From images to language: how neural networks learn to predict

Day 3: CNNs, language models, training, attention, and Transformers

Day 3 of 4

Fatih Kansoy
Worcester College, University of Oxford

We will follow information from input to prediction

Part I	**CNNs**	finish the path from response maps to image classes
Part II	**Language models**	follow a host-built request to next-token probabilities
Part III	**Training and generation**	see where targets come from and how text is produced
Part IV	**Attention**	let each token gather the earlier clues it needs
Part V	**Transformers**	repeat and refine those context updates

Finish: separate model prediction from evidence and enforceable control.

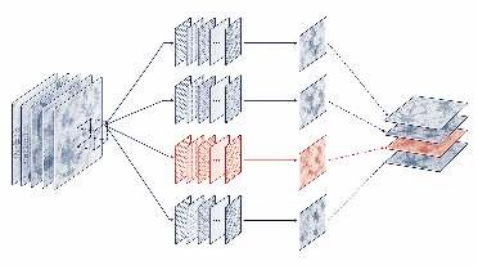
One running question: why does `out` become likely after `Leila ... took it`—and what else is needed before that prediction becomes a trustworthy assistant response?

Part I

CNNs turn feature maps into a class decision

How does a stack of local response maps become one class prediction?

Recap: one filter makes one map; many filters make the stack



1. Look locally
test one small image patch at a time

2. Reuse one filter
repeat the same visual test everywhere to make one response map

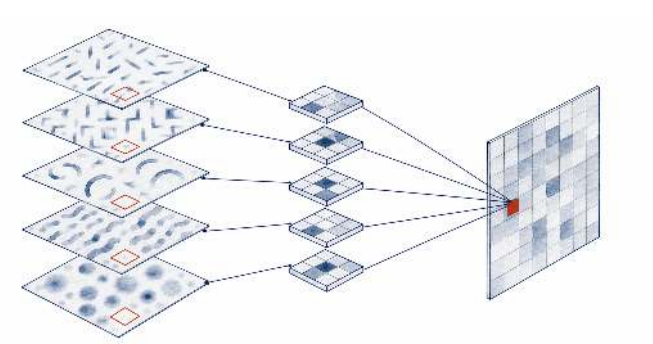
3. Learn many filters
different visual tests contribute different maps to the stack

Local connection = read a small patch. Shared weights = reuse the same filter everywhere.

Today's starting point: the response-map stack is already available; the remaining job is to turn it into 37 breed scores.

▸ One local calculation ▸ Shapes and parameters

Deeper layers combine simple clues into more meaningful ones



Earlier layers: simple clues

separate maps respond to edges, curves, and textures

Next layer: combine nearby clues

a new filter checks whether several clues meet in the right arrangement

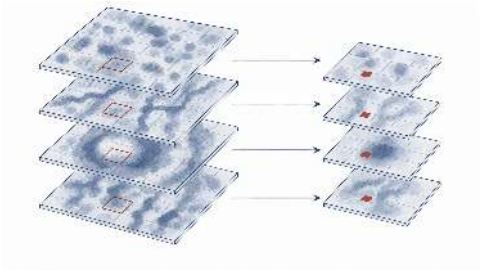
Result: richer evidence

the combination can represent an object part and draw on a wider image area

Receptive field: the part of the original image that can influence one response. It grows as local layers are stacked.

▸ Why the field gets wider

Pooling keeps useful evidence while relaxing its exact position



The evidence moves with the object

the same visual clue may appear in a nearby feature-map cell

Pooling keeps the local signal

keep the strongest nearby response and replace several positions with one

Max pooling: each 2×2 block becomes its largest value. Channels stay separate, and pooling learns no weights.

▸ See one 2×2 pooling calculation

One average per map removes the remaining spatial grid

$$\underbrace{\text{one final response map}}_{\text{where a feature appeared}} \xrightarrow{\text{average every position}} \underbrace{\text{one summary}}_{\text{how strongly it appeared}} .$$

One map $\longrightarrow$ one number

average the whole map instead of keeping every final position

Sixty-four maps $\longrightarrow$ 64 summaries

the image is now represented by a short list of feature strengths

Global average pooling: average positions within each map. Maps remain separate, and no weights are learned.

▸ Calculation and flatten comparison

Each breed combines the 64 summaries in its own way

64 summaries $\longrightarrow$ 37 breed scores $\xrightarrow{\text{softmax}}$ 37 probabilities.

Each breed score uses its own learned mixture of the same 64 summaries.

One breed: combine all the evidence

positive weights add supporting clues; negative weights subtract conflicting clues

All breeds: use different mixtures

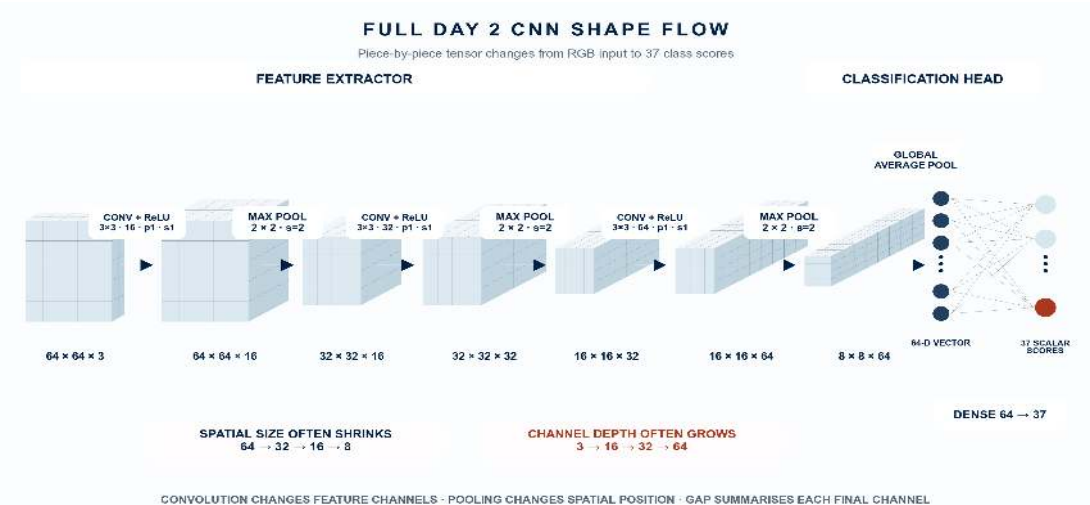
repeat with a different set of learned weights to produce one score per breed

Score, or logit: a breed's weighted evidence before softmax. No single feature map belongs to one breed.

Once the 37 logits exist, the Day 1 learning loop takes over: softmax, loss, backpropagation, and parameter updates.

▸ Calculate one breed score

▸ Recall softmax with three scores



local filters build maps · pooling shrinks the grid · GAP summarises maps · the dense head produces scores

A CNN scores a fixed list of breeds once. A language model will score a much larger list of possible next text pieces repeatedly.

▸ Exact shapes and parameters

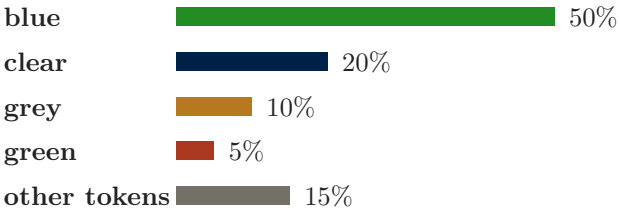
Part II

Language models: what an LLM receives and predicts

How does one message become the model's next-token probabilities?

One output position begins as a probability menu

CONTEXT c: `The sky is`



Illustrative probabilities. "Other tokens" is 15% combined, not 15% each. The full vocabulary distribution sums to 100%.

- w_1 names the **first new-token position**; it does not yet mean "blue".
- Every vocabulary token is a possible value of w_1. If c already ended in `blue`, then w_1 would be the token after `blue`—not `blue` itself.
- A decoding rule selects one value. For the next slides, suppose it selects $w_1 =$ `blue`.

Before selection, w_1 has many alternatives. After selection, $w_1 =$ `blue`.

Each selected token creates the next prediction step

TEXT ALREADY GIVEN TO THE MODEL

`The sky is ...`

This starting text is the **context**, written c.

`blue`	`today`	`.`
w_1	w_2	w_3
first new token	second new token	third new token

1. The first distribution offered many values for w_1; suppose the decoder selected $w_1 =$ `blue`.
2. The text becomes `The sky is blue`. The model creates a new distribution for w_2; suppose the decoder selects $w_2 =$ `today`.
3. The model creates another distribution for w_3; suppose the decoder selects $w_3 =$ `.` A full stop is another token, so generation may still continue.

Positions: w_1, w_2, w_3 **selected values:** `blue`, `today`, `.`

One generated path combines the probabilities from its steps

CHOSEN VALUES

$$c = \texttt{The sky is}, \qquad w_1 = \texttt{blue}, \qquad w_2 = \texttt{today}, \qquad w_3 = \texttt{.}$$

$$\underbrace{p_\theta(w_1, w_2, w_3 \mid c)}_{\text{three-token continuation}} = \underbrace{p_\theta(w_1 \mid c)}_{\text{first token}} \times \underbrace{p_\theta(w_2 \mid c, w_1)}_{\text{second token}} \times \underbrace{p_\theta(w_3 \mid c, w_1, w_2)}_{\text{third token}}$$

Left side: one chosen path

After the values are fixed, $p_\theta(w_1, w_2, w_3 \mid c)$ is the probability assigned to this exact three-token path. It is not the question used to choose w_1.

Right side: three small predictions

Each factor came from a different next-token distribution: before `blue`, after `blue`, and after `today`.

Generation uses one distribution at a time. The path probability is calculated from the values selected along the way.

The same one-token rule builds a continuation of any length

$$p_\theta(w_1, \ldots, w_T \mid c) = \prod_{t=1}^{T} p_\theta(w_t \mid c, w_{<t})$$

c = context already given to the model
w_t = token produced at step t
T = total number of generated tokens
$\prod$ = multiply the term for every step

t = current prediction step, from 1 to T
$w_{<t}$ = earlier tokens $w_1, \ldots, w_{t-1}$
$p_\theta(a \mid b)$ = probability of a, given b, from weights θ
`<EOS>` = special token that can signal the end

How does generation stop? A full stop is only punctuation. The model may produce `<EOS>` as its next token; the host may also stop at a length limit or another configured stopping rule.

We have previewed what leaves one model call. Next, rewind to what the application sends into that call.

▸ Probability and log probability

Pressing Send packages more than the visible message

A SIMPLIFIED REQUEST PACKET

```
{
 "messages": [
  {"role":"system",    "content":"Follow platform safety and privacy policy."},
  {"role":"developer", "content":"Use tools for current facts; report uncertainty."},
  {"role":"user",      "content":"Hello, what is the weather in Oxford?"},
  {"role":"tool",      "content":"Oxford weather data returned with observation time."}
 ],
 "tools": [
  "get_weather"
 ]
}
```

This entire packet—not only the `user` sentence—is given to the language model.

Simplified illustration; exact fields vary across applications and APIs.

The host decides what enters this model call

Host system
the application around the model. Before a call, it can collect:

- application instructions;
- the user's message;
- selected conversation history;
- relevant documents or tool results.

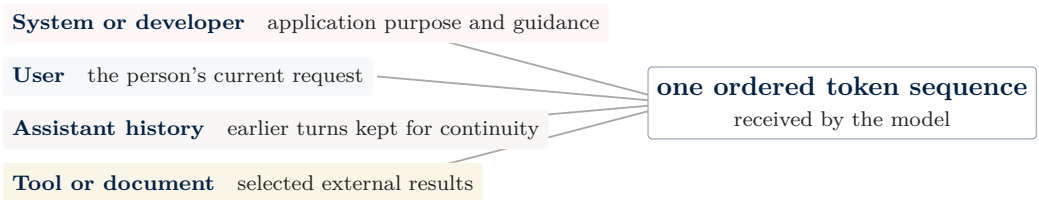
ordered request packet $\longrightarrow$ one token sequence received by the model

The visible user sentence may be only one part of this packet. Exact fields and formats differ across applications and providers.

The host has assembled the text. The model must now turn that text into tokens and numbers.

Message roles tell the model why each piece of text is present

Message role a label that identifies the purpose of one part of the request and how it relates to the other instructions.



System prompt high-priority guidance supplied by the service or application. It is part of the request, not another neural-network layer. A system prompt can guide the model, but it is not authentication, a file permission, or an operating-system security boundary.

▸ Request assembly and authority

Text is split into tokens before the model can process it

> **Token** One item from the model's fixed text vocabulary. A token can be a whole word, part of a word, punctuation, or a space joined to text.

ILLUSTRATIVE EXAMPLES

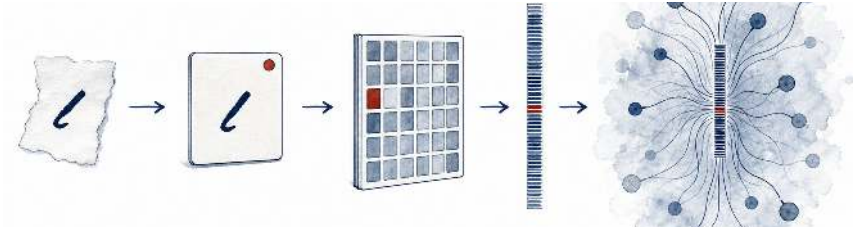
`weather`	`un` `believ` `able`	`?`	`Oxford`
may be one whole-word token	one word may require several tokens	punctuation can be a token	a leading space may belong to the token

These splits are illustrative. Exact tokens differ across models.

- The **vocabulary** is the model's finite list of possible tokens.
- A **tokeniser** applies fixed rules to split text into vocabulary tokens and replace each token with an integer ID.

A word is a human language unit; a token is a model input unit.

Each token ID retrieves a numerical starting point



token text ⟶ token ID ⟶ embedding-table row ⟶ starting vector

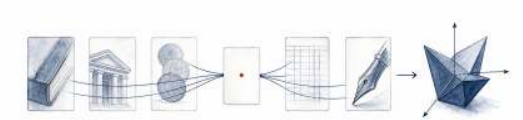
- A **vector** means a list of numbers.
- The **embedding table** stores one learned vector for each token ID; the ID selects its row.
- **Position information** is added so the model knows where the token occurs in the sequence.

The token `bank` begins with the same ID and starting vector in `bank loan` and `river bank`.

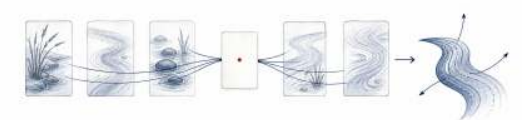
How can the same starting vector acquire two different meanings?

▸ Vectors and position notation

The sentence changes what the same token means



`bank loan`
financial meaning



`river bank`
edge-of-a-river meaning

- Both occurrences begin with the same token ID and starting embedding.
- Transformer layers use the surrounding tokens to change that starting vector.
- The results are different **contextual states**: numerical representations of what `bank` means in each sentence.

But how does the model decide which earlier tokens should influence the current token most?

Each token has an ID and a numbered position

Leila	put	the	cake	in	the	oven	.
1	2	3	4	5	6	7	8

Token
one word or text piece, such as `cake`

Token ID
the integer label used to identify that text piece

Position
the token's numbered place in this sequence

Vocabulary
the model's fixed list of possible token choices

A token ID is a category label, not a measure: token 8 is not "twice as meaningful" as token 4. Different tokenisation rules may split the same sentence differently.

The ID says *which text piece*; the position says *where it appears in this sequence*.

The CNN scoring idea now chooses among vocabulary tokens

CNN evidence	$\longrightarrow$	**37 breed logits**	$\longrightarrow$	**breed probabilities**
text prefix	$\longrightarrow$	**$\lvert\mathcal{V}\rvert$ token logits**	$\longrightarrow$	**next-token probabilities**

Prefix the tokens already available before the next choice.

Logit an unrestricted score before softmax; this is the same kind of score used for the CNN classes.

$|\mathcal{V}|$ the number of entries in the vocabulary.

Softmax converts either set of logits into non-negative probabilities that sum to 100%. The candidate list changes; the final scoring pattern does not.

The model produces the probabilities. A separate selection rule will later choose and append one token.

Earlier words make `it` useful for the next prediction

> Leila put the **cake** in the oven. She washed the dishes. When the timer rang, she took **it** _____

Token identity alone

`it` does not say which earlier object is being discussed

Prefix supplies clues

`cake`, `oven`, and `took` make some continuations more plausible than others

prefix clues + current token `it` $\longrightarrow$ next-token distribution

The task is therefore conditional: estimate the next-token probabilities *given this prefix*, not from `it` in isolation.

Context: the prefix made available for the current prediction. It may contain many tokens; not every token will be equally useful.

The model returns probabilities; decoding chooses one token

> `Leila ... she took it` _____

possible next token	`out`	`back`	`away`	all other tokens together
illustrative probability	**42%**	12%	6%	40%

Language model a model that assigns a probability to every possible next token given a prefix.

Next-token distribution the complete row of probabilities; together they sum to 100%.

These probabilities describe likely continuations. They do not certify that a continuation is true.

The notation $p(\texttt{out} \mid \texttt{Leila ... took it})$ means "the probability of `out`, given the prefix on the right of the vertical bar."

The LLM predicts; the surrounding application operates

text prefix $\longrightarrow$ next-token probabilities $\longrightarrow$ one selected token

Language model a model that assigns probabilities to possible next tokens given the current prefix.

Large language model (LLM) a language model with substantial scale in learned parameters, training data, or computation. “Large” has no universal numerical cutoff.

LLM application the model plus the surrounding software that assembles its input, selects tokens, connects tools, and returns results.

The model is the probability-producing component. The application decides what information reaches it and what may happen with its output.

The entire request and the new answer share a finite context window

$$\underbrace{\text{instructions}}_{\text{application guidance}} + \underbrace{\text{history}}_{\text{earlier turns}} + \underbrace{\text{documents and tools}}_{\text{selected evidence}} + \underbrace{\text{new answer}}_{\text{space to generate}} \leq \underbrace{\text{context window}}_{\text{finite token capacity}}.$$

Context window the finite number of token positions available in one model call. It is temporary working input, unlike the durable learned weights.

More context is not automatically better: irrelevant material uses capacity and may obscure the evidence that matters. The host chooses what enters and reserves space for the answer.

▸ Audit one token budget

A one-line reply can carry a large hidden input

You type: `What about tomorrow?`

The application may also resend instructions, tool descriptions, earlier turns, and selected documents.

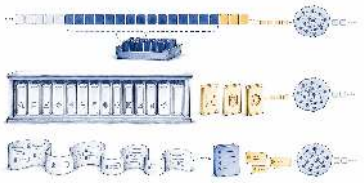
call	background sent again	newly added text
turn 1	instructions + tool descriptions	message 1
turn 2	the same background + turn 1	message 2
turn 3	the same background + turns 1–2	message 3

Input tokens
All the text sent into this call, including the repeated background.

Output tokens
Only the new answer generated by the model.

The visible message length is not the model-call size. Repeated background can dominate the time and computation, yet some of it may contain rules or facts the model still needs.

Systems can reuse, select, or shorten repeated context



1. **Prompt caching reuses computation.** The repeated prefix still occupies context positions, but some earlier work need not be repeated.
2. **On-demand tool loading selects.** Only relevant tool definitions are sent instead of every available tool.
3. **Compaction shortens.** Older detail is replaced with a shorter state; this saves tokens but may lose information.

Next: we now know what enters and leaves an LLM call. How were the probabilities inside the model learned?

▸ What each method saves—and risks

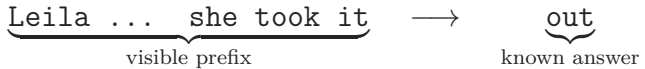
Part III

Text trains the model; generation uses it

How does text teach the probabilities, and how are those probabilities used?

The text already contains the answer used for training

> Leila put the cake in the oven. Later, she took it **out**.



Target the observed token the model should have predicted.

During training, `out` is hidden from this prediction, but it is already present in the original text. The model assigns probabilities to every vocabulary token and is corrected using the known answer.

The text supplies both the question—the prefix—and the answer—the next token. Human-written class labels are not required for every position.

One sentence supplies a prediction exercise at nearly every position

```
                    Leila   ⟶   put
                Leila put   ⟶   the
Leila ...  she took it      ⟶   out
```

Self-supervised learning: create training targets from the data itself. A **left-looking rule** prevents each prediction from reading the later token it is meant to predict; Part IV will show the mechanism.

▸ Why the rows are shifted

The loss asks how much probability reached the observed token

> After `Leila ... she took it`, the observed next token is **`out`**.

token	`out`	`back`	`away`	all others
illustrative probability	**5%**	35%	20%	40%

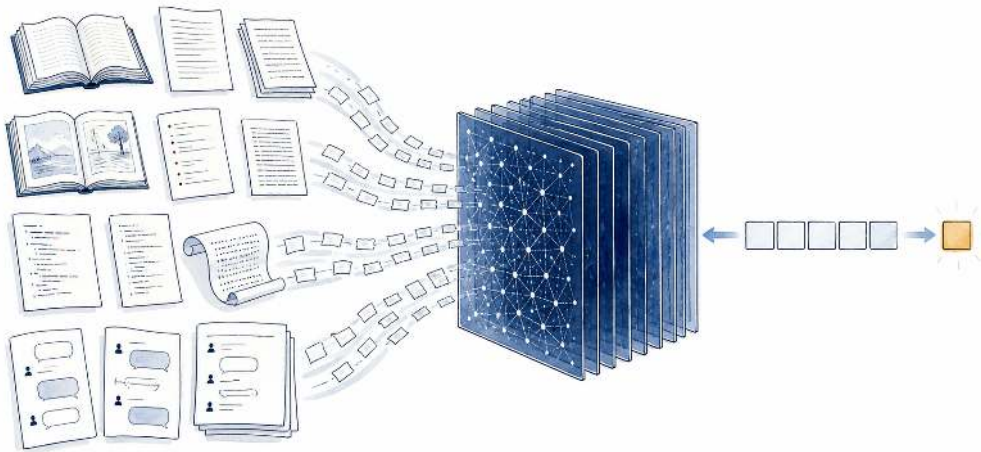
$$\underbrace{\ell}_{\text{token loss}} = -\log \underbrace{p(\texttt{out} \mid \texttt{Leila ... took it})}_{\text{probability assigned to the known answer}} .$$

Loss one number measuring how poorly this prediction matched the observed target. Low target probability gives a large loss; high target probability gives a small loss. Training then adjusts the shared parameters so similar future prefixes place more probability on their observed next tokens.

Softmax turns logits into probabilities that sum to one. The loss rewards any increase in the observed token's probability, not only a change in which token ranks first.

▸ Calculate the loss and gradient

Pretraining repeats that correction across a vast text collection



Pretraining repeats that correction across a vast text collection

- **Read many kinds of text:** prose, dialogue, code, and other token sequences provide different next-token patterns.
- **Reuse one model:** every correction adjusts the same large set of learned parameters.
- **Generalise:** success requires useful predictions on new prefixes, although large models can also memorise some training passages.

Pretraining: broad training before a particular conversation or downstream task. Training changes learned weights; it does not install a reliable search interface over the training collection.

Scale supplies breadth: the same prediction exercise is repeated across many contexts, so one set of weights must support many kinds of continuation.

▸ See the sequence objective

Pretraining creates a continuation model—not yet a helpful assistant

Prefix: `Leila put the cake in the oven. Later, she took it`

If the prefix looks like prose	continue the prose pattern.
If it looks like dialogue	continue the dialogue pattern.
If it contains instructions	it may imitate an answer pattern—but helpfulness and instruction priority are not guaranteed by pretraining alone.

Base model a broadly pretrained next-token predictor before it is specifically shaped to behave as a conversational assistant.

That leaves a behavioural problem: a capable continuation model is not automatically a useful conversational assistant.

Post-training shapes the base model into a chat model

Broad pretraining → **Instruction examples** → **Preference and safety shaping**

predict text across many domains

learn useful request–response patterns

favour better behaviours among possible answers

Post-training additional training after broad pretraining; it may use demonstrations, preferences, safety data, rewards, or tool-use examples.

Instruction tuning post-training on examples of requests and useful responses.

Chat model a post-trained language model shaped to follow conversational instructions and message roles.

Durable change

training updates learned weights and affects later requests

Temporary guidance

a prompt changes the current input; it does not retrain the model

▸ Post-training methods and boundaries

Training can check many known answers; generation creates one unknown token

Training

The full sentence already exists.

- each position has an observed next-token target;
- the left-looking rule hides later answers;
- the known targets supply many loss terms;
- gradients update the model weights.

Generation

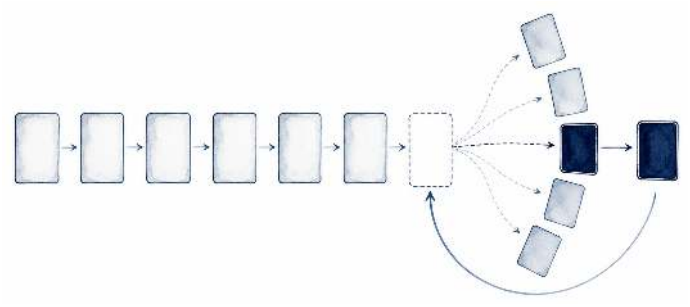
Only the current prefix exists.

- the next token is unknown;
- the model produces one probability distribution;
- a selection rule selects one token;
- that token must be appended before prediction continues.

During training, future tokens exist in the data but are hidden from the relevant prediction. During generation, future output tokens do not yet exist and each new choice depends on earlier selections.

See the training and generation shapes

Generation selects, appends, and asks again



... took it ⟶ out ⟶ of ⟶ the ⟶ oven

Decoding the rule that turns the current vocabulary probabilities into one selected token. It may take the most probable token or sample among plausible alternatives.

Each selected token changes the prefix and therefore changes every later probability. Decoding controls variability; it does not verify whether a statement is true.

▸ Temperature and top-p

Context must change the starting vector for `it`

Leila sliced the **cake** and served **it**.

Here, `it` refers to the cake.

Leila cleaned the **oven** and closed **it**.

Here, `it` refers to the oven.

Both occurrences use the same token ID and begin with the same learned token vector. Their surrounding words are different, so the useful final representations must also be different.

Contextual state the vector for one token after the network has used the surrounding sequence to revise its starting description.

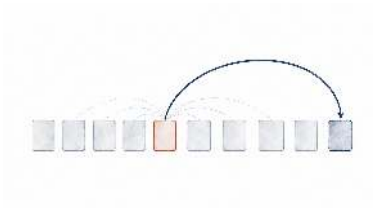
The mechanism question: how does the current token decide which earlier vectors should influence its update most? Attention provides the route.

Part IV

Attention finds the earlier clues that matter now

For `it`, how can the model use `cake` many positions earlier?

To predict after `it`, the model needs a route back to `cake`



CNN: the route is fixed
each image location first receives information from nearby locations.

Language: the useful route changes
the word that helps may be near or far, depending on this sentence.

> Leila put the **cake** in the oven. She washed the dishes. When the timer rang, she took **it**
> ———

Unlike a CNN's fixed local route, the useful language route depends on this sentence. Attention calculates that route from the current token states.

Attention: a learned calculation that lets one token position gather a mixture of information from other allowed positions.

Early sequence models passed only one summary to the writer

Input sentence
Leila put the cake in the oven.

$\longrightarrow$

Output translation
Leila a mis le gâteau dans le four.

Input, or source
the sentence the system reads

Output
the sentence the system produces

The reader had to compress the whole input into one fixed-length list of numbers before the writer began. Older descriptions may call that list “memory” or “context”; here it simply means one numerical summary.

▸ How the designs changed

A longer sentence still had to fit into the same-size summary

Short input: few details to remember

> Take the **cake** out.

→ **one summary**
one fixed number of values

Long input: many details to remember

> Leila put the **cake** in the oven. She washed the dishes. When the timer rang, she took **it** out.

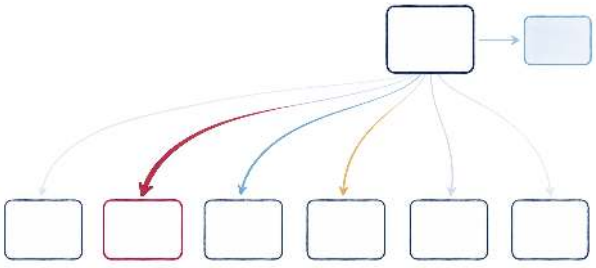
→ **the same summary size**
still the same number of values

Fixed-size
the summary does not gain more space when the input grows

Bottleneck
one limited route through which every input detail must pass

If `cake` becomes unclear in that summary, the writer cannot look back and recover it from the original word records.

Attention keeps each word and chooses the useful clues



Word being understood
here, `it`; the model is finding what it refers to.
Technical name: current word

Word records kept separately
one vector for each numbered word place; `cake` remains available.
Technical name: available positions

Attention chooses and mixes
give each record an importance number; larger means more information is used

> Leila put the **cake** in the oven ⋯ she took **it** out.

For this occurrence of `it`, `cake` should receive the largest share; its information will dominate the update.

▸ Old route versus attention route

The attention pattern is recomputed for each current word

> **Leila** put the **cake** in the **oven**. **She** washed the dishes. Later, she took **it** out.

word or step being updated	useful earlier clue	what that clue may supply
Who does `She` refer to?	`Leila`	the person being discussed
What does `it` refer to?	`cake`	the object being discussed
Next word after `took it`	`oven`	support for the relation “out of”

Training is not given the links in this table. It adjusts the model so that connections useful for prediction tend to receive more influence.

Attention works like searching labelled information cards

Current job: work out what `it` refers to in the Leila–cake sentence.

Query q: search request

made from `it`; asks: "What earlier thing does this word refer to?" It is used to search the saved word records.

Key k: search label

made for every saved word; answers: "What kind of clue could I provide?" It is compared with the query to judge whether this record is useful.

Value v: information card

made for every saved word; supplies: "Here is the information I contribute." It is mixed into the update when its key is a strong match.

For `it`, the query searches for a likely earlier object; extttcake's key may match that request, and its value can then contribute information. Every word makes all three learned number vectors.

▸ How the three views are calculated

The request from `it` is checked against every saved label

Plain-English intuition for this query: "Which earlier thing was taken out?"

saved word j	plain-English intuition for its key	match score
`cake`	an earlier object that can be moved	2—strong
`oven`	a place something can be taken from	1—medium
`timer`	a signal for when an action happens	0—weak

`cake` matches the current need best, so it receives the largest score. Softmax will turn the three scores into usable shares.

j simply means "the saved word being checked now." The labels and scores above are teaching interpretations; the trained network creates its actual scores from the numerical query and keys.

▸ Dot products, shapes, and the full equation

Softmax divides a 100% attention budget among the words

Think of 100 units of listening time: stronger matches should be heard more.

saved word	cake	oven	timer
match score	2	1	0
attention share	66.5%	24.5%	9.0%

$$66.5\% + 24.5\% + 9.0\% = 100\%.$$

Softmax: the calculation that turns arbitrary match scores into positive shares adding to 100%.
Attention weight: one temporary share for one saved word; the model recalculates it for each current word and sentence.

▸ Worked score-to-share trace

Use the shares as volume controls on the saved information

$$\underbrace{0.665}_{\text{loud}}\ \underbrace{\boldsymbol{v}_{\texttt{cake}}}_{\text{cake card}} + \underbrace{0.245}_{\text{quieter}}\ \underbrace{\boldsymbol{v}_{\texttt{oven}}}_{\text{oven card}} + \underbrace{0.090}_{\text{quiet}}\ \underbrace{\boldsymbol{v}_{\texttt{timer}}}_{\text{timer card}}$$

1. Set the volume
multiply each information card by its attention share

2. Mix the cards
add the scaled number lists coordinate by coordinate

3. Update `it`
combine the result with the current vector for `it`

The result is a new information mixture dominated by `cake`. It changes the vector for this occurrence of `it`.

Technical names: each information card is a **value vector**; the new blend for `it` is its **attention output**. No word is copied.

▸ Worked two-coordinate mixture

Every position gets its own attention update

$$\underbrace{T \text{ starting token vectors}}_{\text{one per numbered slot}} \longrightarrow \underbrace{T \text{ updated token vectors}}_{\text{one per numbered slot}}.$$

current word	its query is matched with allowed keys	its result
`Leila`	the words this position may use	updated `Leila` vector
`cake`	the words this position may use	updated `cake` vector
`it`	the words this position may use	updated `it` vector

T is the number of token positions in the row. A **position** is simply one numbered token slot. Every row gets its own scores, attention shares, and value mixture.

The causal mask implements the left-looking rule inside attention

> Leila put the cake in the oven ⋯ she took it [**next token?**]

word relative to `it`	may attention use it?	reason
`cake`, `timer`, `it`	yes	already present in the prompt
`out`	no	this is the future answer

This is the left-looking rule from Part III: the model may use the prefix seen so far, but not the later answer it is learning to predict.

Causal mask: the rule that blocks positions to the right of the current position. A blocked position receives an attention share of zero.

▸ Compare three visibility rules

Attention is one learned exchange of information—not yet a full model

numbered token vectors ⟶ query–key match ⟶ attention shares ⟶ value mixture ⟶ updated vectors

> The updated vector for **it** can now include information gathered from **cake**, which can help the model predict **out**.

What attention does
chooses which allowed token vectors influence each update, how strongly they influence it, and what numerical information is mixed.

What this does not guarantee
human-like understanding, factual correctness, or a complete explanation of why the final prediction was made.

Next question: how can a deep network preserve, process, and repeat that update until it produces logits?

▸ Cost and interpretation limits

Part V

Transformers repeat and refine context updates

What turns one attention update into a stable network that produces token scores?

One attention update is useful; it is not yet a complete model

> Leila put the **cake** in the oven. She washed the dishes. When the timer rang, she took **it** _____

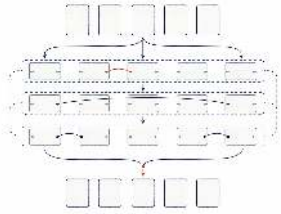
$$\underbrace{\text{old state for } \texttt{it}}_{\text{what was already present}} + \underbrace{\text{attention update}}_{\text{information gathered from } \texttt{cake}} \longrightarrow \underbrace{\text{richer state for } \texttt{it}}_{\text{one revised vector}}.$$

Transformer a neural network that repeatedly updates one vector for every token position. The 2017 design made attention the main route for positions to exchange information.

Transformer block one repeatable processing stage inside that network.

A complete block preserves useful information, processes a proposed update, and returns the same number of token vectors for the next block to refine.

Several heads can compare the same sentence differently



> For `it`, one learned comparison may emphasise **cake**, another may use **oven**, and another may use **took**.

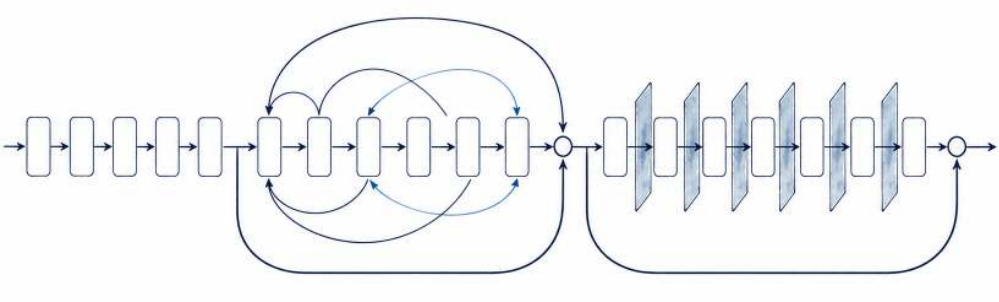
Attention head one learned query–key–value calculation.

Multi-head attention several heads run in parallel; their numerical outputs are placed side by side and remixed into one update.

The coloured relations are illustrations, not permanent labels. Training learns the patterns, and heads may overlap.

▸ How several heads become one update

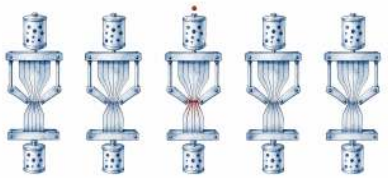
A block alternates two jobs: communicate, then transform



1. **Communicate**: multi-head attention lets token vectors gather information from permitted positions.
2. **Transform**: a small neural network processes each updated token vector separately.

MLP, or multilayer perceptron: the familiar dense neural network from Day 1, applied separately to every numbered token position.

The MLP is the familiar dense network reused at every token



- **Attention communicates:** it brings information from other allowed token positions.
- **The MLP transforms:** it applies a small dense neural network to one token vector at a time.
- **The rule is shared:** every position in this layer uses the same learned MLP, but produces a different result because its vector differs.

"Position-wise" means that token positions do not communicate during this step; their communication already happened in attention.

▸ Calculate one small MLP

Keep the old state, then add only a useful correction

$$\underbrace{\text{new token state}}_{\text{what continues}} = \underbrace{\text{old token state}}_{\text{carried directly}} + \underbrace{\text{learned update}}_{\text{attention or MLP contribution}} .$$

1. the **old state** carries information already present;
2. the **learned operation** proposes an update;
3. coordinate-by-coordinate addition produces the new state.

Residual connection, or skip connection a direct route that carries the incoming state around a learned operation and adds the operation's update to it.
The learned sublayer only has to propose a useful correction. If that update is near zero, the information already present can pass through largely unchanged.

Residual means addition. It is not a second memory, a replacement, or a decision to skip running the learned operation.

▸ Add two example vectors

Normalisation keeps repeated updates on a workable scale

One token row before rescaling

$$\begin{bmatrix} 0.2 & 8.1 & -5.4 & 1.7 & \cdots \end{bmatrix}$$

$$\downarrow$$

The same row after rescaling

$$\begin{bmatrix} -0.1 & 1.4 & -1.2 & 0.3 & \cdots \end{bmatrix}$$

schematic values; the appendix shows an exact calculation

Layer normalisation
rescale the numerical features inside one token vector before the next learned calculation.

Each token row is rescaled separately, followed by a learned scale and shift. Words are not averaged together; every token keeps its own position. The next attention or MLP calculation therefore receives a more predictable numerical scale.

▸ Work through one exact rescaling

One block communicates, adds, transforms, and adds again

token states $\longrightarrow$ normalise $\longrightarrow$ attention $\longrightarrow$ add the old states

intermediate states $\longrightarrow$ normalise $\longrightarrow$ MLP $\longrightarrow$ add the intermediate states

Attention changes each state using information from other allowed positions. The MLP then changes the features inside each state. Residual additions keep the old information available around both operations.

This shows one common pre-normalised block: normalisation occurs before each learned operation. The appendix gives the equations and shapes.

▸ Expand the equations into six steps

Depth lets later blocks work with already-enriched token states

> Leila put the **cake** in the **oven**. She washed the dishes. When the timer rang, she **took it** _____

$$\underbrace{T\times d}_{\text{starting vectors}} \longrightarrow \underbrace{T\times d}_{\text{block 1}} \longrightarrow \underbrace{T\times d}_{\text{block 2}} \longrightarrow \cdots \longrightarrow \underbrace{T\times d}_{\text{final vectors}} .$$

Layer one application of a Transformer block with its own learned parameters.

Depth several layers applied one after another.

Why the shape stays the same each block changes the information inside the vectors, not the number of token slots T or their width d.
Later layers compare states that already contain context gathered by earlier layers. This mirrors the CNN story: later layers process representations built by earlier layers.

The sentence progression is explanatory. Real layers and heads have distributed, overlapping roles rather than fixed human-written jobs.

▸ Audit shapes through one block

Every decoder block keeps the same left-looking boundary

> Leila put the cake in the oven ⋯ she took **it** **out** [later tokens]

Block 1 → **Block 2** ⟶ **Final block**

use `it` and tokens to its left | use `it` and tokens to its left | use `it` and tokens to its left

Self-attention query, key, and value vectors come from the same table of token states.

Decoder-only Transformer uses causal self-attention in every block: each position may use itself and earlier positions, but not later ones.

More depth gives more rounds of processing; it does not allow any layer to look at the future answer.

The original 2017 Transformer contained both a reader and a writer. The decoder-only path followed here keeps a causal, writer-like stack.

▸ Compare Transformer families

The final token state produces the distribution we met earlier

> Leila put the cake in the oven ⋯ she took **it** _____

$$\underbrace{\boldsymbol{z}}_{\text{one vocabulary score per entry}} = \underbrace{\boldsymbol{h}_{\text{current}}}_{\text{final vector at it}} \underbrace{W_{\text{out}}}_{\text{learned output weights}} + \underbrace{\boldsymbol{b}_{\text{out}}}_{\text{learned biases}} .$$

possible next token	`out`	`back`	`away`
illustrative score, or logit	3.2	1.1	0.6

This opens the black box from Part II: the Transformer builds the final state, the output map produces one logit for every vocabulary entry, and softmax produces the next-token distribution. Selection still belongs to decoding.

GPT puts the language-model task inside a decoder-only Transformer

Generative	at generation time, a decoding rule selects a token, appends it, and prediction repeats.
Pre-trained	learns broad text patterns from a large text collection before use.
Transformer	the decoder-only causal architecture that constructs contextual vectors.

GPT = Generative Pre-trained Transformer

> For `Leila ... took it`, attention gathers relevant clues, stacked blocks refine the state, and the output map scores `out`, `back`, `away`, and every other vocabulary token.

Translation, summarisation, code completion, and conversation are applications built around trained Transformer models—not definitions of attention.

Architecture explains how scores are computed; training explains how the weights were shaped; decoding explains how scores become text.

Finish

A useful AI system needs more than the model

Who supplies evidence, enforces permissions, and controls real actions?

The model proposes; the host decides what may happen

host assembles context $\longrightarrow$ model proposes $\longrightarrow$ host checks $\longrightarrow$ tool executes

1. The **model** maps the supplied token context to probabilities for text or a structured tool request.
2. The **host system** authenticates the user, checks permissions and approvals, and decides whether a proposed action is allowed.
3. A **tool** performs an external operation such as search, calculation, file access, or a database query.

A prompt can tell the model what it should request. It cannot grant file permissions or override execution limits; those controls belong to the host.

▸ Guidance versus enforcement

A likely continuation is not a verified claim

Prediction

`out` may be the most likely continuation of our sentence.

Evidence

likelihood alone does not prove that Leila really removed the cake.

Control

a fluent instruction does not grant permission to inspect a file or operate a tool.

Support important claims with evidence **check consequential outputs** **control real actions**

The model can still produce unsupported, outdated, or internally inconsistent text. Better prompting or more deterministic decoding does not close the gap between probable wording and verified truth.

The complete system separates prediction, evidence, and control

- **CNNs** use fixed local sharing to build visual evidence for a fixed class list.
- **Language models** learn which token is likely after a prefix.
- **Attention and Transformers** construct the contextual evidence used for that token choice.
- **The host system** must add selected evidence, permissions, execution, and accountability.

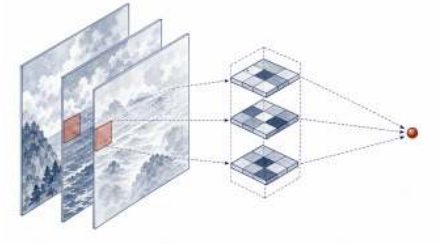
Probability can guide a prediction. It cannot, by itself, verify a fact or authorise an action.

Reference

Technical appendix

Worked calculations, notation, shape checks, and system details.

Appendix: one filter combines all channels at one location



1. Same location
take the same small window from every input channel

2. One filter
use one weight grid per channel; multiply matching cells and add everything

3. One response
add one bias and apply ReLU; the red dot is the match strength here

$$\text{response} = \text{ReLU}(\text{weighted sum of this patch} + \text{bias})$$

◂ Return to the CNN recap

▸ Next: numerical example

Appendix: one filter calculation

For visibility, this example shows one channel. The RGB calculation repeats the products across three channels before the same final sum.

$$\begin{bmatrix} 0.1 & 0.5 & 0.9 \\ 0.1 & 0.5 & 0.9 \\ 0.1 & 0.5 & 0.9 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} &3\big[(0.1)(-1) + (0.5)(0) + (0.9)(1)\big] \\ &= 3(-0.1 + 0 + 0.9) \\ &= \boxed{2.4}. \end{aligned}$$

$$a = \mathrm{ReLU}(2.4 - 0.5) = \mathrm{ReLU}(1.9) = \boxed{1.9}.$$

Negative weights subtract evidence, zero weights ignore a cell, and positive weights add evidence. Bias shifts the threshold; ReLU keeps positive responses and replaces negative ones with zero.

Return to the filter recap

Appendix: derive one response map's size

H, W: input height and width; k: filter size; p: padding; s: stride.

Step 1: add padding. An input $H \times W$ becomes $(H + 2p) \times (W + 2p)$.

Step 2: fit the filter. A $k \times k$ filter can travel $H + 2p - k$ cells vertically and $W + 2p - k$ cells horizontally.

Step 3: count stride-sized moves, then include the starting position.

$$H' = \left\lfloor \frac{H + 2p - k}{s} \right\rfloor + 1, \qquad W' = \left\lfloor \frac{W + 2p - k}{s} \right\rfloor + 1.$$

Example: $H = W = 64,\ k = 3,\ p = 1,\ s = 1.$

$$H' = W' = \frac{64 + 2(1) - 3}{1} + 1 = 64.$$

One filter therefore produces 64×64 responses, arranged as one map.

◂ Return to the map stack

▸ Next: filters and parameters

Appendix: derive output depth and parameter count

C_{in}: maps entering the layer; C_{out}: different filters learned.

Step 1: count one filter's weights. It has a $k \times k$ grid for each of the C_{in} input maps: $k^2 C_{\text{in}}$ weights.

Step 2: add one bias for that filter. This gives $k^2 C_{\text{in}} + 1$ parameters per filter.

Step 3: learn C_{out} different filters. Each filter contributes one output map:

$$\text{total parameters} = \big(k^2 C_{\text{in}} + 1\big) C_{\text{out}}, \qquad \text{output shape} = H' \times W' \times C_{\text{out}}.$$

Example: 3×3 filters, $C_{\text{in}} = 3$, $C_{\text{out}} = 16$:

$$\underbrace{(3 \times 3 \times 3) + 1}_{\text{28 per filter}} \implies \underbrace{28 \times 16}_{\text{448 parameters}}, \qquad \text{output} = 64 \times 64 \times 16.$$

H, W do not enter the parameter count: the same filter is reused at every spatial position.

Return to the map stack

Appendix: receptive-field arithmetic

Let r_ℓ be the field size and j_ℓ the spacing between neighbouring cells, measured on the original image:

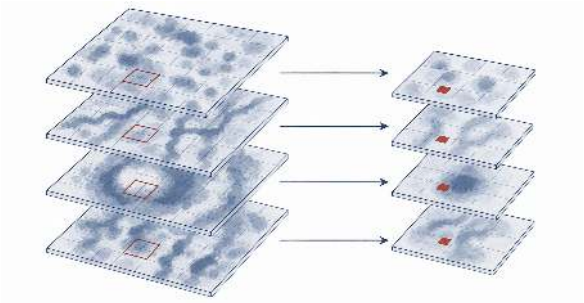
$$r_\ell = r_{\ell-1} + (k_\ell - 1) j_{\ell-1}, \qquad j_\ell = j_{\ell-1} s_\ell, \qquad r_0 = j_0 = 1.$$

operation	Conv 1	Pool 1	Conv 2	Pool 2	Conv 3	Pool 3
kernel k	3	2	3	2	3	2
stride s	1	2	1	2	1	2
field r	3	4	8	10	18	22
spacing j	1	2	2	4	4	8

> One final 8×8 activation can therefore depend on a nominal 22×22 region of the 64×64 input.

◂ Return to the receptive field

Appendix: one max-pooling calculation



Step 1: choose one 2×2 block.

$$\begin{bmatrix} 0.1 & \mathbf{0.8} \\ 0.3 & 0.2 \end{bmatrix} \xrightarrow{\max} \boxed{0.8}.$$

Step 2: move two cells and repeat.
Four exact positions become one local maximum.

Step 3: see what changes across the whole stack.

$$\underbrace{64 \times 64}_{\text{rows and columns in each map}} \times \underbrace{16}_{\text{separate maps}} \xrightarrow{2\times2 \text{ max, } s=2} \underbrace{32 \times 32}_{\text{half as many rows and columns}} \times \underbrace{16}_{\text{same maps}}.$$

H: rows; W: columns; C: separate feature maps; $s = 2$: move the 2×2 window two cells each time. In general, $H \times W \times C \to \frac{H}{2} \times \frac{W}{2} \times C$. Pooling does not mix the maps or learn weights.

◂ Return to pooling

Appendix: calculate one global average

Step 1: take one final response map.

$$\begin{bmatrix} 0 & 1 & 0 \\ 0.5 & 0.8 & 0.2 \\ 0 & 1 & 1 \end{bmatrix}$$

Step 2: add every position and divide by the number of positions.

$$\frac{0+1+0+0.5+0.8+0.2+0+1+1}{9} = \boxed{0.5}.$$

Step 3: repeat separately for every final map.

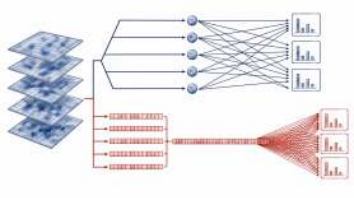
$$8 \times 8 \times 64 \longrightarrow 64 \text{ summaries.}$$

GAP discards the exact final arrangement but keeps one average strength for each learned feature map.

◂ Return to GAP

▸ Next: compare with flatten

Appendix: GAP is compact; flatten preserves layout



Global average pooling

$8 \times 8 \times 64 \to 64$ values. The head needs $64(37) + 37 = \boxed{2{,}405}$ parameters.

Flatten

$8 \times 8 \times 64 \to 4{,}096$ values. The same head needs $4096(37) + 37 = \boxed{151{,}589}$ parameters.

GAP is compact and position-insensitive; flatten preserves layout. The parameter difference appears in the dense head.

Return to GAP

Appendix: exact shapes and parameters

step	operation	output shape	parameters
input	resized photograph	$64 \times 64 \times 3$	0
1	Conv 3×3, 16 filters, $p = 1, s = 1$, ReLU	$64 \times 64 \times 16$	448
2	max pool 2×2, $s = 2$	$32 \times 32 \times 16$	0
3	Conv 3×3, 32 filters, $p = 1, s = 1$, ReLU	$32 \times 32 \times 32$	4,640
4	max pool 2×2, $s = 2$	$16 \times 16 \times 32$	0
5	Conv 3×3, 64 filters, $p = 1, s = 1$, ReLU	$16 \times 16 \times 64$	18,496
6	max pool 2×2, $s = 2$	$8 \times 8 \times 64$	0
7	global average over positions	64	0
8	dense, $64 \to 37$ breeds	37	2,405
total			**25,989**

A convolution learns $\big(k^2 C_{\text{in}} + 1\big) C_{\text{out}}$ parameters. H, W do not enter this count because one filter is reused at every spatial position.

◂ Return to the complete pipeline

Appendix: calculate one breed score

For visibility, this toy head uses three summaries instead of 64.

input	summary g_m	learned weight W_{tm}	product
map 1	0.8	0.6	0.48
map 2	0.2	−0.3	−0.06
map 3	0.5	0.4	0.20

$$z_t = 0.48 - 0.06 + 0.20 + \underbrace{0.10}_{\text{bias}} = \boxed{0.72}.$$

> The summaries change with the image. The weights and bias are learned from training data and then reused for every image.

Return to one breed score

Appendix: compare all breed scores to get probabilities

> The previous slide gave breed A a score of 0.72. That is a **raw score**, not a 72% probability. We must compare it with the scores of the other breeds.

For visibility, suppose the same calculation produced scores for only three breeds instead of 37.

breed	raw score	positive amount	share of total
A (previous slide)	0.72	$e^{0.72} = 2.05$	52.1%
B	0.20	$e^{0.20} = 1.22$	31.0%
C	−0.40	$e^{-0.40} = 0.67$	16.9%
total		3.94	100%

$$\Pr(\text{breed A}) = \frac{2.05}{2.05 + 1.22 + 0.67} = \boxed{52.1\%}.$$

Make every score positive, then divide it by the total across all breeds.

Softmax: compares all raw scores and turns each into a positive share. The shares keep the same ranking and add to 100%. Technical form: $p_t = e^{z_t} / \sum_k e^{z_k}$; t is one breed and k runs over all breeds.

Return to breed scores

Appendix: tokens, token IDs, vocabulary, and prefix

Tokenizer the fixed procedure that splits text into tokens and maps each token to an integer ID.

1. **Start with text**

   `Leila put the cake in the oven.`

2. **Split into vocabulary entries**

   [`Leila`] [`put`] [`the`] [`cake`] [`in`] [`the`] [`oven`] [`.`]

3. **Replace each entry by its token ID**

   [4812, 711, 279, 13401, 304, 279, 17819, 13] (illustrative IDs).

Vocabulary the complete token-to-ID lookup list.

Prefix the ordered token IDs already available before the next prediction.

Tokens need not be whole words: a tokenizer may use punctuation, common word pieces, or byte-level pieces. Token IDs are arbitrary labels, so their numerical size carries no meaning.

◂ Return to tokens

▸ Next: continuation probability

Appendix: continuation probability is a product of local choices

Notation first

- c: the prefix already present, here `Leila ... took it`;
- y_n: continuation token number n;
- $y_{<n}$: continuation tokens selected before position n;
- N: the number of tokens in the continuation being scored.

Step 1: score each local choice using everything available so far

$$p_\theta(y_n \mid c, y_{<n}).$$

Step 2: multiply the local conditional probabilities

$$\underbrace{p_\theta(y_{1:N} \mid c)}_{\text{probability of this continuation}} = \prod_{n=1}^{N} \underbrace{p_\theta(y_n \mid c, y_{<n})}_{\text{one next-token probability}}.$$

Step 3: check a three-token example

$$p(\texttt{out, of, the} \mid c) = 0.42 \times 0.55 \times 0.62 = 0.14322.$$

The multiplication is the probability chain rule, not an extra neural-network layer. Training normally sums negative log probabilities because logs turn products into numerically stable sums.

◂ Return to language-model probabilities

▸ Next: shifted training targets

Appendix: one token sequence becomes shifted training rows

Start with one observed sequence:

$$(x_1, x_2, x_3, x_4, x_5, x_6) = (\texttt{Leila}, \texttt{put}, \texttt{the}, \texttt{cake}, \texttt{in}, \texttt{the}).$$

row	prefix visible to the prediction	target	target position
1	`Leila`	`put`	x_2
2	`Leila put`	`the`	x_3
3	`Leila put the`	`cake`	x_4
4	`Leila put the cake`	`in`	x_5
5	`Leila put the cake in`	`the`	x_6

Input row $(x_1, \ldots, x_{T-1})$. **Target row** $(x_2, \ldots, x_T)$.

The rows are shifted by one position. The left-looking rule makes prediction row t depend only on $x_1, \ldots, x_t$, while its observed answer is x_{t+1}.

The table describes five prediction problems, not five unrelated sentences. Later, the attention section names the mechanism that prevents a position from reading the future target.

◂ Return to many targets

▸ Next: loss and gradient

Appendix: one target, from score to correction

1. Turn all logits into probabilities.

$$p_j = \frac{e^{z_j - m}}{\sum_k e^{z_k - m}}, \quad m = \max_k z_k, \quad \sum_j p_j = 1.$$

z_j is the score for vocabulary entry j. Subtracting the largest score m keeps the calculation stable without changing the answer.

2. Read the observed target's probability.

$$p_{\text{target}} = p(\texttt{out} \mid \texttt{Leila ... took it}) = 0.05.$$

3. A low target probability gives a large loss.

$$\ell = -\log(0.05) = \boxed{2.996}.$$

For comparison, $p_{\text{target}} = 0.50$ gives the smaller loss $\ell = 0.693$.

4. Send a correction to every logit.

$$\frac{\partial \ell}{\partial z_j} = p_j - \mathbf{1}[j = \text{target}].$$

$\mathbf{1}[\cdot]$ is 1 for the observed target and 0 otherwise. Backpropagation carries these corrections through the network; an optimiser then updates its parameters.

◂ Return to the loss story

▸ Next: the sequence objective

Appendix: the next-token objective across one sequence

For T observed tokens $x_1, \ldots, x_T$, the model defines

$$p_\theta(x_2, \ldots, x_T \mid x_1) = \prod_{t=1}^{T-1} p_\theta(x_{t+1} \mid x_1, \ldots, x_t).$$

Training minimises the average negative log probability of the observed next tokens:

$$\mathcal{L}(\theta) = -\frac{1}{T-1} \sum_{t=1}^{T-1} \log p_\theta(x_{t+1} \mid x_{\leq t}).$$

θ	all learned model parameters.
$x_{\leq t}$	the prefix $x_1, \ldots, x_t$ available to prediction row t.
x_{t+1}	the observed target immediately after that prefix.
$T-1$ terms	every token except the first can serve as a target in this sequence.

Real training averages valid targets across many sequences and ignores padding. The objective says what to improve; backpropagation and the optimiser determine how parameters change.

Return to pretraining

Next: post-training

Appendix: pretraining, post-training, and prompting change different things

stage	learning signal	what changes	main purpose
pretraining	next tokens in broad text	model parameters	learn general continuation patterns
supervised instruction tuning	example desired responses	model parameters	learn response and tool-use patterns
preference or reinforcement methods	preferences, rewards, critiques, or verified outcomes	model parameters	favour desired behaviour
prompting at use time	current instructions and evidence	current context only	steer this request without retraining

RLHF reinforcement learning from human feedback: a family of methods using human-derived preferences or rewards.

DPO direct preference optimisation: a method that trains directly from preferred versus rejected responses under a particular objective.

Important boundary neither acronym means all post-training. Datasets, evaluators, objectives, and exact recipes differ across model families.

A prompt can strongly change behaviour because it changes the current input. It does not write new knowledge into the trained weights.

Return

Next: training versus generation

Appendix: training and generation use the same model differently

Training with T observed tokens

$$\underbrace{(x_1, \ldots, x_{T-1})}_{\text{model inputs}} \longrightarrow \underbrace{(x_2, \ldots, x_T)}_{\text{known targets}}.$$

Each prediction row can be compared with its observed target because the full training sequence already exists. Efficient implementations may evaluate many known targets together; the mechanism depends on the architecture taught later.

Generation with a current prefix of length t

$$(x_1, \ldots, x_t) \longrightarrow p(x_{t+1} \mid x_{\leq t}) \longrightarrow \widehat{x}_{t+1} \longrightarrow (x_1, \ldots, x_t, \widehat{x}_{t+1}).$$

The newly selected token $\widehat{x}_{t+1}$ must be appended before the next distribution $p(x_{t+2} \mid x_{\leq t}, \widehat{x}_{t+1})$ is defined.

Teacher forcing during training, prediction row t conditions on the observed earlier tokens $x_{\leq t}$, rather than on a generated version of that prefix.

Evaluating several known training targets together does not imply parallel autoregressive generation. The logical dependency between successive selected tokens remains.

◂ Return to training versus generation

▸ Next: decoding

Appendix: temperature and top-p change selection

Temperature rescales logits before softmax.

$$p_j^{(\tau)} = \frac{\exp(z_j/\tau)}{\sum_k \exp(z_k/\tau)}, \qquad \tau > 0.$$

For logits $(3.2, 1.1, 0.6)$:

τ	token 1	token 2	token 3	effect
0.5	0.980	0.015	0.005	sharper distribution
1.0	0.836	0.102	0.062	original scale
2.0	0.616	0.216	0.168	flatter distribution

Top-p sampling keeps the smallest high-probability set whose total reaches a threshold.
For $(0.60, 0.25, 0.10, 0.05)$, top-$p = 0.80$ keeps the first two tokens because $0.60 + 0.25 = 0.85$, then renormalises them to approximately $(0.706, 0.294)$.
Greedy decoding select $\arg\max_j p_j$. It is a separate rule; the temperature equation is not defined at exactly $\tau = 0$.

Temperature and top-p alter variability among token choices. They do not add evidence, correct outdated context, or turn probability into factual confidence.

◂ Return to generation

▸ Next: attention inputs

Appendix: from token ID to current vector

Step 1: give the text piece an integer address.
At numbered position t, let x_t be the vocabulary ID of that token. For example, x_4 is the ID used to look up `cake`.

The ID is only an address in a table. A larger ID does not mean a larger, later, or more similar word.

Step 2: look up its learned token vector.

$$\boldsymbol{e}_t = E[x_t].$$

E is the learned embedding table; $E[x_t]$ selects one row. That selected row is a d-number vector.

Step 3: add where this occurrence sits in the row.

$$\boldsymbol{h}_t^{(0)} = \boldsymbol{e}_t + \boldsymbol{p}_t.$$

$\boldsymbol{p}_t$ carries position information. $\boldsymbol{h}_t^{(0)}$ is the starting vector used by the first layer.

Step 4: let later layers gather sentence information.

$$\boldsymbol{h}_t^{(0)} \longrightarrow \boldsymbol{h}_t^{(1)} \longrightarrow \cdots \longrightarrow \boldsymbol{h}_t^{(\ell)}.$$

t: numbered token slot; d: number of coordinates in one token vector; ℓ: layer number. The superscript is a processing stage, not a mathematical power.

◂ Return to starting vectors
▸ Next: historical route

Appendix: three designs and what each changed

Encoder = reader
turns the input sentence into numerical vectors

Decoder = writer
uses those vectors to produce the output sentence

period	route from input to output	practical change
2014	the reader compresses the whole input into one fixed-length summary; the writer receives that same summary throughout	shows that a neural network can learn sentence-to-sentence translation, but every detail crosses one summary bottleneck
2014/15	the writer keeps all input-word vectors available and builds a new weighted summary at each output step	the writer can give direct influence to the particular input words useful now
2017	the Transformer uses attention as the main way positions exchange information	positions within one training layer can be processed in parallel; scaled dot-product attention and multiple attention calculations are combined

Primary papers: Cho et al. (2014); Sutskever, Vinyals & Le (2014); Bahdanau, Cho & Bengio (2014/15); Vaswani et al. (2017).

Return to the single-summary story

Appendix: one summary versus step-specific access

Old route: one unchanged summary

1. The reader makes one vector for each input word: $\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_T$.

2. Only the final reader vector is passed across: $\boldsymbol{c} = \boldsymbol{h}_T$.

3. Every output step must use that same $\boldsymbol{c}$. It cannot directly return to a particular $\boldsymbol{h}_j$.

Attention route: a new summary each step

1. At output step t, compare the writer's current state with every input-word vector: $e_{tj} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$.

2. Turn those match scores into shares: $\alpha_{tj} = \text{softmax}_j(e_{tj})$.

3. Build this step's summary: $\boldsymbol{c}_t = \sum_{j=1}^{T} \alpha_{tj} \boldsymbol{h}_j$.

T: number of input words; j: one input position; t: one output step; e_{tj}: match score; α_{tj}: attention share; $\boldsymbol{c}_t$: summary built for step t.

The decisive change is access: replace one permanent $\boldsymbol{c}$ with a step-specific $\boldsymbol{c}_t$ that can draw directly from every input-word vector.

Return to the intuitive solution

Appendix: how token states become queries, keys, and values

1. Stack the current token vectors.

$$H \in \mathbb{R}^{T \times d}.$$

There are T rows—one per token position—and d numbers in each row.

2. Learn three remapping matrices.

$$W_Q, W_K \in \mathbb{R}^{d \times d_k}, \qquad W_V \in \mathbb{R}^{d \times d_v}.$$

A projection is a learned weighted remapping. Three matrices let the same token state perform three different jobs.

3. Apply all three remappings to every token row.

$$Q = HW_Q, \qquad K = HW_K, \qquad V = HW_V.$$

result	shape	how to read row i
Q	$T \times d_k$	$\boldsymbol{q}_i$: what current position i needs
K	$T \times d_k$	$\boldsymbol{k}_i$: how position i can be matched
V	$T \times d_v$	$\boldsymbol{v}_i$: what position i can contribute

d_k and d_v are the chosen widths of these new vectors. Every entry in W_Q, W_K, W_V is learned during training.

◂ Return to the Q/K/V intuition

▸ Next: combine Q, K, and V

Appendix: the attention equation, one step at a time

$$\underbrace{O}_{\text{updated vectors}} = \underbrace{\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)}_{\text{attention-share table } A} \underbrace{V}_{\text{information to mix}} .$$

1. $QK^\top$: compare every current position with every candidate.
The result is a $T \times T$ score table. Cell (i, j) compares query $\boldsymbol{q}_i$ with key $\boldsymbol{k}_j$.

2. Divide by $\sqrt{d_k}$: keep score sizes numerically stable.
This rescales the numbers; it does not change which positions are allowed.

3. Add M, then apply softmax across each row.
M is the mask table. A blocked cell receives $-\infty$; softmax turns each allowed row into non-negative shares summing to one.

4. Multiply AV: use each share row to mix the value vectors.
The output $O \in \mathbb{R}^{T \times d_v}$ contains one updated vector for each of the T current positions.
softmax is applied separately to each row. The equation is compact, but it is exactly the four operations above: compare, scale, block, and mix.

◂ Return to the score story

▸ Next: numerical trace

Appendix: scores become attention shares step by step

For one deliberately small update at `it`, use

$$q_{\texttt{it}} = 1, \qquad (k_{\texttt{cake}}, k_{\texttt{oven}}, k_{\texttt{timer}}, k_{\texttt{out}}) = (2, 1, 0, 10), \qquad d_k = 1.$$

1. Multiply the query by every key.

$$(1)(2),\ (1)(1),\ (1)(0),\ (1)(10) = (2, 1, 0, 10).$$

2. Block the unseen future answer.

$$(2, 1, 0, 10) \longrightarrow (2, 1, 0, -\infty).$$

3. Exponentiate the allowed scores.

$$(e^2, e^1, e^0, e^{-\infty}) = (7.389, 2.718, 1, 0).$$

Their sum is 11.107.

4. Divide every entry by that sum.

$$\frac{(7.389, 2.718, 1, 0)}{11.107} = \boxed{(0.665, 0.245, 0.090, 0)}.$$

The keys and scores are constructed teaching values. Real queries and keys have many learned coordinates; their dot product supplies each score.

◂ Return to attention shares

▸ Next: mix the values

Appendix: mix the value vectors step by step

Use the attention shares from the previous page:

$$(\alpha_{\texttt{cake}}, \alpha_{\texttt{oven}}, \alpha_{\texttt{timer}}) = (0.665, 0.245, 0.090)$$

and the simple two-coordinate value vectors

$$\boldsymbol{v}_{\texttt{cake}} = (1, 0), \qquad \boldsymbol{v}_{\texttt{oven}} = (0, 1), \qquad \boldsymbol{v}_{\texttt{timer}} = (1, 1).$$

Step 1: multiply each value by its share.

$$\begin{aligned} 0.665(1, 0) &= (0.665, 0), \\ 0.245(0, 1) &= (0, 0.245), \\ 0.090(1, 1) &= (0.090, 0.090). \end{aligned}$$

Step 2: add the scaled vectors coordinate by coordinate.

$$\boldsymbol{o}_{\texttt{it}} = (0.665, 0) + (0, 0.245) + (0.090, 0.090) = \boxed{(0.755, 0.335)}.$$

Step 3: pass this result forward as the attention output for `it`.

The two coordinates are deliberately artificial so every operation is visible. Real value vectors are wider, and the coordinates have learned distributed meanings rather than human-written labels.

◂ Return to the value mixture

Appendix: why divide attention scores by $\sqrt{d_k}$

Step 1: a dot product adds one product per coordinate.

$$\boldsymbol{q}_i \boldsymbol{k}_j^\top = \sum_{r=1}^{d_k} q_{ir} k_{jr}.$$

d_k is the number of coordinates in each query and key vector.

Step 2: wider vectors usually produce more variable raw sums.

With more terms being added, the raw dot products can spread farther apart. Softmax may then become extremely sharp, which can make learning less stable.

Step 3: divide by the square root of the width.

$$s_{ij} = \frac{\boldsymbol{q}_i \boldsymbol{k}_j^\top}{\sqrt{d_k}}.$$

This keeps the typical score scale more comparable as d_k changes.

Scaling changes only the score size before softmax. It learns no new parameters and does not decide which positions the mask allows. In the teaching trace $d_k = 1$, so the divisor is 1.

Return to the full equation

Appendix: available positions depend on the task

For one attention row, i is the current token slot being updated and j is a candidate slot it might use. A visibility rule decides which j positions are available.

visibility rule	positions available to row i	why it is used
causal mask	the current slot and everything to its left: $j \leq i$	next-token prediction; prevents the model from seeing a future answer
full, or bidirectional, attention	every real token slot on both sides	tasks where the whole input is already available, such as classifying a sentence
padding mask	every real token slot, but not blank placeholders added only to make batch rows the same length	stops artificial blank positions from contributing information

Available
the candidate may receive a non-zero attention share

Blocked
the candidate is removed before the shares are calculated

"Available position" is not a special storage location. It means a numbered token slot that the visibility rule allows the current token to use.

Return to the next-token mask

Next: read a causal mask row by row

Appendix: read a four-token causal mask row by row

For four numbered token positions, the permission table is

	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	✓	×	×	×
$i=2$	✓	✓	×	×
$i=3$	✓	✓	✓	×
$i=4$	✓	✓	✓	✓

Row $i=1$ the first token can use only token 1; tokens $2, 3, 4$ are still future.

Row $i=2$ the second token can use tokens 1 and 2, but not 3 or 4.

Row $i=3$ the third token can use tokens $1, 2, 3$, but not future token 4.

Row $i=4$ the fourth token can use every token seen so far.

Before softmax, every × cell receives $-\infty$. Since $e^{-\infty}=0$, its attention share becomes exactly zero. A padding mask may be added to the causal mask to block artificial blank slots as well.

◂ Return to the next-token mask

Appendix: attention cost and interpretation limits

Why the score table grows quickly

If the sequence contains T token positions:

1. each of the T current positions has one query;
2. each query is compared with T keys;
3. the table therefore contains $T \times T = T^2$ scores.

$$T \longrightarrow 2T \qquad \Rightarrow \qquad T^2 \longrightarrow 4T^2.$$

Doubling the text length creates four times as many query–key comparisons. The learned projection parameters depend on vector widths, not on the length of this particular text.

What an attention row can show

It records which value vectors were mixed for one current position and in what proportions.

What it cannot prove by itself

- that a statement is factually supported;
- that a source is being cited;
- the complete causal reason for the final prediction.

Attention maps describe one mixing operation; they are not automatically explanations of the model's full reasoning.

Return to the attention summary

Appendix: several attention heads become one update

Step 1: begin with the same token-state table

$$\boldsymbol{H} \in \mathbb{R}^{T \times d}, \qquad d_h = \frac{d}{h}.$$

T: token positions d: model width h: number of heads d_h: width used by one head.

Step 2: each head makes its own Q, K, and V views

For head r,

$$\boldsymbol{Q}^{(r)} = \boldsymbol{H}W_Q^{(r)}, \qquad \boldsymbol{K}^{(r)} = \boldsymbol{H}W_K^{(r)}, \qquad \boldsymbol{V}^{(r)} = \boldsymbol{H}W_V^{(r)} \in \mathbb{R}^{T \times d_h}.$$

Step 3: each head runs the familiar attention calculation

$$\boldsymbol{O}^{(r)} = \mathrm{softmax}_{\mathrm{row}}\left(\frac{\boldsymbol{Q}^{(r)}\boldsymbol{K}^{(r)\mathsf{T}}}{\sqrt{d_h}} + \boldsymbol{M}\right)\boldsymbol{V}^{(r)} \in \mathbb{R}^{T \times d_h}.$$

Step 4: place head features side by side, then remix them

$$\boldsymbol{C} = \mathrm{Concat}_{\mathrm{columns}}\left(\boldsymbol{O}^{(1)}, \ldots, \boldsymbol{O}^{(h)}\right) \in \mathbb{R}^{T \times d}, \qquad \mathrm{MHA}(\boldsymbol{H}) = \boldsymbol{C}W_O \in \mathbb{R}^{T \times d}.$$

Concatenate means "place features side by side." The learned matrix W_O remixes them and restores width d, so the result can be added to $\boldsymbol{H}$.

◂ Return to multi-head attention ▸ Next: one MLP calculation

Appendix: calculate the MLP at one token position

For one token row, use the Day 1 structure $\boldsymbol{m}_i = \mathrm{ReLU}(\boldsymbol{x}_i W_1 + \boldsymbol{b}_1)W_2 + \boldsymbol{b}_2$.

Step 1: choose a two-number input and a three-unit hidden layer

$$\boldsymbol{x}_i = (1, -2), \qquad W_1 = \begin{pmatrix} 1 & -1 & 2 \\ 0.5 & 1 & -1 \end{pmatrix}, \qquad \boldsymbol{b}_1 = (1, 1, -1).$$

Step 2: first weighted layer, then ReLU

$$\boldsymbol{x}_i W_1 + \boldsymbol{b}_1 = (1, -2, 3), \qquad \mathrm{ReLU}(1, -2, 3) = (1, 0, 3).$$

Step 3: return from three hidden features to two output features

$$W_2 = \begin{pmatrix} 0.1 & 0.2 \\ 0 & 0 \\ 0.1 & 0.1 \end{pmatrix}, \qquad \boldsymbol{b}_2 = (-0.1, 0),$$

$$\boldsymbol{m}_i = (1, 0, 3)W_2 + \boldsymbol{b}_2 = (0.4, 0.5) + (-0.1, 0) = \boxed{(0.3, 0.5)}.$$

i names one token position. The same learned numbers are reused at every position, but each row is calculated independently.

Return to the position-wise MLP

Next: residual addition

Appendix: residual addition preserves the old state

Step 1: state entering the addition

$$\boldsymbol{u}_i = (0.7, -0.4).$$

This is the current two-feature state at token position i.

Step 2: learned MLP update

$$\boldsymbol{m}_i = (0.3, 0.5).$$

This is a separate, shape-matched update chosen to demonstrate residual addition; it is not computed from the state above.

Step 3: add matching coordinates

$$\boldsymbol{h}'_i = \boldsymbol{u}_i + \boldsymbol{m}_i = (0.7, -0.4) + (0.3, 0.5) = \boxed{(1.0, 0.1)}.$$

What if the learned update is zero?

$$\boldsymbol{u}_i + (0, 0) = \boldsymbol{u}_i.$$

◂ Return to residual connections

▸ Next: normalisation

Appendix: layer normalisation, one token row at a time

Use one four-feature token state and omit ε only to keep this teaching arithmetic exact:

$$\boldsymbol{h}_i = (1, 1, 3, 3), \qquad d = 4.$$

1. Mean across the four features

$$\mu_i = \frac{1+1+3+3}{4} = 2.$$

2. Centre the features

$$\boldsymbol{h}_i - \mu_i \mathbf{1} = (-1, -1, 1, 1).$$

3. Variance and standard deviation

$$\sigma_i^2 = \frac{1+1+1+1}{4} = 1, \qquad \sigma_i = 1.$$

4. The normalised row

$$\boldsymbol{z}_i = (-1, -1, 1, 1).$$

5. Learned scale and shift

Let

$$\boldsymbol{\gamma} = (2, 1, 0.5, 1), \qquad \boldsymbol{\beta} = (0, 0.5, 0, -0.5).$$

Then

$$\boldsymbol{\gamma} \odot \boldsymbol{z}_i + \boldsymbol{\beta} = \boxed{(-2, -0.5, 0.5, 0.5)}.$$

Implementations use a small $\varepsilon > 0$ for numerical safety. The calculation runs across width d inside this row—not across positions or the batch.

◂ Return to layer normalisation

▸ Next: the complete block

Appendix: one decoder block, expanded into six steps

1. Normalise the incoming token-state table: $\widetilde{\boldsymbol{H}} = \mathrm{LN}_1(\boldsymbol{H})$.
2. Communicate through multi-head attention: $\boldsymbol{A} = \mathrm{MHA}(\widetilde{\boldsymbol{H}})$.
3. Add the attention update: $\boldsymbol{U} = \boldsymbol{H} + \boldsymbol{A}$.
4. Normalise the updated table: $\widetilde{\boldsymbol{U}} = \mathrm{LN}_2(\boldsymbol{U})$.
5. Transform every row separately: $\boldsymbol{F} = \mathrm{MLP}(\widetilde{\boldsymbol{U}})$.
6. Add the MLP update: $\boldsymbol{H}' = \boldsymbol{U} + \boldsymbol{F}$.

One row: audit two additions with illustrative vectors

$$\boldsymbol{h}_i = (1, -2), \quad \boldsymbol{a}_i = (-0.3, 1.6), \quad \boldsymbol{u}_i = \boldsymbol{h}_i + \boldsymbol{a}_i = (0.7, -0.4),$$

$$\boldsymbol{f}_i = (0.3, 0.5), \qquad \boldsymbol{h}'_i = \boldsymbol{u}_i + \boldsymbol{f}_i = \boxed{(1.0, 0.1)}.$$

These chosen vectors audit residual addition; they are not one full parameter-level forward pass. In a real block, attention can use every allowed row, while the MLP is calculated independently at row $\boldsymbol{i}$.

This is one common pre-normalised block; other arrangements exist.

Return to the two block equations

Next: shape and parameter audit

Appendix: shape and parameter audit for one small block

Use $T = 4$ token positions, width $d = 8$, $h = 2$ heads, head width $d_h = 4$, and MLP width $d_{\text{ff}} = 16$.

current-input activation	shape
$\boldsymbol{H}, \mathrm{LN}_1(\boldsymbol{H})$	4×8
one head's $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$	4×4 each
one head's scores and shares	4×4 each
one head output $\boldsymbol{O}^{(r)}$	4×4
joined heads, MHA output, $\boldsymbol{U}$	4×8
MLP hidden table	4×16
MLP update $\boldsymbol{F}$, output $\boldsymbol{H}'$	4×8

Learned parameters, omitting biases

$$\underbrace{3hd\,d_h}_{Q,K,V} + \underbrace{d^2}_{W_O} = 3(2)(8)(4) + 64 = 256.$$

$$\underbrace{2d\,d_{\text{ff}}}_{\text{two MLP matrices}} = 2(8)(16) = 256.$$

$$\underbrace{2(2d)}_{\text{two normalisations}} = 32, \qquad \text{block total} = \boxed{544}.$$

Parameter: a learned number reused for every input. **Activation**: a value created for this input. Each head's attention table contains T^2 activations, but it is not a learned T^2-parameter table.

◂ Return to depth and preserved shape

▸ Next: Transformer families

Appendix: Transformer family, GPT path, and output shape

family	self-attention rule	typical information route
encoder-only	full: both sides are visible	build a representation from a complete input
encoder–decoder	encoder full; decoder causal	decoder also reads the encoder through cross-attention; the original Transformer used this design
decoder-only	causal: current and earlier positions	continue a sequence; the GPT path taught here uses this family

Repeated decoder stack

$$\boldsymbol{H}^{(\ell+1)} = \mathrm{Block}_{\ell}(\boldsymbol{H}^{(\ell)}), \qquad \ell = 0, \ldots, L-1.$$

ℓ: layer number L: total number of layers $\boldsymbol{H}^{(\ell)}$: token states entering layer ℓ.

Output map $\widetilde{\boldsymbol{H}}$ is the final normalised token-state table.

$$\widetilde{\boldsymbol{H}} W_{\mathrm{out}} + \mathbf{1}_T \boldsymbol{b}_{\mathrm{out}} = \boldsymbol{Z} \in \mathbb{R}^{T \times |\mathcal{V}|}.$$

$|\mathcal{V}|$: vocabulary size row t of $\boldsymbol{Z}$: scores for the token after position t final row: scores for the unseen next token.

In "GPT," T means Transformer. In these equations, T means positions.

◂ Return to decoder-only attention ◂ Return to the GPT bridge

Appendix: the service serialises roles before the Transformer sees tokens

$$\underbrace{\text{application records}}_{\substack{\text{instructions, user, history,}\\ \text{tools, documents}}} \longrightarrow \underbrace{\text{chat template}}_{\text{role markers and ordering}} \longrightarrow \underbrace{\text{token IDs}}_{x_1,\ldots,x_T} \longrightarrow \underbrace{H^{(0)}}_{\text{starting token states}} .$$

record	ordinary-language purpose	important caution
system/developer	application instructions and boundaries	names and precedence are provider-specific
user	current request	lower-priority text should not override higher instructions
assistant history	earlier turns retained for continuity	generated text may contain mistakes
tool/document	external results added by the host	distinguish evidence from embedded instructions

Chat template: provider-specific formatting that converts role-labelled records into one token sequence. Role markers, tool schemas, and formatting also use tokens. "System prompt" is an application convention, not a separate architecture component.

Instruction authority guides the model's response. Authentication, permissions, and approvals still belong to the surrounding host system.

◂ Return to message roles

▸ Next: context accounting

Appendix: audit one finite context budget

Suppose an application allows 4,096 token positions for one request:

component	tokens	why it is present
system and developer instructions	800	workflow guidance
selected conversation history	600	earlier-turn continuity
selected documents	1,200	current evidence
tool definitions and results	400	actions and returned state
reserved space for the new answer	500	positions not yet used
total planned use	3,500	596 positions remain unused

Context accounting count every input component and reserve output space before calling the model.

If the packet exceeds the limit, the host must omit, retrieve, truncate, or compact material. Each choice changes the evidence or instructions available.

A cached prefix still occupies token positions in the request. Caching may reduce repeated computation or price; it does not enlarge the model's context window. Product rules and billing categories vary.

Appendix: current efficiency methods save different resources

method	plain-English mechanism	mainly saves	does not automatically do
prompt or context cache	reuse computation for an unchanged leading prefix	repeated compute, latency, or price	shorten context or create token capacity
on-demand tool loading	insert only relevant tool definitions when needed	input tokens and attention work	guarantee the right tool was selected
compaction or context editing	replace old detail with shorter state	input tokens in later requests	preserve every discarded detail
KV cache during generation	retain earlier keys and values within a request	repeated decoder computation	remove sequential dependence or storage

Stable-prefix rule put reusable instructions and examples before request-specific material when a provider's cache uses exact prefix matching.

Measurement rule compare quality, latency, input tokens, output tokens, and cost separately. A technique can improve one while leaving another unchanged.

Product behaviour changes over time. These distinctions are stable; provider thresholds, prices, cache-retention rules, and tool-search APIs are date-sensitive.

Return

Next: model and host

Day 3 technical reference

Appendix

Appendix: instructions guide the model; controls constrain the system

question	model-side mechanism	host-side control
what should be done?	system/developer instructions and post-trained behaviour	application policy and workflow logic
what information is available?	token context supplied to the model	retrieval, database access, and context selection
may this action occur?	model can propose or decline a tool call	authentication, authorisation, approvals, and allow-lists
what actually happened?	model reads a returned result if supplied	tool execution, error handling, logs, and audit records

One safe tool loop

request packet ⟶ model proposal ⟶ host validation
⟶ tool execution ⟶ returned result ⟶ next response.

The host may reject, modify, or require human approval for a proposed action. Treating a prompt as a security boundary confuses behavioural guidance with enforceable control.

Return

Return to the closing limits