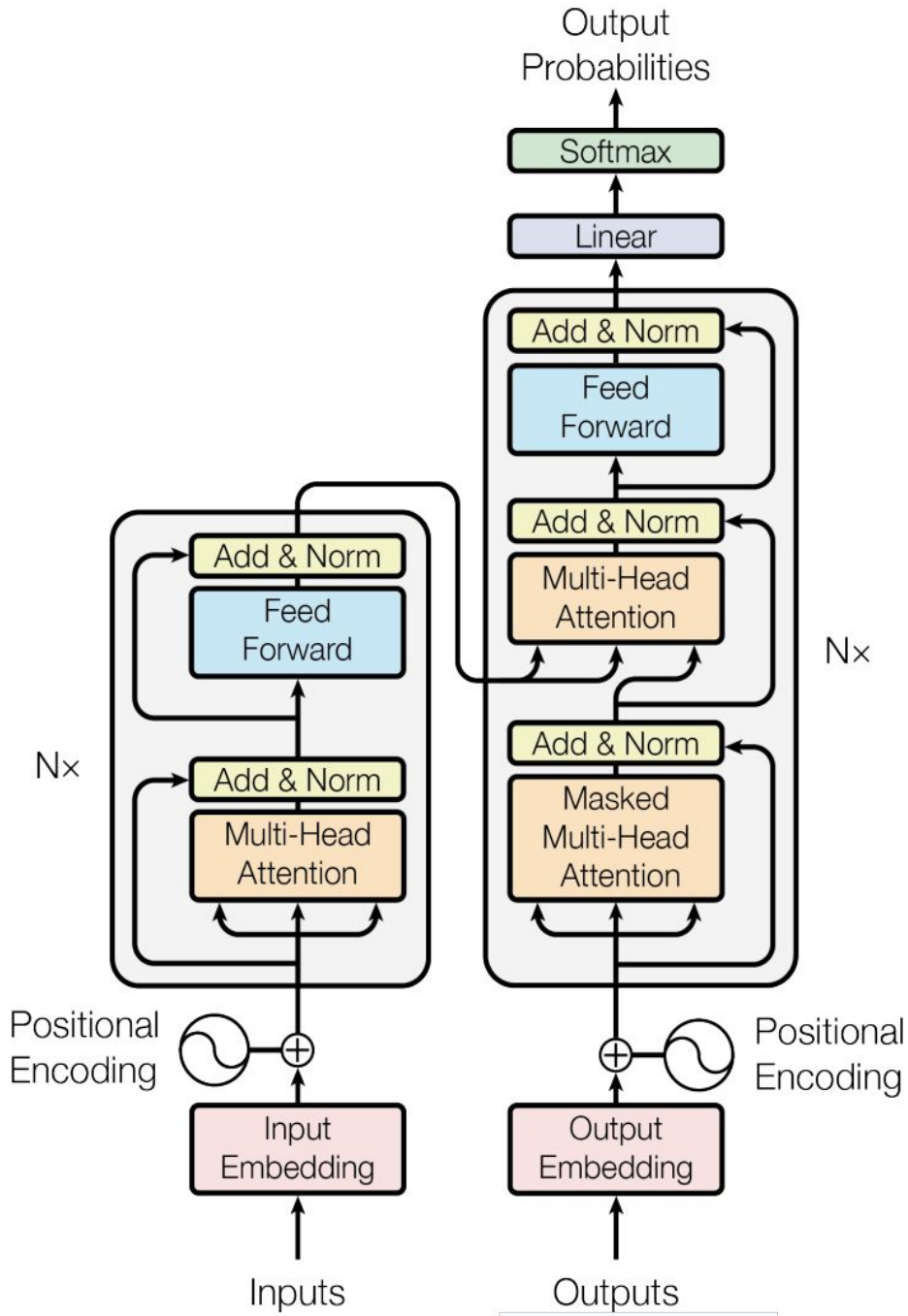
Deep Network Development Lecture 11

Transformers and variants, basic NLP topics, Visual Transformers, Advantages

Presented by:
Arron Pirku

BERT

Encoder



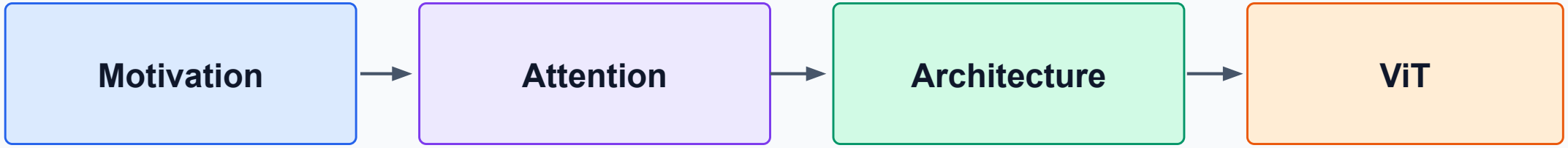
GPT

Decoder

Transformer Models

From subword tokenization to self-attention, masking, encoder–decoder models, and Vision Transformers

Lecture focus: why each component exists, not only what the equations say.



Motivation: Why Transformers?

- Before studying the Transformer architecture itself, we need to understand the problem it was designed to solve. Many types of data are naturally sequential: text, speech, video, DNA, time series, and even images when represented as patches.

- The central difficulty is that sequence models must handle order, context, and variable length. In language, the meaning of a token often depends not only on nearby tokens, but also on tokens much earlier or later in the sequence.

- Guiding question: How can we represent and process sequences efficiently while still preserving meaningful context?

From Text to Tokens

- A Transformer cannot directly process raw text. First, the text must be converted into a sequence of tokens, and then each token must be converted into a vector.

- Character-level tokenization avoids unknown words but creates very long sequences. Word-level tokenization creates shorter sequences but huge vocabularies and many rare or unseen forms.

- Subword tokenization, such as Byte Pair Encoding, is a compromise: it keeps sequences shorter than characters while remaining flexible enough to represent rare words, inflections, names, and new terms.

Sequences appear everywhere

The same architectural ideas apply beyond text.

Text	Speech	Video	DNA	Images
words / subwords	frames / features	image frames	symbols	patch tokens

Transformer models are sequence processors. Their first question is always: what are the elements of the sequence, and how are those elements represented as vectors?

Order matters: “dog bites man” is different from “man bites dog”.
Context matters: the meaning of a token depends on nearby and sometimes far-away tokens.
Length varies: inputs can have 5 tokens, 500 tokens, or far more.

Naive representations: characters or words

Useful starting point, but not ideal for modern language models.

Character tokens avoid unknown words but create long sequences.
Word tokens create shorter sequences but produce huge vocabularies and many unknown or rare forms.
Subword tokens are a compromise: reusable pieces that shorten sequences while preserving morphology and rare-word information.

Subword Tokenization and BPE

- Subword tokenization represents frequent pieces as single units and rare words as combinations of reusable units.
- Byte Pair Encoding starts from characters or bytes, repeatedly merges the most frequent adjacent pair, and builds a fixed vocabulary of useful word pieces.
- This is important for LLMs because it reduces sequence length compared with characters, avoids many unknown-word problems compared with full words, and preserves surface-form information that stemming or lemmatization may destroy.

Why character-level tokenization is costly

Transformers compare many token pairs, so sequence length matters a lot.

Example:

“The transformer generalized to unseen words.”

Character-level representation may produce dozens of tokens even for a short sentence.
Attention cost is approximately quadratic in sequence length: comparing 40 tokens with 40 tokens is much cheaper than comparing 4000 with 4000.
Very long character sequences make context windows expensive and make learning long-range structure harder.

[T, h, e, _, t, r, a, n, s, f, o, r, m, e, r, _, g, e, n, e, r, a, l, i, z, e, d, _, t, o, _, u, n, s, e, e, n, _, w, o, r, d, s, .]

Discussion

“If we double the number of tokens, what happens to the number of token-pair comparisons?”

Answer:
The number of token-pair comparisons becomes **four times larger**.
In self-attention, every token is compared with every other token, so the number of comparisons grows roughly as: n^2

Example:
100 tokens → 10,000 comparisons
200 tokens → 40,000 comparisons

Why word-level tokenization is not enough

Words look simple, but language is full of rare, inflected, and newly created forms.

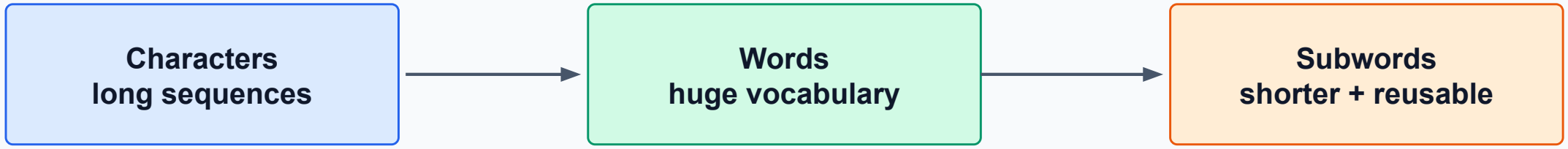
A word vocabulary can become enormous, especially in morphologically rich languages.
Rare words receive weak training signal; unseen words become out-of-vocabulary tokens.
Classical preprocessing such as stemming or lemmatization can reduce vocabulary size, but it may discard information needed by modern generative models.
LLMs often need the exact surface form: punctuation, endings, names, code fragments, spelling, and formatting can matter.

Problem: “normalize too much” and the model loses information.

Problem: “normalize too little” and the model sees too many rare words.

Subword tokenization: the compromise

The goal is to represent frequent pieces as units and rare words as combinations of units.

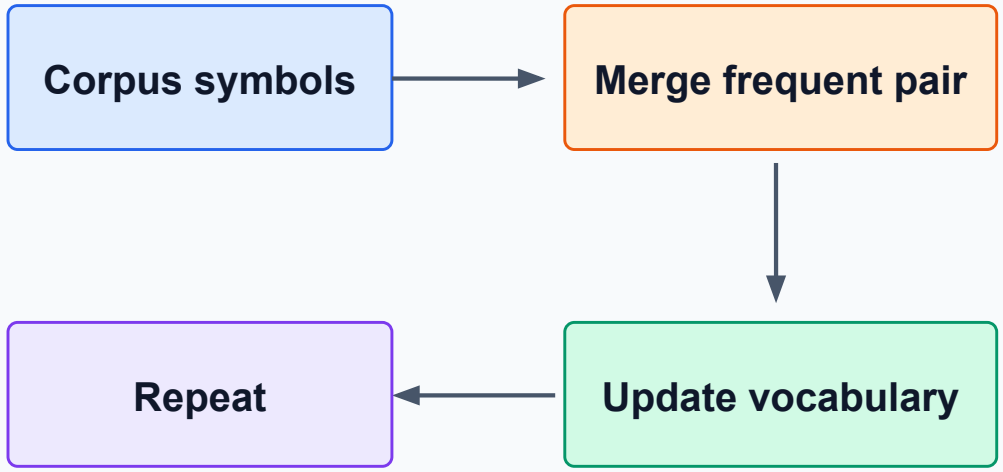


Frequent words may become single tokens.
Common prefixes, suffixes, and word pieces can be reused across many words.
Rare words can still be represented rather than replaced by an unknown token.
The model can learn patterns at a useful granularity: often larger than characters but smaller than full words.

Byte Pair Encoding (BPE): core idea

Repeatedly merge the most frequent adjacent pair.

Start from a very small alphabet: characters or bytes.
Count adjacent symbol pairs in a training corpus.
Merge the most frequent pair into a new symbol.
Repeat many times until the target vocabulary size is reached.
At inference time, apply the learned merges to split new text into known subword tokens.



BPE example on a small phrase

The exact merges depend on corpus statistics, but the principle is simple.

Initial characters:

t h e _ t r a n s f o r m e r _ g e n e r a l i z e d _ t o _ u n s e e n _ w o r d s

Frequent pairs:

t h → th e r → er e d → ed u n → un

After merges:

[_the, _transform, er, _general, ized, _to, _un, seen, _words]

The sequence becomes shorter than character-level tokenization.
The vocabulary remains much smaller and more flexible than full-word tokenization.
The model keeps surface-form information that stemming or lemmatization might remove.

Limits of Recurrent Models

- Before Transformers, recurrent neural networks were a natural choice for sequence modeling. They process tokens one by one and carry information forward through a hidden state.
- This left-to-right structure naturally prevents future-token leakage, but it also creates limitations: computation is hard to parallelize across time, long-distance dependencies are difficult to preserve, and important early information must fit into a limited hidden state.
- In encoder–decoder RNNs, the bottleneck becomes especially clear: one vector may need to summarize an entire input sequence.

Why LLMs usually prefer subword tokens

This is the practical motivation before we move to Transformers.

Shorter than characters: lower attention cost and longer usable context.
More robust than words: rare names, typos, inflections, and new terms can be built from known pieces.
Less destructive than stemming: the model can still see endings, punctuation, formatting, and exact forms.
Language-independent enough to work across many scripts, domains, and even code.
A fixed vocabulary becomes possible while still covering almost any input string.

Discussion

“What information might be lost if we stem or lemmatize everything before training a generative model?”

Answer:
If we stem or lemmatize everything, we lose important surface-form information that a generative model needs in order to produce natural text. For example, the model may lose information about tense, number, gender, case, person, word form, and sometimes even style or tone.

“The students were running”
may become something like
“the student be run”

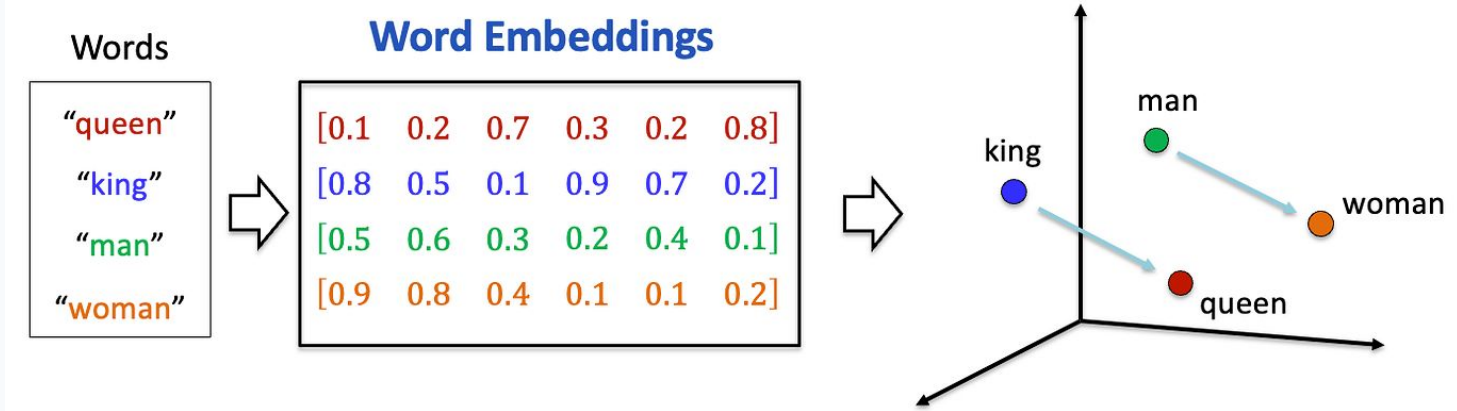
Word embeddings: only a brief bridge

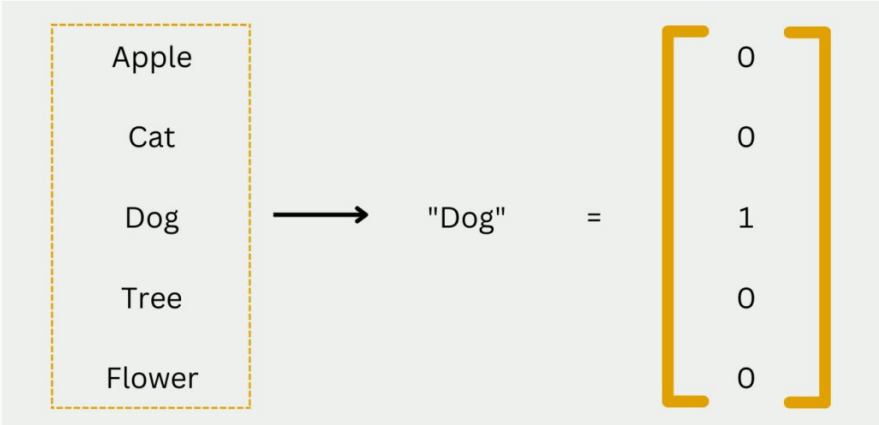
Word2Vec is useful historically, but Transformers learn contextual token representations.

Classic embeddings assign each word or token a learned vector.
Similar vectors can capture semantic or syntactic similarity.
Word2Vec-style embeddings are static: a word has one vector before seeing its sentence context.
Transformers begin with token embeddings too, but then repeatedly update each token vector using context through self-attention.
In classic embeddings, every word or token is represented by a vector, which is just a list of numbers. The idea is that words with similar meanings or similar grammatical behavior should have similar vectors. For example, the vectors for **king** and **queen** may be close, and the vectors for **run**, **walk**, and **jump** may also be close.
In Word2Vec-style embeddings, the vector is **static**. That means a word usually has the same vector every time it appears, no matter where it appears.

Static embedding
“bank” has one base vector

Contextual representation
“bank” depends on the sentence





RNN next-token prediction

Before masking, recall how a unidirectional sequence model avoids cheating.

Input: [<START>, t1, t2, t3, t4]

Target: [t1, t2, t3, t4, t5]

For an N → N next-token task, the target sequence is shifted by one position.
A unidirectional RNN processes left to right.
At time step i, information from future tokens has not yet been processed.
Therefore the model cannot simply copy the answer from the future.

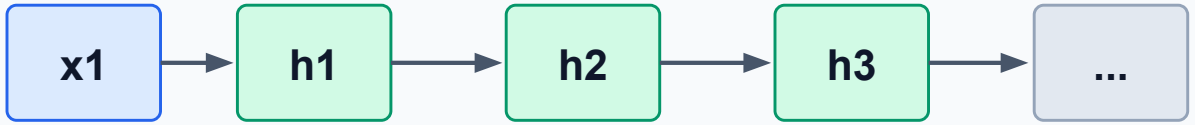
Discussion

“Why is the shifted target not trivial for a left-to-right RNN?”

Answer:
The shifted target is not trivial for a left-to-right RNN because the correct target token is always one step in the future, and the model has not seen that future token yet. Since information only flows forward through time, the model must learn patterns from the previous context instead of simply copying the answer.

RNNs are strong but sequential

The hidden state is the memory carried forward through time.

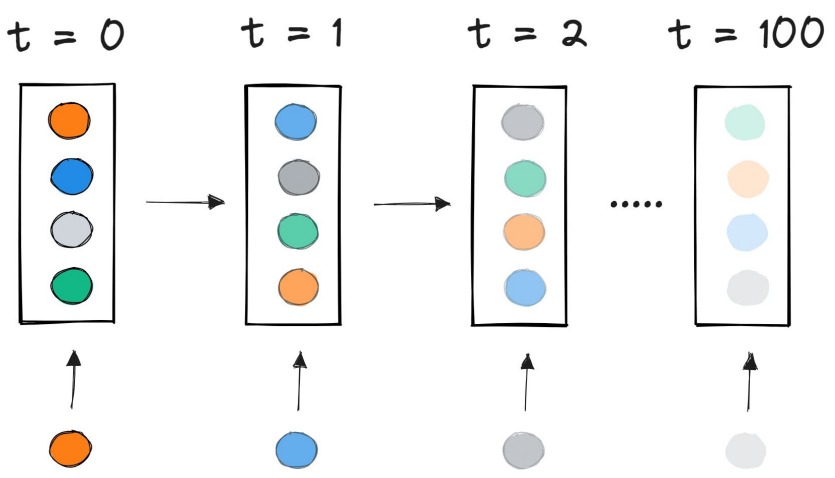


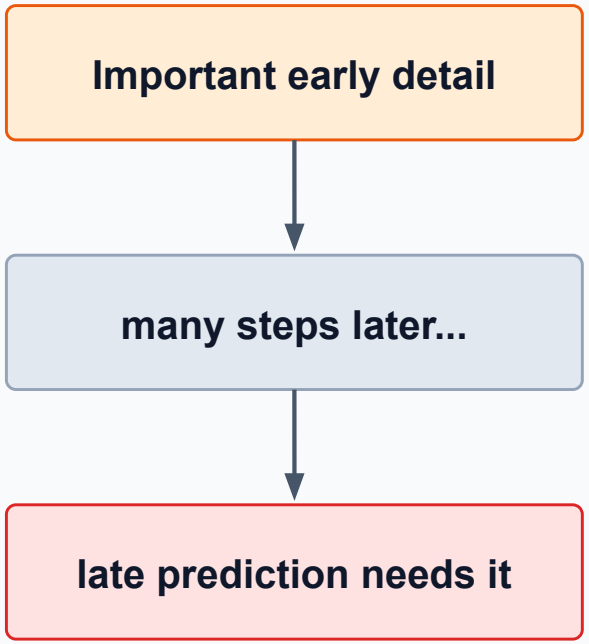
The same recurrent cell is reused at each step.
This parameter sharing makes RNNs natural for variable-length sequences.
But each step depends on the previous step, so computation is hard to parallelize across time.
Long dependencies must be carried through many state updates.

Why RNNs struggle with long context

The path from an early token to a late prediction can be very long.

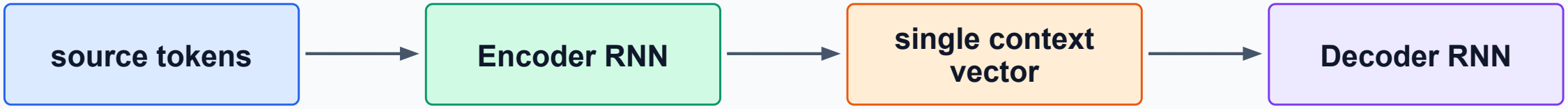
Long gradient paths can cause vanishing or exploding gradients.
A fixed-size hidden state must decide what to remember and what to discard.
If the model later needs an early detail, there is no direct way to “look back” unless it was stored in the state.
Evaluation of one sequence is inherently sequential: step t must wait for step t−1.





Seq2Seq: the bottleneck problem

Classic encoder–decoder RNNs compress the source sequence into one vector.



The encoder must summarize everything useful into one compact representation.
For short sentences this can work; for long sentences it becomes a narrow information channel.
The decoder may need different source words at different output steps.
Attention was introduced to let the decoder access encoder states directly.

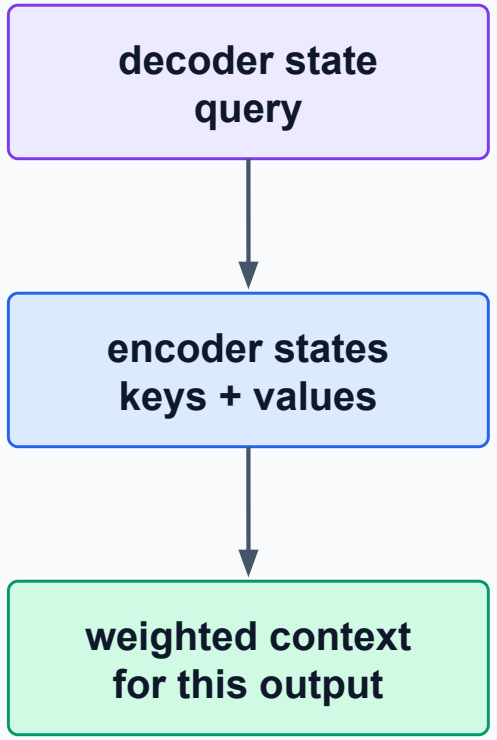
Attention: The Core Idea

- Attention changes how a model uses context. Instead of forcing the model to store everything in one fixed-size vector, attention allows the model to look back at different parts of the input whenever it needs them.
- At each step, the model asks: which tokens are most relevant right now? The answer is represented by attention weights.
- In translation, attention can be understood as soft alignment: different output words may focus on different source words.

Attention: a better context vector

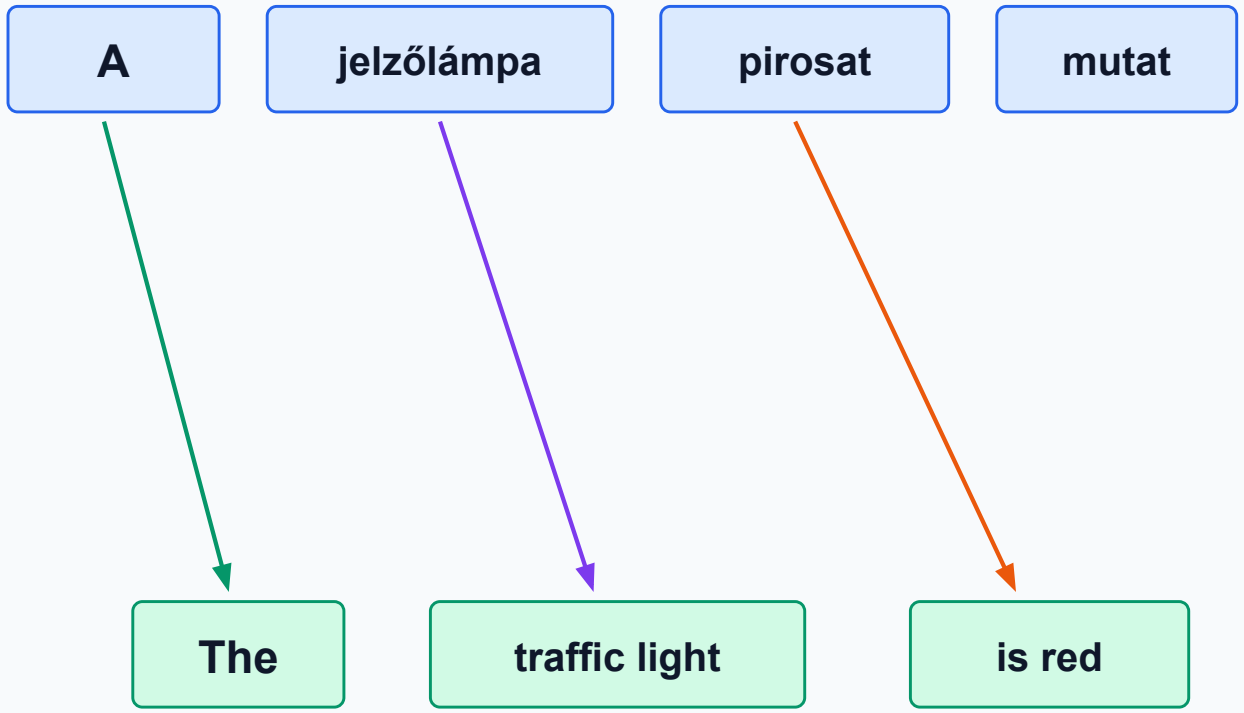
Instead of one fixed summary, compute a context that changes at each decoding step.

At each output step, ask: which input positions are most relevant now?
The model assigns an attention weight to each encoder state.
The context is a weighted sum of encoder states.
This creates a direct path from useful input positions to the current output prediction.



Attention as “soft alignment”

In translation, different output words align with different input words.



The attention matrix can be interpreted as an alignment map: high weights show which source tokens the model uses for each target token.

Query, Key, and Value

- Transformer attention is commonly explained using three roles: query, key, and value.

- The query represents what a token is looking for. The key represents what each token offers for comparison. The value represents the information that will be retrieved and mixed after relevance is computed.

- The mechanism compares queries with keys, normalizes the scores into weights, and uses those weights to combine the values.

Attention formula in words

The mechanism has three conceptual steps.

1. Score
How compatible is this query with each key?

2. Normalize
Turn scores into weights with softmax.

3. Mix values
Return a weighted sum of value vectors.

In compact notation:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Q = questions we ask from each position.
K = descriptors used to decide which positions match a query.
V = information we retrieve and mix once weights are known.

Discussion

"Why do we divide the query–key dot product by $\sqrt{d_k}$ in scaled dot-product attention?"

Answer:

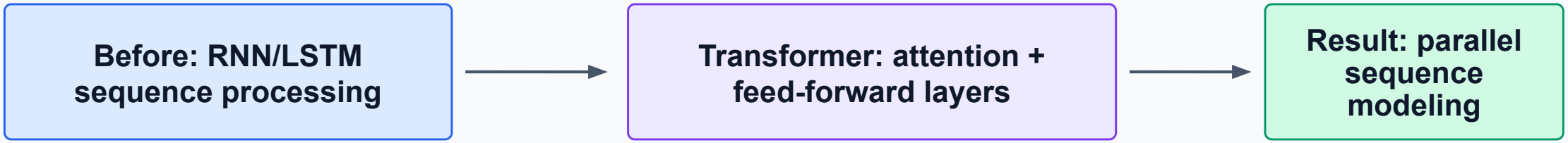
Because when the dimension d_k of the query and key vectors becomes large, their dot products can also become large. Large attention scores make the softmax function too sharp, so one token may receive almost all the attention weight. This can lead to very small gradients and unstable training.

Dividing by $\sqrt{d_k}$ keeps the attention scores in a more stable range before applying softmax.

It resembles vanishing gradients, but it is not the same as the long-chain vanishing gradient problem in RNNs.

“Attention Is All You Need”

Vaswani et al. replaced recurrent processing with stacked attention blocks.



The key idea is not just “use attention”.
The key idea is to make attention the main sequence-processing mechanism.
The model still needs token embeddings, positional information, residual paths, normalization, and feed-forward transformations.

Why remove recurrence?

Transformers trade sequential state updates for pairwise token interactions.

Parallelism: all positions in a layer can be processed together.
Shorter information paths: any token can directly attend to any other token in one layer.
Flexible context: the model can decide which tokens matter based on content.
Better scaling: large matrix operations are efficient on modern hardware.

Trade-off

Full self-attention compares every token with every other token. This gives flexibility, but the cost grows roughly as L^2 with sequence length.

Query–Key–Value intuition

A search engine analogy helps students remember the roles.

Query
What am I looking for?

Key
What does each item advertise?

Value
What information do I retrieve?

The query and key are compared to produce attention scores.
The values are not used to decide relevance; they are what gets mixed after relevance is computed.
In self-attention, Q, K, and V are all derived from the same input sequence using learned linear projections.

Discussion

“In a library search, which part is the search phrase, which part is the book metadata, and which part is the book content?”

Answer:
In the library search analogy:
Query = the search phrase
Example: *“books about neural networks”*
Key = the book metadata used for matching
Example: title, keywords, author, subject tags, abstract
Value = the actual book content or information retrieved
Example: the book itself, the relevant chapter, or the text returned to the reader

The query is what we search for, the keys are what we compare against, and the values are the information we retrieve after finding good matches.

Scaled dot-product attention

The standard Transformer attention operation.

$$\text{Attention}(Q,K,V) = \text{softmax}((QK^T) / \sqrt{d_k}) V$$

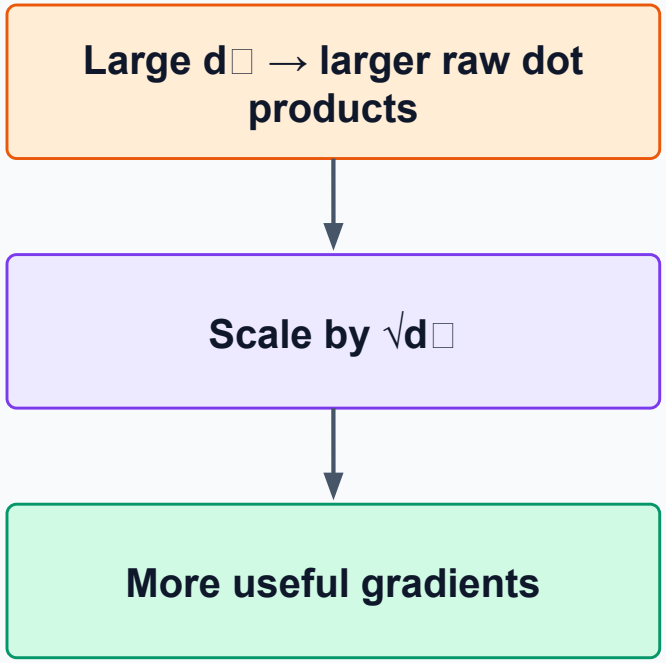
QK^T computes all pairwise query–key similarities.
Dividing by $\sqrt{d_k}$ stabilizes score magnitudes when vector dimension grows.
Softmax turns the scores for each query into a probability-like distribution over keys.
Multiplication by V returns the weighted mixture of value vectors.

scores → weights → mixed values

Why divide by √d□?

Without scaling, large dot products can make softmax too sharp.

Dot products tend to grow in magnitude as vector dimension increases.
Very large positive and negative scores push softmax toward almost one-hot outputs.
When softmax saturates, gradients become small and learning can become unstable.
Scaling keeps score magnitudes in a range where softmax is more trainable.

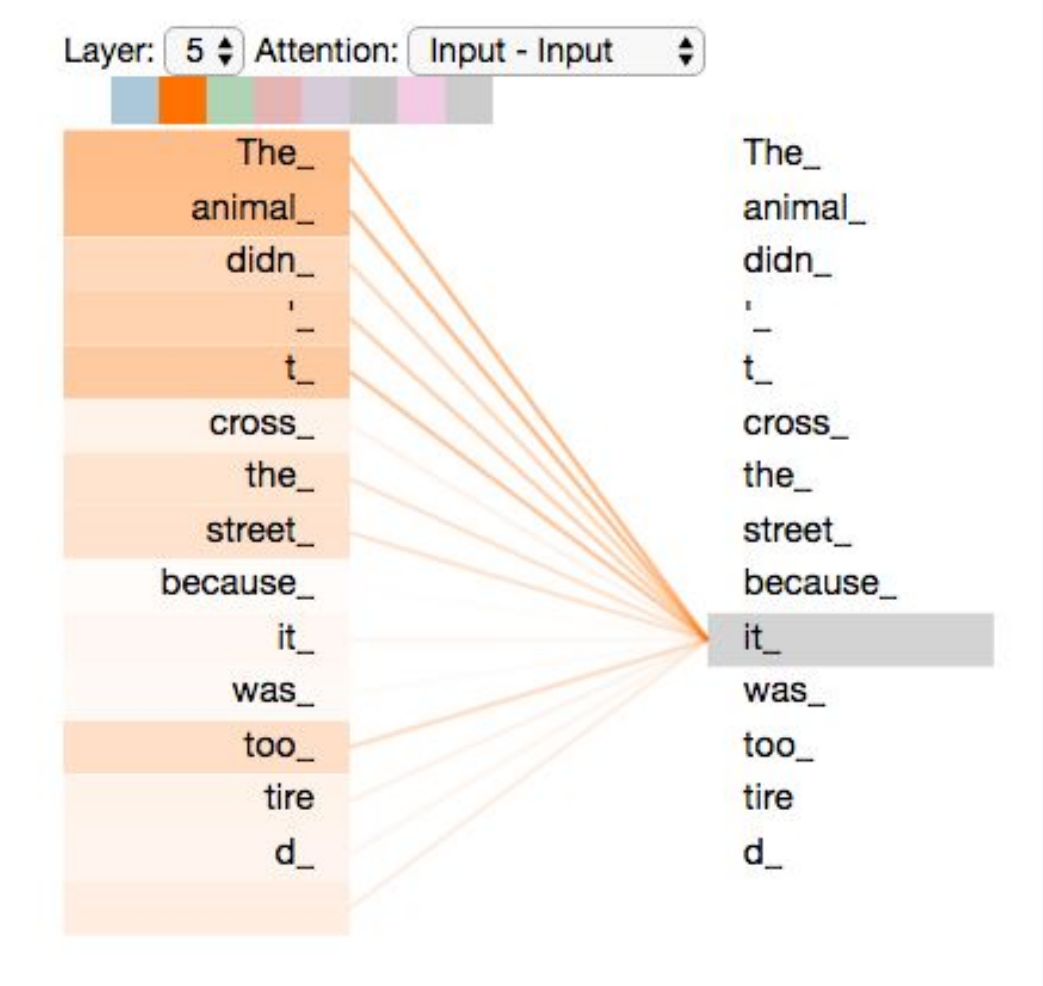


Self-Attention and Multi-Head Attention

- Self-attention is the central operation of the Transformer. Every token can compare itself with every other token in the same sequence.
- This allows the model to build context-dependent token representations. A token can attend to a nearby syntactic relation or to a distant phrase that changes its meaning.
- Multi-head attention repeats this process in parallel. Different heads can learn different relationship types, such as local syntax, long-range dependencies, entity references, or phrase structure.

Self-attention: the core Transformer layer

Every position asks which other positions are useful for updating itself.



Self-attention is not a fixed local window; relevance is content-dependent.
A token can attend to nearby syntax or far-away semantic context.
Stacking layers lets the model build increasingly abstract contextual representations.

Attention weights are data-dependent

This is a major difference from fully connected or convolutional layers.

Learned projection weights WQ, WK, WV are fixed after training.
Attention weights are recomputed for every input sequence.
Therefore the layer adapts its mixing pattern to the content of the current example.
This is why attention can connect distant words when they are semantically related, while ignoring closer but irrelevant tokens.

Fixed learned weights
model parameters

Dynamic attention weights
computed from input

Multi-head attention

One attention pattern is not enough; different heads can focus on different relations.

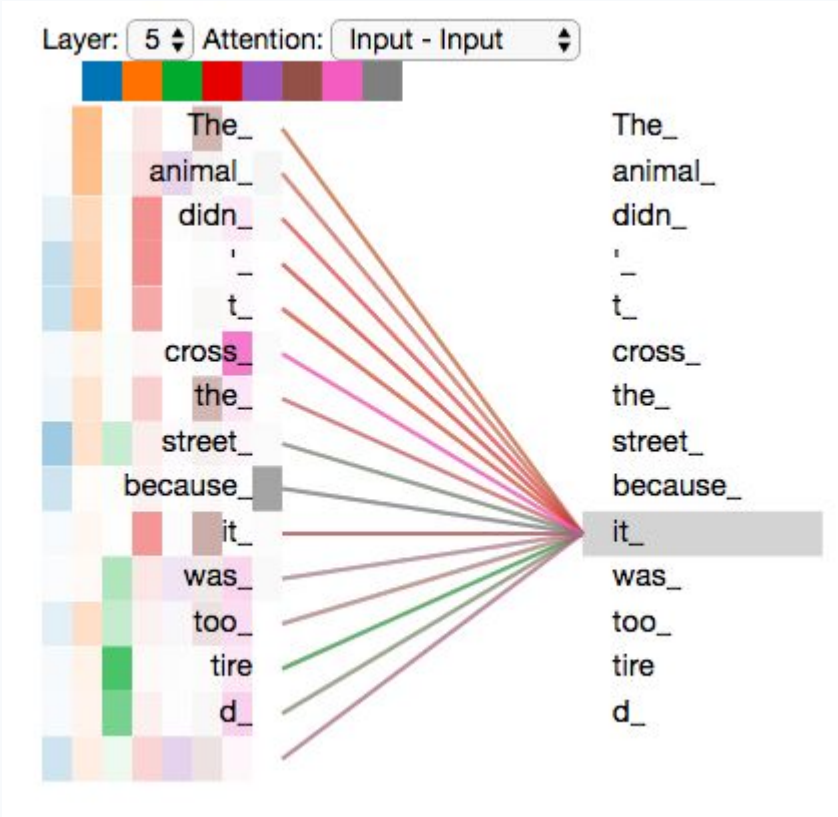
Head 1
syntax / local relation

Head 2
long-range dependency

Head 3
entity relation

Head h
other feature

Each head has its own learned projections for Q, K, and V.
Heads run in parallel on lower-dimensional subspaces.
Outputs are concatenated and projected back to the model dimension.
The model can represent multiple relationship types at the same layer.

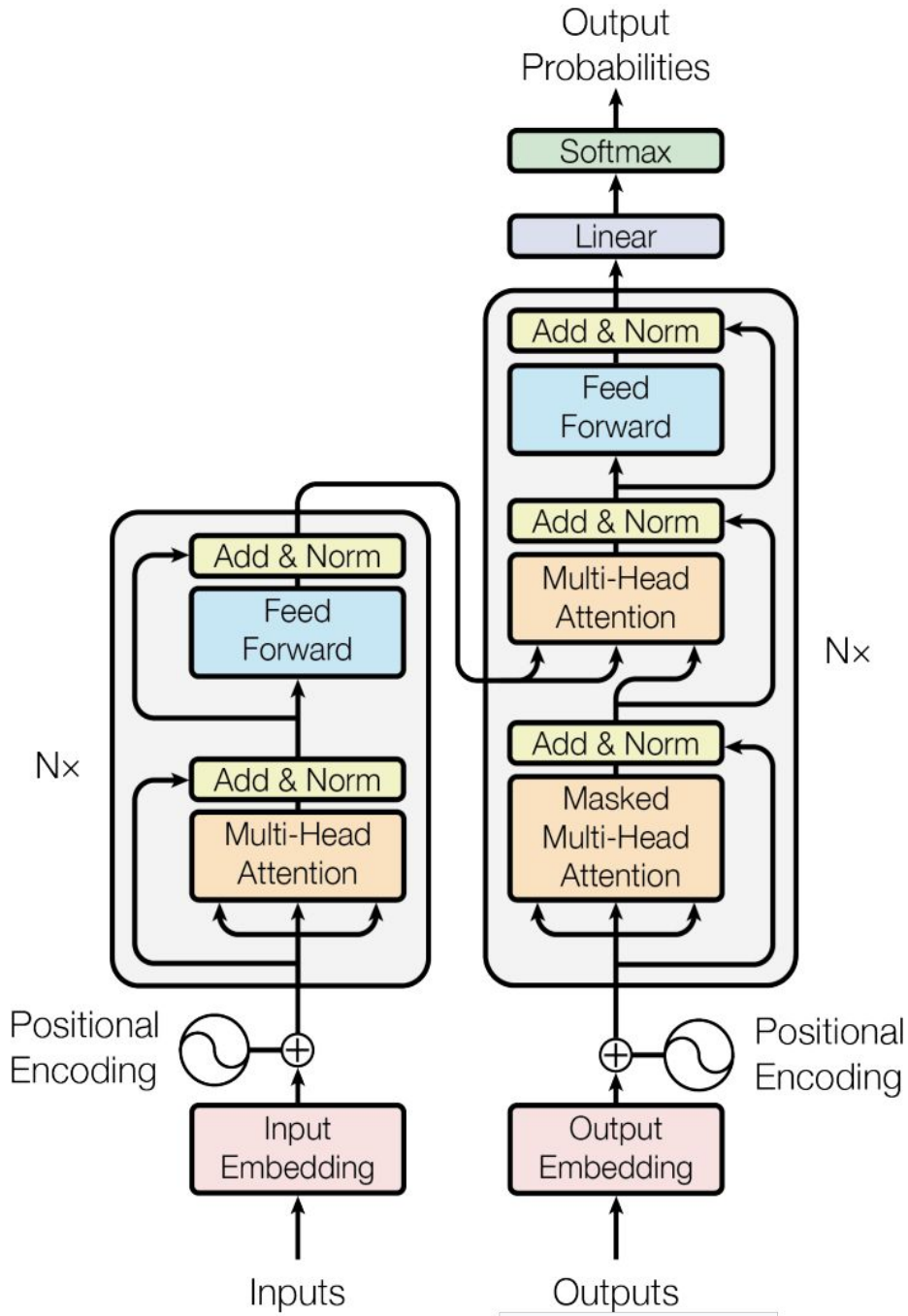


The Transformer Block

- A Transformer is not only attention. A full block combines multi-head attention, a feed-forward network, residual connections, and normalization.
- Attention mixes information between positions. The feed-forward network transforms the features inside each token vector. Residual connections help information and gradients flow through many layers. Layer normalization stabilizes training.
- By stacking many such blocks, the model builds increasingly rich contextual representations.

BERT

Encoder



GPT

Decoder

A Transformer block: two kinds of work

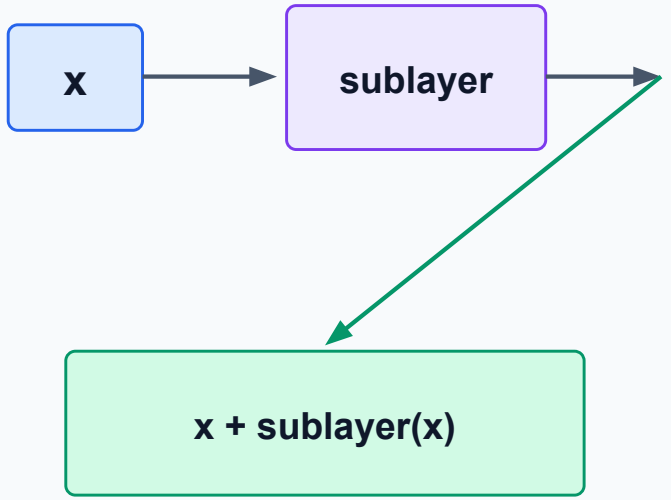
Attention mixes information across positions; FFN mixes features within each position.

Attention answers: which other tokens should this token use?
The feed-forward network answers: after context is gathered, how should each token vector be transformed?
Residual connections and normalization make deep stacks trainable.

Residual connections and LayerNorm

They are not decorative; they stabilize deep training.

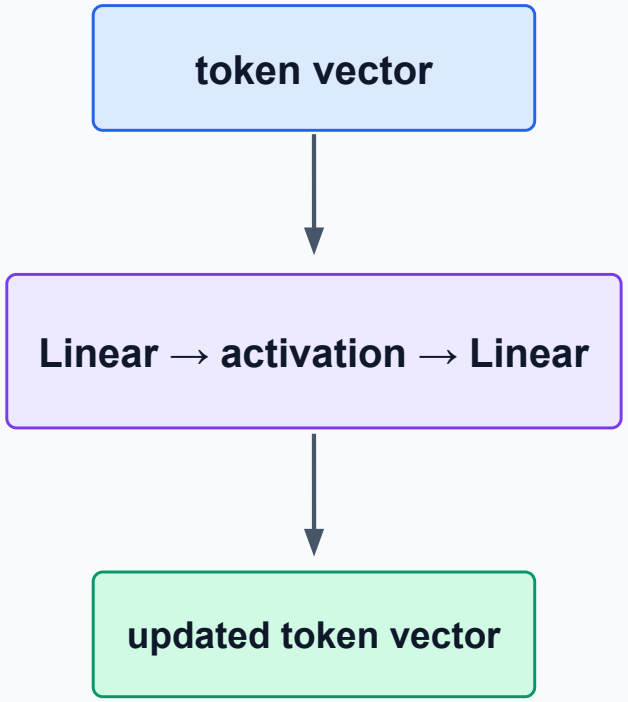
Residual connections provide shortcut paths for information and gradients.
They make it easier for a layer to learn a refinement instead of a complete replacement.
LayerNorm normalizes each token vector, helping stabilize activation scales.
Together they make it practical to stack many attention and feed-forward blocks.



Feed-forward sublayer

Every token receives the same small MLP independently.

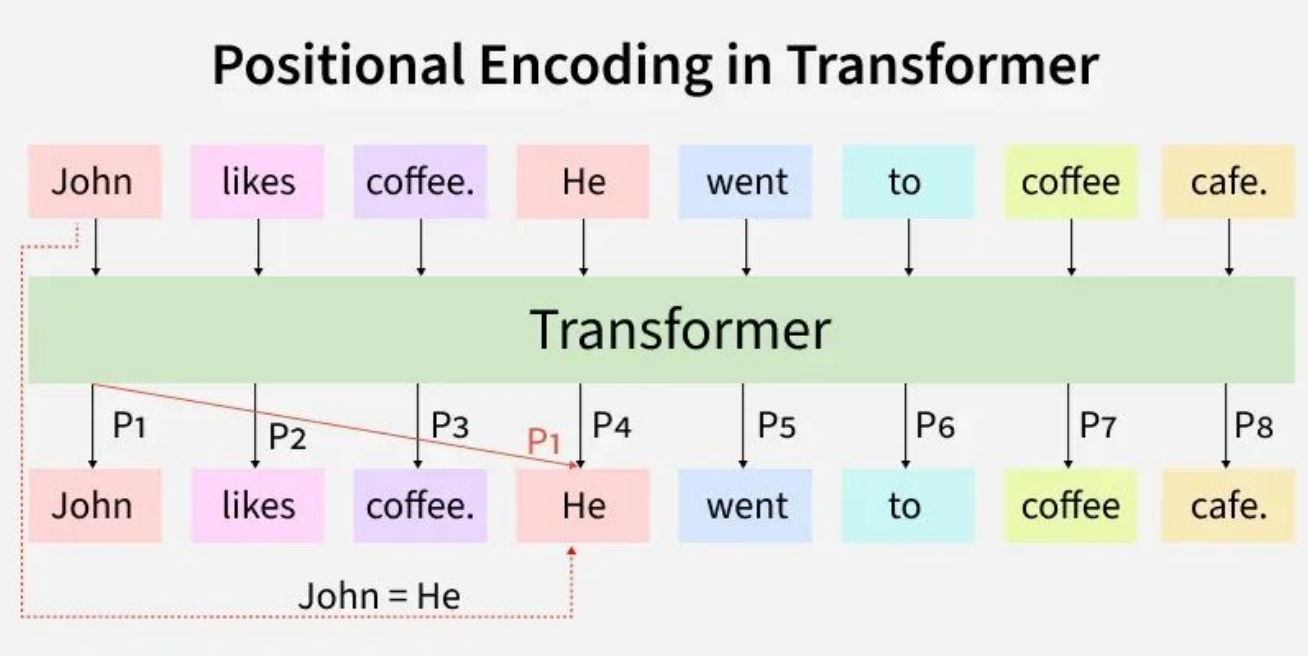
Self-attention mixes information between positions.
The feed-forward network then transforms each position independently.
The same FFN weights are shared across positions, like applying the same operation to every token.
This sublayer increases expressiveness: attention can gather context, and the FFN can process the gathered features.

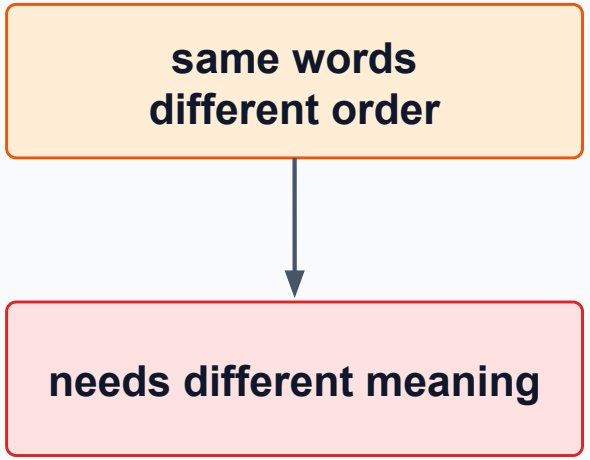


Why positional encoding is needed

Self-attention alone is permutation-equivariant: it does not know original order.

If we permute input tokens, plain self-attention permutes outputs in the same way.
There is no built-in notion of first, second, previous, next, or distance.
But language and time series depend strongly on order.
Therefore we must inject position information into token

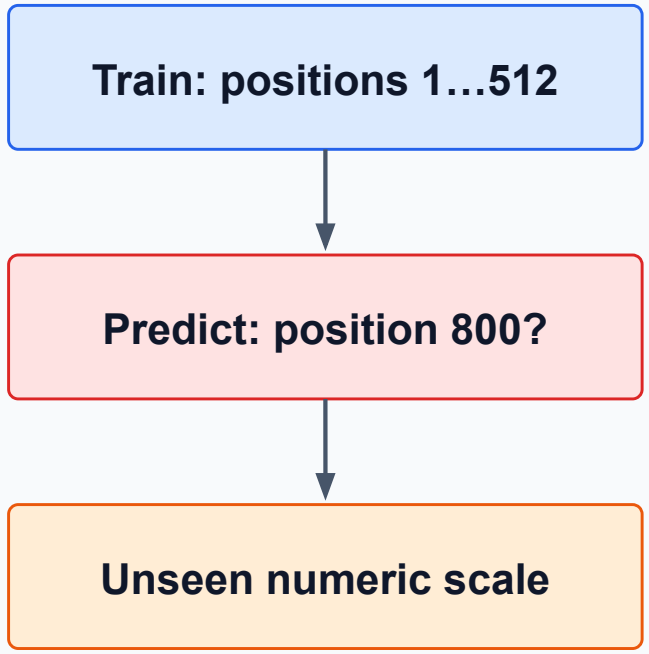




Bad idea #1: just use raw indices

What if we append token index 1, 2, 3, … to the embedding?

During training, the model may only see sequences up to some maximum length.
At prediction time, longer sequences can produce indices outside the training range.
Those unseen index values are out-of-distribution inputs for the network.
The model has no guarantee that it will extrapolate sensibly to positions it never saw.



Bad idea #2: normalize positions to [0,1]

This avoids unseen values, but loses absolute distance information.

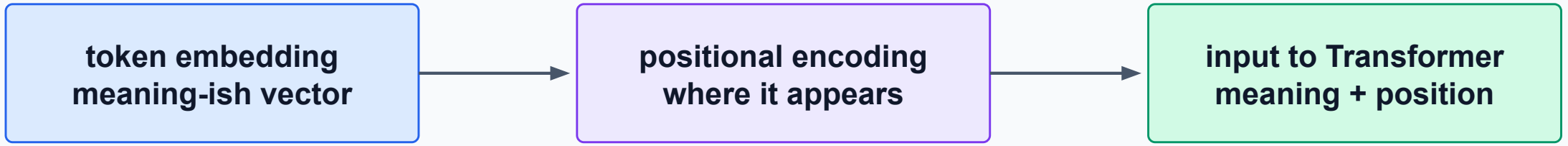
Now every sequence uses values between 0 and 1.
A 10-token sequence has neighbor steps of about 0.1.
A 1000-token sequence has neighbor steps of about 0.001.
The same numerical difference means different absolute distances in different sequence lengths.
The model cannot reliably know what “neighboring token” means in absolute terms.

Length 10
Δ ≈ 0.1

Length 1000
Δ ≈ 0.001

How positional vectors enter the model

The position vector is added to, or sometimes concatenated with, the token embedding.

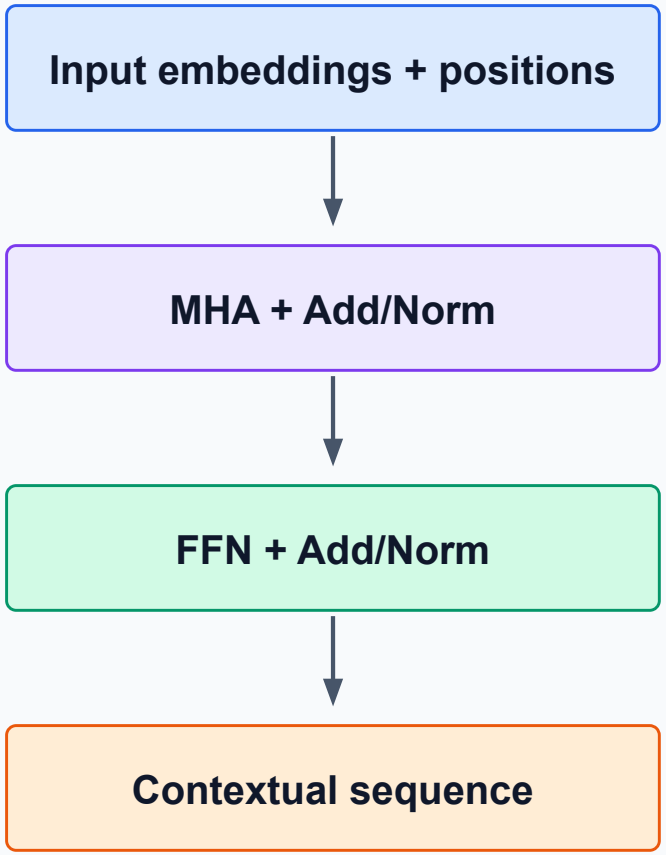


The model sees a vector that contains both token identity and position.
Learned positional embeddings are also common; sinusoidal encodings were used in the original Transformer.
The important principle: self-attention needs explicit order information from somewhere.

Encoder block: bidirectional context

The encoder reads the full input sequence at once.

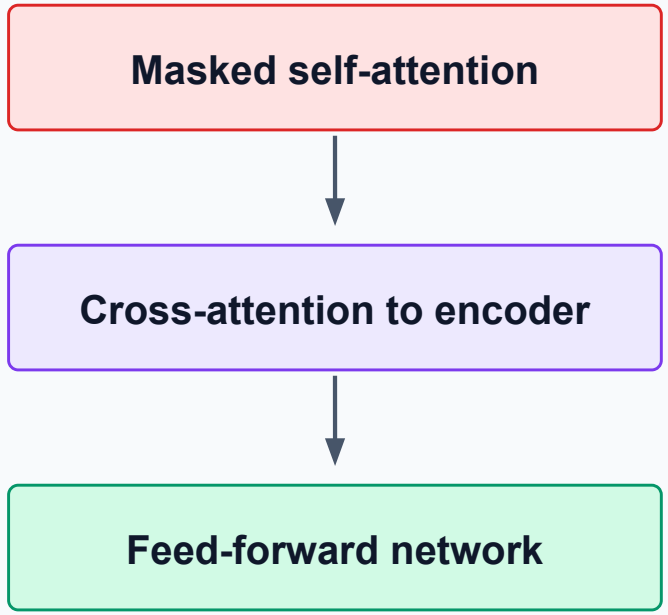
Encoder self-attention is not causally masked.
Each input token can attend to tokens on both its left and right.
This is useful when the entire input is known, such as source text in translation or a sentence for classification.
The final encoder output is a contextualized sequence used by later layers or by the decoder.



Decoder block: causal generation

The decoder generates output tokens left to right.

Decoder self-attention is masked so a position can only use earlier output tokens.
The decoder may also use cross-attention to read the encoder output in translation-style tasks.
During training, teacher forcing allows the whole shifted target sequence to be processed in parallel.
During inference, generated tokens are fed back step by step.



Why masking is needed

Self-attention can otherwise leak the answer from future tokens.

Input: [<START>, _fáj, ni, _fog, _a]

Target: [_fáj, ni, _fog, _a, _fogam]

In an RNN, future tokens cannot influence earlier outputs because processing is left-to-right.
In full self-attention, every position can send information to every other position.
For next-token prediction, this creates a trivial solution: copy the token that should be predicted.
A causal mask blocks attention from a token to future positions.

Masking compared with RNNs

Masking restores the “no future information” rule.

Unidirectional RNN
Information flows left → right

Masked self-attention
Allowed edges only to previous tokens

The model may use token 1 to predict token 2.
The model may use tokens 1…i to predict token i+1.
The model may not use token i+1 to predict token i+1.
This preserves the difficulty of autoregressive prediction while retaining parallel training.

Discussion

“What would happen if the target token were visible to the position that predicts it?”

Answer:
The model would cheat by seeing the answer during training, giving unrealistically good training performance but failing during real generation.
If the target token were visible, the task would become **trivial**.
The model could simply **copy the correct answer** instead of learning to predict it from previous context.
Ex: Input t3, Target to predict t4 If self-attention allows this position to look at t4 then the model does not need to learn language structure, grammar, or meaning. It can just attend to the future token and copy it.
That is why we use **causal masking** in decoder-style Transformers. The mask blocks each position from seeing future tokens, so the model must predict the next token using only the tokens before it.

Causal mask implementation

Set forbidden attention scores to a very negative value before softmax.

Allowed attention pattern for positions 1…5:

✓	×	×	×	×
✓	✓	×	×	×
✓	✓	✓	×	×
✓	✓	✓	✓	×
✓	✓	✓	✓	✓

Rows are query positions; columns are key positions.
The upper triangle is masked because those keys are in the future.
After softmax, masked positions receive attention weight zero.

Mask before softmax → impossible future attention

Training with teacher forcing

Parallel training is possible even though inference is autoregressive.

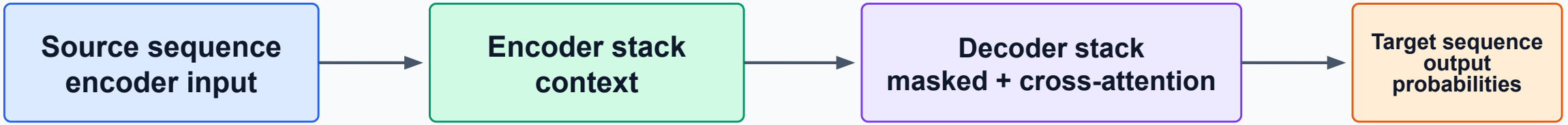
During training, the correct previous tokens are supplied as decoder input.
The target sequence is shifted by one position.
Causal masking prevents access to the current or future target tokens.
During inference, the model uses its own previous predictions, so errors can propagate.

Training
full shifted target available + mask

Inference
generate one token at a time

Encoder–decoder Transformer

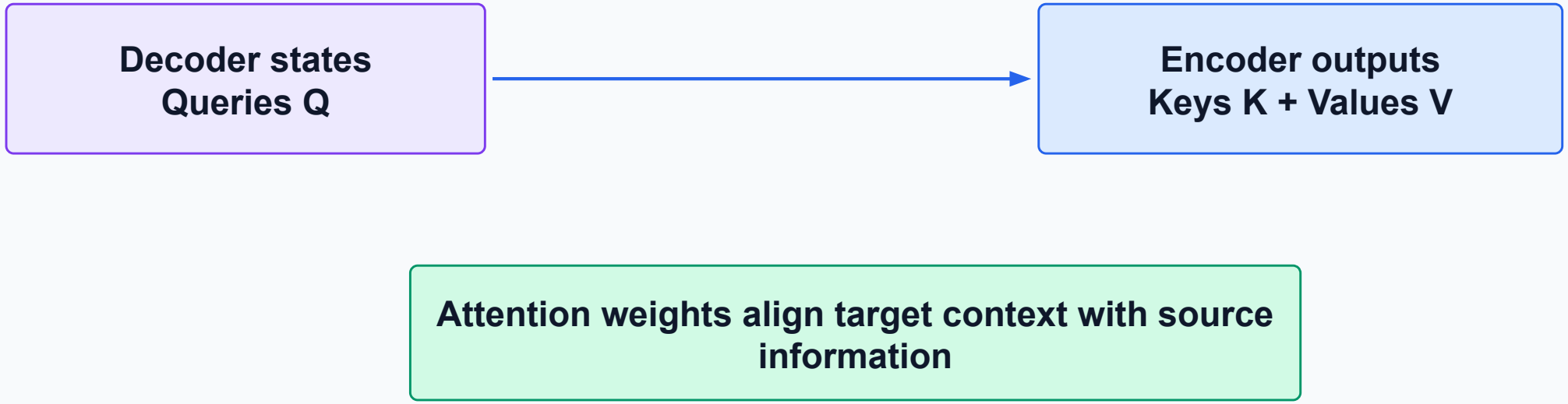
Used when input and output are different sequences, such as translation.



The encoder reads the source bidirectionally.
The decoder reads generated target tokens causally.
Cross-attention connects target-side queries to source-side keys and values.

Cross-attention: connecting two sequences

The decoder asks questions; the encoder provides keys and values.



Self-attention: Q, K, V come from the same sequence.
Cross-attention: Q comes from one sequence; K and V come from another.
In translation, this lets each generated target token retrieve the relevant source-side information.

Transformer Model Families

- Different Transformer models keep different parts of the original architecture depending on the task.
- Encoder-only models, such as BERT-style models, are useful when the full input is available and the task requires understanding, classification, tagging, or retrieval.
- Decoder-only models, such as GPT-style models, use causal masking and are designed for next-token prediction and generation. Encoder–decoder models are used when one sequence must be transformed into another, such as translation or summarization.

Architecture variants

Different Transformer families keep different parts of the full architecture.

Encoder-only
BERT-style
classification, tagging, retrieval

Decoder-only
GPT-style
next-token generation

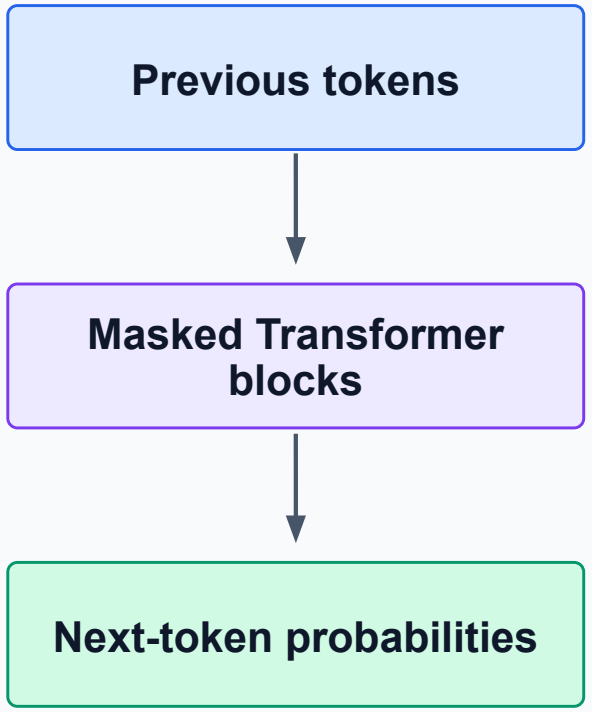
Encoder–decoder
translation, summarization
input → output sequence

Encoder-only models usually use bidirectional attention over known input.
Decoder-only models use causal masking and generate from left to right.
Encoder–decoder models use bidirectional source encoding plus causal target decoding and cross-attention.

GPT-style decoder-only models Generative Pretrained Transformer

A language model trained to predict the next token.

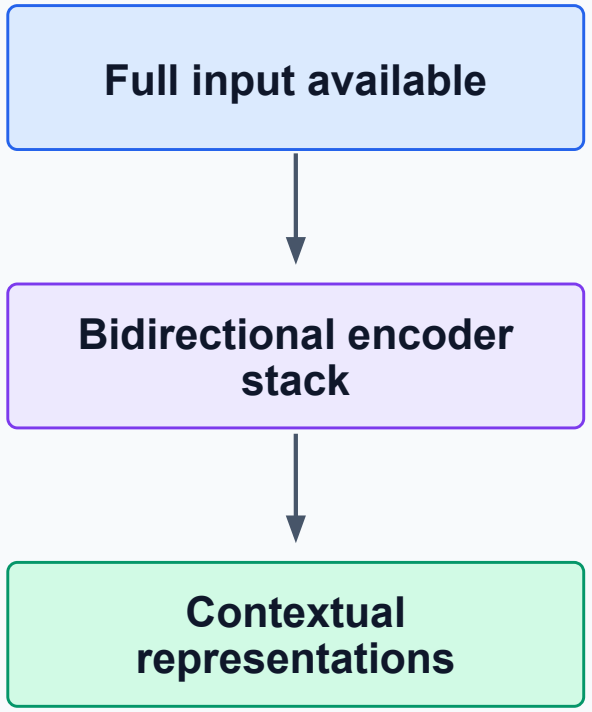
Input is a sequence of previous tokens.
Causal self-attention prevents future leakage.
The model outputs a distribution over the vocabulary at each position.
Large-scale pretraining teaches broad language patterns; task-specific behavior can be obtained through fine-tuning or instruction tuning.



BERT-style encoder-only models Bidirectional Encoder Representations from Transformers

A model that sees both left and right context.

The model uses bidirectional self-attention because the full input is known.
Unlike GPT, which can only look at words to its left, BERT sees words to both the **left and right** of a target word simultaneously.
This allows it to understand complex meanings, like whether "bank" refers to a river or a financial institution
Common pretraining tasks include masked-token recovery and sentence-pair objectives in classic BERT.
During training, BERT isn't asked to predict the "next" word. Instead, random words in a sentence are "masked" (hidden), and the model must use the surrounding context to fill in the blanks
A special classification token can aggregate sequence-level information.
Encoder-only models are natural for classification, semantic similarity, tagging, and retrieval-style tasks.



Subwords + attention in real text

The tokenizer chooses pieces; attention learns how pieces interact.

A long or rare word may be split into several subword tokens.
Self-attention can connect subword pieces belonging to the same word, or connect them to context elsewhere.
The model does not require a handcrafted stemmer; it can learn useful morphology from data.
But tokenization still affects sequence length, cost, and what patterns are easy to learn.

Tokenizer: strings → token IDs

Embedding: IDs → vectors

Transformer: vectors → contextual vectors

Advantages of Transformers

Short, focused advantages section before moving to vision.

Parallelizable across sequence positions within a layer.
Direct paths between distant tokens help long-range dependencies.
Dynamic attention weights adapt to each input.
The same architecture works for language, images, speech, time series, and multimodal data once inputs are tokenized.
Pretraining on large unlabeled corpora produces reusable representations.

Discussion

“Which advantage is most important for training very large language models?”

Answer:
The key advantage for training very large language models is that Transformers are highly parallelizable, making large-scale training much faster and more efficient.
A secondary important advantage is that attention gives shorter paths between distant tokens, helping the model learn long-range dependencies.

Advantages and Limitations

- Transformers became dominant because they solve several weaknesses of recurrent models. They are easier to parallelize, create shorter information paths between distant tokens, and dynamically decide which tokens are relevant for each input.

- However, this flexibility has a cost. Full self-attention compares every token with every other token, so memory and computation grow quadratically with sequence length.

- Transformers also need explicit positional information, and in some domains they require large amounts of data because they have fewer built-in assumptions than specialized architectures.

Limitations of full attention

The strength of attention is also its cost.

Quadratic attention cost: every token can compare with every other token.
Long contexts require large memory for attention matrices.
Transformers need positional information because sequence order is not built in.
They often require large data and compute to outperform models with stronger inductive biases.
For images, plain ViT lacks some CNN assumptions such as locality and translation equivariance.

Different attention styles

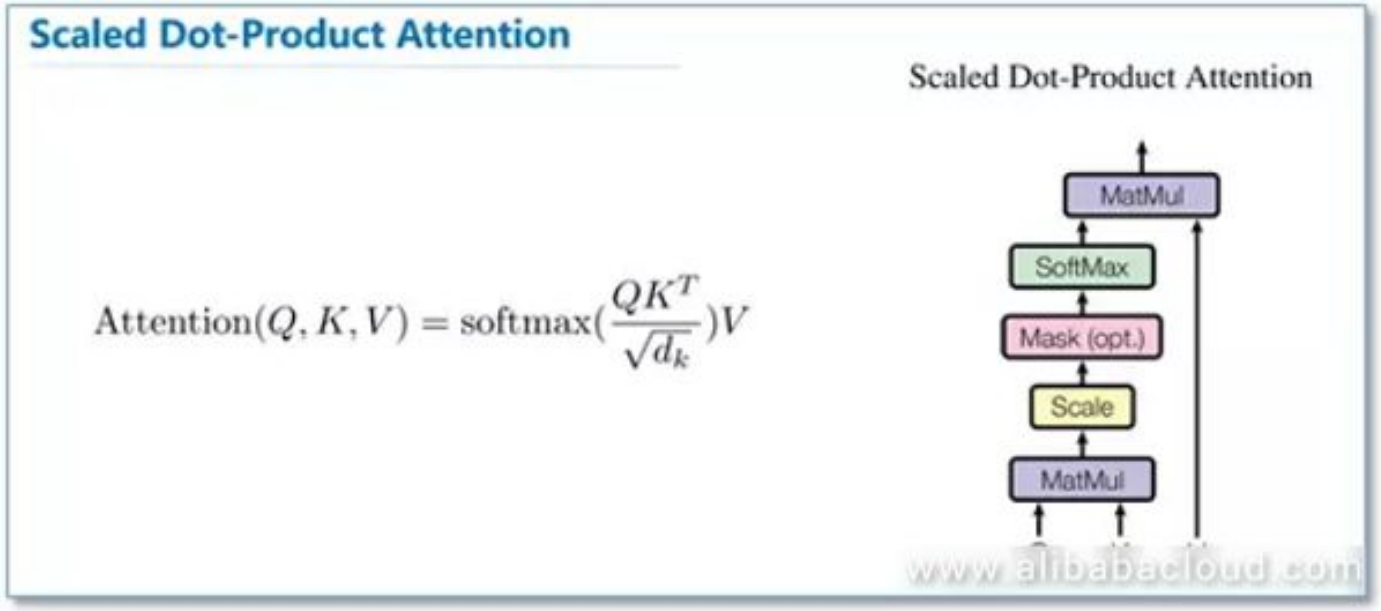
Same general idea, different scoring or visibility rules.

Additive attention
learned scoring network

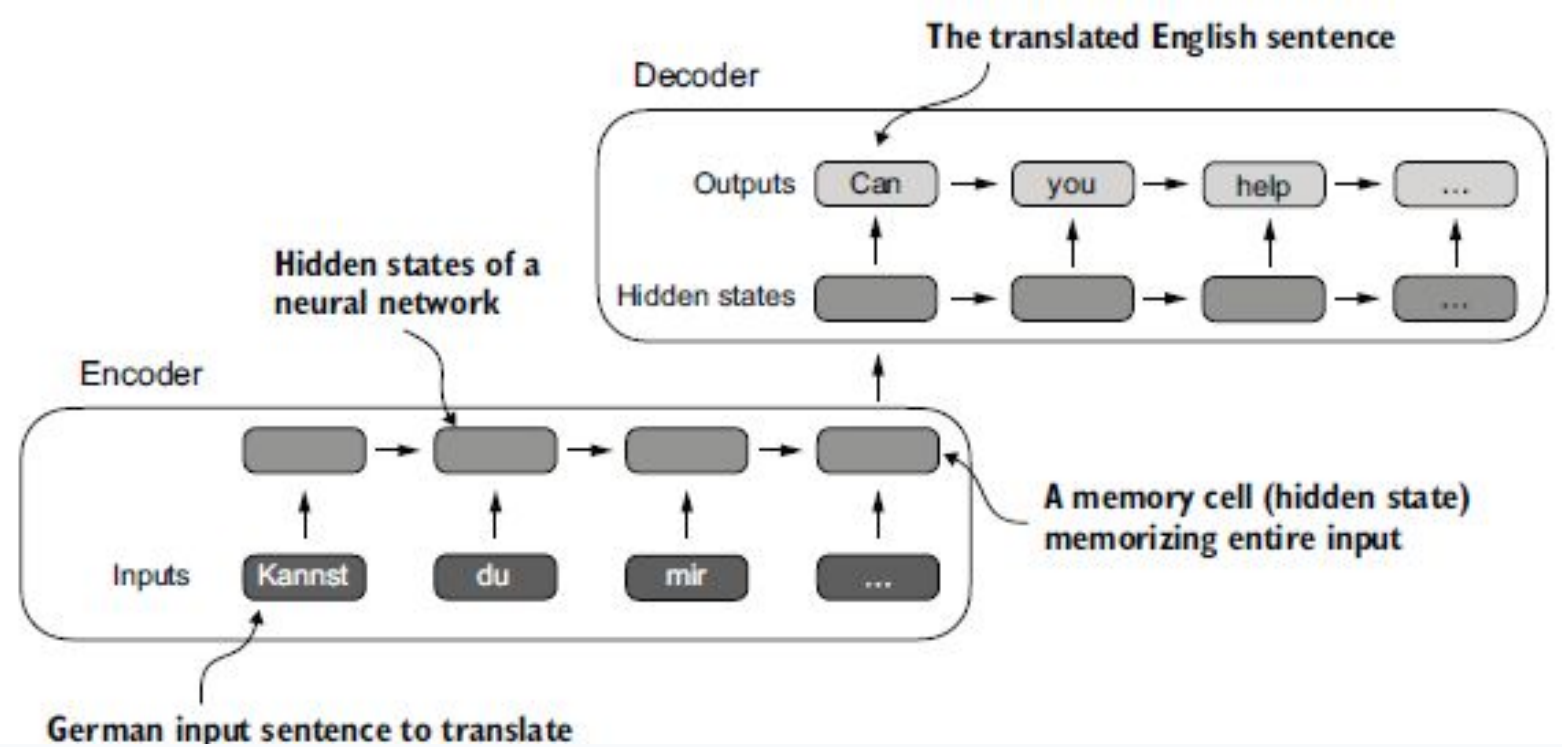
Dot-product attention
QK^T similarities

Scaled dot-product
$QK^T / \sqrt{d}$

Additive attention was common in early RNN encoder–decoder models.
Dot-product attention is efficient because it maps well to matrix multiplication.
Scaled dot-product attention is the standard form used in the original Transformer.



Example taken from LLM from scratch by Sebastian Rashschka. Before transformer models became popular, encoder–decoder RNNs were widely used for machine translation. In this approach, the encoder processes a sequence of tokens from the source language and produces a hidden state—a compact representation of the entire input sequence. The decoder then uses its final hidden state to generate the translation, producing tokens one at a time.



Self, cross, and causal attention

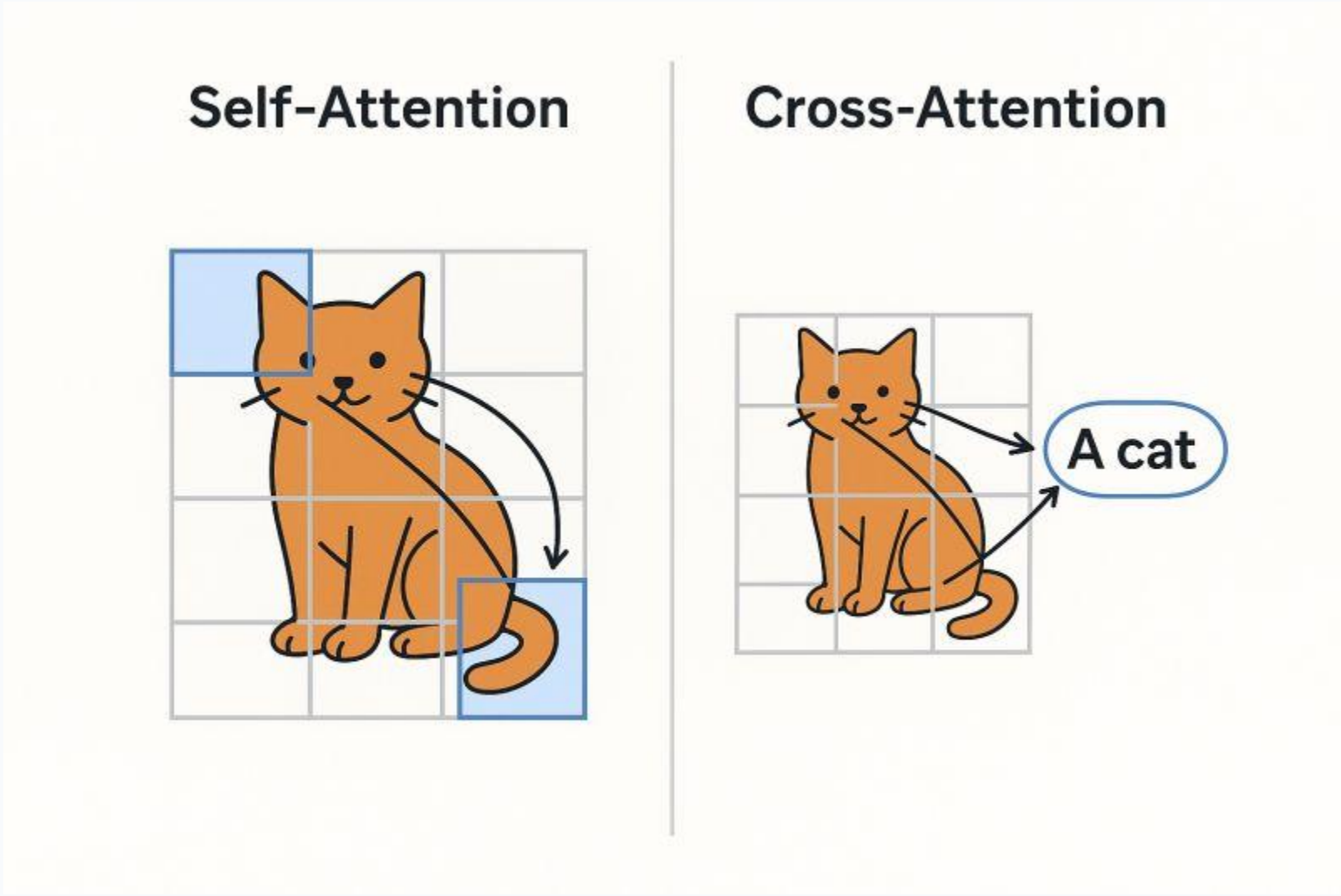
Visibility rules define what information can flow where.

Self-attention
within one sequence

Cross-attention
between two sequences

Causal attention
only past positions

Self-attention asks: how do tokens in this sequence relate to each other?
Cross-attention asks: which source tokens are useful for the current target context?
Causal attention asks the same question under a restriction: do not use future tokens.


Self-Attention
Cross-Attention
A cat

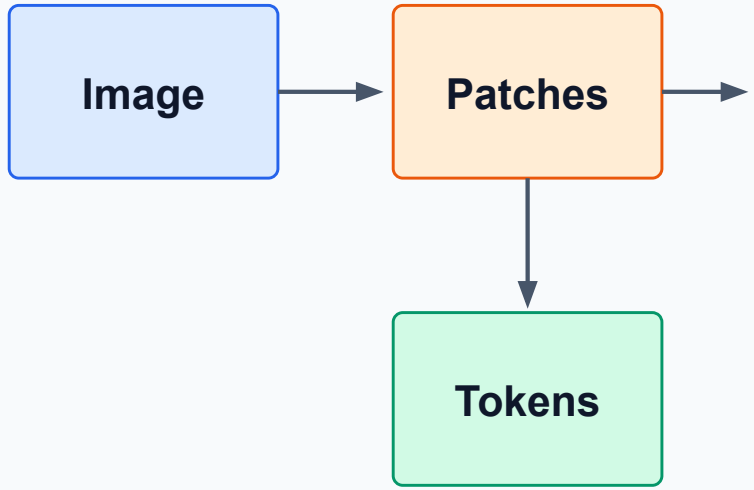
Transformers for Images

- Transformers were first developed for symbolic sequences such as text, but the same idea can be applied to images if we convert the image into a sequence.
- A Vision Transformer splits an image into fixed-size patches. Each patch becomes a token, similar to a word or subword token in language.
- These patch tokens are embedded, given positional information, and processed by a Transformer encoder. This chapter explains how ViTs work and how they differ from convolutional neural networks.

Vision Transformers: why images can be sequences

Transformers operate on token sequences, so images must be tokenized.

A text tokenizer converts a string into tokens.
A Vision Transformer converts an image into patch tokens.
Each image patch is flattened and projected into an embedding vector.
A sequence of patch embeddings can be processed by a Transformer encoder.




DESPICABLE
ME
IN THEATERS
JULY 09, 2010
IN EYE-POPPING REAL 3D

Vision Transformers: Process

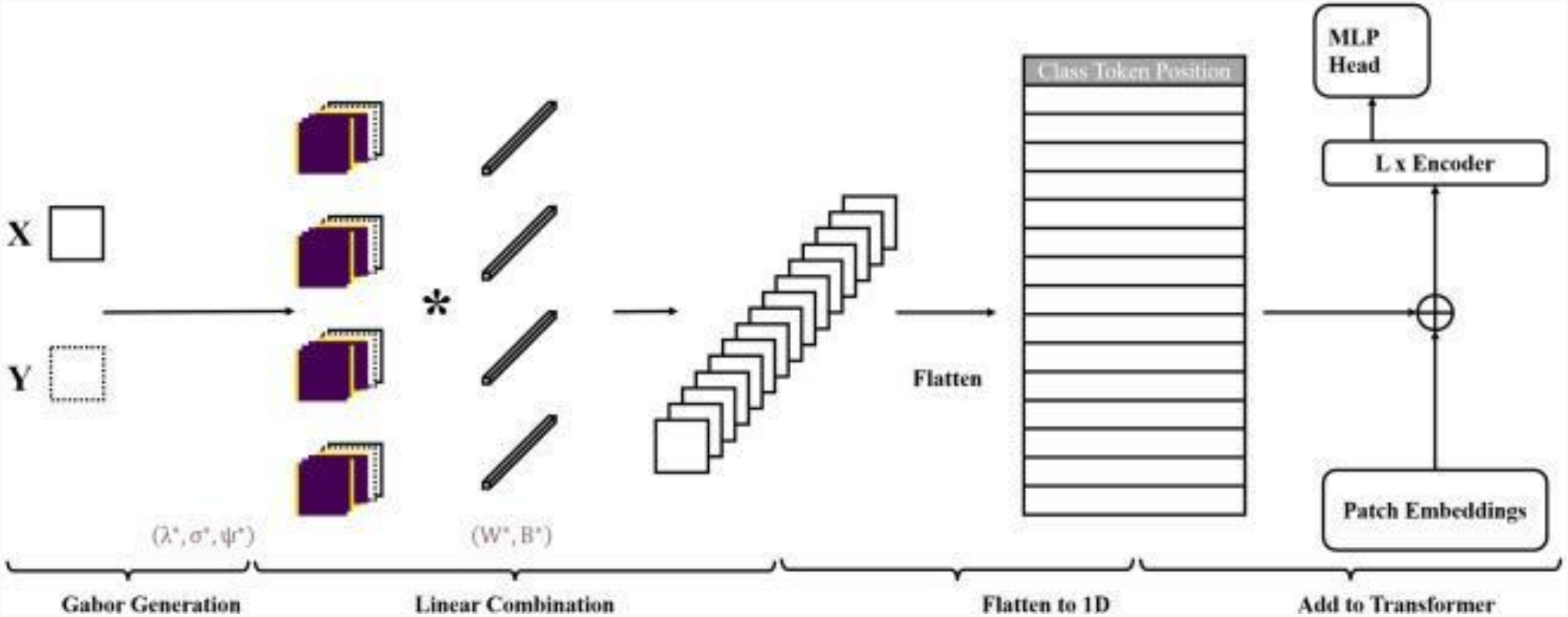
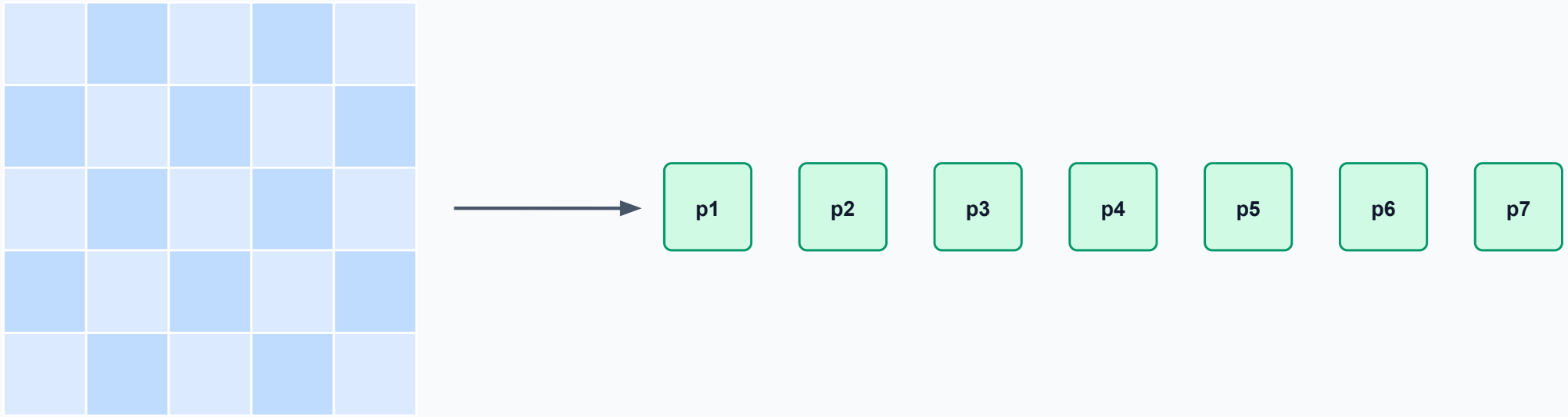


Image patch tokenization

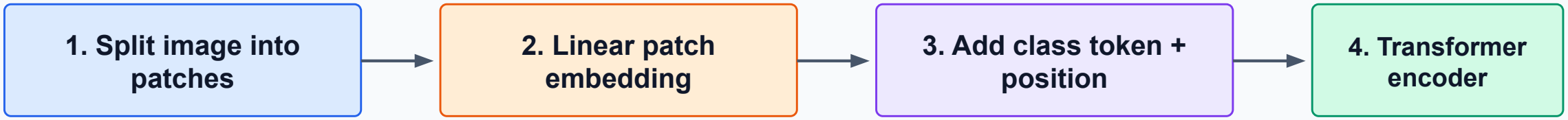
An image is split into fixed-size patches such as 16×16 pixels.



Patch size controls sequence length: smaller patches mean more tokens.
More tokens preserve finer detail but increase attention cost.
Each patch token receives a learned embedding and a positional signal.
A special class token is often prepended for image-level classification.

Vision Transformer pipeline

From image to class prediction.



The Transformer encoder updates patch representations using self-attention.
The class token can aggregate information from all patches.
A final classification head maps the class-token representation to class probabilities.

ViT positional information

Patches need position too: patch order alone is not enough.

A patch token by itself does not know where it came from in the image.
ViTs add positional embeddings to patch embeddings.
Unlike CNNs, a plain ViT does not automatically encode local 2D neighborhood structure.
The model must learn spatial relationships from position signals and data.

Patch content
what is in the patch

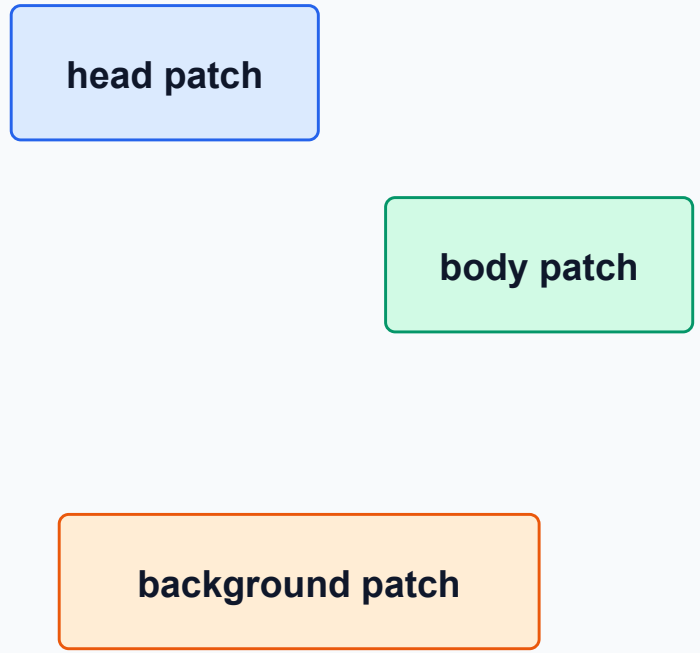
Patch position
where it is in the image

Token vector
content + position

What self-attention does in images

Patch tokens can gather information from other patches.

A patch containing part of an object can attend to other patches containing related parts.
Global attention allows long-range interactions from early layers.
This differs from CNNs, where receptive fields grow gradually through local convolutions.
Attention maps can sometimes be interpreted as which image regions influenced a decision.



ViT versus CNN inductive bias

CNNs build in image assumptions; ViTs rely more on data and attention.

CNN

Local filters: edges/textures are natural.
Translation equivariance is built in.
Strong image-specific inductive bias.

Vision Transformer

Patch tokens processed by attention.
Global relations possible early.
Less image-specific bias; often needs more data or good pretraining.

This is why early ViTs were strongest at large scale, and why many later models reintroduce locality or hierarchy.

Beyond vanilla ViT: DETR, Swin, and hierarchical ideas

Transformers are adapted to vision tasks by changing tokens, attention, and outputs.

DETR applies an encoder–decoder Transformer to object detection, using learned object queries.
Swin Transformer limits attention to local windows and shifts windows between layers to connect neighborhoods.
Hierarchical ViT variants merge patches as depth increases, resembling feature pyramids.
The common theme: keep attention, but add structure to control compute and fit visual tasks.

Discussion

“Why might local windows be a good idea for images?”

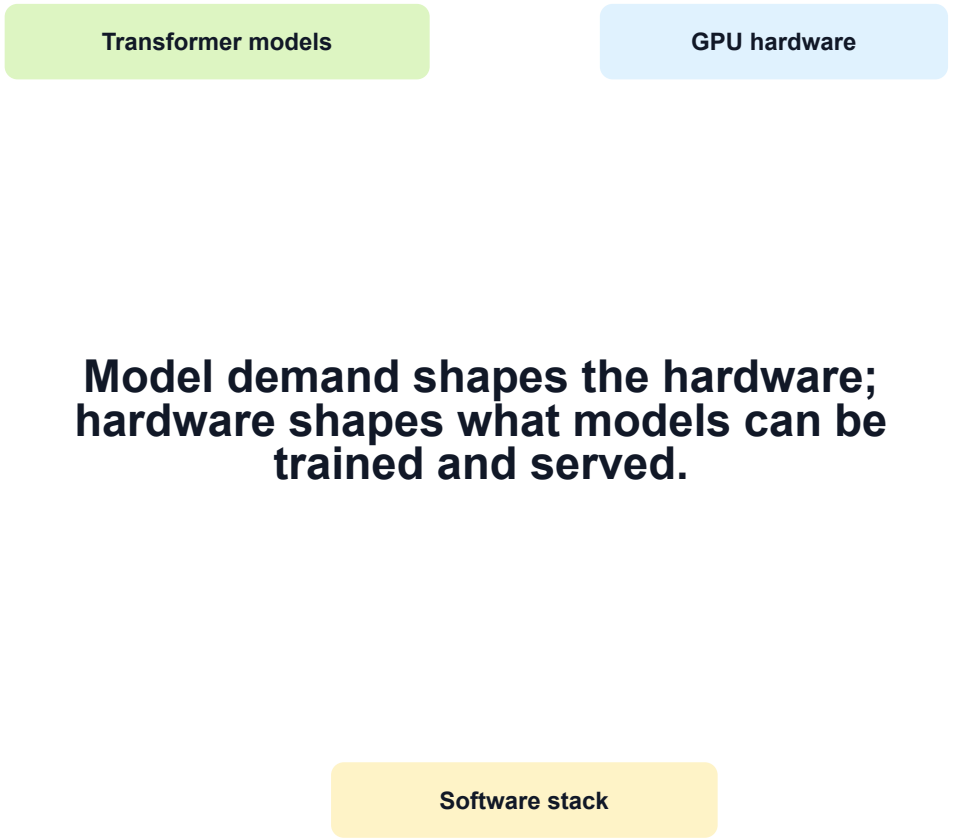
Answer:
Local windows are useful for images because **most visual information is local**. Nearby pixels or patches usually form edges, textures, corners, and object parts. For example, to recognize an eye, wheel, or leaf, the model first needs to understand small neighboring regions before combining them into larger structures.
Local windows also reduce computation. Full self-attention compares every image patch with every other patch, which becomes expensive for large images. If attention is computed only inside small windows, the model performs far fewer comparisons.

Industry example: NVIDIA designs around Transformers

Why mention NVIDIA at the all?

Transformers are not only a model architecture. They became important enough that modern AI hardware and deployment software are optimized around their computation patterns.

Large matrix multiplications dominate training and inference.
Attention, feed-forward layers, and KV-cache management need specialized optimization.
Lower precision can reduce memory and increase throughput if accuracy is preserved.



Sources: NVIDIA Hopper Architecture; NVIDIA Blackwell Architecture; NVIDIA Transformer Engine documentation.

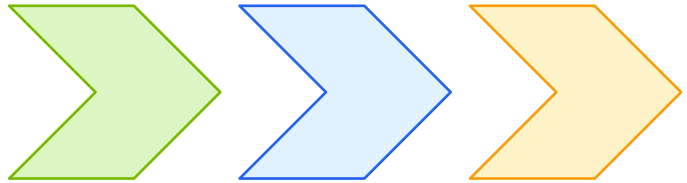
Hardware: Transformer Engine and low precision

Industry example

The hardware target is the expensive part of Transformers:

huge dense matrix multiplications and large activation / weight memory.

NVIDIA Transformer Engine accelerates Transformer models on NVIDIA GPUs and supports FP8 on Hopper, Ada, and Blackwell GPUs.
Hopper Tensor Cores support mixed FP8 and FP16 precision to accelerate Transformer calculations.
Blackwell introduces a second-generation Transformer Engine aimed at LLM and Mixture-of-Experts training and inference.



Lower precision can mean more throughput and lower memory use, but the system must manage numerical stability.

Sources: NVIDIA Transformer Engine overview; NVIDIA Hopper Architecture; NVIDIA Blackwell Architecture.

Software: serving LLMs efficiently with TensorRT-LLM

Industry example

Training is only half the story. Deployed LLMs must generate tokens for many users with low latency and controlled memory use.

TensorRT-LLM is NVIDIA's library for optimizing and deploying LLM inference on NVIDIA GPUs.
It includes optimizations such as quantization, paged attention / KV-cache management, request scheduling, and parallelism strategies.
These optimizations matter because decoder-only Transformers repeatedly reuse past keys and values while generating one token at a time.

During generation:

previous tokens → KV cache → next token

Efficient inference is largely about doing this loop quickly, repeatedly, and for many sequences at once.

Quantization | **Paged attention** | **Batching**

Sources: NVIDIA TensorRT-LLM documentation; NVIDIA TensorRT documentation.

Industrial example: architecture, hardware, and deployment co-evolve

Transformers became a central workload for modern AI systems.

That is why companies optimize the full stack around them:

Model design	Hardware design	Serving software
Attention variants, masking, KV cache, model parallelism	Tensor Cores, lower precision formats, memory bandwidth	TensorRT-LLM, quantization, scheduling, batching

Discussion

“If computation is the expensive part, what design choices can reduce cost without losing too much model quality?”

DLSS + Frame Generation: Transformers Beyond Text

NVIDIA uses Transformer-based neural rendering to improve image quality and smoothness in real-time games.

1. Super Resolution

Render fewer pixels, then reconstruct a higher-quality frame.

Transformer model uses current frame, motion/history, and learned visual priors.

Goal: sharper image, fewer artifacts, better temporal stability.

2. Frame Generation

AI predicts extra frames between traditionally rendered frames.

DLSS 4 Multi Frame Generation can generate multiple additional frames.

DLSS 4.5 adds Dynamic MFG and a 2nd-generation Transformer model.

3. Why Transformers fit

Images and video have spatial + temporal relationships.

Attention can model long-range dependencies across pixels/patches/frames.

Same lecture idea: compare elements, weight relevance, mix information.

Takeaway: Transformers are not only for language — the same attention idea can reconstruct and generate visual information in games.

Note: frame generation improves perceived smoothness, but generated frames are not the same as new game-simulation/input updates; latency handling such as NVIDIA Reflex matters.

Sources: NVIDIA DLSS Developer page; NVIDIA DLSS 4 / DLSS 4.5 GeForce News; NVIDIA Research DLSS 4.

Frame Generation and DLSS NVIDIA example

Final Takeaways

- The main idea of Transformers is that sequence processing can be done through attention rather than recurrence.

- Subword tokenization prepares text in a practical form. Embeddings convert tokens into vectors. Positional encoding adds order. Self-attention allows tokens to exchange information. Masking prevents future-token leakage. Encoder, decoder, and cross-attention structures adapt the architecture to different tasks.

- Vision Transformers show that the same core idea can extend beyond language when data is represented as tokens. The key question is not only what components Transformers contain, but why each component is needed.

References and source material

Main materials used to structure this lecture.

Vaswani et al. (2017), “Attention Is All You Need.”

Bahdanau et al. (2015), neural machine translation with attention.

Luong (2016), neural machine translation and attention variants.

Devlin et al. (2018), BERT.

Radford et al. (2018), GPT.

Dosovitskiy et al. (2021), Vision Transformer.

NVIDIA DLSS Developer page; NVIDIA DLSS 4 / DLSS 4.5 GeForce News; NVIDIA Research DLSS 4.


ABSOLUTE
CINEMA