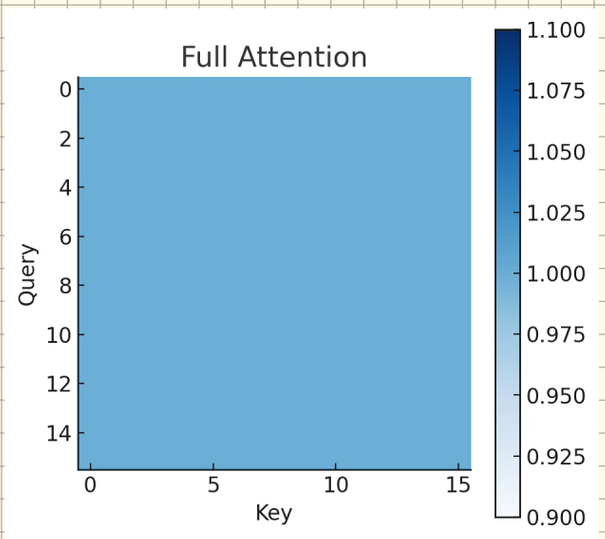
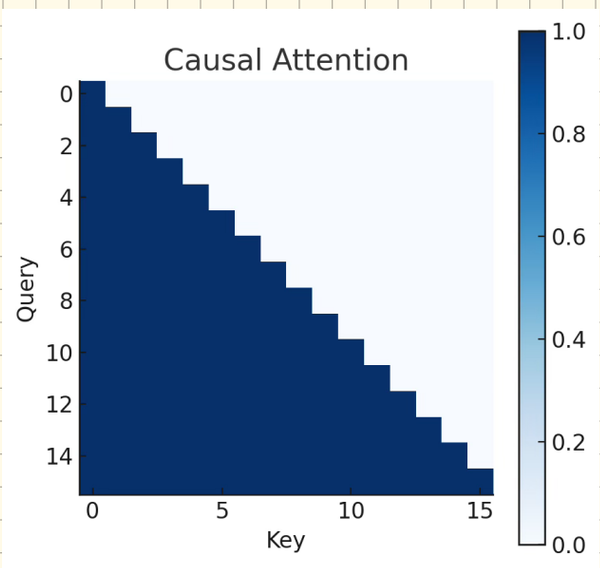
3. Sliding Window Attention

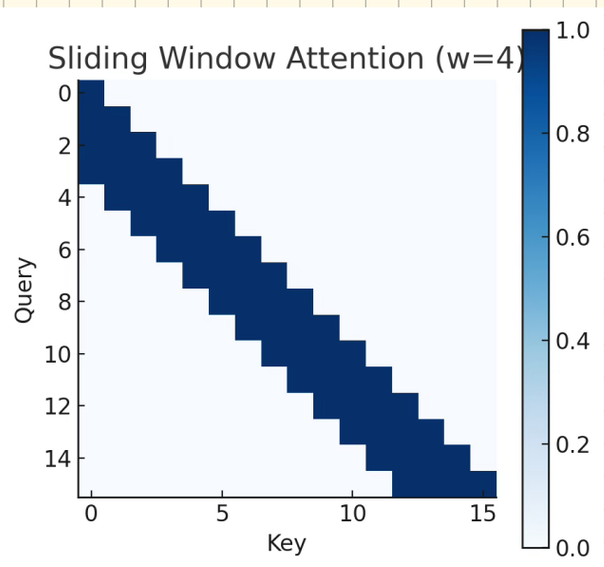
Sliding window attention is a technique in Transformer models that limits the attention span to a fixed-size window around each token rather than attending to the entire sequence.

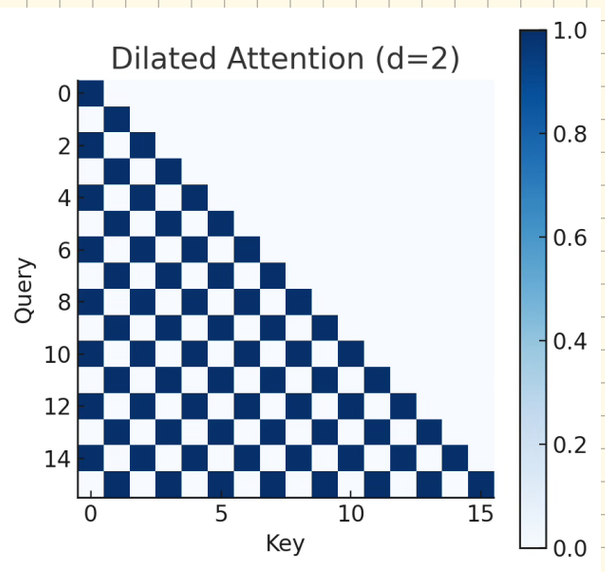
- Instead of computing attention over all tokens (which is expensive for long sequences), each token only attends to a subset—a "window" of tokens nearby.
- The window "slides" over the sequence as you move from token to token.
- This reduces the computational complexity from quadratic ($O(n^2)$) to linear ($O(n * \text{window\_size})$), making it scalable to longer sequences.

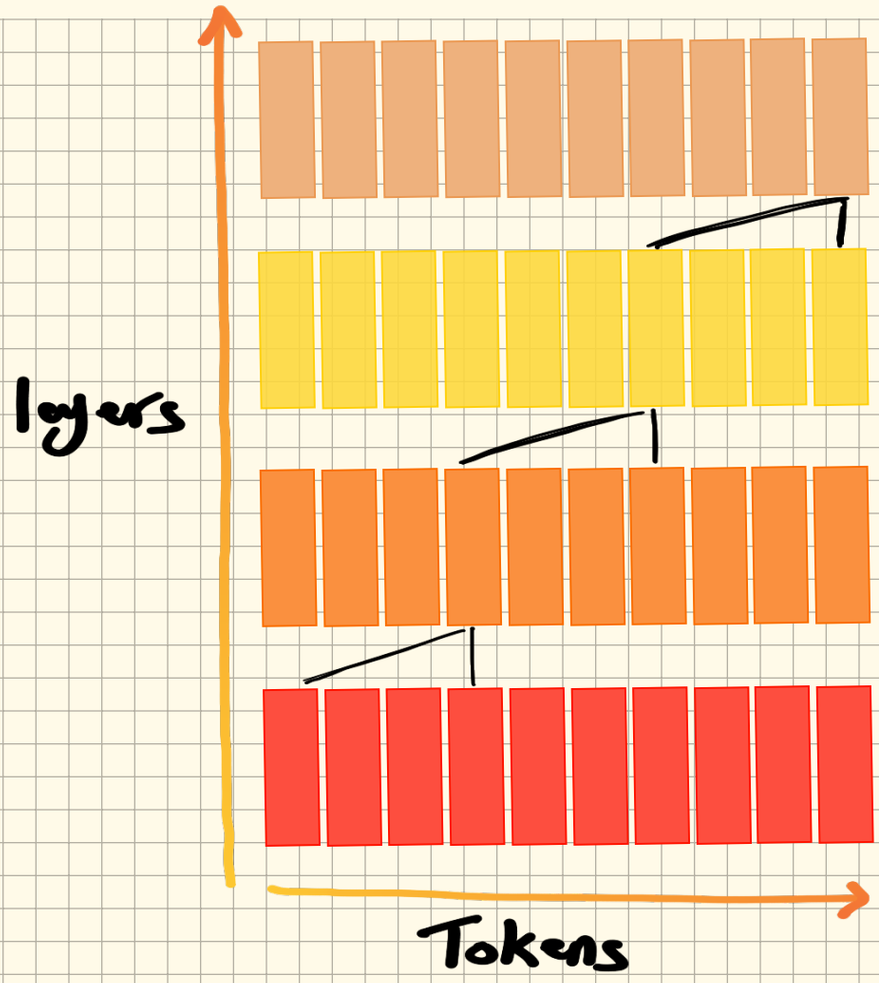
Use cases: Long document modeling, speech, or any sequence where local context matters more than distant tokens.





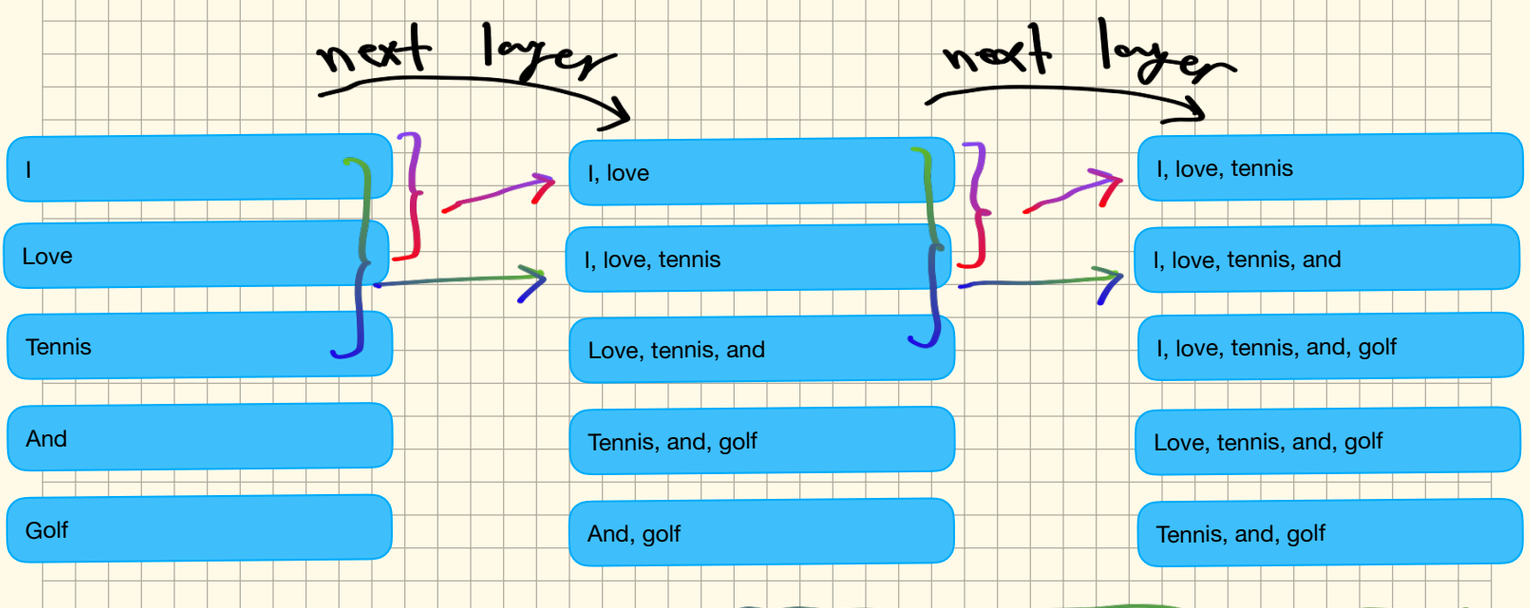






SWA example

sentence: I love tennis and golf



🧠 Types of Attention Mechanisms

Type	Access Pattern	Cost	Best For
1. **Full Attention**	Each token attends to all	$O(n^2)$	Small to medium sequences
2. **Causal Attention**	Each token attends to past	$O(n^2)$	Language modeling (GPT)
3. **Sliding Window**	Fixed-size local window	$O(n \cdot w)$	Long documents, low memory
4. **Dilated Attention**	Every *k*-th token	$O(n \cdot \log n)$	Periodic structure, compression
5. **Global + Sliding**	Local + few global tokens	$O(n \cdot w + g^2)$	QA, summarization, retrieval
6. **Block Sparse**	Attention in and between blocks	$O(n \cdot \sqrt{n})$	Balanced memory/speed (Longformer)
7. **Linear Attention**	Approximated attention (kernel)	$O(n)$	Real-time, ultra-long sequences
8. **State-Space (e.g. Mamba)**	Not attention-based	$O(n)$	Streaming, high-efficiency tasks

1. Rolling Buffer

A **rolling buffer** (also called a circular buffer or ring buffer) is a fixed-size data structure that continuously overwrites old data with new data in a cyclic manner.

- Imagine a buffer with a fixed length N.
- When new data comes in, it's added at the "current position."
- Once the buffer is full, new data overwrites the oldest data in the buffer.
- The "rolling" part means the buffer pointer moves forward with each new element and loops back to the start when it reaches the end.

Use cases: Useful in streaming data, real-time systems, or any scenario where you want to keep a limited recent history without reallocating memory.

2. KV Cache

KV cache stands for **Key-Value cache**, mostly used in Transformer models for attention mechanisms, especially during inference (generation).

- Transformers compute attention by using *keys (K)*, *values (V)*, and *queries (Q)*.
- When generating text token-by-token, recalculating keys and values for all previous tokens repeatedly is expensive.
- The KV cache stores previously computed key and value tensors for all tokens processed so far.
- This allows the model to reuse those cached keys and values when processing the next token, speeding up inference.

In short: It's a memory of previous keys and values to avoid redundant computation during sequence generation.

Mathematical Formulation:

Given token embeddings x_t, compute:

$$K_t = x_t W^K, \quad V_t = x_t W^V$$

These are stored in memory:

$$\text{Cache}_t = \{K_1, ..., K_t\}, \{V_1, ..., V_t\}$$

At each new step $t+1$, the model computes:

$$Q_{t+1} = x_{t+1} W^Q$$

$$\text{Attention}(Q_{t+1}, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_{t+1} K_{1:t}^T}{\sqrt{d_k}}\right) V_{1:t}$$

Feature	KV Cache	Rolling Buffer
Purpose	Stores key-value pairs from attention computations	Manages the sequence of input tokens
Benefit	Speeds up attention computations by reusing past results	Keeps token history manageable for long sequences
Efficient processing	Reduces repeated work during inference	Avoids excessive memory use by discarding older tokens
Impact on speed	Faster generation since we don't recompute attention	Smooths processing by handling input in fixed-size chunks
Impact on memory	Saves computational overhead and memory per token	Uses a sliding window to stay within memory limits
Size relation	Size is constrained by the rolling buffer's window length	Defines the maximum context length; KV cache cannot exceed this

📌 Example 1: Repeat each element 3 times (flat)

python Copy

```python
import torch

x = torch.tensor([1, 2, 3])
out = torch.repeat_interleave(x, 3)
print(out)  # tensor([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

📌 Example 2: Different repeats for each element

python Copy

```python
x = torch.tensor([1, 2, 3])
repeats = torch.tensor([1, 2, 3])
out = torch.repeat_interleave(x, repeats)
print(out)  # tensor([1, 2, 2, 3, 3, 3])
```

📌 Example 3: Along a specific dimension

python Copy

```python
x = torch.tensor([[1, 2], [3, 4]])
# Repeat rows (dim=0)
print(torch.repeat_interleave(x, 2, dim=0))
# tensor([[1, 2],
#         [1, 2],
#         [3, 4],
#         [3, 4]])

# Repeat columns (dim=1)
print(torch.repeat_interleave(x, 2, dim=1))
# tensor([[1, 1, 2, 2],
#         [3, 3, 4, 4]])
```