Language Models
How Machines Understand Text

From Next Token Prediction to GPT

Nipun Batra | IIT Gandhinagar

The Story So Far

Week	Domain	Key Insight
6	Vision	Images = grids of pixels → CNNs
7	Language	Text = sequences of tokens → ?

A Shocking Revelation

ChatGPT, Claude, Gemini, LLaMA...

These AI systems that can:

- Write essays and code
- Answer complex questions
- Translate languages
- Have conversations

Are all playing ONE simple game:

> Guess the next word. Repeat.

Wait, That's It?

Yes. The entire field of Large Language Models is built on:

"Given some text, predict what word comes next."

```
"The capital of France is ___"  →  "Paris"
"To be or not to ___"          →  "be"
"print('Hello ___"             →  "World')"
```

But How Does Prediction = Intelligence?

If you're **really good** at predicting what comes next...

You need to "know"	To predict
Geography	"The capital of France is **Paris**"
Shakespeare	"To be or not to **be**"
Python syntax	"print('Hello **World')**"
Physics	"F = m**a**"
Reasoning	"2 + 2 = **4**"

Good prediction requires implicit understanding.

Today's Agenda

1. **The Core Idea** - Next token prediction
2. **Building a Character-Level LM** - From scratch
3. **The Counting Era** - Bigrams and N-grams
4. **Word Embeddings** - Words as vectors
5. **The Attention Revolution** - Transformers (intuition)
6. **Temperature & Sampling** - Controlling generation
7. **Modern LLMs** - Scale is all you need

Part 1: The Core Idea

It's All About Prediction

The One Question

Every language model answers **one simple question**:

> **"Given what I have seen so far, what word comes next?"**

Example: "The capital of France is ___" → **"Paris"**

That's it. **Predict the next word. Repeat until done.**

You Already Use This!

Application	You type...	Suggestion
Phone Keyboard	"I'm running ___"	`late`
Google Search	"how to make ___"	`money`, `pancakes`
Gmail	"Thanks for the ___"	`quick response!`

> All of these are next-word prediction models!

The Mathematical View

"The capital of France is ___" → Probability distribution:

Word	P(word | context)
Paris	0.85
the	0.02
London	0.01
beautiful	0.01
...	0.11

All probabilities sum to 1.0

ChatGPT: Just Prediction!

Prompt	Prediction	Appears to Know
"F = m"	"a"	Physics
"To be or not to"	"be"	Shakespeare
"E = mc"	"²"	Einstein
"print('Hello"	"')"	Python

If you predict well enough, you **appear** to understand everything.

The Generation Algorithm

```python
def generate_text(prompt, model):
    tokens = tokenize(prompt)

    while not done:
        # Step 1: Predict probabilities for ALL possible next tokens
        probs = model(tokens)

        # Step 2: Sample one token
        next_token = sample(probs)

        # Step 3: Add to sequence and repeat
        tokens.append(next_token)

    return tokens
```

That's ALL ChatGPT does!

Part 2: The Counting Era

Bigrams: The Simplest Model

The Simplest Language Model

Idea: Count what letter usually follows each letter.

Training: `aabid`, `priya`, `zeel`, `nipun`

After	Saw	Probability
`a`	`a` once, `b` once	P(a
`z`	`e` once	P(e

This is a **Bigram** model (pairs of 2 characters).

Generating with Bigrams

```
Step 1: Start with "." (beginning token)
        Sample from row "." → Got 'a'

Step 2: Current = 'a'
        Sample from row "a" → Got 'b'

Step 3: Current = 'b'
        Sample from row "b" → Got 'i'

Step 4: Current = 'i'
        Sample from row "i" → Got 'd'

Step 5: Current = 'd'
        Sample from row "d" → Got "." (DONE!)

Result: "abid" ← Looks like a real name!
```

Why Bigrams Fail

Sentence: "Alice picked up the golden key. She walked to the door and tried to open it with the ___"

Model	What It Sees
Bigram	Only "the" (previous word)
Human	"golden key" (from earlier)

Bigrams have NO MEMORY of earlier context!

Let's Train a Character-Level LM!

Dataset: Shakespeare's complete works (~1MB of text)

```python
# Download Shakespeare
import urllib.request
url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"
text = urllib.request.urlopen(url).read().decode('utf-8')

print(len(text))  # 1,115,394 characters
print(text[:200])
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?
```

Shakespeare Character Statistics

```python
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(f"Vocabulary: {vocab_size} characters")
print(chars)
```

```
Vocabulary: 65 characters
['\n', ' ', '!', '$', '&', "'", ',', '-', '.', '3', ':', ';', '?',
 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Only 65 unique characters! Much simpler than 50K words.

Character to Number Mapping

```python
# Create mappings
stoi = {ch: i for i, ch in enumerate(chars)}  # string to int
itos = {i: ch for i, ch in enumerate(chars)}  # int to string

# Encode text
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])

# Example
print(encode("hello"))  # [46, 43, 50, 50, 53]
print(decode([46, 43, 50, 50, 53]))  # "hello"
```

Training Data: Predict Next Character

Input: "First Citize"

Target: "irst Citizen"

```python
# Create training pairs
block_size = 8  # Context length

for i in range(len(text) - block_size):
    context = text[i : i + block_size]
    target = text[i + block_size]
    print(f"'{context}' → '{target}'")
```

```
'First Ci' → 't'
'irst Cit' → 'i'
'rst Citi' → 'z'
'st Citiz' → 'e'
```

Simple Character-Level LM in PyTorch

```python
import torch
import torch.nn as nn

class CharLM(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, batch_first=True)
        self.output = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        x = self.embed(x)          # [batch, seq, embed]
        out, _ = self.rnn(x)       # [batch, seq, hidden]
        logits = self.output(out)  # [batch, seq, vocab]
        return logits
```

Generated Shakespeare (After Training)

Untrained model: Random garbage

```
xZk$
;3q!Ybz:FwM'hUiP-Rn
```

After 1000 steps:

```
HARKE:
The soun the of the have bea the me
```

After 10000 steps:

```
ROMEO:
What light through yonder window breaks?
It is the east, and Juliet is the sun!
```

> Same model, just more training = better predictions!

N-grams: More Context

Model	Context	Limitation
Bigram	1 word	Too little
Trigram	2 words	Still limited
4-gram	3 words	Better
10-gram	9 words	Storage explodes!

Problem: With vocabulary 50K, 10-gram needs $50K^{10}$ entries!

Part 3: Word Embeddings

Words as Vectors

The Representation Problem

How do we represent words for a neural network?

Bad idea: One-hot encoding

```
"cat" → [1, 0, 0, 0, ..., 0]  (50,000 zeros!)
"dog" → [0, 1, 0, 0, ..., 0]

cat · dog = 0  (orthogonal = unrelated!)
```

But cats and dogs ARE related!

Word Embeddings

Idea: Learn a dense vector for each word!

```
"cat" → [0.2, -0.5, 0.8, 0.1, ...]  (maybe 300 dims)
"dog" → [0.3, -0.4, 0.7, 0.2, ...]

cat · dog = 0.9  (similar!)
```

Embeddings Capture Meaning

Famous example from Word2Vec (2013):

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

The vector arithmetic works because embeddings capture semantic relationships!

Embedding in PyTorch

```python
import torch.nn as nn

# Create embedding layer
vocab_size = 50000
embed_dim = 256

embedding = nn.Embedding(vocab_size, embed_dim)

# Get vector for word index 42
word_idx = torch.tensor([42])
vector = embedding(word_idx)  # shape: [1, 256]
```

Part 4: The Memory Problem

RNNs: Passing the Baton

Fixed Windows Aren't Enough

Story: "Alice picked up the golden key. She walked to the door..."

Model Type	Sees	Missing
Fixed window (3 words)	"to the door"	"key"
We need	Everything	-

RNNs: The Relay Race

Idea: Pass information forward like a baton.

```
 "The"      "cat"      "sat"      "on"       "the"
   ↓          ↓          ↓          ↓          ↓
  [h₀] →     [h₁]  →    [h₂]  →   [h₃]  →    [h₄]
           (pass)     (pass)    (pass)     (pass)
```

Hidden state `h` carries "memory" of previous words.

RNN: The Telephone Game Problem

Sequence Length	Memory Quality
10 words	Clear
50 words	Fuzzy
100+ words	Lost!

RNNs suffer from "vanishing gradients" — they forget old information!

LSTM and GRU help but don't fully solve this.

Part 5: The Attention Revolution

"Just Look Back!"

The Brilliant Idea (2017)

What if, instead of compressing everything...
We could just **look back** at everything directly?

Approach	Sees	Limitation
Fixed window	Last few words	Very limited
RNN	Blurry summary	Degrades over time
ATTENTION	Any word directly!	None!

Attention: The Library Analogy

You're at a library with a question:

Step	Action
1. Query	"What opens doors?"
2. Scan Keys	"key" (relevant!), "door" (related), "Alice" (not relevant)
3. Read Values	Mostly from "key"!

$$\text{Attention} = \text{softmax}(QK^T/\sqrt{d}) \cdot V$$

Why Attention is Powerful

Text: "The animal didn't cross the street because **it** was too tired."

What does "it" refer to?

Word	Attention Score
animal	**0.75**
street	0.15
other	0.10

The model **learns** to connect "it" to "animal"!

Self-Attention

Every word attends to **every other word**:

```
"The cat sat on the mat"

"cat" attends to: "The"(0.1), "cat"(0.2), "sat"(0.4), "on"(0.1)...
"mat" attends to: "on"(0.2), "the"(0.3), "cat"(0.3)...
```

All in parallel — no sequential bottleneck!

The Transformer (2017)

"Attention Is All You Need"

```
┌──────────────────┐
│  Self-Attention  │  ← Every word sees every word
├──────────────────┤
│  Feed-Forward    │  ← Process information
├──────────────────┤
│  (Repeat 96x)    │  ← Stack many layers
└──────────────────┘
```

GPT-4 has ~120 transformer layers!

Part 6: Modern LLMs

From GPT to ChatGPT

Scaling Up

Feature	Toy Model	GPT-4
Vocabulary	27 (letters)	100,000 (tokens)
Embedding size	2 dims	12,288 dims
Layers	1	~120
Parameters	~1,000	175+ BILLION
Training data	1,000 names	500B+ tokens
Context	3 chars	128K tokens

Same algorithm. Just MUCH bigger.

Tokenization: Not Words, Not Characters

Approach	Example	Problem
Characters	"hello" → 5 tokens	Too slow
Words	"unhappiness" = 1 token	Millions needed
Subwords	"un" + "happiness"	Best of both!

LLMs use ~50K-100K tokens (subwords).

Tokenization Examples

Text	Tokens
"Hello world"	["Hello", " world"]
"ChatGPT"	["Chat", "G", "PT"]
"unhappiness"	["un", "happiness"]

"How many r's in strawberry?" fails because the model sees ["str", "aw", "berry"]!

Training an LLM

```
1. COLLECT DATA
   - Web crawl (trillions of tokens)
   - Books, Wikipedia, code

2. PRE-TRAINING
   - Objective: Predict next token
   - Massive compute (thousands of GPUs)

3. FINE-TUNING
   - Instruction following
   - RLHF (Reinforcement Learning from Human Feedback)
```

RLHF: Making ChatGPT Helpful

Pre-trained model: Great at next-word prediction

Problem: Doesn't follow instructions well

Solution: RLHF

1. Humans rank model responses
2. Train reward model on rankings
3. Fine-tune LLM to maximize reward

This is what makes ChatGPT **conversational**!

Temperature: The Creativity Knob

When sampling the next token, we apply temperature `T`:

$$P_{\text{adjusted}}(w) = \frac{\exp(\text{logit}_w / T)}{\sum_i \exp(\text{logit}_i / T)}$$

Temperature	Effect	Best For
T = 0	Always pick highest prob	Facts, code, math
T = 0.7	Some randomness	Conversation
T = 1.0	Original distribution	Creative writing
T > 1.0	More random	Brainstorming

Temperature Visualization

```
Prompt: "The cat sat on the ___"

Original probabilities (T=1.0):
  mat: ████████████████ 40%
  bed: ████████ 20%
  floor: ██████ 15%
  sofa: ████ 10%
  other: ██████ 15%

Low temperature (T=0.3):
  mat: █████████████████████████ 85%
  bed: ██ 8%
  floor: █ 4%
  sofa: ░ 2%
  other: ░ 1%

High temperature (T=2.0):
  mat: ██████ 25%
  bed: █████ 22%
  floor: ████ 18%
  sofa: ████ 17%
  other: ████ 18%
```

Sampling Strategies

Strategy	How It Works	Effect
Greedy	Always pick max prob	Deterministic, boring
Temperature	Scale logits by 1/T	Control randomness
Top-k	Only sample from top k	Avoid rare tokens
Top-p (nucleus)	Sample from smallest set with prob ≥ p	Dynamic cutoff

```
# Using HuggingFace transformers
output = model.generate(
    input_ids,
    temperature=0.7,
    top_k=50,
    top_p=0.95,
    do_sample=True
)
```

Why Sampling Matters

Prompt: "Write a poem about the ocean"

Greedy (T=0):

```
The ocean is blue.
The ocean is deep.
The ocean is big.
```

With sampling (T=0.8):

```
Azure whispers dance on moonlit waves,
Where ancient secrets swim in salty caves,
The tide embraces shores with gentle might,
As starfish dream beneath the fading light.
```

> Same model, same prompt — temperature changes everything!

Emergent Abilities

As models get bigger, **new abilities emerge**:

Size	Capabilities
Small (100M)	Grammar, simple completion
Medium (1B)	Factual Q&A, basic reasoning
Large (100B+)	Complex reasoning, code, creativity

All from the same objective: **predict the next token!**

Key Takeaways

1. **LLMs predict the next token** — that's it!
2. **Embeddings** represent words as vectors
3. **Attention** lets models look at ALL context
4. **Transformers** stack attention + feed-forward layers
5. **Scale matters** — same algorithm, more parameters
6. **Temperature** controls creativity vs. accuracy

The Big Picture: What Makes ChatGPT "Chat"?

We now have a model that predicts text well. But...

What Base Model Does	What We Want
"The capital of France is" → "Paris"	Great!
"What is 2+2?" → "? I don't know..."	Bad!
"Help me write code" → Random code	Not helpful!

> Base models are great at completing text, but terrible at following instructions!

Next week: How do we fix this?

Preview: The LLM Training Pipeline

```
STEP 1: Pre-training           (This week!)
─────────────────────
• Predict next token on internet text
• Learns grammar, facts, reasoning
• Result: "Base model" (text completer)

STEP 2: Supervised Fine-Tuning (SFT)    (Next week!)
─────────────────────────────────────
• Train on (instruction, response) pairs
• Learns to follow instructions
• Result: "Instruction model"

STEP 3: Alignment (RLHF/DPO)            (Next week!)
─────────────────────────────
• Human feedback on what's "good"
• Learns to be helpful, harmless, honest
• Result: "AI Assistant" (ChatGPT, Claude)
```

The Journey of a Language Model

Stage	Data	Output
Base model	Trillions of web tokens	Text completion
+ SFT	~100K instruction pairs	Follows instructions
+ RLHF	Human preferences	Helpful assistant

> Same architecture, different training = very different behavior!

Next week: We'll see how SFT and RLHF transform a text predictor into ChatGPT.

You Now Understand LLMs!

Next: From Language Model to Assistant

What we learned:

- Next token prediction is the core idea
- Embeddings, attention, transformers
- Temperature controls generation

Next week:

- How to make models follow instructions (SFT)
- How to align models with human values (RLHF)
- The full ChatGPT training pipeline

Lab: Build a character-level LM, experiment with generation