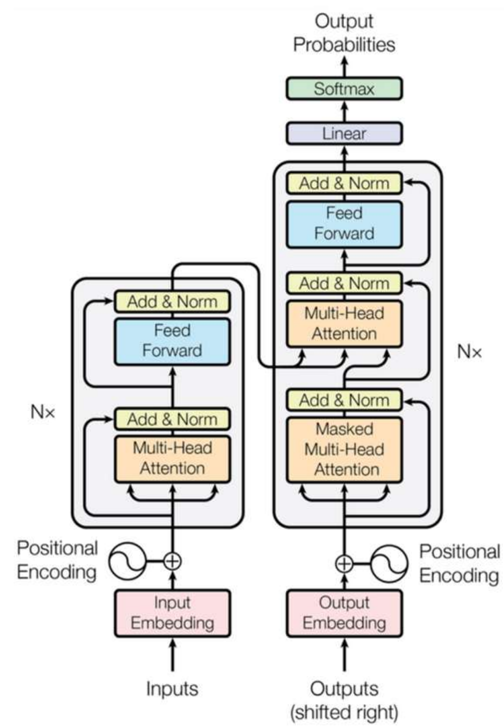
# Introduction to

# (1) AI-Assisted Coding
# (2) Autoregressive Large Language Model (LLM)
# (3) Prompt Engineering 201




Minha Hwang

## AI-ASSITED CODING DEMO

- NOTEBOOK: *AI_ASSITED_CODING.IPYNB*

# LANGUAGE MODEL (LM) - AUTOREGRESSIVE

**LM (language model)**: predict next word given input

- Input: text
- Output: next word prediction


Image source: https//web.Stanford.edu/class/cs224n

# AUTOREGRESSIVE LANGUAGE MODELS: PROBABILITY KERNEL

- A language model is a **probability kernel $\mu$** given a prefix of words: $\mu: X \rightarrow Pr(Y)$
  - Stochastic in nature: **A same prefix $X$** can give a **random output** sampled from a probability distribution $\mu_X$(i.e., generative) → A key reason for factual inaccuracy, inconsistency or hallucination (making stuff up)

- A language model calculates $Pr(s)$ given a sequence of words: $s = (w_1, w_2, \ldots\ldots., w_{T-1}, w_T)$

- An autoregressive language model calculates this **conditional on a previous sequence of words**:

$$Pr(s) = Pr(w_1, w_2, \ldots\ldots., w_{T-1}, w_T)$$
$$= \prod_{t=1}^{T} Pr(w_t | w_1, w_2, \ldots\ldots., w_{t-1})$$

  - **Next-word prediction**: Given a prefix $(w_1, w_2, \ldots\ldots., w_{t-1})$, calculate the probability of the next word $w_t$ (Conceptually same to time series with path dependence)

Source: Prof. Kyunghyun Cho

# AUTOREGRESSIVE LANGUAGE MODELS: SIMPLE EXAMPLE

- 4-word sentence example: "I am a student"

$$Pr(s) = Pr(w_1, w_2, w_3, w_4) = Pr(w_1) \times Pr(w_2|w_1) \times Pr(w_3|w_1, w_2) \times Pr(w_4|w_1, w_2, w_3)$$

- All you need is **"counting"** (if there are large amounts of data)

$$Pr(w_2|w_1) = \frac{count(w_1, w_2)}{count(w_1)} \longrightarrow \text{2-grams (Bigrams)}$$

$$Pr(w_3|w_1, w_2) = \frac{count(w_1, w_2, w_3)}{count(w_1, w_2)} \longrightarrow \text{3-grams (Trigrams)}$$

$$Pr(w_4|w_1, w_2, w_3\ ) = \frac{count(w_1, w_2, w_3, w_4)}{count(w_1, w_2, w_3)} \longrightarrow \text{4-grams}$$

- Problems:
  - This requires **a lot of space (RAM)**
  - Count-based language models **cannot generalize**: A certain sentence **does not appear** in the corpus

Source: Prof. Kyunghyun Cho

# TWO TYPES OF LARGE LANGUAGE MODEL (LLM)

## Base LLM: Pre-Training

predict next word, based on text training data

- Self-supervised

> Once upon a time, there was a unicorn
> that lived in a magical forest with
> all her unicorn friends

> What is the capital of France?
> What is France's largest city?
> What is France's population?
> What is the currency of France?

## Instruction Tuned LLM: Post Training

Tries to follow instructions

Fine-tune on instructions and good attempts at following those instructions (SFT)

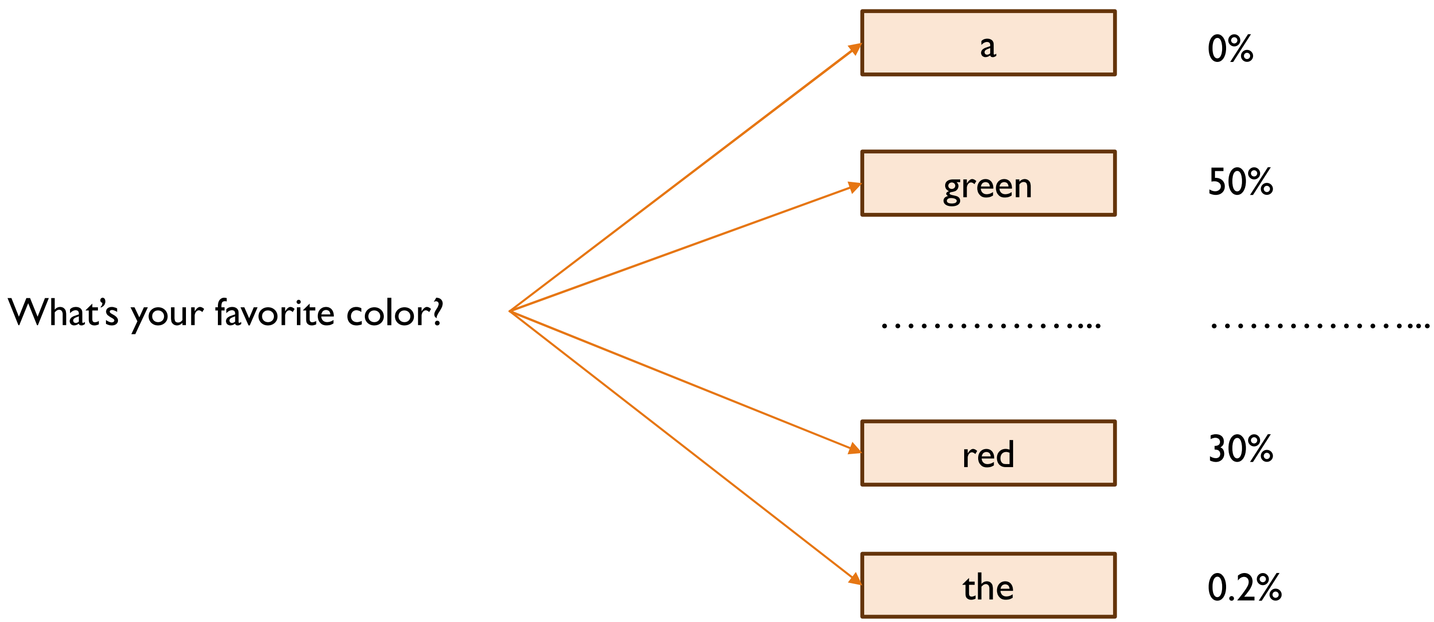
- Human Labeled Data: Instruction – Response Pair
- RLHF (Reinforcement Learning with Human Feedback) or DPO (Direct Preference Optimization)

> What is the capital of France?
> The capital of France is Paris.

Helpful, Honest, Harmless (Style)

Source: DeepLearning.AI – ChatGPT Prompt Engineering

# SAMPLING: KEY TO UNDERSTAND PARAMETERS FOR LLM



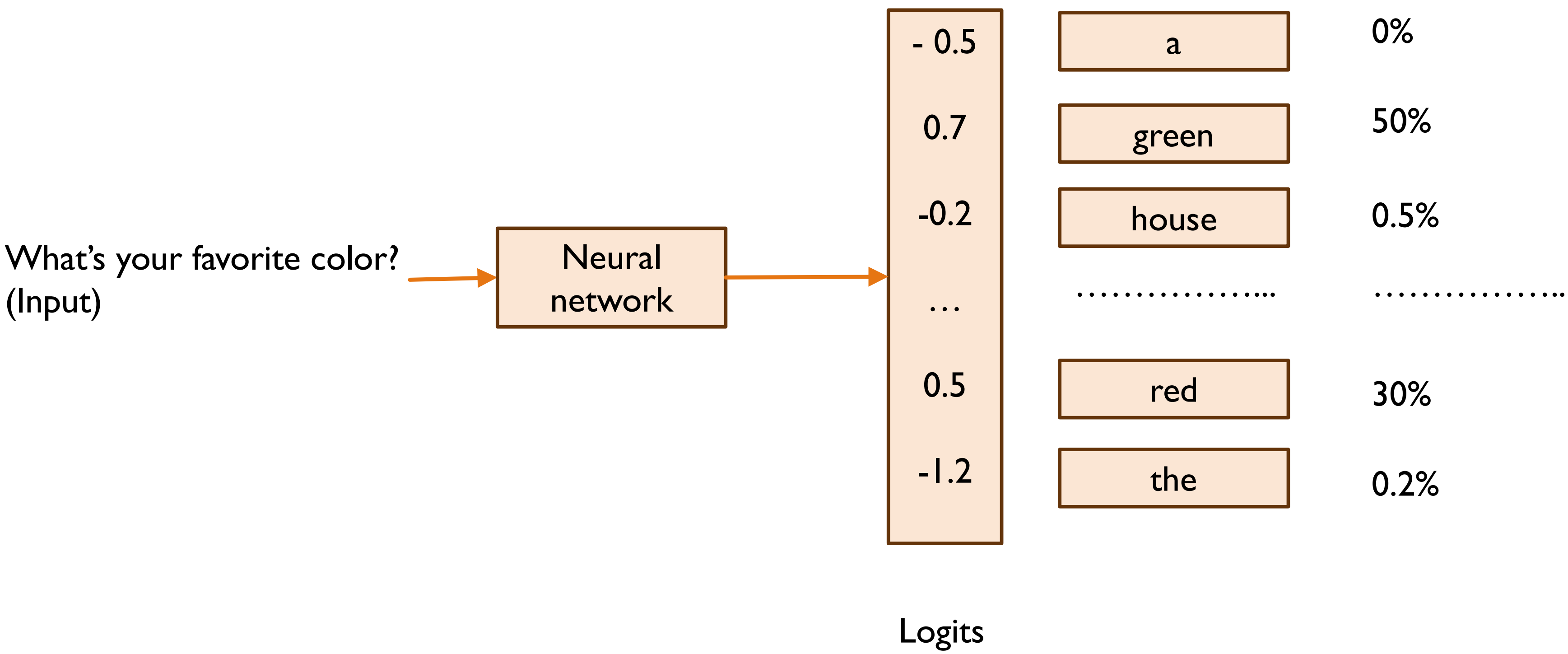

Probability distribution over the vocabulary

**SoftMax:**
Multiple classes

- **Greedy sampling:** Always pick the outcome with highest probability (boring outcome)
- Sample the next token **according to the probability distribution** over all possible values

Source: AI Engineering

# UNDER THE HOOD




Source: AI Engineering

# TEMPERATURE: CONTROLS RANDOMNESS/CREATIVITY (1/2)

- A constat used to adjust the logits vector before the softmax transformation
- Logits are divided by temperature
- Higher temperature reduces the probabilities of common tokens: increase the probabilities of rarer tokens
  - (+) More creative model responses
  - (-) More hallucination, inconsistency

### 1. Adjusting Logits with Temperature

Given a logit vector $z$, temperature $T$ scales the logits as follows:

$$z' = \frac{z}{T}$$

where:

- $z$ is the original logit vector output from the model.
- $T$ is the **temperature** hyperparameter.
- $z'$ is the adjusted logit vector.
- If $T = 1$, logits remain unchanged.
- If $T > 1$, logits are **flattened** (making the probability distribution more uniform).
- If $T < 1$, logits are **sharpened** (making high probabilities even higher and low probabilities even lower).

### 2. Applying Softmax to Adjusted Logits

Once we have the temperature-scaled logits $z'$, we apply the **softmax function**:

$$P_i = \frac{e^{z'_i}}{\sum_j e^{z'_j}}$$

where:

- $P_i$ is the probability of token $i$.
- $z'_i = \frac{z_i}{T}$ is the temperature-scaled logit.
- The denominator ensures that the probabilities sum to 1.

# TEMPERATURE (2/2)

```
{
  "prompt": "Give me a list of 5 unusual ice cream flavors.",
  "max_tokens": 30,
  "temperature": 0.7
}
```

**Example:**

- With temperature = 0.7, you may get some unique, creative flavors like "Lavender Basil" or "Sriracha Peanut."
- If you set temperature = 0.0, the model's answer might be more "plain" or it might return more common "unique" flavors, like "Mint Chocolate Chip."

**What it is:**

- A value between 0 and 2 (commonly 0 to 1) that controls the "creativity" or randomness in the model's outputs.

**Why it matters:**

- **Lower temperature** (e.g., 0–0.3): The output is **more deterministic and focused**, which is helpful for tasks requiring **factual or logical consistency** (e.g., summarization, factual Q&A).
- **Higher temperature** (e.g., 0.7–1): The output becomes **more creative**, with **increased risk of straying off-topic** or introducing **less relevant details**.

# LOGPROBS: LOG PROBABILITIES

**What it is**

- Log Probs: Probability in log scale (helps to reduce underflow problems from small probabilities: e.g., vocabulary size of 100,000)
  - If the probabilities look random, the model hasn't learned much
  - Helpful for debugging
- Log Probs = Log(Probs): Per-token probabilities for analysis
- Log Probs can be used as a level of confidence (i.e., high likely tokens)

**Why it matters**

- Understanding model confidence: You can see the probability distribution over tokens.
- Useful for advanced applications: You might re-rank or filter tokens yourself based on those probabilities.
- Helps in prompt engineering: By looking at tokens with high or low probabilities, you may identify ambiguous phrasing in your prompt that is leading to inconsistent output

# TOP_P (NUCLEUS SAMPLING)

**What it is**

- An alternative (or complementary) way of controlling randomness. top_p is the **cumulative probability threshold**. The model will consider **only the tokens within the "top" probability mass**.

**Why it matters**

- When top_p is smaller (e.g., 0.1), the model focuses on **highly probable tokens**, making responses **more conservative and repetitive**.
- When top_p is larger (close to 1.0), the model considers a **wider distribution of potential tokens**, yielding **more diverse outputs**.

# PENALTIES (1/2)

## Frequency_Penalty

**What it is**

- A value between -2.0 and 2.0 that penalizes or rewards new tokens based on their frequency so far in the text. A higher positive value will make the model *less likely* to repeat tokens it has already used.

**Why it matters**

- Reduces redundancy: If the model tends to get repetitive, increasing frequency_penalty discourages repeated tokens.
- Negative values can encourage repetition if that's desired (rare).

## Presence_Penalty

**What it is**

- Similar to frequency_penalty but slightly different in logic. A higher positive value penalizes tokens that have already appeared in the text, *regardless* of how many times they appear.

**Why it matters**

- Encourages novelty: Even if a word has appeared once, the model is more likely to try something new.
- Helps avoid repeating entire lines or phrases.

# PENALTIES: EXAMPLES (2/2)

## Frequency_Penalty

**Example:**

```json
{
  "prompt": "Write a paragraph describing an autumn day.",
  "max_tokens": 60,
  "frequency_penalty": 1.0
}
```

- The model will try not to overuse the same words like “leaves” or “trees,” leading to more varied vocabulary.

## Presence_Penalty

**Example:**

```json
{
  "prompt": "Give me instructions on how to plant a rose garden.",
  "presence_penalty": 0.5
}
```

- With a `presence_penalty` of 0.5, the model will try to avoid using the same keywords repeatedly, so it might use synonyms or rephrase steps for variety.

# MAX_TOKENS

```
{
  "prompt": "Write a short story about a space explorer who discovers a new planet.",
  "max_tokens": 50
}
```

**Example:**

Here, we tell the model to generate at most 50 tokens. That might result in only 3–5 sentences of output (depending on the tokens). If you need a **longer story**, you'd increase **max_tokens to 200**+

**What it is:**

- This parameter sets the **maximum number of tokens (word pieces)** that the **model can generate in its response**.

**Why it matters:**

- Controlling length: If you have a strict limit on how long the output can be (e.g., a tweet-like format), you can set **max_tokens** to cap the model's response.
- Efficiency: A **smaller max_tokens** will typically be **cheaper and faster**. However, it might **cut off the response mid-sentence** if it's too restrictive.

# STOP (OR STOP_SEQUENCES)

**What it is**

- A string or list of strings the model will stop generating tokens upon encountering. This effectively truncates the output at a certain pattern.

**Why it matters**

- Truncating the response safely: For example, you might want to stop the model when it starts a new line or sees a certain sentinel token.

- Useful for structured outputs: If you're generating JSON or code, you can define a stop sequence that ends the code block.

**Example:**

json Copy

```json
{
  "prompt": "List three advantages of renewable energy:\n1.",
  "stop": ["4."]
}
```

- The model will stop generating once it begins writing "4." (or hits the token boundary for "4."), ensuring only three advantages are listed.

# N AND BEST_OF

## What they are

- **n**: The number of completions to generate for each prompt.
- **best_of**: When used, it generates multiple completions server-side but returns only the "best" one according to some criterion (usually likelihood).

## Why they matter

- **n** can help you sample multiple possible outputs for a creative or brainstorming scenario in a single request.
- **best_of** is useful to get the highest probability completion from a set, though note it uses more compute (and thus may cost more).

**Example:**

json Copy

```json
{
  "prompt": "Suggest a tagline for an eco-friendly reusable water bott
  "n": 3,
  "best_of": 3
}
```

- The API might internally generate 3 completions and return the best one (if you also set `best_of`). If `n=3` without `best_of`, you'll simply get all 3 completions back.

## PROMPT ENGINEERING 201 DEMO

- *NOTEBOOK:  PROMPT_ENGINEERING.IPYNB*

DEMO