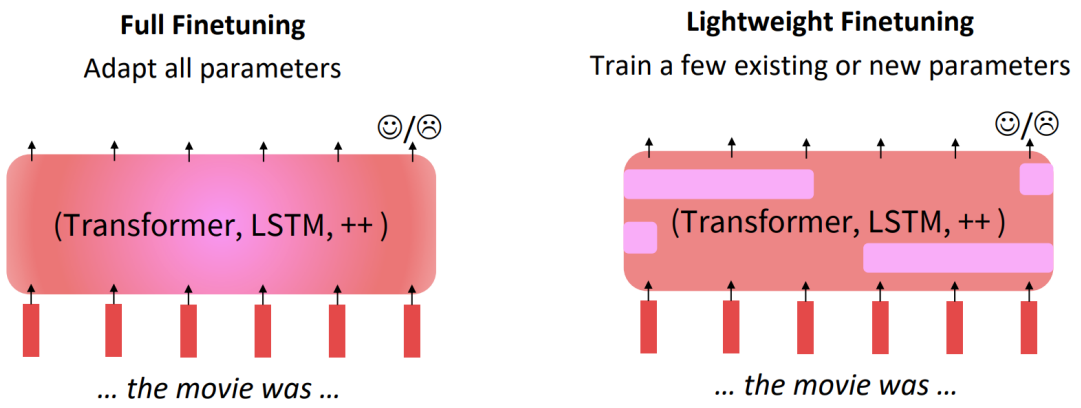



Figure 14: Full Finetuning vs. Parameter-Efficient Finetuning

## Parameter-Efficient Finetuning: Prefix-Tuning, Prompt tuning

Prefix-Tuning adds a prefix of parameters, and freezes all pretrained parameters. The prefix is processed by the model just like real words would be. Advantage: each element of a batch at inference could run a different tuned model.

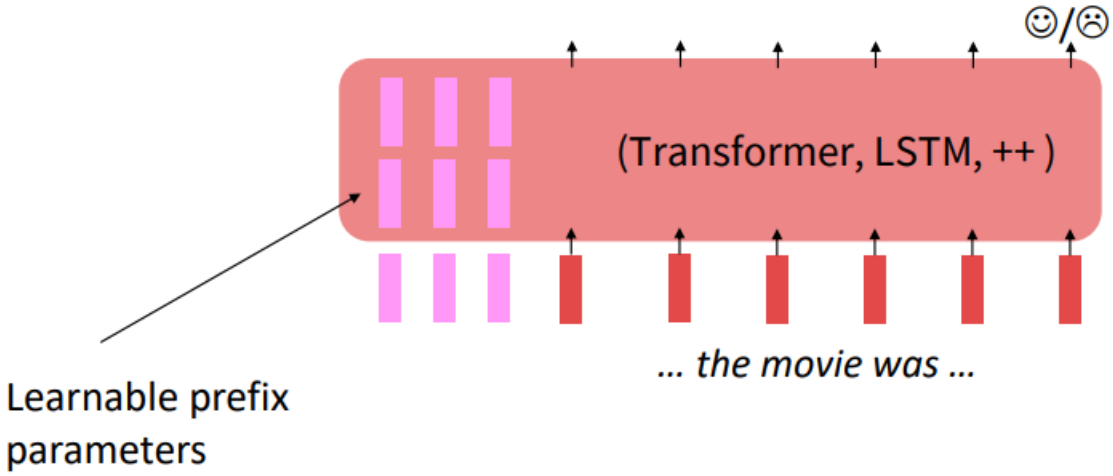



Figure 15: Full Finetuning vs. Prefix-Tuning

## Parameter-Efficient Finetuning: Low-Rank Adaptation (LORA)

Low-Rank Adaptation (LoRA) is a technique used to efficiently fine-tune large, pretrained language models by learning a low-rank "difference" between the pretrained and fine-tuned weight matrices. Instead of updating the entire weight matrix, LoRA introduces a low-rank approximation that captures the essential adjustments needed for fine-tuning.

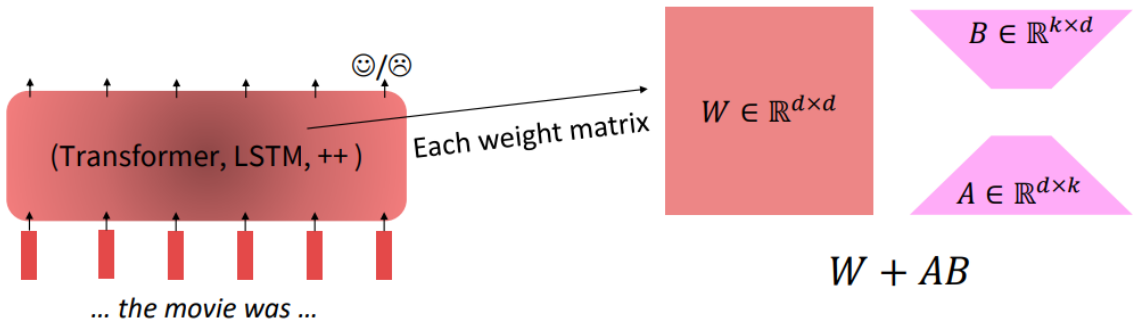



Figure 16: Full Finetuning vs. Low-Rank Adaptation

**Concept of Low-Rank Adaptation**

In large pretrained models, fine-tuning all weights can be computationally expensive and may require a significant amount of data. LoRA addresses this by adding a low-rank matrix to the pretrained weights during fine-tuning, which allows the model to adjust to the new task in an efficient and scalable way. This approach is generally simpler and requires fewer parameters to learn compared to other fine-tuning techniques like prefix-tuning.

**Mathematical Formulation**

Let:

- $W_0 \in \mathbb{R}^{d \times d}$ be the pretrained weight matrix, where $d$ is the dimensionality of the weight matrix.
- $W$ be the fine-tuned weight matrix.

Instead of directly modifying $W_0$, LoRA introduces a low-rank update $\Delta W$ to represent the "difference" between $W_0$ and $W$. This update is given by:

$$\Delta W = AB$$

where:

- $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times d}$, with $r \ll d$. The rank $r$ is much smaller than $d$, allowing the update $\Delta W$ to be low-rank.
- The matrices $A$ and $B$ are learned during fine-tuning, while the pretrained weights $W_0$ remain fixed.

Thus, the fine-tuned weight matrix $W$ can be expressed as:

$$W = W_0 + AB$$

This approach reduces the number of parameters to be learned to $2dr$ (since $A$ and $B$ together have $dr + dr = 2dr$ parameters), making the fine-tuning process more efficient. By focusing on this low-rank difference, LoRA enables the model to adapt to new tasks without directly modifying the pretrained weights.

**Training Process**

During fine-tuning:

- The pretrained weight matrix $W_0$ remains fixed.
- The matrices $A$ and $B$ are optimized to minimize the task-specific loss function.
- After training, the final weight matrix $W = W_0 + AB$ incorporates the task-specific adjustments learned through the low-rank update.

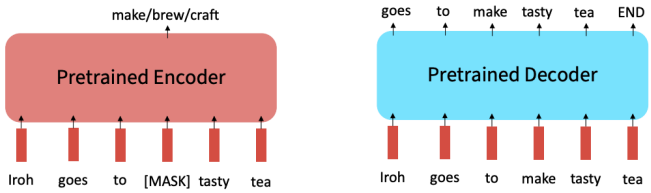
**Benefits of LoRA**

LoRA's low-rank adaptation allows the model to maintain the general knowledge encoded in the pretrained weights while efficiently incorporating new task-specific information. By learning only the "difference" needed to adapt to the new task, LoRA reduces the computational cost and parameter requirements compared to full fine-tuning, making it a highly effective method for adapting large models.

## Limitations of pretrained encoders

Those results looked great! Why not use pretrained encoders for everything?

If your task involves generating sequences, consider using a pretrained decoder; BERT and other pretrained encoders don't naturally lead to nice autoregressive (1-word-at-a-time) generation methods.

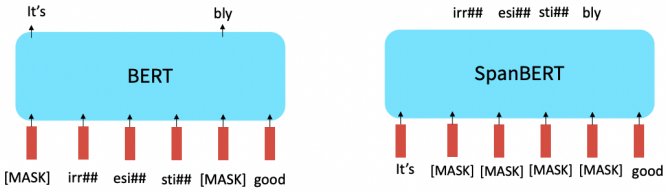



## Extensions of BERT

You'll see a lot of BERT variants like RoBERTa, SpanBERT, ...

Some generally accepted improvements to the BERT pretraining formula:

- **RoBERTa**: mainly just train BERT for longer and remove next sentence prediction!
- **SpanBERT**: masking contiguous spans of words makes a harder, more useful pretraining task.




A takeaway from the RoBERTa paper: more compute, more data can improve pretraining even when not changing the underlying Transformer encoder.

| Model | data | bsz | steps | SQuAD (v1.1/2.0) | MNLI-m | SST-2 |
|---|---|---|---|---|---|---|
| RoBERTa | | | | | | |
| with BOOKS + WIKI | 16GB | 8K | 100K | 93.6/87.3 | 89.0 | 95.3 |
| + additional data (§3.2) | 160GB | 8K | 100K | 94.0/87.7 | 89.3 | 95.6 |
| + pretrain longer | 160GB | 8K | 300K | 94.4/88.7 | 90.0 | 96.1 |
| + pretrain even longer | 160GB | 8K | 500K | **94.6/89.4** | **90.2** | **96.4** |
| $\text{BERT}_{\text{LARGE}}$ | | | | | | |
| with BOOKS + WIKI | 13GB | 256 | 1M | 90.9/81.8 | 86.6 | 93.7 |

## Pretraining Decoders

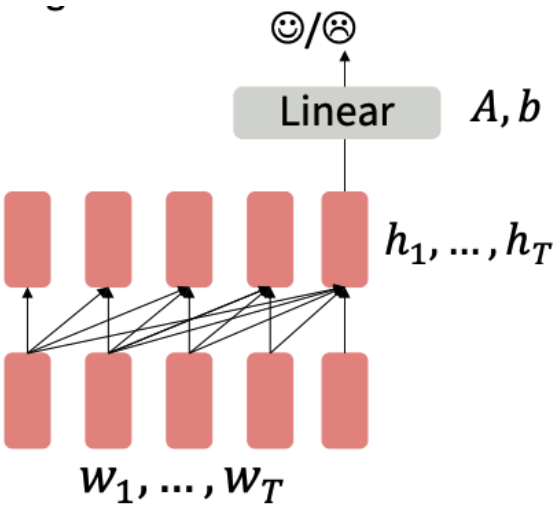
When using language model pretrained decoders, we can ignore that they were trained to model $p(w_t|w_{1:t-1})$.

We can finetune them by training a softmax classifier on the last word's hidden state.

$$h_1, \ldots, h_T = \text{Decoder}(w_1, \ldots, w_T)$$
$$y \sim Ah_T + b$$

Where $A$ and $b$ are randomly initialized and specified by the downstream task.

Gradients backpropagate through the whole network.



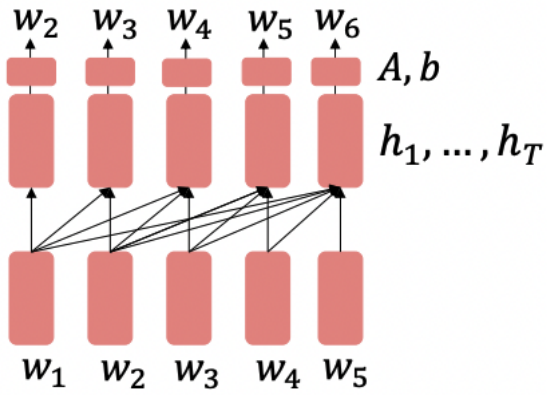

[Note how the linear layer hasn't been pretrained and must be learned from scratch.]

The basic architecture is extremely similar to the encoder model and comprises a series of transformers that operate on learned word embeddings. However, the goal is different. The encoder aimed to build a representation of the text that could be fine-tuned to solve a variety of more specific NLP tasks. Conversely, the decoder has one purpose: to generate the next token in a sequence. It can generate a coherent text passage by feeding the extended sequence back into the model. It's natural to pretrain decoders as language models and then use them as generators, finetuning their $p_\theta(w_t|w_{1:t-1})$. This is helpful in tasks **where the output is a sequence** with a vocabulary like that at pretraining time!

- Dialogue (context = dialogue history)
- Summarization (context = document)

$$h_1, \ldots, h_T = \text{Decoder}(w_1, \ldots, w_T)$$
$$w_t \sim Ah_{t-1} + b$$

Where $A, b$ were pretrained in the language model!




[Note how the linear layer has been pretrained.]

## Generative Pretrained Transformer (GPT)

2018's GPT was a big success in pretraining a decoder!

- Transformer decoder with 12 layers, 117M parameters.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges.
- Trained on BooksCorpus: over 7000 unique books.
  - Contains long spans of contiguous text, for learning long-distance dependencies.
- The acronym "GPT" never showed up in the original paper; it could stand for *"Generative PreTraining"* or *"Generative Pretrained Transformer"*.

How do we format inputs to our decoder for **finetuning tasks?**
**Natural Language Inference:** Label pairs of sentences as *entailing/contradictory/neutral*.
**Entailment example**:

- Premise: *The man is in the doorway*
- Hypothesis: *The person is near the door*

meaning the premise entails the hypothesis, that i can believe the hypothesis if i believe the premise.
Here's roughly how the input was formatted, as a sequence of tokens for the decoder:
**[START] The man is in the doorway [DELIM] The person is near the door [EXTRACT].**
The linear classifier is applied to the representation of the `[EXTRACT]` token to predict "yes" (entailment) or "no" (no entailment). GPT results on various natural language inference datasets.

## GPT2

We mentioned how pretrained decoders can be used *in their capacities as language models*. **GPT-2**, a larger version (1.5B) of GPT trained on more data, was shown to produce relatively convincing samples of natural language.

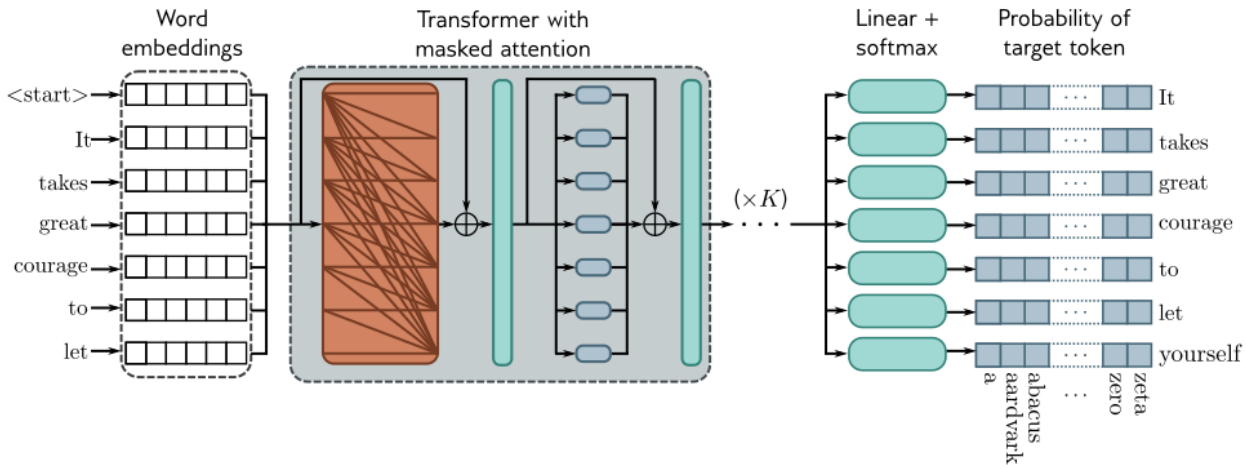
| |
|---|
| **Context (human-written):** In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English. |
| **GPT-2:** The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.<br><br>Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.<br><br>Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. |

## GPT-3, In-context learning, and very large models

So far, we've interacted with pretrained models in two ways:

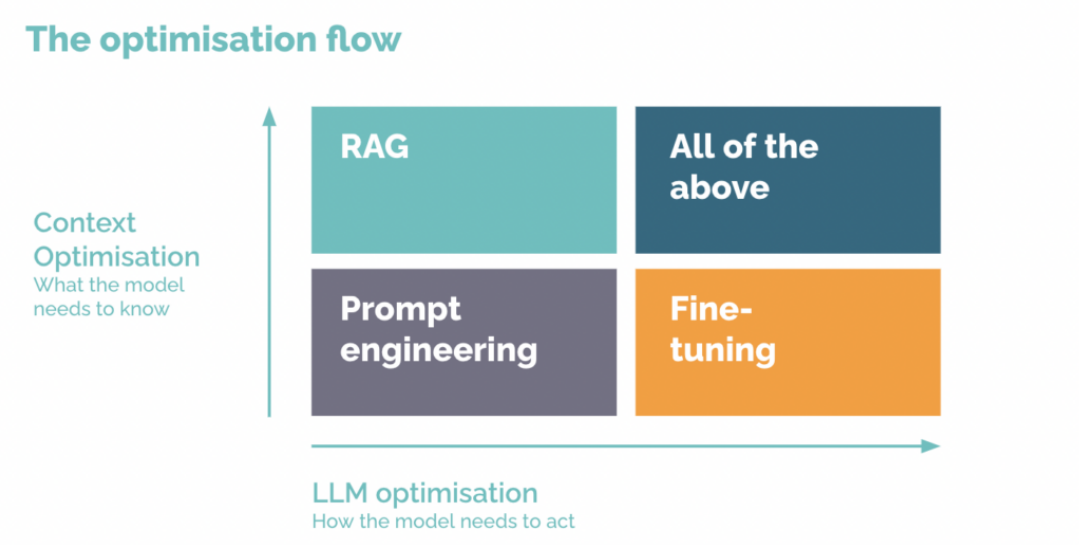
- Sample from the distributions they define (maybe providing a prompt)
- Fine-tune them on a task we care about and take their predictions.

Very large language models seem to perform some kind of learning without gradient steps simply from examples you provide within their contexts (no fine tuning).
**GPT-3 has 175 billion parameters.**



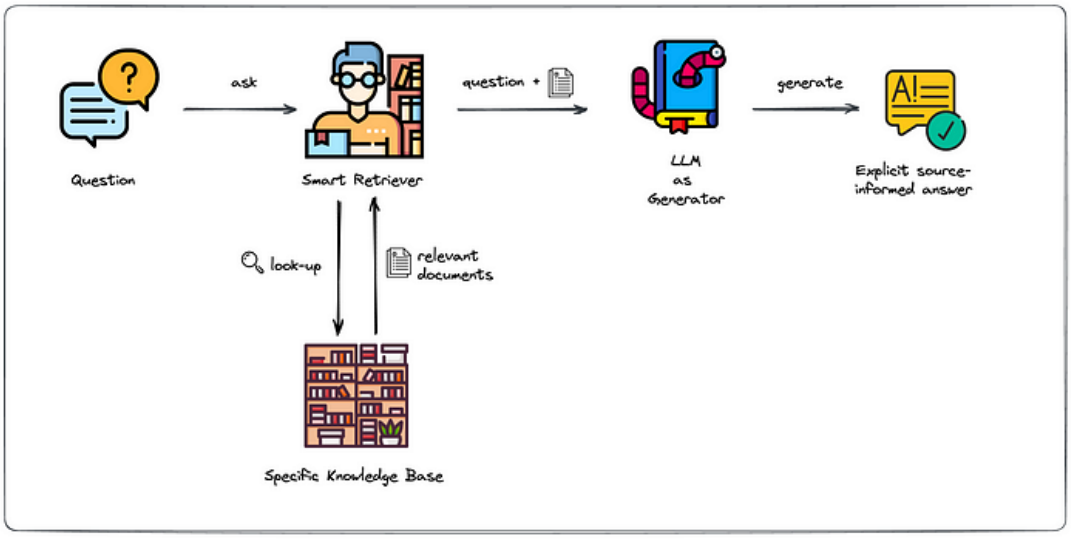

Training GPT3-type decoder network. The tokens are mapped to word embeddings with a special `<start>` token at the beginning of the sequence. The embeddings are passed through a series of transformers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and the embeddings of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the following ground truth token in the sequence. In other words, at position one, we want to maximize the probability of the token *It*; at position two, we want to maximize the probability of the token *takes*; and so on. Masked self-attention ensures the system cannot cheat by looking at subsequent inputs. The autoregressive task has the advantage of making efficient use of the data since every word contributes a term to the loss function. However, it only exploits the left context of each word.

# Training and Adaptation of GPT Models




## 1- Retrieval-Augmented Generation (RAG)

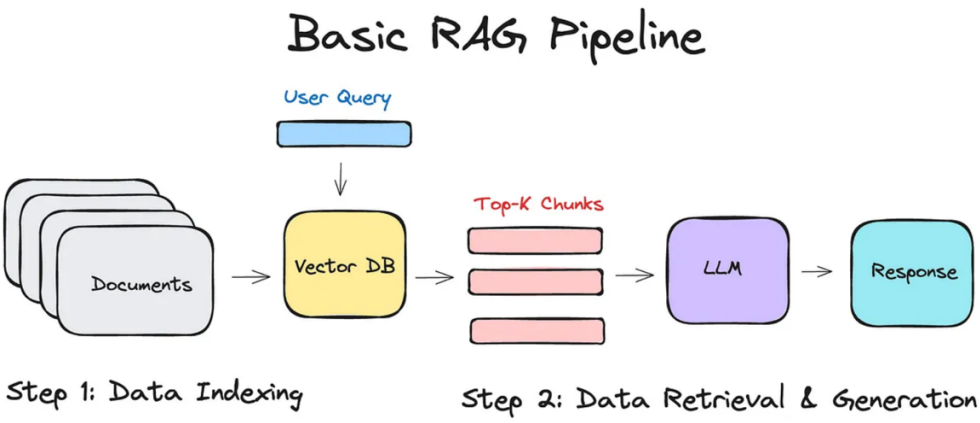
Retrieval-Augmented Generation (RAG) is a technique that enhances language models by incorporating external information retrieval into the generation process. Instead of relying solely on pre-trained knowledge, the model retrieves relevant documents from an external knowledge base before generating responses. This approach improves factual accuracy, reduces hallucinations, and allows the model to stay up to date without requiring frequent retraining.




RAG consists of two key components: a **retriever** and a **generator**. The retriever searches for relevant documents from a knowledge source, such as a vector database (e.g., FAISS) or a traditional search engine. The generator then conditions its response on the retrieved documents along with the original query.

The process works as follows:

- The query is first transformed into an embedding representation.
- The retriever searches for the most relevant documents based on similarity.
- The retrieved documents are combined with the query and fed into the model.
- The model generates an answer using both the retrieved knowledge and its pre-trained understanding.

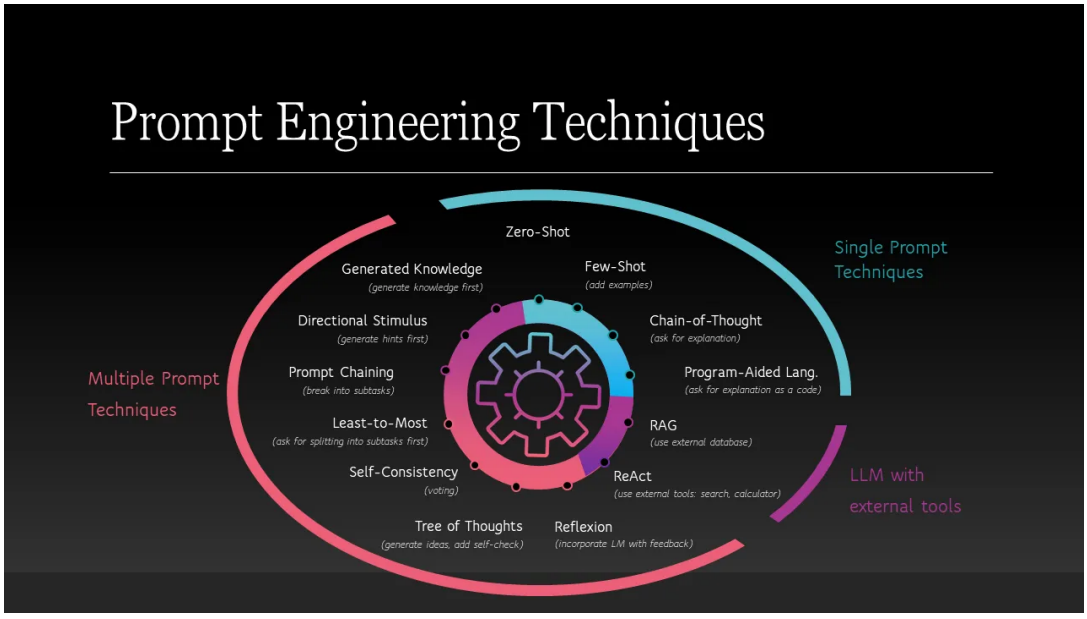



Using RAG provides several advantages. It significantly improves factual reliability by incorporating external knowledge, allowing the model to provide up-to-date information. It is also useful in domain-specific applications such as legal or medical AI, where access to accurate external sources is crucial. Additionally, RAG reduces the need for frequent fine-tuning, since new knowledge can be retrieved dynamically.

However, RAG also comes with challenges. The retrieval process can introduce latency, especially when working with large-scale document collections. If the retriever fails to find relevant documents, the model's response quality may degrade. Furthermore, ensuring scalability and maintaining an efficient retrieval system is critical for real-time applications.

## 2- Prompt Engineering

Prompt engineering is the practice of designing effective input prompts to guide the behavior of language models. Since large language models (LLMs) do not require explicit retraining to adapt to new tasks, carefully crafted prompts can significantly influence the quality and accuracy of their responses. This technique is crucial for optimizing model performance in various applications.




There are several key prompting strategies:

**1. Zero-Shot Prompting:** The model is given a direct query without any examples. While efficient, it may not always produce the most reliable responses.

**Summarization**

**Prompt:** "Summarize the following paragraph in one sentence: [input paragraph]"

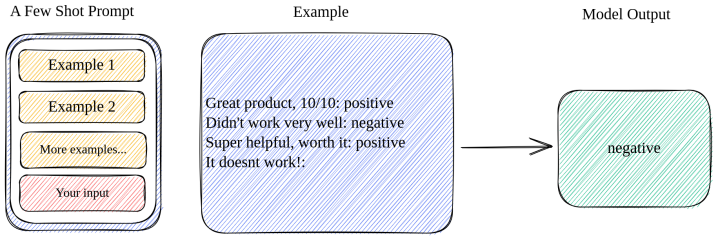
**Creative Writing**

**Prompt:** "Write a short story about a time traveler visiting ancient Rome. Include sensory details and dialogue."

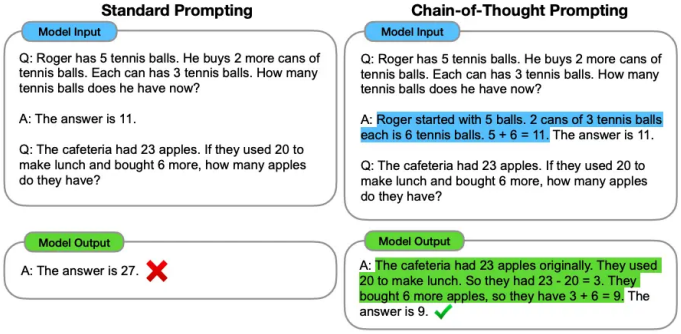
**Translation**

**Prompt:** "Translate the following English text to Spanish: [input text]"

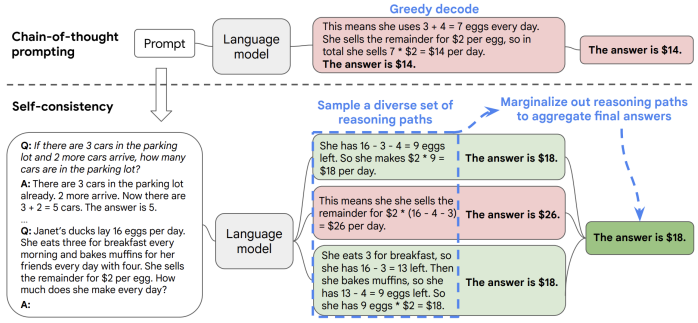
**2. Few-Shot Prompting:** The prompt includes a few examples of input-output pairs, helping the model generalize better to the task. This improves performance without requiring fine-tuning.



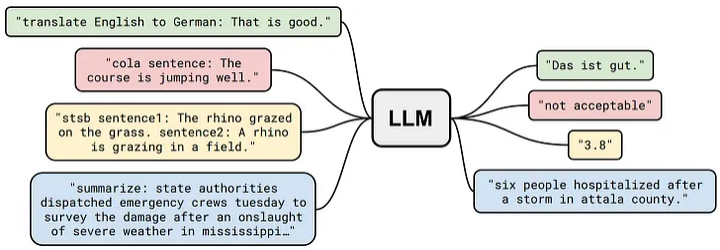

**3. Chain-of-Thought (CoT) Prompting:** The model is encouraged to reason step-by-step before producing a final answer. This is particularly useful for mathematical and logical reasoning tasks.



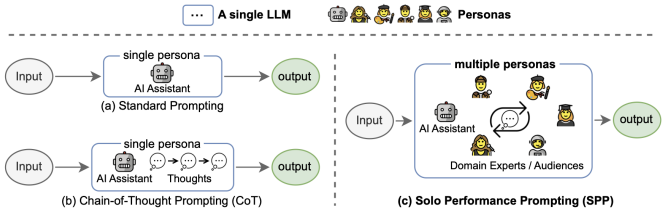

**4. Self-Consistency Prompting:** The model generates multiple responses using CoT reasoning, and the final answer is selected based on majority agreement, improving reliability.

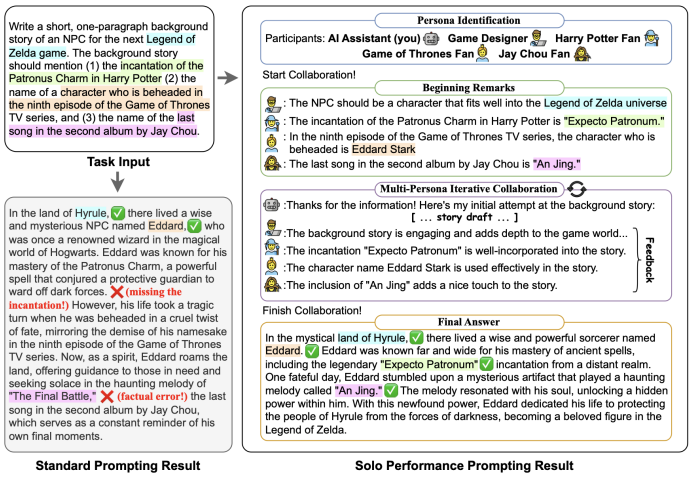


**5. Instruction-Based Prompting:** The prompt explicitly describes how the model should respond, such as "Explain this in simple terms" or "Summarize this article in two sentences."




**6. Persona-Based Prompting:** The model is instructed to respond in a specific style, such as "Answer as if you were a financial analyst."



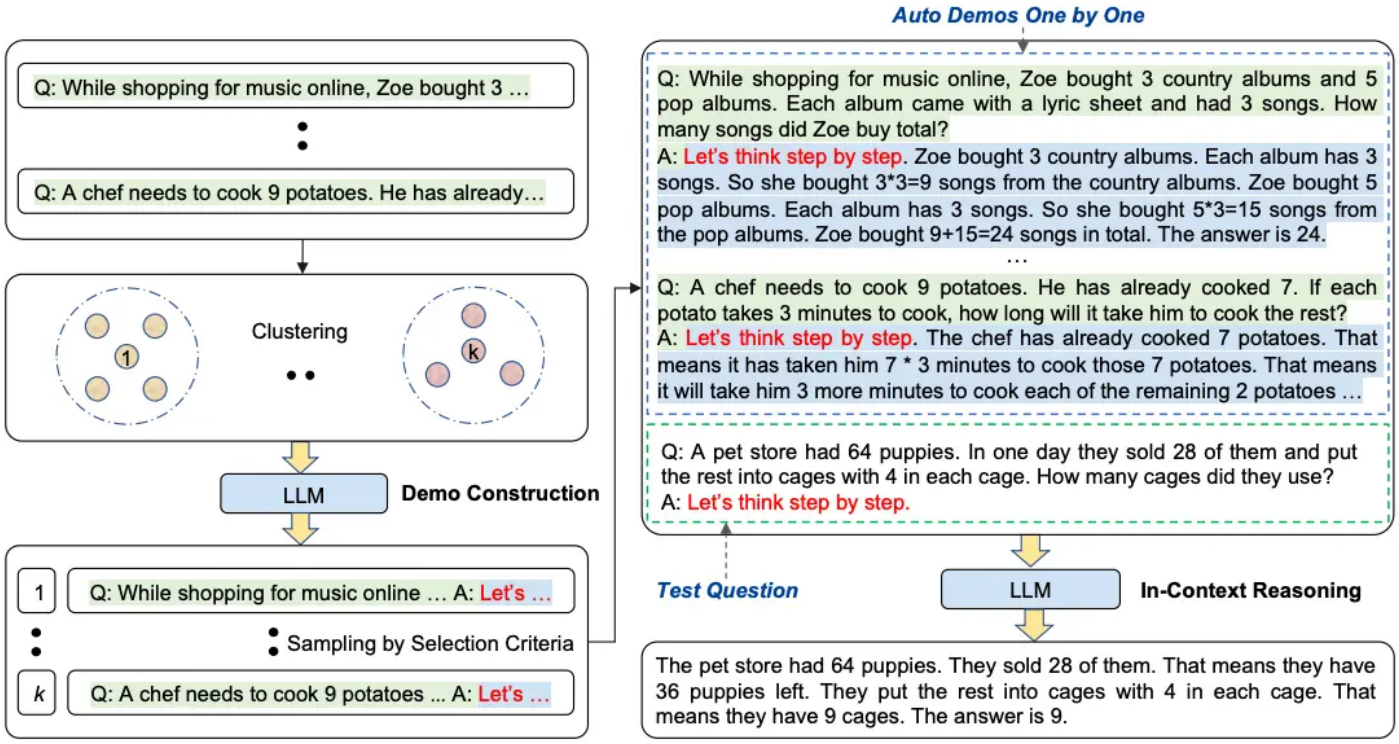

## 7. Automatic Chain-of-Thought (Auto-CoT)

Automatic Chain-of-Thought (Auto-CoT) prompting is a method designed to construct high-quality demonstrations for reasoning tasks automatically. Instead of relying on manually created examples, Auto-CoT uses a systematic approach to generate reasoning demonstrations by clustering similar questions and sampling representative ones from each cluster. This ensures diverse and effective demonstrations without human intervention.

The Auto-CoT process consists of two main steps:

1. **Question Clustering:** Questions from a dataset are partitioned into clusters based on their semantic similarity. This ensures that similar reasoning patterns are grouped together.
2. **Demonstration Sampling:** From each cluster, a representative question is selected, and its reasoning chain is generated using a language model with Zero-Shot Chain-of-Thought (CoT) prompting. Simple heuristics, such as the "Let's think step by step" prompt, are used to create these reasoning chains.

Once the reasoning demonstrations are constructed, they are included as examples in the prompt for in-context learning. This allows the language model to apply reasoning skills to new, unseen questions based on the provided demonstrations.




## Steps in Automatic Chain-of-Thought (Auto-CoT)

**Algorithm 1** Cluster

**Require:** A set of questions $\mathcal{Q}$ and the number of demonstrations $k$
**Ensure:** Sorted questions $\mathbf{q}^{(i)} = [q_1^{(i)}, q_2^{(i)}, \ldots]$ for each cluster $i$ $(i = 1, \ldots, k)$

**procedure** CLUSTER($\mathcal{Q}$, $k$)
  **for** each question $q$ in $\mathcal{Q}$ **do**
    Encode $q$ by Sentence-BERT
  Cluster all the encoded question representations into $k$ clusters
  **for** each cluster $i = 1, \ldots, k$ **do**
    Sort questions $\mathbf{q}^{(i)} = [q_1^{(i)}, q_2^{(i)}, \ldots]$ in the ascending order of the distance to the cluster center
  **return** $\mathbf{q}^{(i)}$ $(i = 1, \ldots, k)$

**Algorithm 2** Construct

**Require:** Sorted questions $\mathbf{q}^{(i)} = [q_1^{(i)}, q_2^{(i)}, \ldots]$ for each cluster $i$ $(i = 1, \ldots, k)$, empty demonstration list $\mathbf{d}$
**Ensure:** Demonstration list $\mathbf{d} = [d^{(1)}, \ldots, d^{(k)}]$

**procedure** CONSTRUCT($\mathbf{q}^{(i)}, \ldots, \mathbf{q}^{(k)}$)
  **for** each cluster $i = 1, \ldots, k$ **do**
    **for** each question $q_j^{(i)}$ in $\mathbf{q}^{(i)}$ **do**
      Generate rationale $r_j^{(i)}$ and answer $a_j^{(i)}$ for $q_j^{(i)}$ using Zero-Shot-CoT
      **if** $q_j^{(i)}, r_j^{(i)}$ satisfy selection criteria **then**
        Add $d^{(i)} = [\text{Q: } q_j^{(i)}, \text{A: } r_j^{(i)} \circ a_j^{(i)}]$ to $\mathbf{d}$
        **break**
  **return** $\mathbf{d}$

### 1. Question Clustering

The first step is to partition the given set of questions $Q$ into $k$ clusters using a clustering algorithm. The process involves the following:

- Each question is transformed into a vector representation using Sentence-BERT.
- The contextualized vectors are averaged to produce fixed-size question embeddings.
- The $k$-means clustering algorithm is applied to group these embeddings into $k$ clusters.
- Within each cluster $i$, questions are sorted into a list:

$$q^{(i)} = [q_1^{(i)}, q_2^{(i)}, \ldots]$$

  in ascending order of their distance to the cluster center.

**2. Demonstration Sampling**

In this step, reasoning demonstrations are constructed for each cluster $i$. The key steps are as follows:

- The question $q_j^{(i)}$ closest to the cluster center is selected first from the sorted list $q^{(i)}$, where $q^{(i)}$ is the list of questions in cluster $i$, sorted by their distance to the cluster center.
- A prompted input is formatted as:
$$[\text{Q: } q_j^{(i)}, \text{ A: [P]}]$$
where $[P]$ is the prompt "Let's think step by step."
- The prompted input is passed through a language model using Zero-Shot-CoT, which generates:
  - $r_j^{(i)}$: The rationale (step-by-step reasoning).
  - $a_j^{(i)}$: The final answer.
- The demonstration $d_j^{(i)}$ is created by concatenating the question, rationale, and answer:
$$d_j^{(i)} = [\text{Q: } q_j^{(i)}, \text{ A: } r_j^{(i)} \circ a_j^{(i)}]$$
where $\circ$ denotes concatenation.

**Selection Criteria:**

- A demonstration $d_j^{(i)}$ is selected only if:
  - $q_j^{(i)}$ has no more than 60 tokens.
  - $r_j^{(i)}$ contains no more than 5 reasoning steps.

This approach ensures scalable and high-quality reasoning demonstrations by clustering similar questions and automatically constructing reasoning chains.

**Key Features of Auto-CoT:**

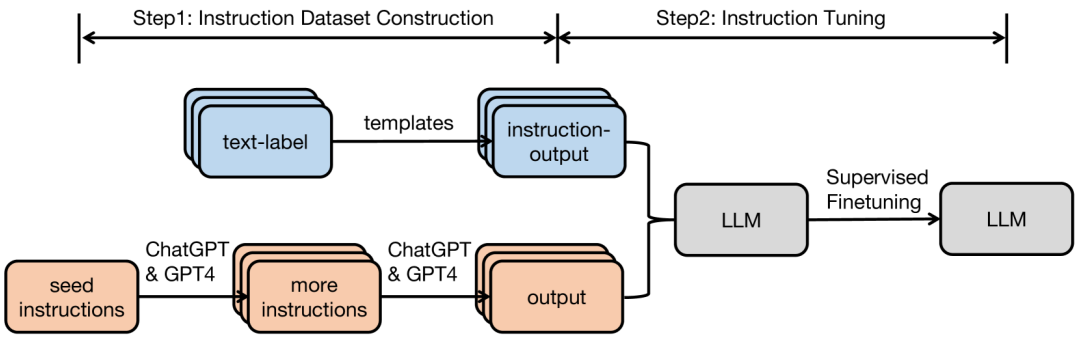
- **Automated Demo Construction:** The process eliminates the need for manually curated reasoning examples, making it scalable and efficient.
- **Cluster-Based Selection:** By sampling representative questions from clusters, Auto-CoT ensures diversity and relevance in the constructed demonstrations.
- **Improved Reasoning Performance:** Demonstrations created via Auto-CoT enhance the model's ability to perform complex reasoning tasks.

## Summary:

Well-constructed prompts can enhance model interpretability, improve accuracy, and reduce hallucinations. However, prompt engineering requires experimentation, as slight changes in phrasing can lead to different results. It is an essential technique for maximizing the capabilities of generative AI without modifying the model's internal parameters.

## 3- Instruction Tuning

Instruction tuning is a fine-tuning method for large language models that focuses on aligning their behavior with human instructions. Instead of training the model on raw data alone, instruction tuning uses datasets that pair user instructions with appropriate outputs, enabling the model to better understand and follow explicit directions.




**Key Principles:**

- Models are trained on diverse examples of instructions and corresponding responses to improve generalization across tasks.
- The goal is to enhance the model's capability to handle unseen instructions effectively by learning patterns in how humans expect instructions to be followed.
- Instruction tuning complements existing pretraining and fine-tuning steps, acting as an additional phase for improving usability and alignment.

**How It Works:**

**Instructions for MC-TACO question generation task**

- **Title:** Writing questions that involve commonsense understanding of "event duration".
- **Definition:** In this task, we ask you to write a question that involves "event duration", based on a given sentence. Here, event duration is defined as the understanding of how long events typically last. For example, "brushing teeth", usually takes few minutes.
- **Emphasis & Caution:** The written questions are not required to have a single correct answer.
- **Things to avoid:** Don't create questions which have explicit mentions of answers in text. Instead, it has to be implied from what is given. In other words, we want you to use "instinct" or "common sense".

**Positive Example**

- **Input:** Sentence: Jack played basketball after school, after which he was very tired.
- **Output:** How long did Jack play basketball?
- **Reason:** the question asks about the duration of an event; therefore it's a temporal event duration question.

**Negative Example**

- **Input:** Sentence: He spent two hours on his homework.
- **Output:** How long did he do his homework?
- **Reason:** We DO NOT want this question as the answer is directly mentioned in the text.
- **Suggestion:** -

- **Prompt:** Ask a question on "event duration" based on the provided sentence.

1. **Instruction-Response Dataset:** A dataset is created where each example consists of:

$$(\text{Instruction}, \text{Response})$$

   For example:

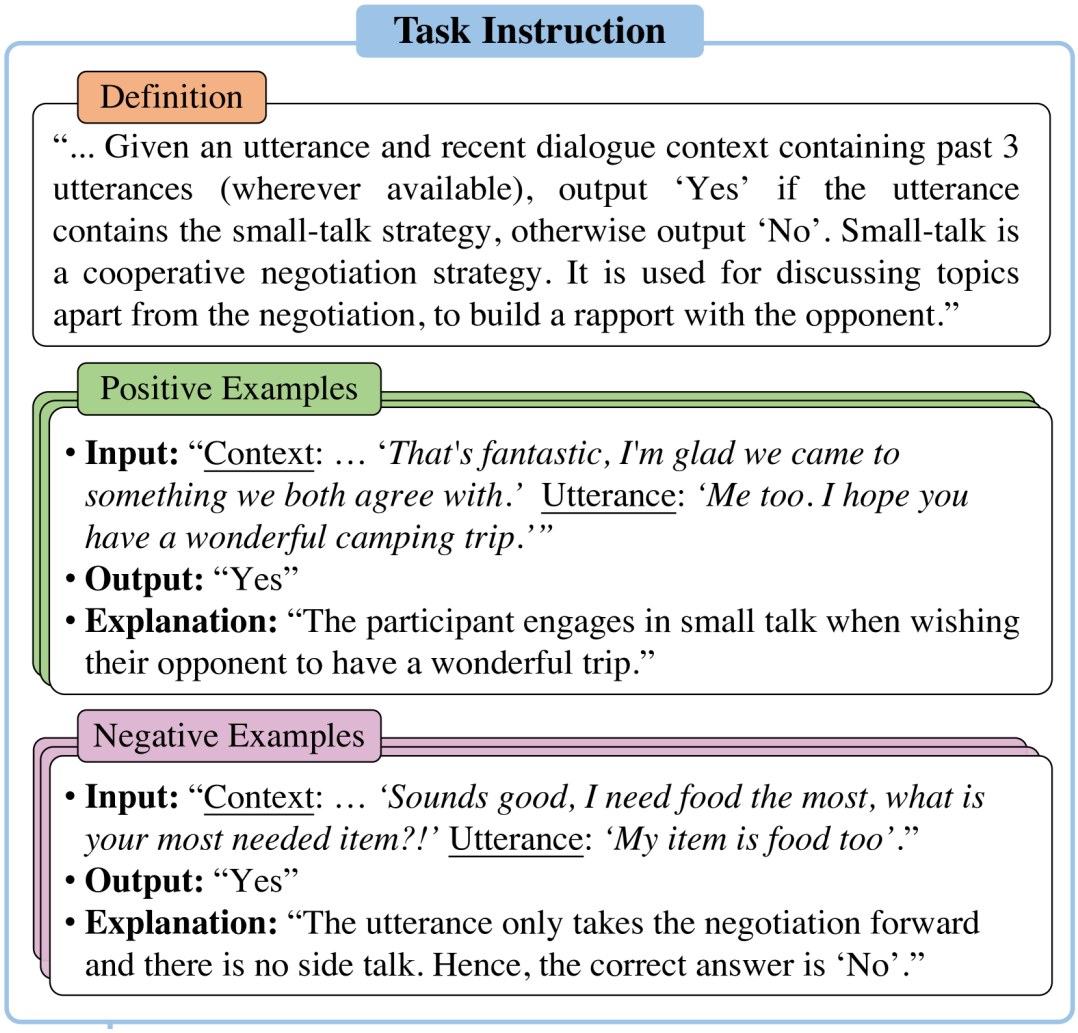
   - Instruction: *"Summarize the following paragraph in one sentence."*
   - Response: *"The paragraph discusses the impact of climate change on polar bears."*

2. **Fine-Tuning Process:** The model is fine-tuned on this dataset using supervised learning, minimizing the loss between the model's generated response and the ground truth response.

3. **Diverse Tasks:** The dataset includes a variety of tasks, such as:

   - Summarization
   - Question answering
   - Code generation
   - Translation

   This diversity ensures that the model can generalize to new tasks not explicitly included in the dataset.




**Advantages:**

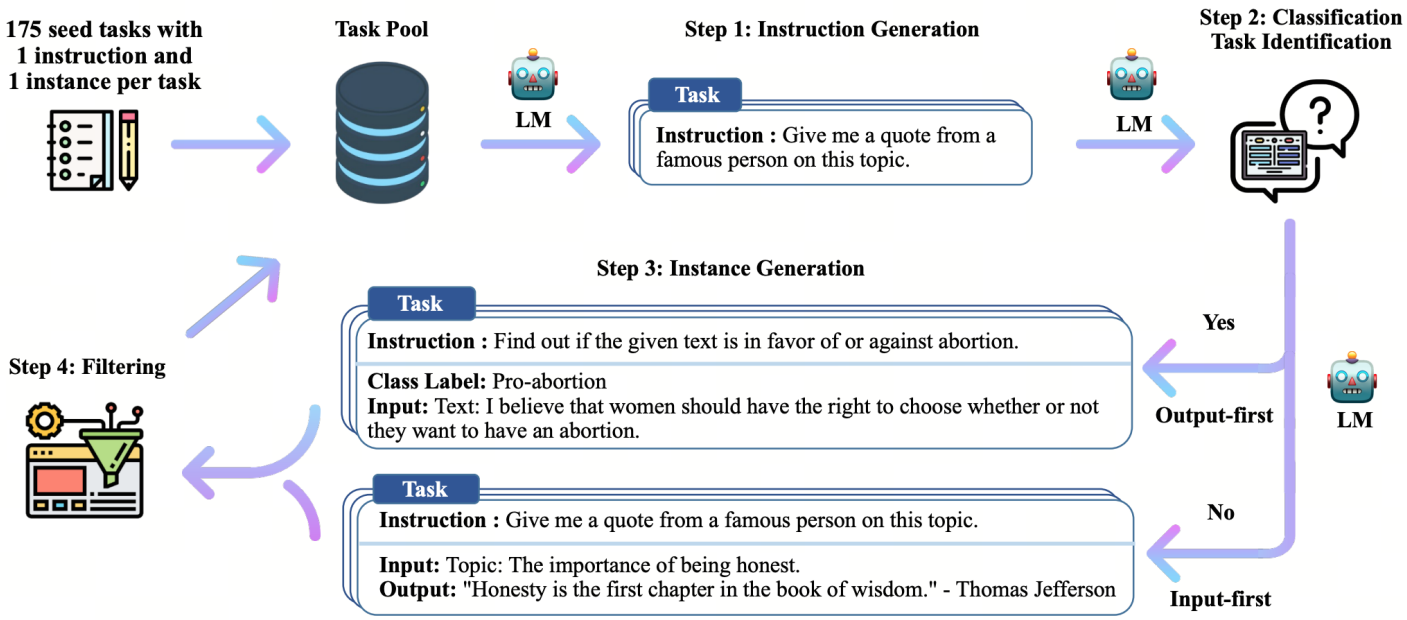
- **Improved Alignment:** The model becomes more responsive to user needs by understanding and fulfilling instructions more accurately.

- **Ease of Use:** Users no longer need to craft elaborate prompts to get the desired behavior, as the model is already fine-tuned to follow simple instructions.
- **Multi-Task Learning:** By learning from diverse instruction-response pairs, the model develops a strong ability to generalize across domains and tasks.

**Challenges:**

- **Dataset Quality:** The quality of the instruction-response dataset is critical. Poorly constructed datasets can lead to undesirable behavior.
- **Bias Amplification:** If the dataset contains biased instructions or responses, the model may inadvertently amplify these biases.
- **Task-Specific Limitations:** Instruction tuning does not replace task-specific fine-tuning for highly specialized use cases.

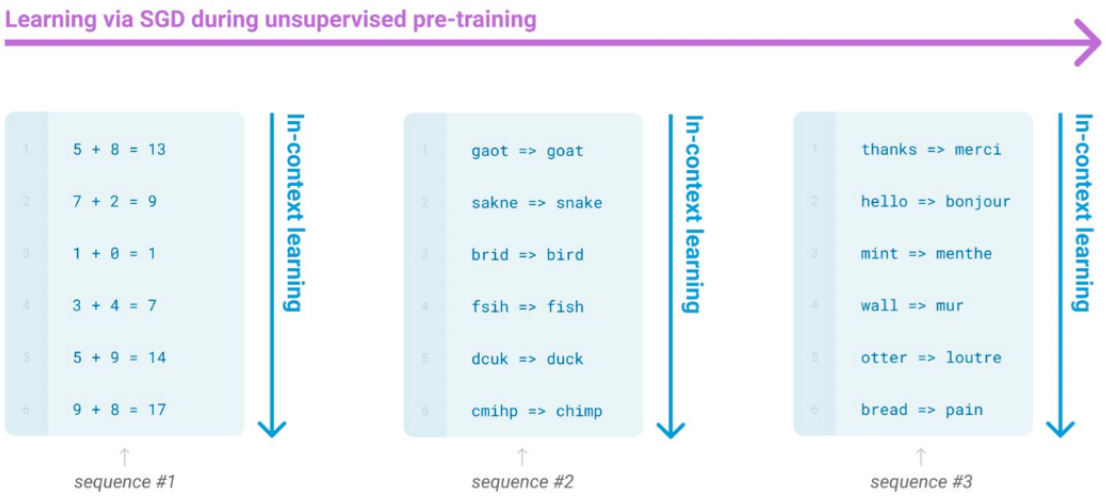
## 4- Self-Instruct:




**ChatGPT/GPT-4/GPT-3.5 Turbo introduced a further instruction-tuning idea.**
The in-context examples seem to specify the task to be performed, and the conditional distribution mocks performing the task to a certain extent.

## 5- In-context learning:

we will provide a word and an arrow connecting it to the translation and the goal is by providing enough example the model will learn how to map otter to its translation **loutre**.




Language model meta-learning. During unsupervised pre-training, a language model develops a broad set of skills and pattern recognition abilities. It then uses these abilities at inference time to rapidly adapt to or recognize the desired task. We use the term "in-context learning" to describe the inner loop of this process, which occurs within the forward-pass upon each sequence. The sequences in this diagram are not intended to be representative of the data a model would see during pre-training, but are intended to show that there are sometimes repeated sub-tasks embedded within a single sequence.

### Other methods:

- Low-Rank Adaptation (LoRA)
- Reinforcement Learning from Human Feedback (RLHF)
- Prefix-Tuning
- Adapter layers
- ...

## Scaling laws

Scaling laws in Transformer models refer to empirical observations that model performance improves predictably with increases in three main factors: model size (number of parameters), dataset size, and computational power. Studies have shown that as we scale up these factors, the model's ability to generalize and perform on downstream tasks often continues to improve logarithmically, although with diminishing returns. This insight has driven the development of increasingly large models, as researchers seek to leverage these scaling laws to build more powerful and versatile language models.

**Can we predict model performance based on the amount of data & parameters?**

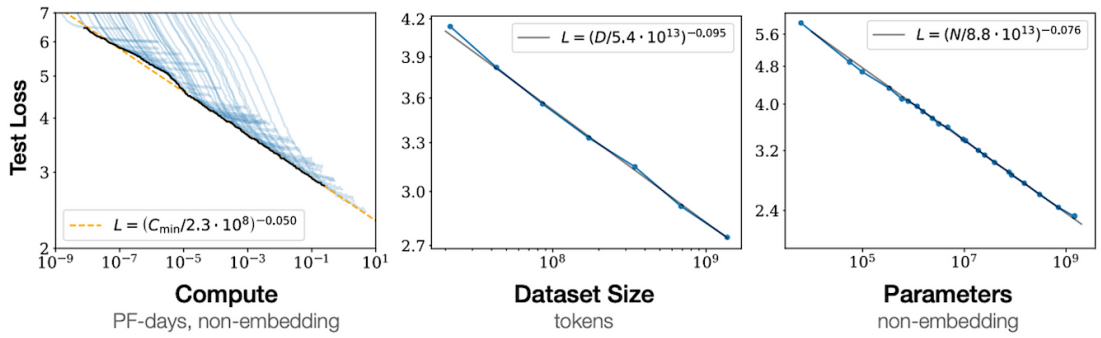



Figure 17: Scaling Laws

Empirical observation: scaling up models (larger models) leads to reliable gains in perplexity.

**Scaling can help identify model size – data tradeoffs**

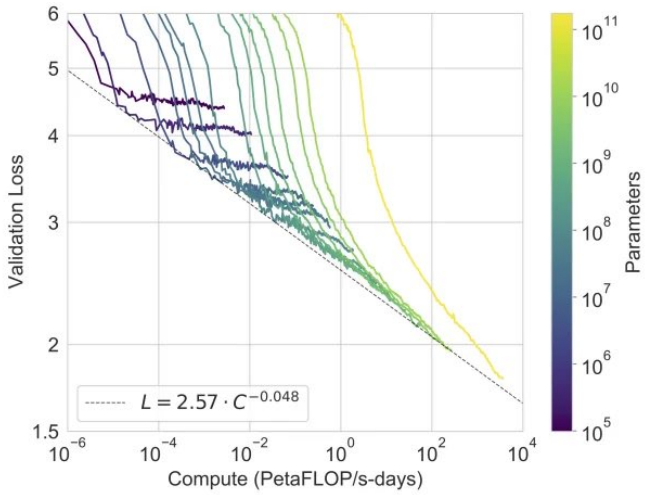



Figure 18: Scaling laws vs model size

Modern observation: train a big model that's not fully converged.

## Scaling laws for many other interesting architecture decisions

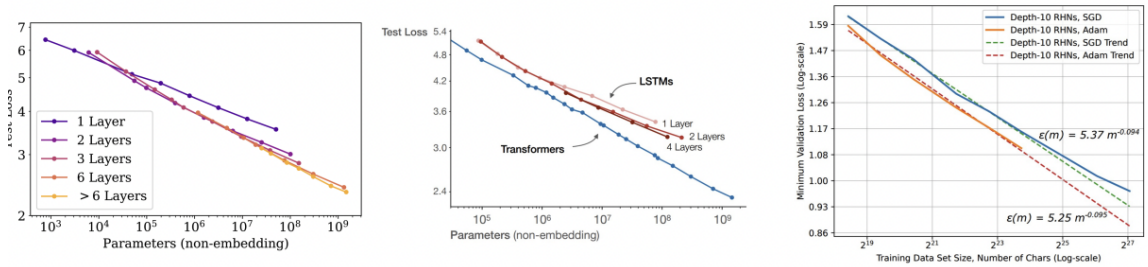



Figure 19: Scaling laws - other architectures

Predictable scaling helps us make intelligent decisions about architectures etc.

## Scaling Efficiency: how do we best use our compute

GPT-3 was **175B parameters** and trained on **300B** tokens of text.

Roughly, the cost of training a large transformer scales as **parameters * tokens**.

Did OpenAI strike the right parameter-token data to get the best model? No.

| Model | Size (# Parameters) | Training Tokens |
|---|---|---|
| LaMDA (Thoppilan et al., 2022) | 137 Billion | 168 Billion |
| GPT-3 (Brown et al., 2020) | 175 Billion | 300 Billion |
| Jurassic (Lieber et al., 2021) | 178 Billion | 300 Billion |
| *Gopher* (Rae et al., 2021) | 280 Billion | 300 Billion |
| MT-NLG 530B (Smith et al., 2022) | 530 Billion | 270 Billion |
| *Chinchilla* | 70 Billion | 1.4 Trillion |

**This 70B parameter model is better than the much larger other models!**

Figure 20: Chinchilla - Scaling efficiency

## What kinds of things does pretraining teach?

There's increasing evidence that pretrained models learn a wide variety of things about the statistical properties of language. Taking our examples from the start of class:

- *Northeastern University is located in _____, Massachuset.* [Trivia]
- *I put ___ fork down on the table.* [Syntax]
- *The woman walked across the street, checking for traffic over ___ shoulder.* [Coreference]
- *I went to the ocean to see the fish, turtles, seals, and _____.* [Lexical semantics/topic]
- *Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was ___.* [Sentiment]
- *Iroh went into the kitchen to make some tea. Standing next to Iroh, Zuko pondered his destiny. Zuko left the _____.* [Some reasoning – this is harder]
- *I was thinking about the sequence that goes 1, 1, 2, 3, 5, 8, 13, 21, _____.* [Some basic arithmetic; they don't learn the Fibonacci sequence]

- Models also learn – and can exacerbate racism, sexism, all manner of bad biases.

## Sometimes it also memorizes copyrighted material

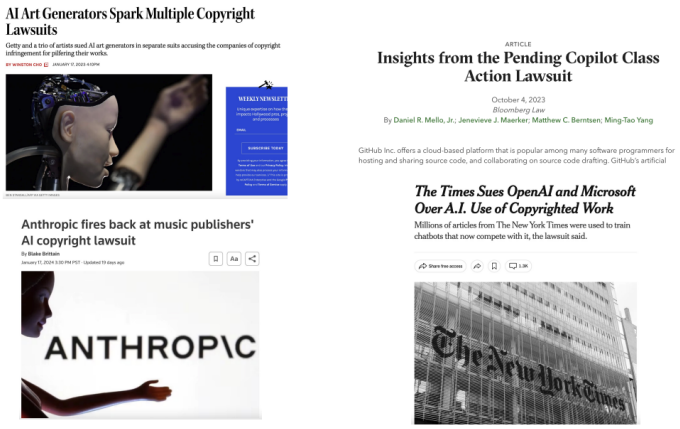



Figure 21: Memorizes copyrighted material

## Sometimes it learns some things we don't want..

- **Membership inference** lets you recover parts of the training data.
- Sometimes this training data is semi-private material from the web (addresses, emails).
- It learns the prejudices and biases of human beings who write online.

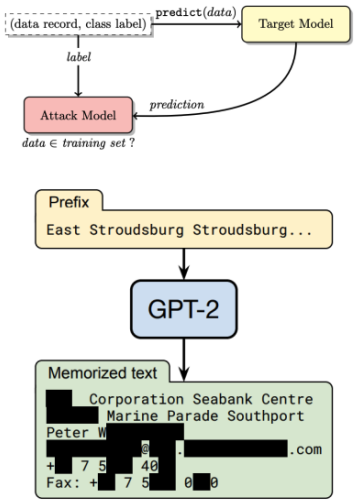



Figure 22: Memorizes copyrighted material

# Pretraining encoder-decoders: what pretraining objective to use?

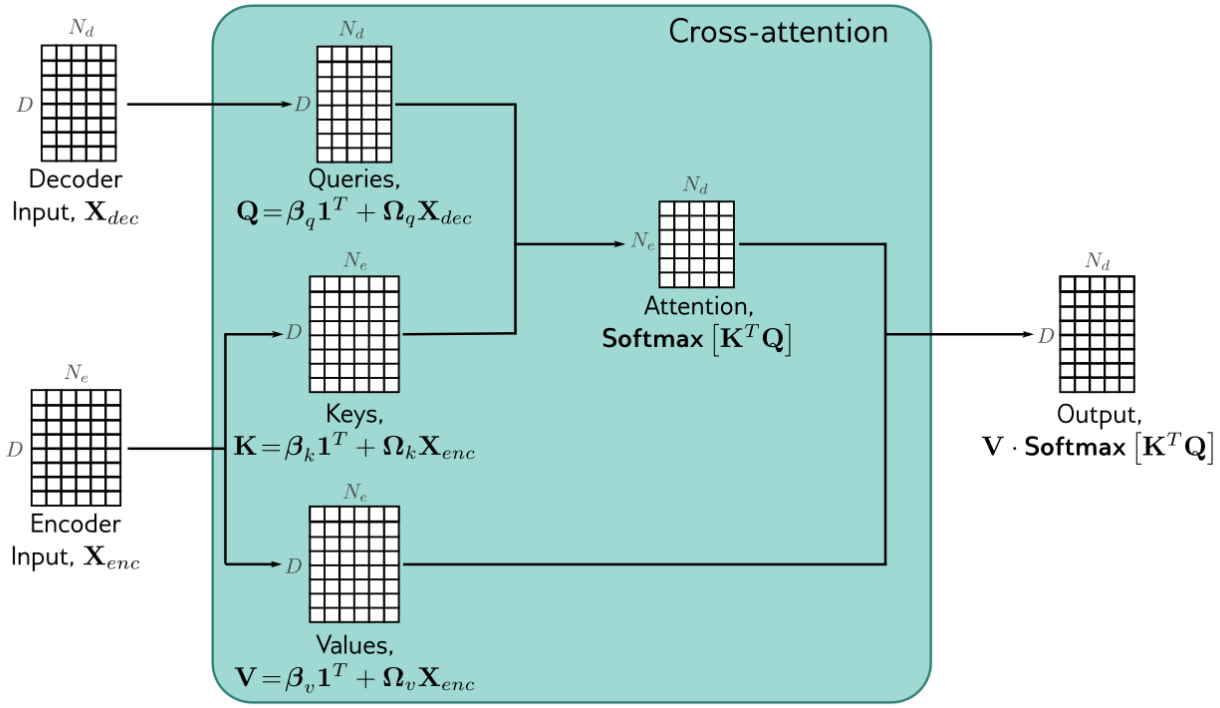



Figure 23: Cross-attention. The flow of computation is the same as in standard self-attention. However, the queries are calculated from the decoder embeddings Xdec , and the keys and values from the encoder embeddings Xenc . In the context of translation, the encoder contains information about the source language, and the decoder contains information about the target language statistics.

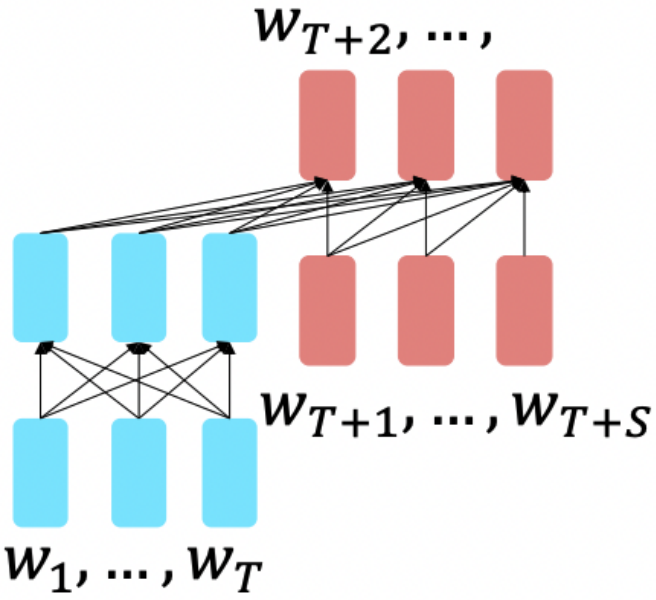
For **encoder-decoders**, we could do something like *language modeling*, but where a prefix of every input is provided to the encoder and is not predicted.

$$h_1, \ldots, h_T = \text{Encoder}(w_1, \ldots, w_T)$$

$$h_{T+1}, \ldots, h_{T+S} = \text{Decoder}(w_{T+1}, \ldots, w_{T+S}, h_1, \ldots, h_T)$$

$$y_i \sim Ah_i + b, \quad i > T$$

The **encoder** portion can benefit from bidirectional context; the **decoder** portion is used to train the whole model through language modeling, autoregressively predicting and then conditioning on one token at a time.




## Transformers for long sequences

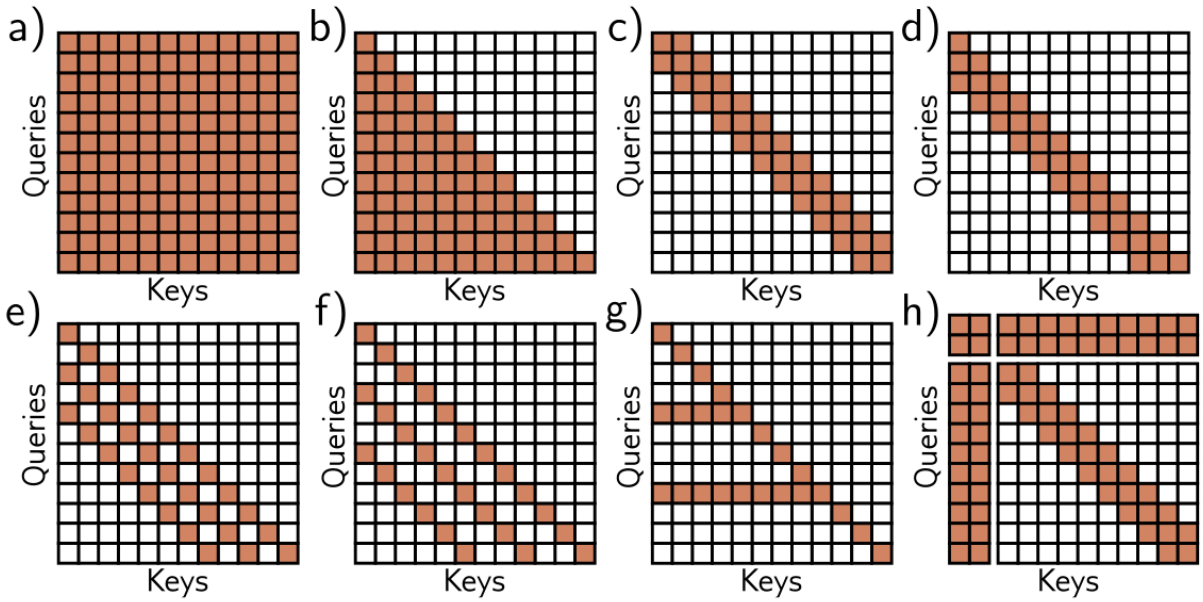


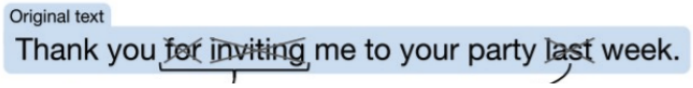

Figure 24: Interaction matrices for self-attention. a) In an encoder, every token interacts with every other token, and computation expands quadratically with the number of tokens. b) In a decoder, each token only interacts with the previous tokens, but complexity is still quadratic. c) Complexity can be reduced by using a convolutional structure (encoder case). d) Convolutional structure for decoder case. e–f) Convolutional structure with dilation rate of two and three (decoder case).g) Another strategy is to allow selected tokens to interact with all the other tokens (encoder case) or all the previous tokens (decoder case pictured). h) Alternatively, global tokens can be introduced (left two columns and top two rows). These interact with all of the tokens as well as with each other.

Since each token in a transformer encoder model interacts with every other token, the computational complexity scales quadratically with the length of the sequence. For a decoder model, each token only interacts with previous tokens, so there are roughly half the number of interactions, but the complexity still scales quadratically. These relationships can be visualized as interaction matrices (figure 26a–b).
This quadratic increase in the amount of computation ultimately limits the length of sequences that can be used. Many

methods have been developed to extend the transformer to cope with longer sequences. One approach is to prune the self-attention interactions or, equivalently, to sparsify the interaction matrix (figures 26c-h). For example, this can be restricted to a convolutional structure so that each token only interacts with a few neighboring tokens. Across multiple layers, tokens still interact at larger distances as the receptive field expands. As for convolution in images, the kernel can vary in size and dilation rate.
A pure convolutional approach requires many layers to integrate information over large distances. One way to speed up this process is to allow select tokens (perhaps at the start of every sentence) to attend to all other tokens (encoder model) or all previous tokens (decoder model). A similar idea is to have a small number of global tokens that connect to all the other tokens and themselves. Like the $< cls >$ token, these do not represent any word but serve to provide long-distance connections.

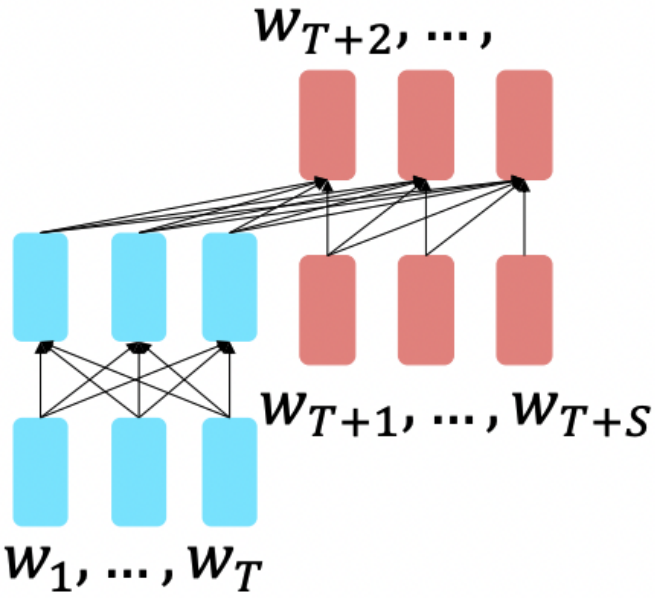
## Span Corruption: T5 Model (Raffel et al., 2018)

*Replace different-length spans from the input with unique placeholders; decode out the spans that were removed!*

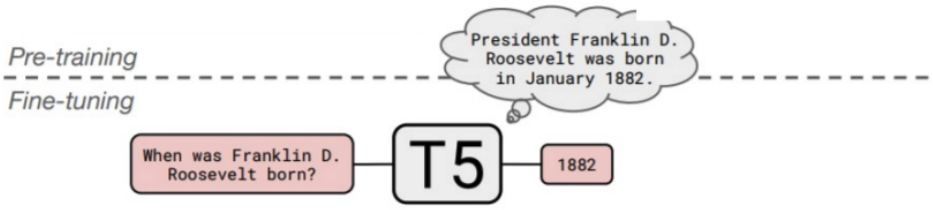



This is implemented in text preprocessing: it's still an objective that looks like **language modeling** at the decoder side.

**Raffel et al., 2018** found encoder-decoders to work better than decoders for their tasks, and span corruption (denoising) to work better than language modeling.




A fascinating property of T5: it can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters.
NQ: Natural Questions
WQ: WebQuestions
TQA: Trivia QA
All "open-domain" versions




| | NQ | WQ | TQA dev | TQA test | |
|---|---|---|---|---|---|
| Karpukhin et al. (2020) | **41.5** | 42.4 | **57.9** | – | |
| T5.1.1-Base | 25.7 | 28.2 | 24.2 | 30.6 | **220 million params** |
| T5.1.1-Large | 27.3 | 29.5 | 28.5 | 37.2 | **770 million params** |
| T5.1.1-XL | 29.5 | 32.4 | 36.0 | 45.1 | **3 billion params** |
| T5.1.1-XXL | 32.8 | 35.6 | 42.9 | 52.5 | **11 billion params** |
| T5.1.1-XXL + SSM | 35.2 | **42.8** | 51.9 | **61.6** | |

## Transfer Learning to Downstream Tasks

There are several input transformations to handle inputs for different types of tasks. The following image from the paper shows the structures of the models and input transformations used to carry out various tasks.

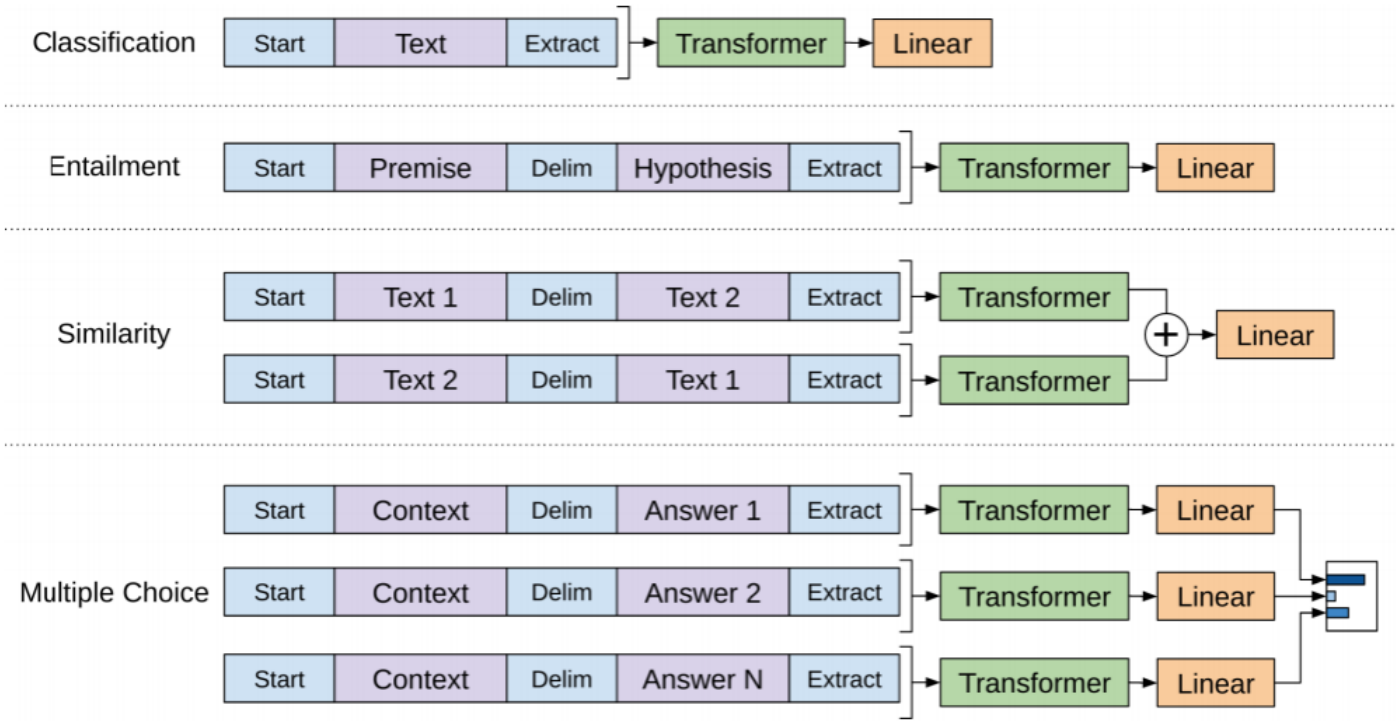



Figure 25: Transfer Learning

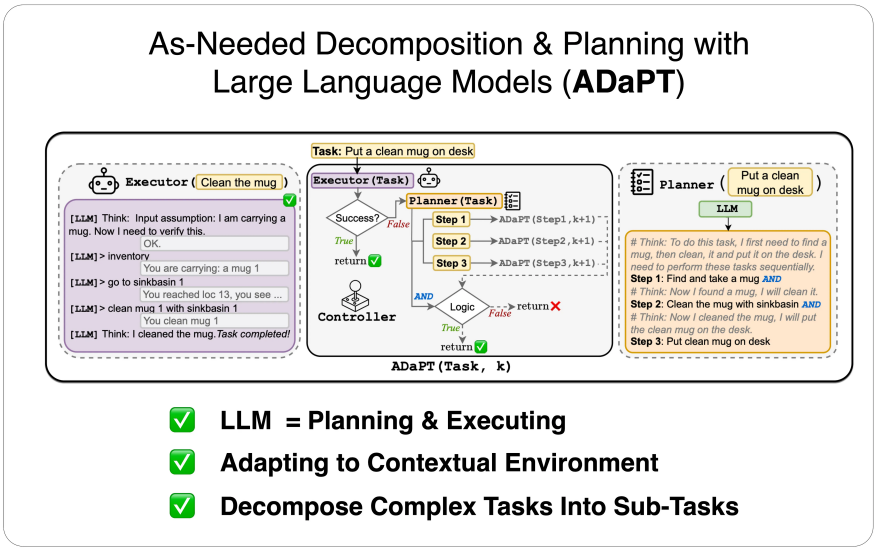
# Key Advanced Techniques in LLMs

## Reasoning Models in LLMs

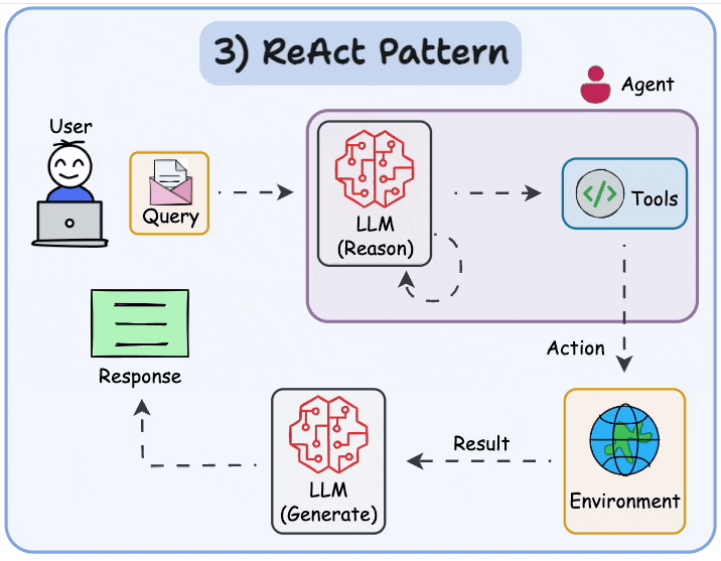
**Reasoning models** are large language models designed or trained to perform multi-step logical inference, rather than producing answers through direct pattern completion alone. Unlike standard next-token predictors, reasoning models explicitly *structure their internal thinking process*, often breaking problems into intermediate steps or plans before generating a final output.

**Key characteristics of reasoning models:**

- **Step-by-step inference:** The model generates intermediate reasoning chains, often guided by techniques such as *Chain-of-Thought (CoT)* prompting.
- **Planning and decomposition:** The model may plan a sequence of steps, solve subproblems individually, and combine the results into a coherent answer.




- **Tool use and external grounding:** In advanced systems, reasoning models can call APIs, query databases, or retrieve documents during their reasoning process (e.g., ReAct framework).



- **Improved interpretability:** The intermediate reasoning traces can be inspected, making the model's decision-making more transparent.

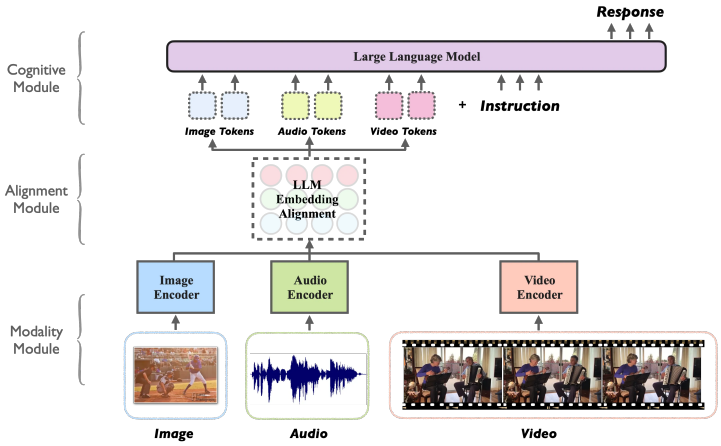
**Example:**

```
Q: If a train travels 60 miles in 1 hour, how far will it go in 4 hours?
Reasoning: In 1 hour, it travels 60 miles. In 4 hours, 60 × 4 = 240 miles.
Answer: 240 miles.
```

Here, the model explicitly reasons about the intermediate calculation rather than memorizing an answer.

**Recent advancements** in reasoning models combine CoT with *self-consistency*, *tree search*, and *planning algorithms*, leading to significantly improved performance in math, coding, and multi-hop question answering.

## Multimodal Models in LLMs




**Multimodal models** are language models that can process and integrate information from multiple input modalities, such as text, images, audio, video, and structured data. Unlike traditional text-only LLMs, multimodal models learn to understand, reason over, and generate content across different types of inputs.

**Key characteristics of multimodal models:**

- **Unified understanding:** They combine signals from multiple sources (e.g., text and image) to form richer and more context-aware representations.
- **Cross-modal reasoning:** The model can reason jointly across modalities, for example answering questions about an image or explaining audio content using text.
- **Flexible input and output:** These models accept and generate different combinations of modalities — for example, image-to-text, text-to-image, or text+image-to-text.
- **Grounded responses:** By linking language with visual or auditory information, multimodal models produce outputs that are more precise, concrete, and aligned with real-world context.

**Example:**

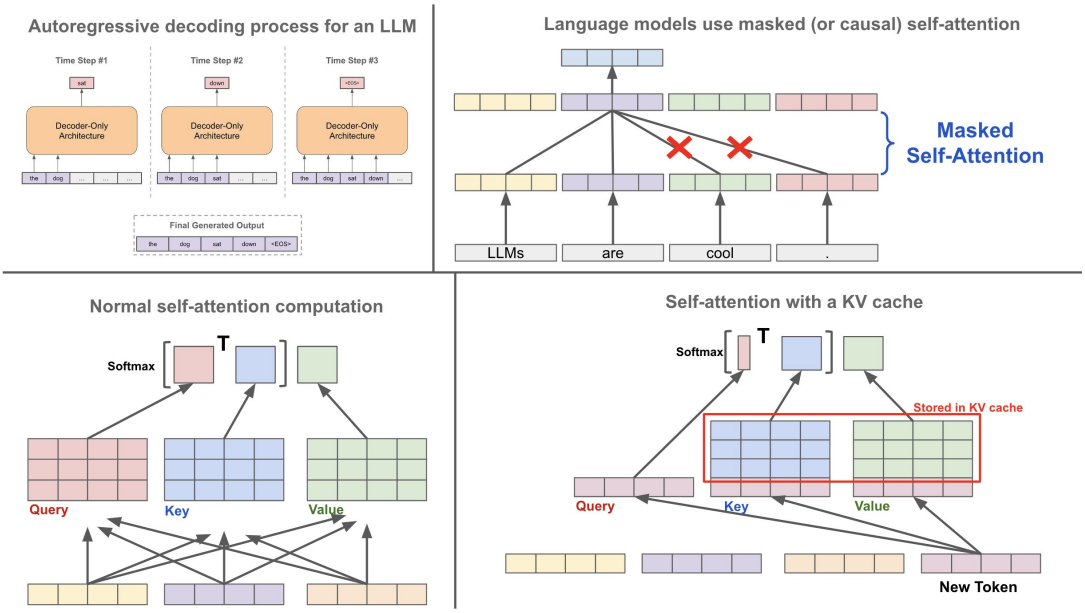
```
Input: [Image of a cat sitting on a sofa] + ‘‘Describe what you see.’’
Model Output: ‘‘A gray cat is sitting comfortably on a blue sofa.’’
```

**Another example:**

```
Input:  ‘‘What object is the person holding?’’  + [Image of a person with a red umbrella]
Model Output:  ‘‘The person is holding a red umbrella.’’
```

**Why multimodality matters:** By combining different information sources, multimodal models move beyond purely linguistic reasoning and gain the ability to perceive and understand the world in a more human-like way. This enables a wide range of applications, from visual question answering and document understanding to interactive assistants that process text, images, and speech together.

## Key–Query Caching in Transformer Models




In Transformer-based language models, each layer computes attention using three components:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

where $X$ is the input, and $W_Q, W_K, W_V$ are learned projection matrices.

During autoregressive generation, tokens are processed sequentially. At every time step $t$, the model computes:

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_t K_{1:t}^\top}{\sqrt{d_k}}\right) V_{1:t}$$

where $d_k$ is the dimension of the key vectors.

**Naive computation (without caching):** For each new token, the model recomputes all previous $K$ and $V$ vectors for every attention layer. This leads to:

$$\text{Time complexity per step} \quad \propto \quad O(t \times d_k)$$

and overall cost grows quadratically with sequence length.

**Key–Value Caching:** To make generation efficient, modern LLMs store the $K$ and $V$ vectors computed in previous steps so they don't need to be recomputed again.

Let:

$$K_{1:t-1}^{\text{cache}}, \; V_{1:t-1}^{\text{cache}}$$

denote the cached keys and values from previous steps. When generating the next token $x_t$, the model computes only:

$$K_t = x_t W_K, \quad V_t = x_t W_V$$

and then concatenates:

$$K_{1:t} = [K_{1:t-1}^{\text{cache}}; K_t], \quad V_{1:t} = [V_{1:t-1}^{\text{cache}}; V_t]$$

Finally, the attention becomes:

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_t K_{1:t}^\top}{\sqrt{d_k}}\right) V_{1:t}$$

but now the cost is only from computing the *new* key and value — not recomputing the entire sequence.

**Practical Example:**

Suppose the sequence so far is:

``The cat sits''

At step $t = 4$, we generate the next token.

1. At $t = 1$: Compute $Q_1, K_1, V_1$ and store $K_1, V_1$.
2. At $t = 2$: Retrieve $K_1, V_1$ from cache, compute $K_2, V_2$, and update cache to $[K_1, K_2]$.
3. At $t = 3$: Repeat for $K_3, V_3$.
4. At $t = 4$: When generating the next word after `"The cat sits"`, compute only $K_4, V_4$ and reuse cached $[K_1, K_2, K_3]$ and $[V_1, V_2, V_3]$.

This allows the model to compute attention for step 4 without recomputing steps 1–3.

**Why this matters:**

- **Efficiency:** Speeds up autoregressive decoding by avoiding redundant computation.
- **Memory–time tradeoff:** Requires extra memory to store $K$ and $V$ but drastically reduces latency.
- **Scalability:** Makes long-context inference feasible in real-time applications.

**Summary Formula:**

$$\boxed{\text{Attention}_t = \text{softmax}\left(\frac{Q_t [K^{\text{cache}}; K_t]^\top}{\sqrt{d_k}}\right) [V^{\text{cache}}; V_t]}$$

Caching turns what would have been a full recomputation at every step into a single incremental update.

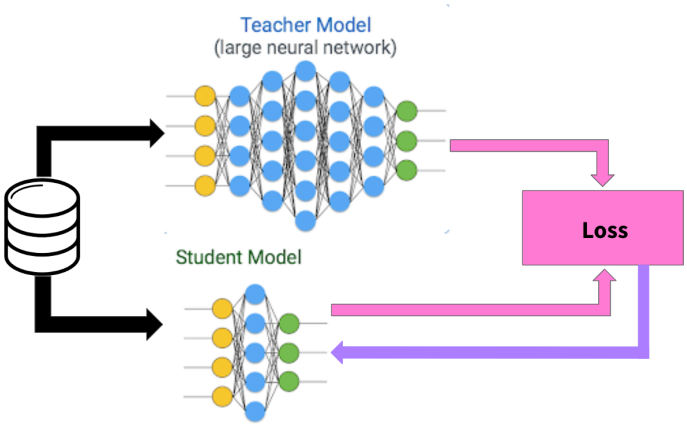
## Efficiency, Scaling, and Distillation / Quantization

Large language models (LLMs) contain billions of parameters and require significant compute for training and inference. To make these models practical for deployment, several efficiency techniques are used:

1. **Distillation** — compressing a large model into a smaller one.
2. **Quantization** — reducing precision of weights and activations.

These techniques allow models to run faster, consume less memory, and be deployed in real-time systems without losing much accuracy.

**1. Knowledge Distillation:**




*Distillation* compresses a large **teacher model** $T$ into a smaller **student model** $S$ by training $S$ to mimic $T$'s outputs. Given logits $z_T$ and $z_S$ from teacher and student respectively, we use a *softmax with temperature* $\tau$:

$$p_T = \text{softmax}\left(\frac{z_T}{\tau}\right), \quad p_S = \text{softmax}\left(\frac{z_S}{\tau}\right)$$

The distillation loss is:

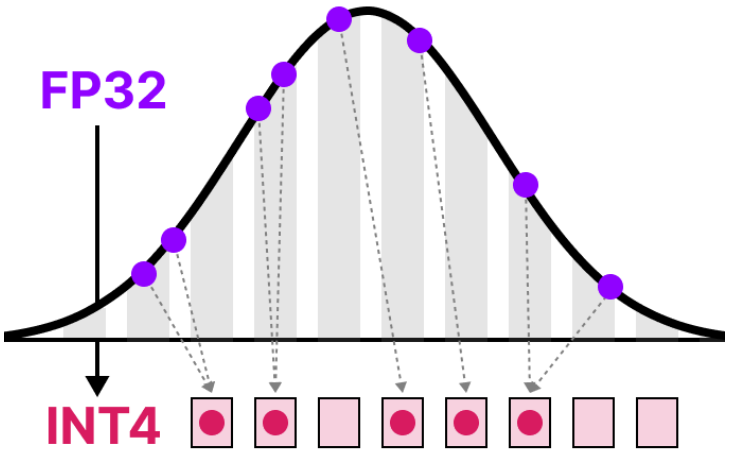
$$\mathcal{L}_{\text{distill}} = -\sum_i p_T^{(i)} \log p_S^{(i)}$$

where $i$ indexes the vocabulary.

- High $\tau$ smooths the probability distribution, revealing "dark knowledge" (e.g., relative probabilities of non-top tokens).
- The student learns not only the correct label but also the teacher's uncertainty structure.

**Example:** If the teacher outputs $\{0.8, 0.1, 0.1\}$ and the student outputs $\{0.7, 0.2, 0.1\}$, the loss will guide the student to get closer to the teacher's distribution.

**2. Quantization:**




Quantization reduces the number of bits used to represent weights and activations. For example, converting 16-bit floating point weights to 4-bit integers.

Let a weight $w$ be quantized to:

$$\hat{w} = \text{round}\left(\frac{w - \min(W)}{\Delta}\right) \times \Delta + \min(W)$$

where:

$$\Delta = \frac{\max(W) - \min(W)}{2^b - 1}$$

and $b$ is the number of bits (e.g., $b = 4$ for 4-bit quantization).

This maps floating-point values into a discrete set of levels, greatly reducing memory footprint:

$$\text{Memory Saving Factor} = \frac{32 \text{ bits}}{b \text{ bits}}$$

For example, 4-bit quantization saves 8× memory compared to 32-bit.

**Effect on inference:**

- Lower precision speeds up matrix multiplication and reduces memory bandwidth.
- Some accuracy may be lost, but careful quantization preserves most of the performance.

## Mixed Precision in LLMs

**Mixed precision** refers to using different numerical precisions (e.g., FP32, FP16, INT8, INT4) for different parts of the model during training or inference. The main idea is to reduce memory footprint and speed up computation *without sacrificing too much accuracy.*

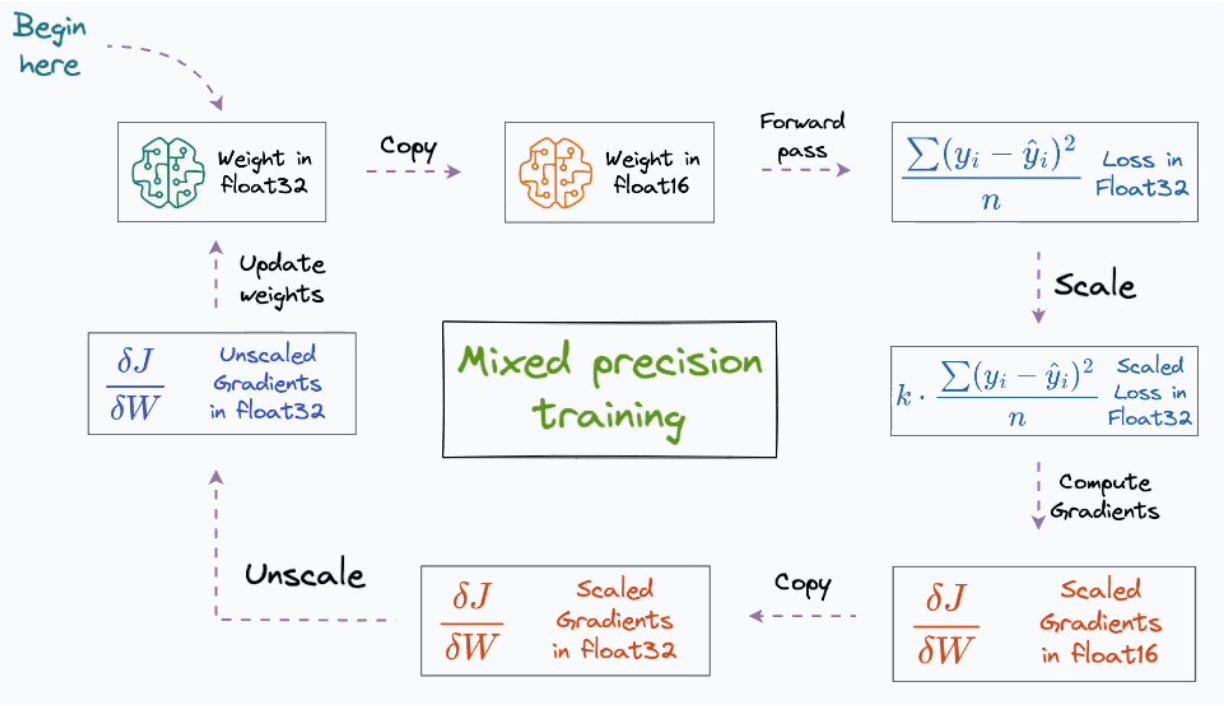



Figure 26: Illustration of mixed precision training with loss scaling and master FP32 weights.

**1. Why not quantize everything?**
While lower precision improves efficiency, not all parts of a neural network are equally tolerant to precision loss. Some tensors are more sensitive than others. Mixed precision selectively applies lower precision to *less sensitive* components and keeps *critical parts* at higher precision.

**2. Common precision assignments:**

- **Weights** ($W$)**:** Often quantized to INT8 or INT4 during inference to save memory.
- **Activations** ($A$)**:** Can also be quantized, typically to INT8 or FP16. Activations are more dynamic, so slightly higher precision is often used.
- **Gradients** ($\nabla W$)**:** During training, these are usually kept in FP16 or FP32 to preserve numerical stability.
- **Optimizer states (e.g., momentum, variance):** Typically stored in FP32 because these accumulate values over many steps and are sensitive to precision errors.
- **LayerNorm, Softmax, and Attention Scores:** These are numerically sensitive operations. They are usually computed in FP32 even if other parts are quantized.

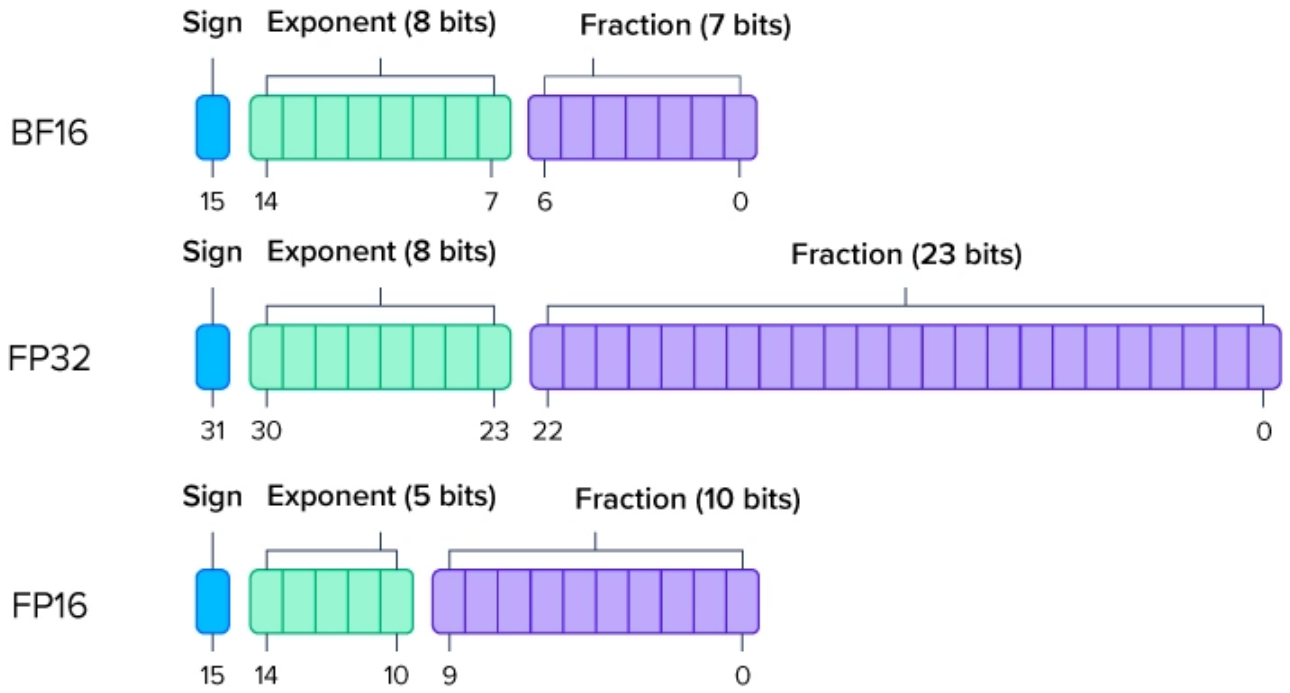
**3. Typical Mixed Precision Forward Pass:**

1. Inputs are processed in FP16 for fast matrix multiplications.
2. Weights are stored in INT8 but are *dequantized* to FP16 or FP32 on the fly:
$$\hat{W} = s \times Q$$
where $Q$ is the quantized weight and $s$ is a scale factor.
3. Attention logits and softmax are computed in FP32 for stability.
4. Outputs may be stored back in lower precision.

**4. Dequantization Step:** Quantized weights are stored compactly but used at higher precision for computation:

$$W_{\text{real}} = s \times W_{\text{int}}$$

where $s$ is typically precomputed per weight group (e.g., per tensor or per channel). This allows models to retain most of the original numerical behavior with far less memory.




**5. Mixed Precision Training Procedure (Step-by-Step):**

1. **FP32 Master Weights:** Maintain a high-precision copy of the model parameters:

$$W_{\text{master}} \in \mathbb{R}^{\text{FP32}}$$

These weights are never directly used in the forward pass; they are updated after each optimization step to preserve numerical stability.

2. **FP16 Weights for Compute:** Cast the FP32 master weights to FP16 for fast computation:

$$W_{\text{fp16}} = \text{cast}(W_{\text{master}}, \text{FP16})$$

3. **Forward Pass:** Compute the forward pass using FP16 weights:

$$\hat{y}_{\text{fp16}} = f(x_{\text{fp16}}; W_{\text{fp16}})$$

Then cast the output to FP32 for stable loss computation:

$$\hat{y}_{\text{fp32}} = \text{cast}(\hat{y}_{\text{fp16}}, \text{FP32})$$

$$L = \ell(y_{\text{fp32}}, \hat{y}_{\text{fp32}})$$

4. **Loss Scaling:** Multiply the loss by a scaling factor $k$ to avoid gradient underflow in FP16:

$$L_{\text{scaled}} = k \times L$$

5. **Backward Pass:** Compute gradients in FP16 using the scaled loss:

$$\nabla W_{\text{fp16}}^{\text{scaled}} = \frac{\partial L_{\text{scaled}}}{\partial W_{\text{fp16}}}$$

6. **Unscale Gradients:** Convert the gradients to FP32 and divide by the scaling factor:

$$\nabla W_{\text{fp32}} = \frac{1}{k} \times \text{cast}\big(\nabla W_{\text{fp16}}^{\text{scaled}}, \text{FP32}\big)$$

7. **Weight Update in FP32:** Update the FP32 master weights with the optimizer (e.g., SGD or Adam):

$$W_{\text{master}} \leftarrow W_{\text{master}} - \eta \, \nabla W_{\text{fp32}}$$

where $\eta$ is the learning rate.

8. **Repeat:** Cast the updated FP32 master weights back to FP16 for the next iteration:

$$W_{\text{fp16}} = \text{cast}(W_{\text{master}}, \text{FP16})$$

**6. Why This Works:**

- Most of the heavy matrix multiplications are done in FP16, which is much faster.
- Sensitive parts (loss, optimizer states, softmax, LayerNorm) remain in FP32 for stability.
- Loss scaling prevents small gradients from vanishing due to FP16's smaller dynamic range.
- FP32 master weights prevent precision drift over time.

**7. Typical Configurations:**

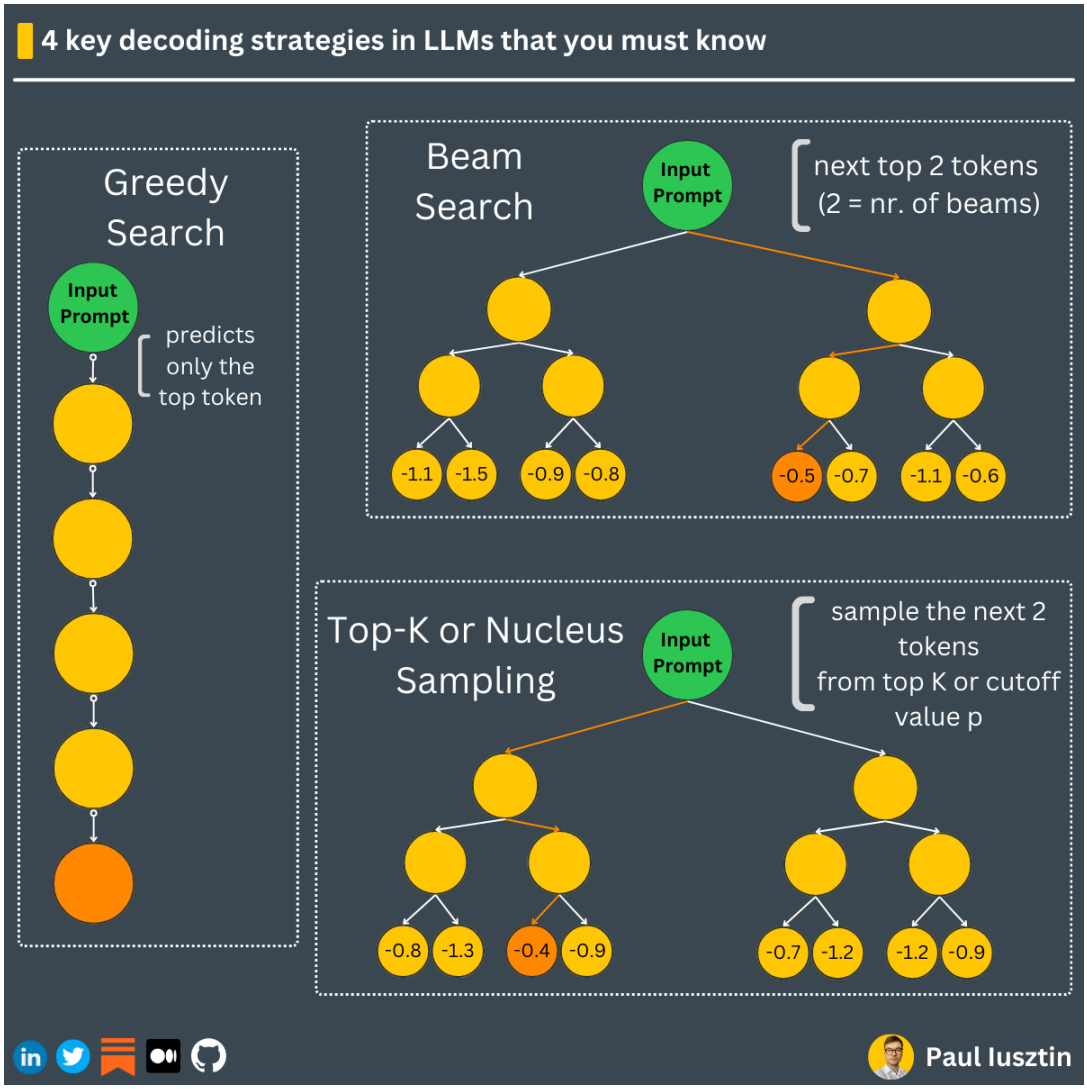
- **FP16 / FP32 mix**: FP16 for most compute, FP32 for critical ops.
- **INT8 weights + FP16 activations**: Common for deployment.
- **INT4 weights + FP16 activations**: More aggressive compression for edge devices.

## Decoding Strategies in Large Language Models

In autoregressive large language models (LLMs), text generation is framed as

$$P(w) = \prod_{t=1}^{T} P(w_t \mid w_{1:t-1}),$$

where at each step $t$, the model outputs a probability distribution over the vocabulary. *Decoding* refers to the strategy used to select the next token $w_t$ from this distribution. Different decoding strategies balance **speed**, **quality**, and **diversity**.




**1. Greedy Search.** At each step, greedy decoding selects the single most likely token:

$$w_t = \arg\max_v P(v \mid w_{1:t-1}).$$

This method is extremely fast and deterministic. However, it is *short-sighted* and can get stuck in locally optimal but globally suboptimal sequences. For example, in language generation, greedy decoding often produces repetitive or generic text.

**2. Beam Search.** Beam search maintains $B$ partial hypotheses ("beams") at each step. For each beam, it expands the top $B$ next-token candidates, forming $B^2$ total candidates, then keeps only the top $B$ based on cumulative log-probability:

$$\text{score}(w_{1:t}) = \sum_{i=1}^{t} \log P(w_i \mid w_{1:i-1}).$$

This procedure repeats until an end-of-sequence token or a maximum length is reached. The final output is the sequence with the highest overall score among beams.

**Example:** For beam width $B = 2$ and sequence length 3, each step expands two candidates, forming a tree of possible sequences. The best-scoring leaf node is chosen at the end.

**3. Top-$k$ Sampling.** Top-$k$ introduces stochasticity. At each step, instead of always picking the highest-probability token, we restrict the distribution to the top $k$ tokens and sample:

$$V_k = \mathrm{TopK}\big(P(\cdot \mid w_{1:t-1}), k\big),$$

$$w_t \sim P(\cdot \mid V_k).$$

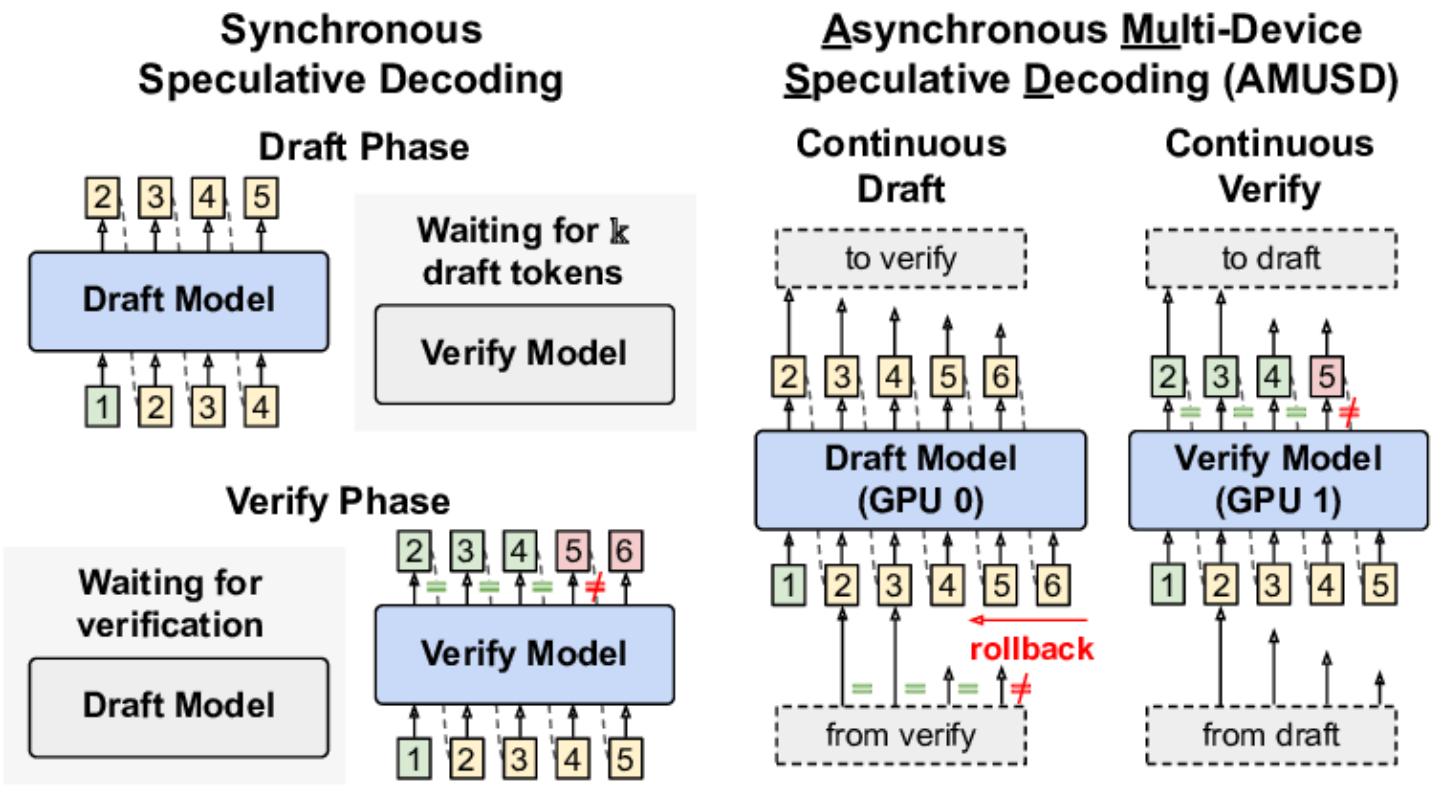
This makes generation more diverse and less deterministic while still avoiding very low-probability tokens.

**4. Nucleus (Top-$p$) Sampling.** Nucleus sampling further refines the idea by selecting the smallest set of tokens $V_p$ whose cumulative probability exceeds a threshold $p$:

$$\sum_{v \in V_p} P(v \mid w_{1:t-1}) \geq p.$$

Then we sample the next token from $V_p$. Unlike Top-$k$, the size of $V_p$ can vary depending on the uncertainty of the model's distribution. This allows the model to adaptively include more or fewer candidate tokens, balancing diversity and coherence.

**5. Speculative / Parallel Decoding.**




Speculative decoding is a recent method to *accelerate* decoding rather than change the distribution.

- Use a small, fast **draft model** $M_d$ to generate $m$ speculative tokens:

$$\tilde{w}_{t+1:t+m} \sim P_d(\cdot \mid w_{1:t}).$$

- Use the **target model** $M_t$ to verify these tokens in parallel:

$$P_t(\tilde{w}_{t+1:t+m} \mid w_{1:t}) = \prod_{i=1}^{m} P_t(\tilde{w}_{t+i} \mid w_{1:t+i-1}).$$

- Compute acceptance ratios
$$r_i = \frac{P_t(\tilde{w}_{t+i} \mid w_{1:t+i-1})}{P_d(\tilde{w}_{t+i} \mid w_{1:t+i-1})}.$$
- Accept tokens where $u_i \sim \mathcal{U}(0,1)$ satisfies $u_i \leq r_i$. Revert to the target model at the first rejection.

**Example:** If the draft model proposes $\tilde{w}_{t+1:t+4} = $ `[the, cat, is, sleeping]` and the target model accepts the first three tokens, decoding resumes from $t+4$. *Three forward steps are replaced with one*, reducing latency.

**Why speculative decoding is powerful:**

- Reduces the number of expensive model forward passes.
- Keeps output distribution identical to the original target model.
- Works well with other decoding strategies (e.g., top-$k$, nucleus, beam search).

**Summary Table:**

| **Technique** | **Determinism** | **Diversity** | **Speed** |
|---|---|---|---|
| Greedy Search | High | Low | Fast |
| Beam Search | High | Low–Medium | Moderate |
| Top-$k$ Sampling | Medium | Medium–High | Fast |
| Nucleus Sampling | Medium | High | Fast |
| Speculative Decoding | Same as base | Same as base | Very Fast |

# References


[1] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.

[2] John Hewitt. "Self-Attention & Transformers.".

[3] Alammar, J (2018). The Illustrated Transformer .

[4] DataCamp

[5] promptingguide