UNDERSTAND & CREATE LLAMA 4 FROM THE GROUND UP

By Vuk Rosić

GOOD TO HAVE

- Intermediate Python
- High school math

Even if you don’t know these, I will explain them.
No need for a high-performance machine, we will not train AI in this course, just learn its architecture, math and code it from the ground up.

REALISTIC EXPECTATIONS

- The course contains everything from high-school maths to advanced **PhD math**, which will feel very dense and require months to understand.
- But math for this LLM is the same as for every other LLM, and very similar to any other transformer — learn once, create any LLM.
- You can watch this video, other YouTube videos, and other courses that I make, gradually grasping these concepts.

KEY CONCEPTS INTRODUCED:

POINT → VECTOR → 3D VECTOR → Large Language Models

- **LLMs:** AIs that you can chat with using text messages.
- **More precisely:** LLMs generate text and are typically used in chat interfaces.
- **Famous LLMs:** GPT (ChatGPT), Claude, Gemini, DeepSeek, Qwen, Grok, Llama.

HOW LLMS WORK

An LLM:

- Takes some input text,
- Analyzes it
- Predicts the next word that makes sense.

Example:

- Input: *"The sun is shining, and the sky is"*
- Output: *"blue."*

LLMs don’t just predict a single word—they:

- Have a vocabulary of words, characters, etc.
- Assign a **probability** to each based on the input context.

Example Probabilities:

- "blue." → 60%
- "clear." → 45%
- "cloudy." → 41%
- "apple." → 0.002%
- "banana." → 0.001%

METHODS FOR WORD SELECTION

1. **Greedy Sampling:**
    - Always pick the most probable word.
    - Example: "blue" (60%).

2. **Random Sampling:**
    - Randomly choose based on the probability distribution.
    - Could choose "cloudy" even if it's not the top choice.

3. **Top-k Sampling:**
    - Choose randomly from the top *k* most probable words.
    - Example: Top 3 = "blue," "clear," "cloudy".

4. **Temperature Sampling:**
    - Adjust the “sharpness” of the distribution.
    - Higher temperature = more randomness.
    - Lower temperature = more deterministic (closer to greedy).
    - Higher temperature gives rare words like "banana" more chance.

Tokens

Anything can be added as a token that AI can generate (predict based on preceding text):

- "car" (words token)
- "b" (a letter token)
- "123" (a number token)
- "🌲" (an emoji token)
- "   " (a Chinese character token)
- "The sun rises in the east." (a sentence token)
- "   " (a Chinese sentence token)
- "a1   b2   c3!@#" (an arbitrary token)

When a token is a word like "car", the model learns:

- It's a vehicle
- It has wheels
- People drive it
- It's used for transportation
- ...

When a token is a letter like "b", the model learns:

- It's the second letter in the English alphabet
- It can start words like "book" or "bird"
- Where and how it's used to construct words
- ...

When a token is a number like "123", the model learns:

- It represents a specific quantity
- It follows mathematical rules
- It appears in contexts like counts, measurements, or dates
- This one also looks like a beginning of a counting sequence, so it can be used when user wants model to count from 1 to 10 for example
- ...

When a token is an emoji like "🌲", the model learns:

- It's an emoji (and what is an emoji)
- It represents a tree or forest
- It's also used to symbolize nature or the outdoors
- ...

When a token is a Chinese character like " ", the model learns:

- It's a Chinese character
- It means "car" or "vehicle"
- It can be part of compound words like "   " (automobile) or "   " (train)
- It's used in contexts related to transportation and vehicles
- It has a specific stroke order and writing pattern
- ...

When a token is a sentence like "The sun rises in the east.", the model learns:

- It describes a natural phenomenon that happens daily
- It has a subject (sun), verb (rises), and prepositional phrase (in the east)
- It's a statement of fact that's universally understood
- It can be used literally or metaphorically in different contexts
- ...

When a token is a Chinese sentence like "   ", the model learns:

- It's in Mandarin Chinese langauge
- It means "We are learning artificial intelligence"
- The action is currently hapenning, it's present continuous, due to
- For example if the story is about somebody learning AI, and the question is what they are doing, LLM can output this token.
- ...

When a token is arbitrary like "a1b2$3]'.fDc3!@#", the model learns:

- It contains a mix of letters, numbers, characters from different languages, and symbols
- It doesn't match common language patterns
- It might be a code, password, or random string
- It lacks semantic meaning
- ...

The question is, which way of tokenizing is best:

"The cat is sleeping",

"The", " cat", " is", " sleeping",

"T", "h", "e", " ", "c", "a", "t", " ", "i", "s", " ", "s", "l", "e", "e", "p", "i", "n", "g", Each token requires same amount of compute to process.

Computational Cost

"The quick brown fox jumps over the lazy dog."

- **Sentence tokenizer**: 1 token = 1x compute
- **Word tokenizer**: 9 tokens = 9x compute
- **Letter tokenizer**: 44 tokens = 44x compute

But there are some issues with word and sentence tokens as well.

- Massive vocabulary: Not only every word, but every version of every word: ["run", "runs", "ran", "running" - this requires high compute to calculate probability distribution.
- If you forget to add any word, e.g., "running", to the vocabulary, the AI will not learn what it means.
- Can't handle misspellings: If text contains incorrect spellings like "runned" or "avocdo", humans can infer meaning, but AI would see it as OOV (out of vocabulary) token, replace it with a special token that indicates that this is completely unkonwn, eg. `<UNK>`.
- Most words are rare and will not appear enough times in the training data for model to learn their meaning well.
- Can not construct new words like letters can.

For sentence level tokens these issues are even more exaggerated.

Quick note: there might be breakthroughs in the future that make word or sentence level tokens the best options, research in this field is very fast and accelerating.

Can we get the best of both worlds - low vocabulay size, ability to construct new words (like letter based tokenizer) and lower computation cost by creating token of multiple letters combined?

Introducing: Subword tokens!

Instead of:
[run, running, play, playing, stay, staying]

it's tokenized as:
[run, play, stay, ing]

- Can be used to construct words like [running, playing, staying] - AI will learn that adding "ing" to a base verb will convert it to present continuous
- Requires less computation then processing each letter separately
- Vocabulary size reduced from 10s of millions to just 100s of thousands
- Better at understanding text with spelling mistakes

Qwen2.5 Tokenizer (at the time of making this, tokenizer for Qwen3 has not been released yet, but it will probably be exactly the same or a bit different, which doesn't matter for our understanding of how tokenizer works)

https://huggingface.co/Qwen/Qwen2.5-7B/raw/main/tokenizer.json

Example of what you will see:

```
"sh": 927,
"ual": 928,
"Type": 929,
"son": 930,
"new": 931,
"ern": 932,
"Ġag": 933,
"AR": 934,
"];Ċ": 935,
```

Tokens on the left, paired with a numbers from 0 to 151664 on the right.

- Vocabulary contains 151664 tokens
- Each token is indexed with a number

To understand what we need numbers for, we need to go back and understand how exactly LLMs "understand and store semantic meaning" of a token.

Introducing: Vector Embeddings

(fancy words, but don't get scared, you will understand it)

Vector embedding is just an array of numbers

[0.43, 0.76, 0.13, 0.05]

Each token in the vocabulary will have a corresponding vector embedding (array of numbers).

"car" = `[0.84, 0.01, 0.31, 0.03]`

"grass" = `[0.01, 0.97, 0.89, 0.82]`

In reality, these vectors are thousands of numbers long, and each number captures some characteristic of this token, e.g.: quickness, aliveness, greenness, fluffiness, playfulness, vehicleness, parallel universeness, regular excerciseness, and who knows what else [we don't] - each feature will be a measure of some characteristic, and it will contribute to the meaning.

AI learns by itself what each feature should be about, and the numerical value of it for each token, and we don't know what exactly those numbers represent.

Let's take these 2 tokens as an example.

"dog" = `[0.42, 0.97, 0.95]`

"cat" = `[0.73, 0.04, 0.01]`

Feature (number) 1: Fluffiness Amount

> *How soft, fuzzy, and likely to leave hair on your favorite hoodie?*

- **Dog** → `0.42`

  Dogs come in all fluff levels, from bald weirdos to walking cotton balls.

- **Cat** → `0.73`

  Living fur clouds. Built for maximum softness.

Feature 2: Willingness to Save Your Life

> *If you're in a lake and can't swim... will it save you?*

- **Dog** → `0.97`

  Leaps in with heroic urgency. Risks life. Brings floaty.

- **Cat** → `0.04`

  The one who pushed you into the water.

Feature 3: Gratefulness

> *How much it appreciates everything you do for it?*

- **Dog** → `0.95`
  Thinks you hung the moon. Worships your every move. Applauds your shoe-tying skills.
- **Cat** → `0.01`
  You are but a mere buttler, void of any purpose beyond servitude.

In reality, features at same positions across tokens don't necessarily encode same characteristic, also different tokens can have unique characteristics, or multiple features could work together to encode characteristics.

Summary

These vector embeddings aren't manually created by human scientists — the AI **learns them** by reading the entire internet (trillions of words), trying to predict the next token as it's reading it, and adjusting vector embeddings for each token and other numbers (parameters, that we will talk about later) so its predicted token matches the actual token in the training data.

As it fails to predict the next token during the training, it updates vector embeddings and other parameters, causing the correct token from the training data to be more likely next time, and incorrect tokens less likely.

How LLMs use vector embeddings

"The car is fast, and the car is red."

- "The" → Vector: [0.12, 0.34, 0.56, 0.78]
- " car" → Vector: [0.84, 0.01, 0.31, 0.03]
- " is" → Vector: [0.23, 0.45, 0.67, 0.89]
- " fast" → Vector: [0.91, 0.12, 0.34, 0.56]
- " and" → Vector: [0.78, 0.90, 0.12, 0.34]
- " red." → Vector: [0.56, 0.78, 0.90, 0.12]
- "," (comma) → Vector: [0.55, 0.66, 0.77, 0.88]

Usually tokens have space as the first character (" car") - it's more efficient then encoding space separately.

LLMs replace each token with the vector embedding.

"The car is fast, and the car is red."

[0.12, 0.34, 0.56, 0.78] [0.84, 0.01, 0.31, 0.03] [0.23, 0.45, 0.67, 0.89] [0.91, 0.12, 0.34, 0.56] [0.78, 0.90, 0.12, 0.34] [0.84, 0.01, 0.31, 0.03] [0.23, 0.45, 0.67, 0.89] [0.56, 0.78, 0.90, 0.12] [0.55, 0.66, 0.77, 0.88] [0.78, 0.90, 0.12, 0.34] [0.56, 0.78, 0.90, 0.12]

But we don't want to tie tokens to vectors, because we want to use same token vocabulary for different versions of the model or models, which will have different vector embeddings for the same token.

So we will also have an array of vector embeddings (array of arrays, matrix), where each vector embedding is placed at the same position corresponding to the index of the token.

```
[
[0.12, 0.34, 0.56, 0.78]
[0.84, 0.01, 0.31, 0.03]
[0.23, 0.45, 0.67, 0.89]
[0.91, 0.12, 0.34, 0.56]
[0.78, 0.90, 0.12, 0.34]
[0.56, 0.78, 0.90, 0.12]
[0.55, 0.66, 0.77, 0.88]
]
```

We will place a vector embedding corresponding to the token with index 0 at position 0.

Later we will pluck out embedding corresponding to the token's index from the embedding matrix.

Mapping to Indices

- "The" : 0
- "car" : 1
- "is" : 2
- "fast," : 3
- "and" : 4
- "red." : 5

"The car is fast, and the car is red."

0 1 2 3 4 0 1 2 5

For each index (0 1 2 3 4 0 1 2 5) we will pluck out corresponding vector embedding (they are arranged to correspond to tokens with the same index.)

[0.12, 0.34, 0.56, 0.78] [0.84, 0.01, 0.31, 0.03] [0.23, 0.45, 0.67, 0.89] [0.91, 0.12, 0.34, 0.56] [0.78, 0.90, 0.12, 0.34] [0.84, 0.01, 0.31, 0.03] [0.23, 0.45, 0.67, 0.89] [0.56, 0.78, 0.90, 0.12] [0.55, 0.66, 0.77, 0.88] [0.78, 0.90, 0.12, 0.34] [0.56, 0.78, 0.90, 0.12]

This allows us to easily swap vector embeddings from different models / versions that use the same tokenizer.

Next lesson: **Coding our own tokenizer** in **lesson_2_coding_tokenizer.py** file