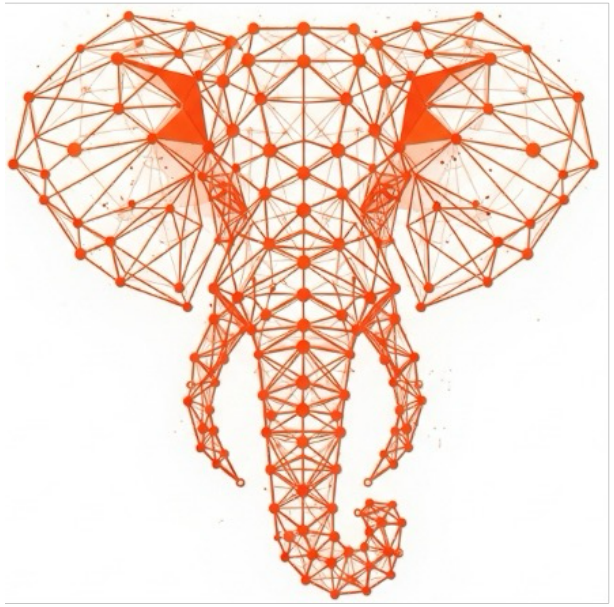
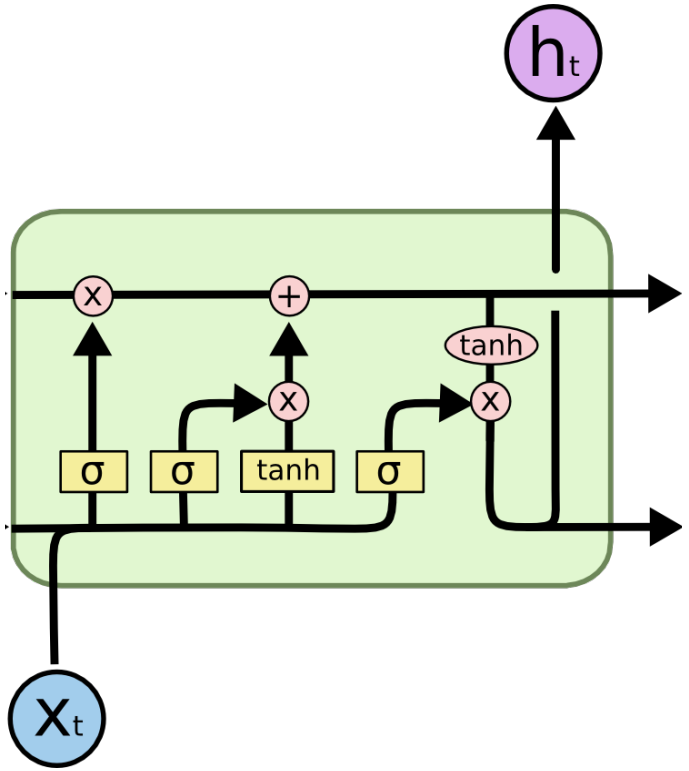


Deep Learning
Advanced Models & Methods

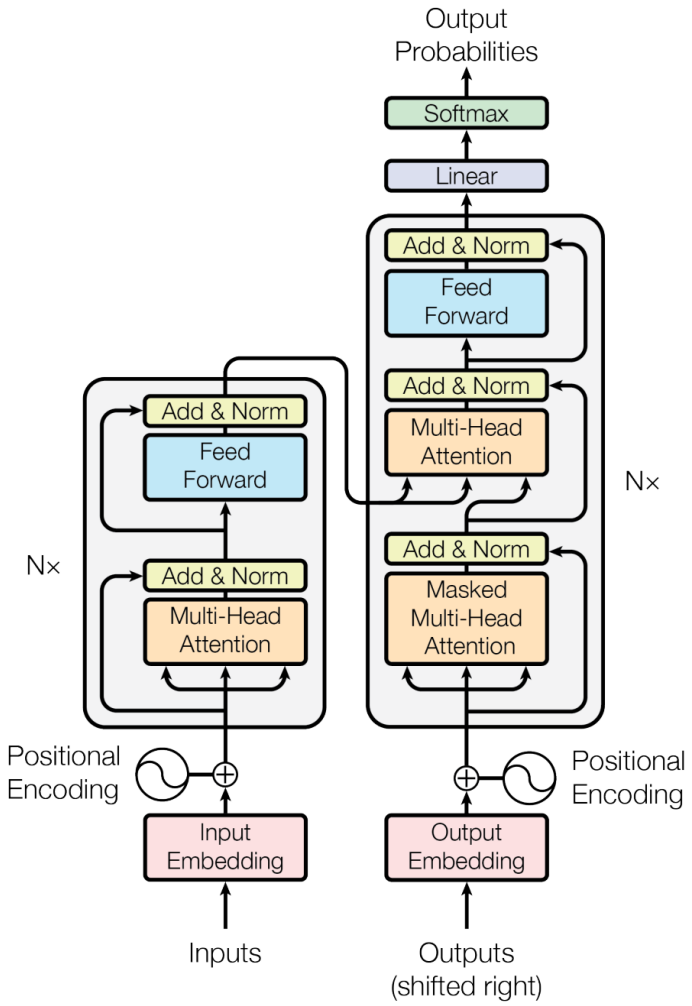
Prof. Antonino Furnari (antonino.furnari@unict.it)
Corso di Laurea Magistrale in Informatica
Dip. di Matematica e Informatica
Università di Catania

Advanced Sequential Modeling

Recurrent Neural Networks vs Transformers



https://colah.github.io/posts/2015-08-Understanding-LSTMs/



What About Convolutions?

An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling

Shaojie Bai [1] **J. Zico Kolter** [2] **Vladlen Koltun** [3]

Abstract

For most deep learning practitioners, sequence modeling is synonymous with recurrent networks. Yet recent results indicate that convolutional architectures can outperform recurrent networks on tasks such as audio synthesis and machine translation. Given a new sequence modeling task or dataset, which architecture should one use? We conduct a systematic evaluation of generic convolutional and recurrent architectures for sequence modeling. The models are evaluated across a broad range of standard tasks that are commonly used to benchmark recurrent networks. Our results indicate that a simple convolutional architecture outperforms canonical recurrent networks such as LSTMs across a diverse range of tasks and datasets, while demonstrating longer effective memory. We conclude that the common association between sequence modeling and recurrent networks should be reconsidered, and convolutional networks should be regarded as a natural starting point for sequence modeling tasks. To assist related work, we have made code available at http://github.com/locuslab/TCN.

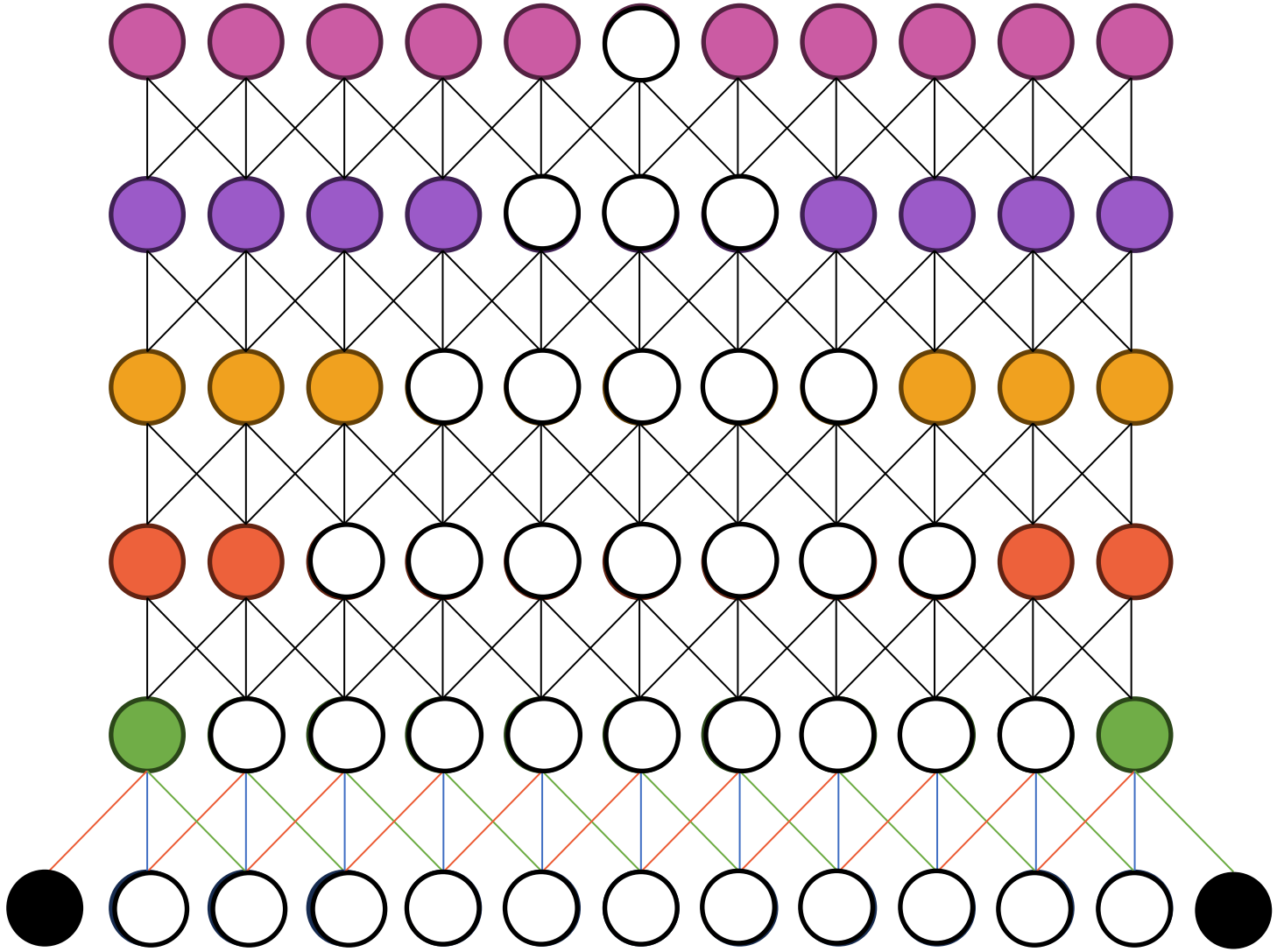
On the other hand, recent research indicates that certain convolutional architectures can reach state-of-the-art accuracy in audio synthesis, word-level language modeling, and machine translation (van den Oord et al., 2016; Kalchbrenner et al., 2016; Dauphin et al., 2017; Gehring et al., 2017a;b). This raises the question of whether these successes of convolutional sequence modeling are confined to specific application domains or whether a broader reconsideration of the association between sequence processing and recurrent networks is in order.

We address this question by conducting a systematic empirical evaluation of convolutional and recurrent architectures on a broad range of sequence modeling tasks. We specifically target a comprehensive set of tasks that have been repeatedly used to compare the effectiveness of different recurrent network architectures. These tasks include polyphonic music modeling, word- and character-level language modeling, as well as synthetic stress tests that had been deliberately designed and frequently used to benchmark RNNs. Our evaluation is thus set up to compare convolutional and recurrent approaches to sequence modeling on the recurrent networks' "home turf".

To represent convolutional networks, we describe a generic temporal convolutional network (TCN) architecture that is

Shows empirically that temporal convolution networks works better than LSTMs in many sequence processing problems.

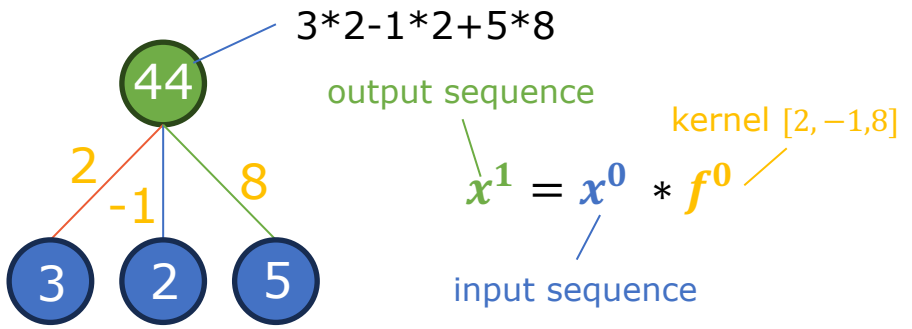
Temporal Convolutional Networks (TCN)



padding (0)

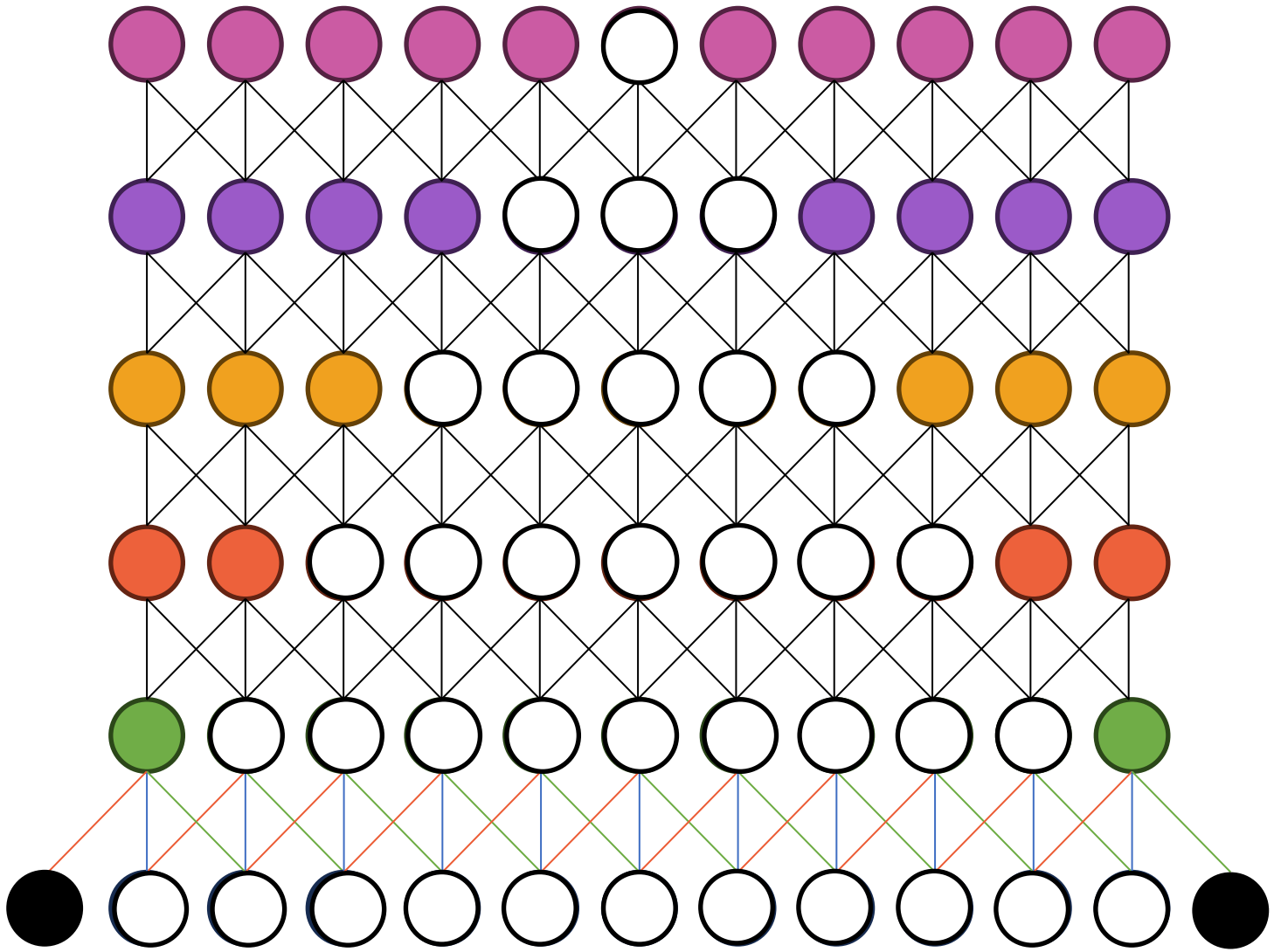
- Sequence-to-sequence model
- Learnable parameters are kernel weights
- Of course, nonlinear activation functions are placed in between layers ☺
- Highly parallelizable
- Growing receptive field

New layer: different kernel $x^2 = x^1 * f^1$



1D input sequence of length L - each element is a D-dimensional vector (e.g., feature frames): $BS \times L \times D$

Depth and sequence length



Problem: how many layers do we need to make sure that each unit can reason globally about the sequence?

It turns out that's:

$$d \geq \left\lceil \frac{L-k}{k-1} + 1 \right\rceil$$

Where d is the number of layers (depth), k is the kernel size, and L is the sequence length.

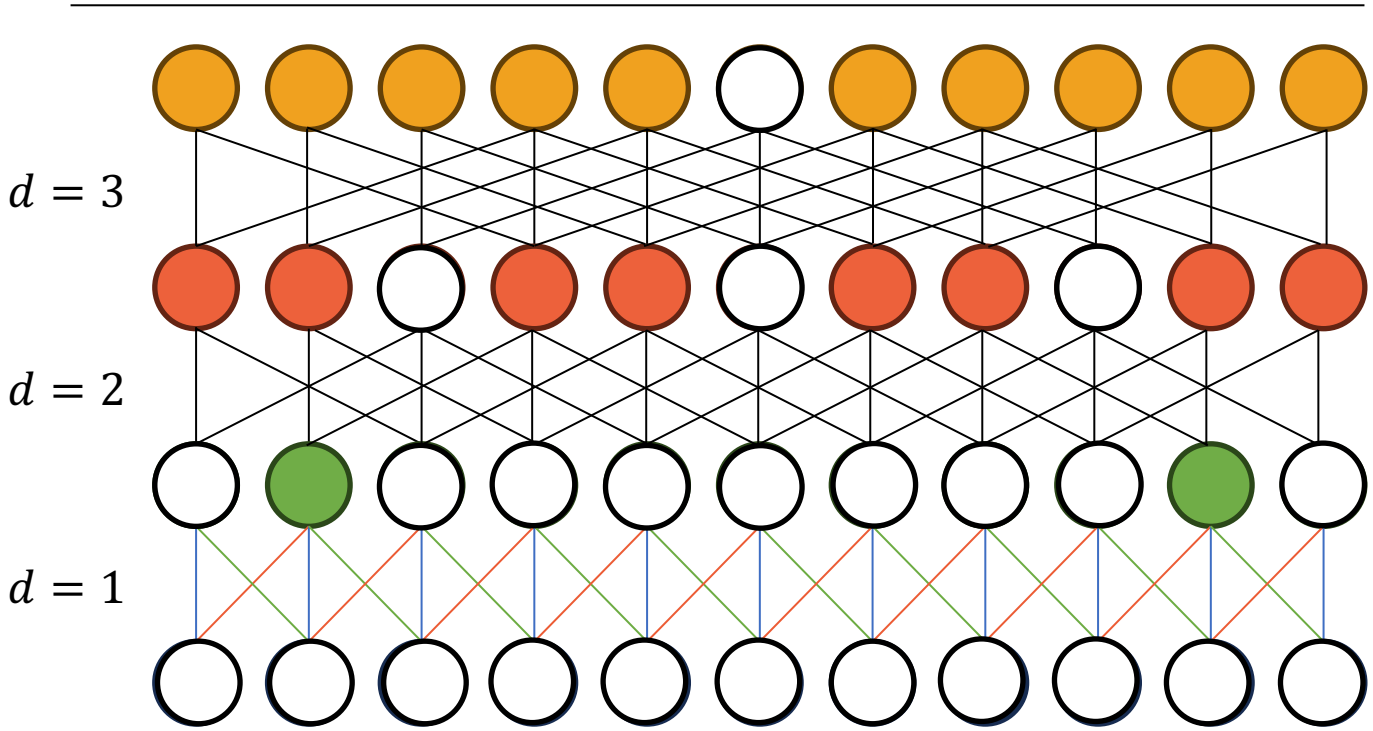
For example:

$$L = 100, k = 3 \Rightarrow d \geq 50$$

C. Lea, M. D. Flynn, R. Vidal, A. Reiter, e G. D. Hager, «Temporal Convolutional Networks for Action Segmentation and Detection», 2016, *arXiv*: arXiv:1611.05267.

Dilated Convolutions

- Three layers are enough to achieve a large receptive field.
- Without dilated convolutions, we needed 5 layers
- Dilated convolutions act like pooling in 2D CNNs



Idea: dilated convolutions

- Perform convolutions with a dilation factor d which skips $d - 1$ units to increase receptive field
- d increases with layers
- As a consequence, we need fewer layers to make sure that each unit «sees» the whole input

In practice:

$$D \geq \left\lceil \log_2 \left(\frac{L-1}{k-1} + 1 \right) \right\rceil$$

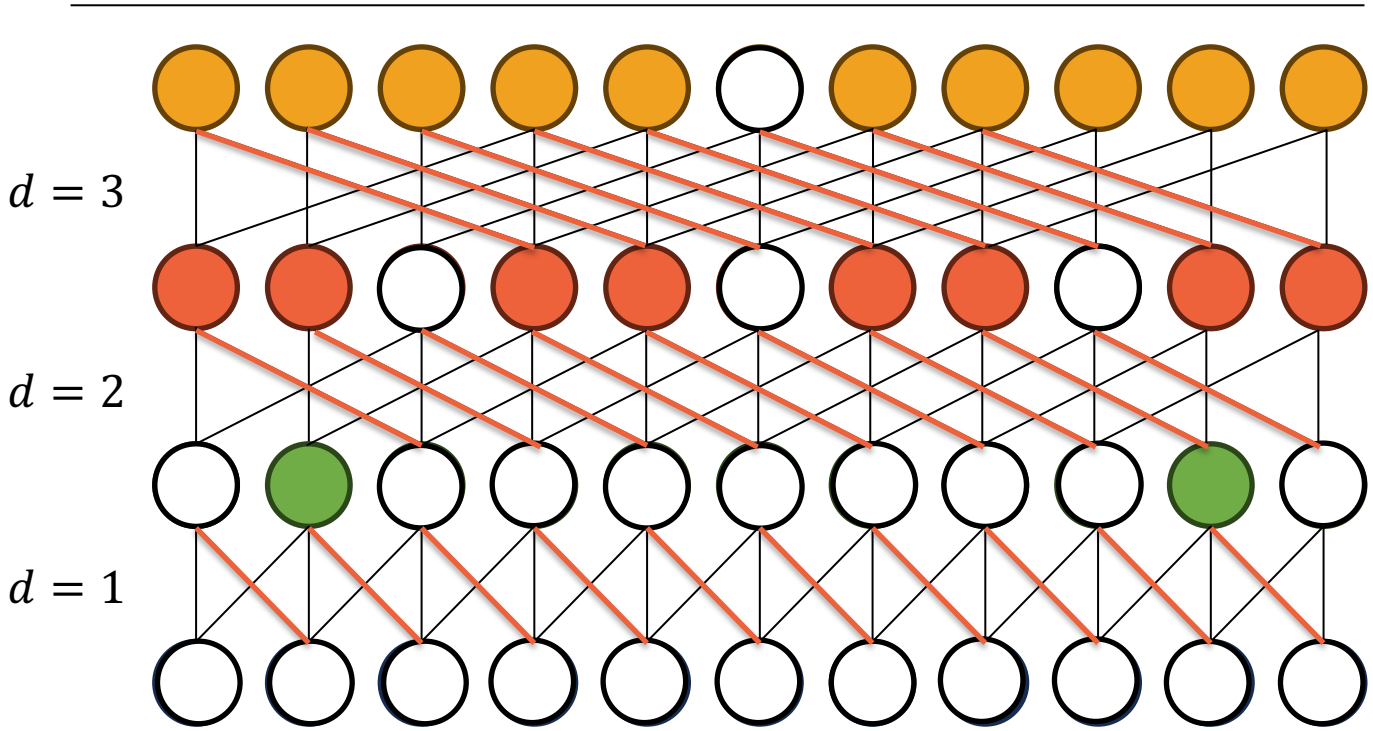
For instance:

$$L = 100, k = 3 \Rightarrow d \geq 6$$

C. Lea, M. D. Flynn, R. Vidal, A. Reiter, e G. D. Hager, «Temporal Convolutional Networks for Action Segmentation and Detection», 2016, *arXiv*: arXiv:1611.05267.

TCN and online processing

The white unit in the top layer sees the whole video, so the model can only be used for offline tasks, and cannot be deployed in an online or streaming fashion.



- As introduced, TCNs are good for «offline» tasks, where the video is processed in batch mode.
- Indeed, in the last layer, each unit actually «sees into the future», so it is necessary to have the whole video available in order to process it online.

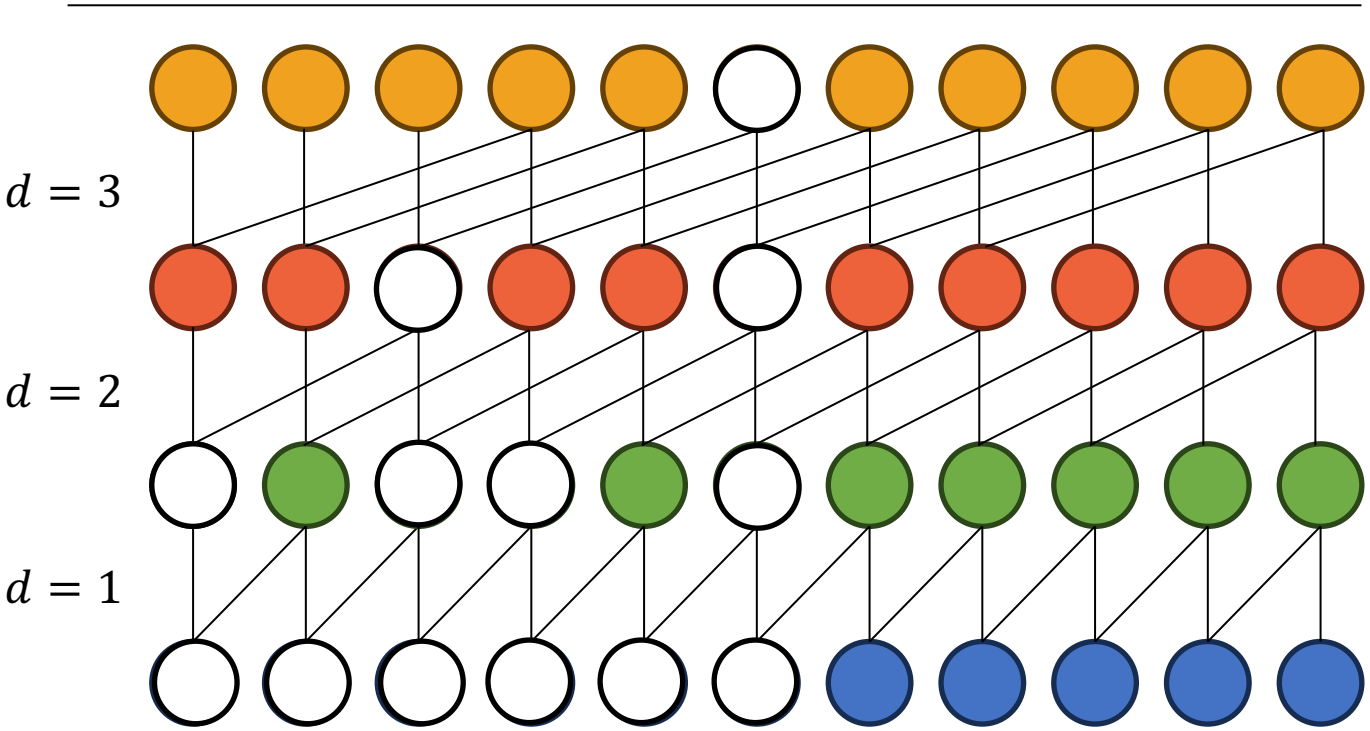
Solution?

Cut the connections looking into the future!

C. Lea, M. D. Flynn, R. Vidal, A. Reiter, e G. D. Hager, «Temporal Convolutional Networks for Action Segmentation and Detection», 2016, *arXiv*: arXiv:1611.05267.

Dilated causal convolutions

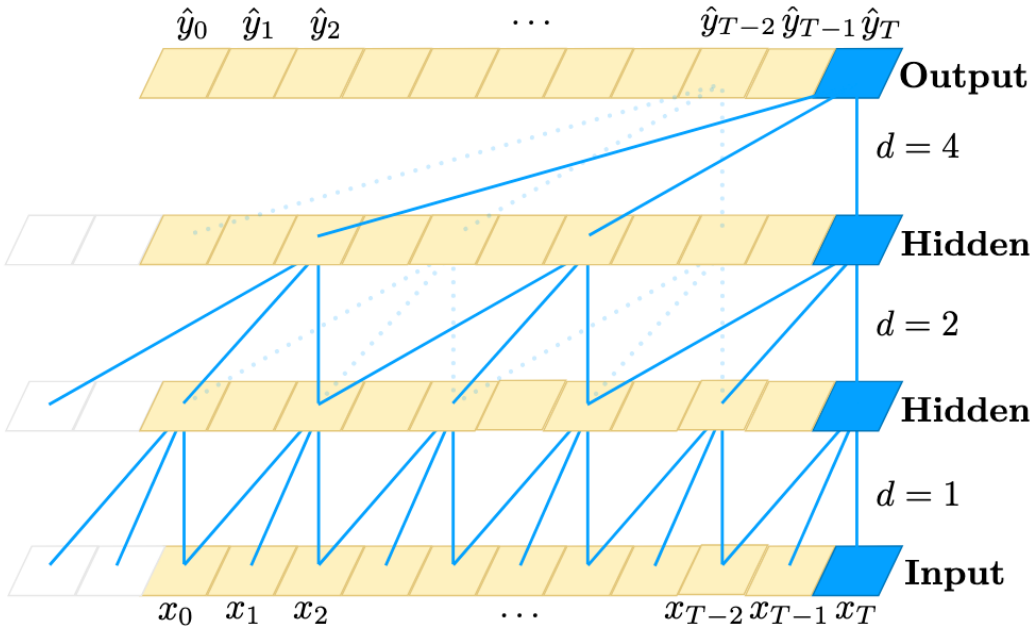
With dilated causal convolutions, we keep the large receptive field, but each unit only looks at past units, hence making TCN suitable for online processing

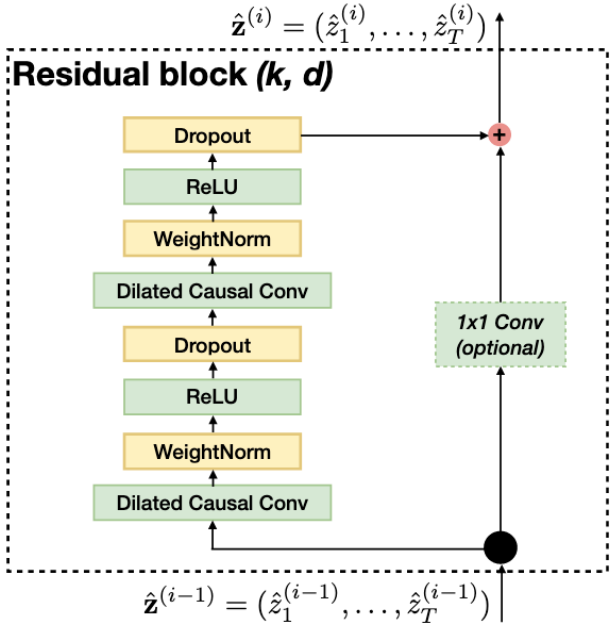


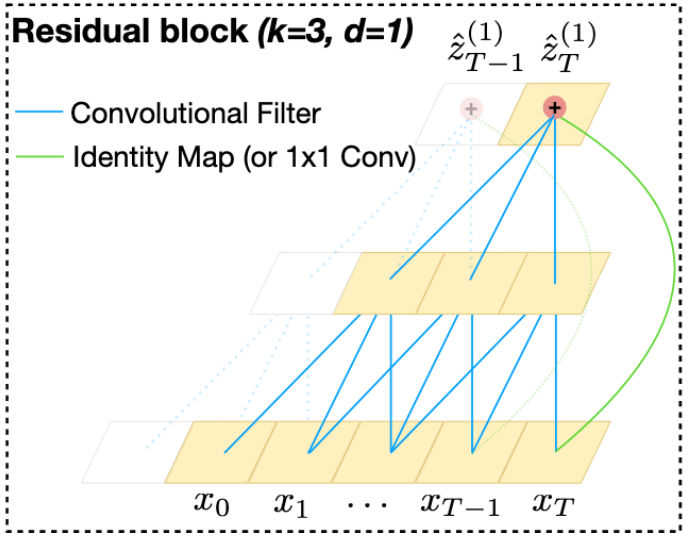
- By removing forward looking connections, at each layer, each unit only "sees" past units.
- In this way, the whole processing is compatible with online/streaming sceanrios.
- It should be noted that, while possible, streaming inference is still not super-straightforward and we need to keep a cache of past observations and activations.
- In practice, we obtain a causal TCN by using a smaller kernel (e.g., k=2) and shifting it backward.

C. Lea, M. D. Flynn, R. Vidal, A. Reiter, e G. D. Hager, «Temporal Convolutional Networks for Action Segmentation and Detection», 2016, *arXiv*: arXiv:1611.05267.

Architectural Details







In practice, TCNs can be implemented in slighlty different ways (as CNNs), including choices of:

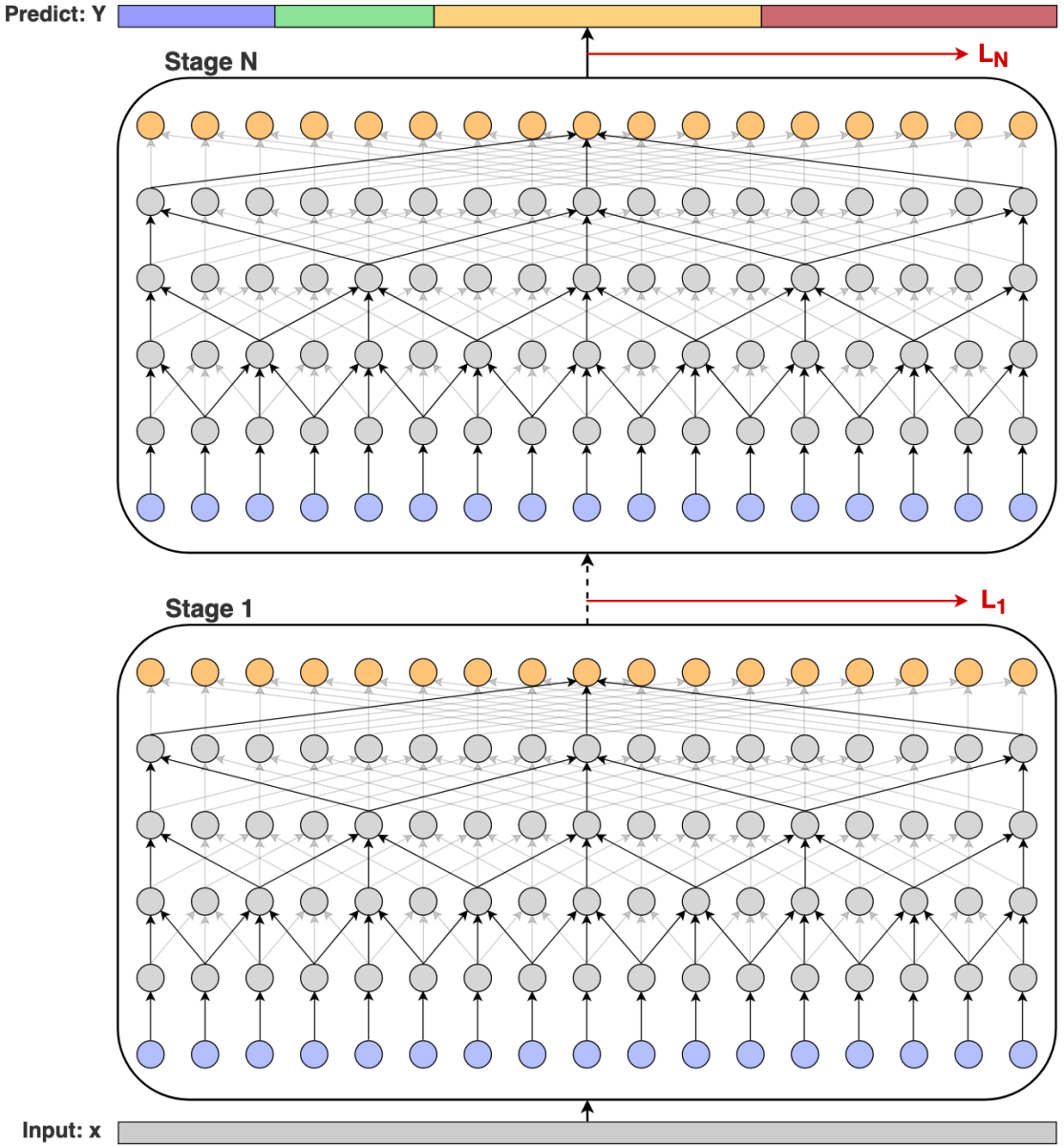
- Activation functions
- Normalization techniques
- Residual connections

Sequence Modeling Task	Model Size (≈)	Models			
		LSTM	GRU	RNN	TCN
Seq. MNIST (accuracy^h)	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	**97.2**
Adding problem T=600 (loss$^\ell$)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T=1000 (loss)	16K	0.0204	0.0197	0.0202	**3.5e-5**
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	**8.10**
Music Nottingham (loss)	1M	3.29	3.46	4.05	**3.07**
Word-level PTB (perplexity$^\ell$)	13M	**78.93**	92.48	114.50	88.68
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^\ell$)	3M	1.36	1.37	1.48	**1.31**
Char-level text8 (bpc)	5M	1.50	1.53	1.69	**1.45**

In 2018, they were shown to be competitive with recurrent networks on many generic sequence processing tasks

S. Bai, J. Z. Kolter, e V. Koltun, «An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling», 2018, *arXiv*: arXiv:1803.01271.

Multi-Stage TCN (MS-TCN) for action segmentaiton

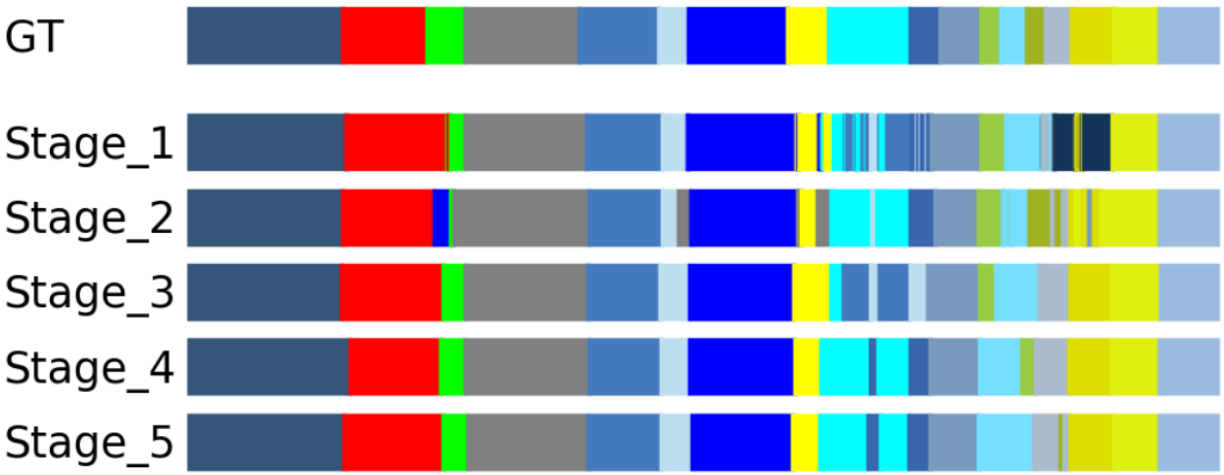


- Pre-extracted features are used as input
- The model operates in «stages»
- In the first stage, the input is the sequence of features and the output is the output segmentation (actually, per-frame probability distributions)
- Subsequent stages take as input segmentations from previous stages and «refine» them, up till the last stage
- Intermediate losses are used $(L_1, \dots, L_N)$ to make sure that intermediate outputs are indeed segmentations
- Uses a-causal convolutions (good for offline processing)

Farha, Yazan Abu, and Jurgen Gall. "Ms-tcn: Multi-stage temporal convolutional network for action segmentation." *IEEE/CVF conference on computer vision and pattern recognition*. 2019.

Multi-Stage TCN (MS-TCN) for action segmentaiton

Example of obtained progressive refinement:



Effect of number of stages

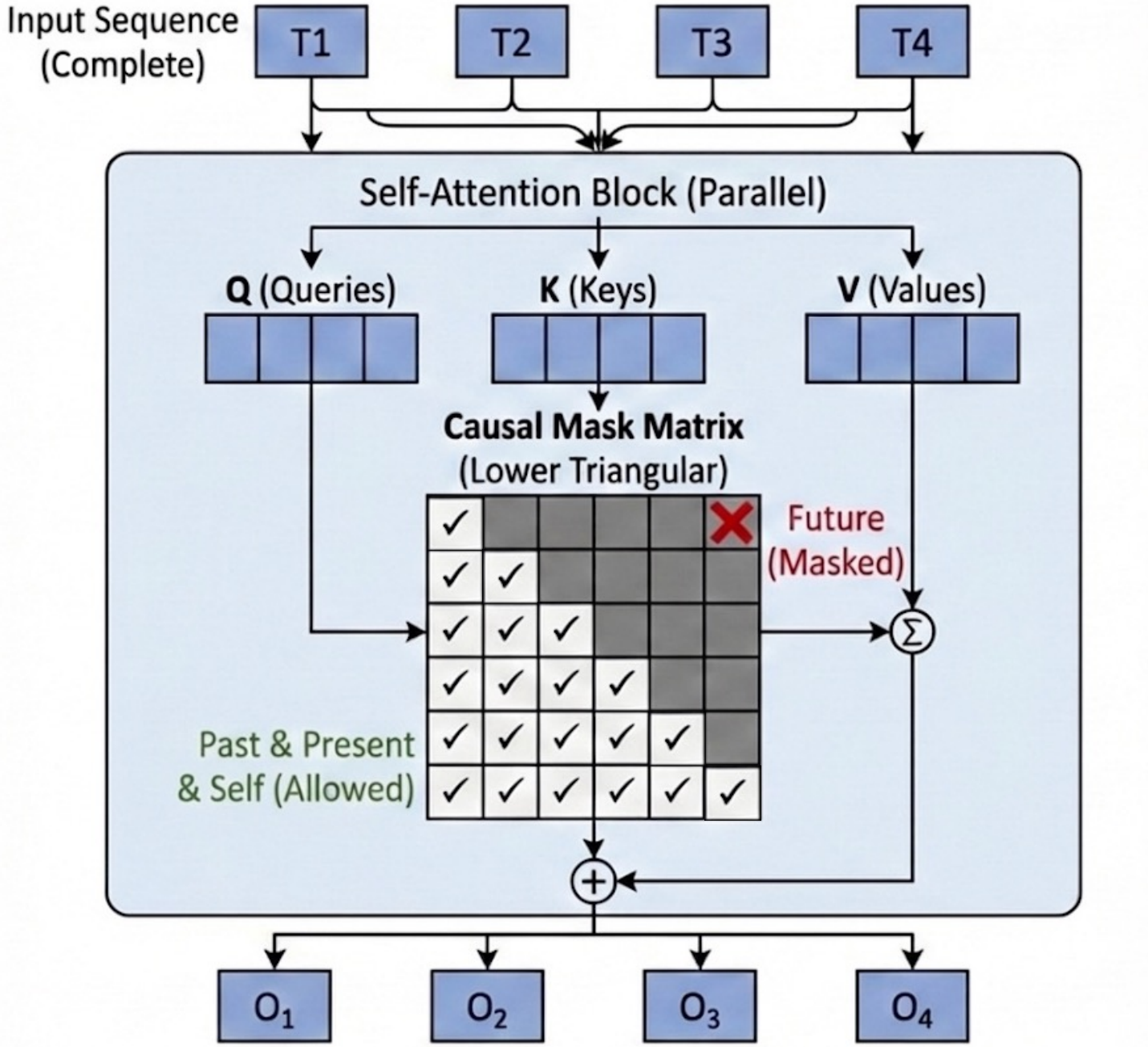
	F1@{10,25,50}			Edit	Acc
SS-TCN	27.0	25.3	21.5	20.5	78.2
MS-TCN (2 stages)	55.5	52.9	47.3	47.9	79.8
MS-TCN (3 stages)	71.5	68.6	61.1	64.0	78.6
MS-TCN (4 stages)	76.3	**74.0**	**64.5**	67.9	**80.7**
MS-TCN (5 stages)	**76.4**	73.4	63.6	**69.2**	79.5

50Salads	F1@{10,25,50}			Edit	Acc
IDT+LM [20]	44.4	38.9	27.8	45.8	48.7
Bi-LSTM [23]	62.6	58.3	47.0	55.6	55.7
ED-TCN [15]	68.0	63.9	52.6	59.8	64.7
TDRN [17]	72.9	68.5	57.2	66.0	68.1
MS-TCN	**76.3**	**74.0**	**64.5**	**67.9**	**80.7**
GTEA	F1@{10,25,50}			Edit	Acc
Bi-LSTM [23]	66.5	59.0	43.6	-	55.5
ED-TCN [15]	72.2	69.3	56.0	-	64.0
TDRN [17]	79.2	74.4	62.7	74.1	70.1
MS-TCN	85.8	83.4	69.8	79.0	76.3
MS-TCN (FT)	**87.5**	**85.4**	**74.6**	**81.4**	**79.2**
Breakfast	F1@{10,25,50}			Edit	Acc
ED-TCN [15]*	-	-	-	-	43.3
HTK [14]	-	-	-	-	50.7
TCFPN [5]	-	-	-	-	52.0
HTK(64) [13]	-	-	-	-	56.3
GRU [21]*	-	-	-	-	60.6
MS-TCN (IDT)	**58.2**	**52.9**	**40.8**	61.4	65.1
MS-TCN (I3D)	52.6	48.1	37.9	**61.7**	**66.3**

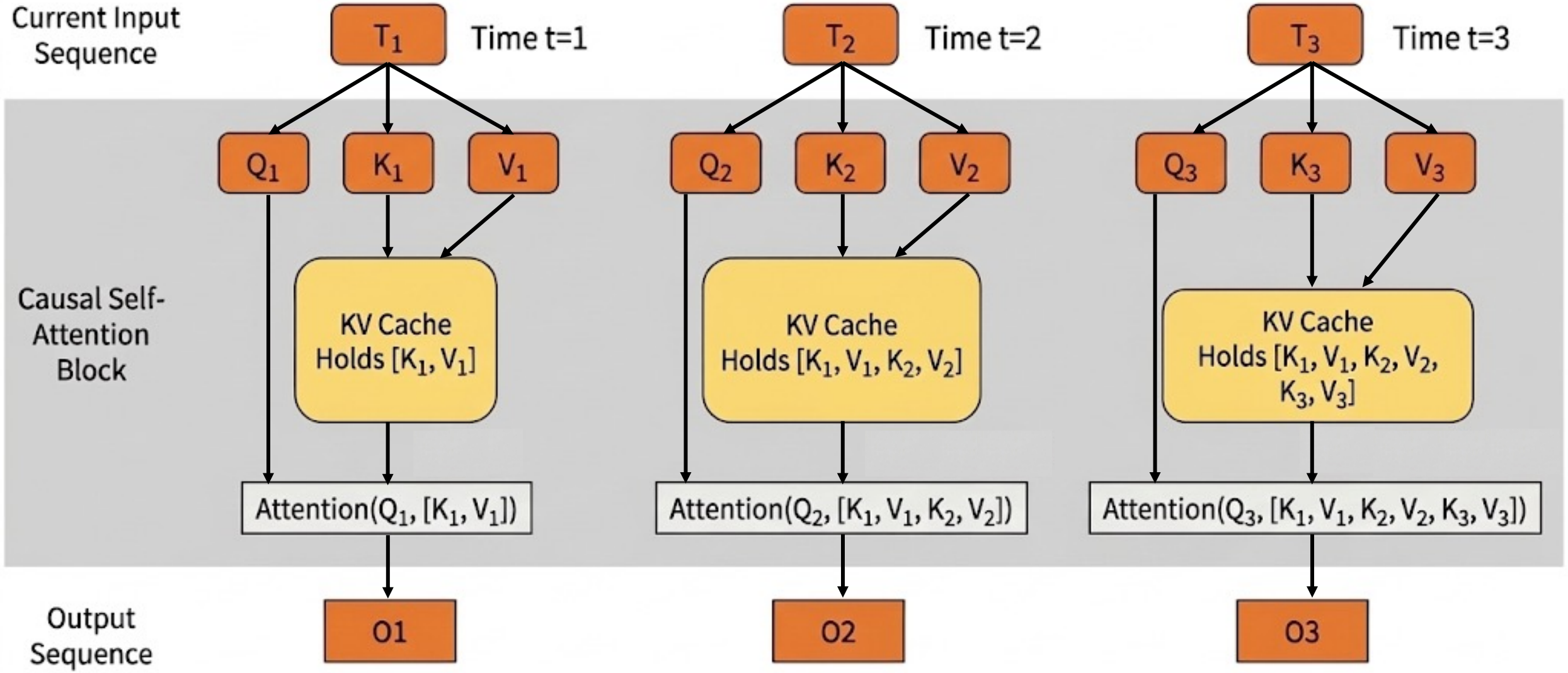
Farha, Yazan Abu, and Jurgen Gall. "Ms-tcn: Multi-stage temporal convolutional network for action segmentation." *IEEE/CVF conference on computer vision and pattern recognition*. 2019.

Wait, isn’t attention all we need?

Can Transformers be Used Online? yes, with causal masking

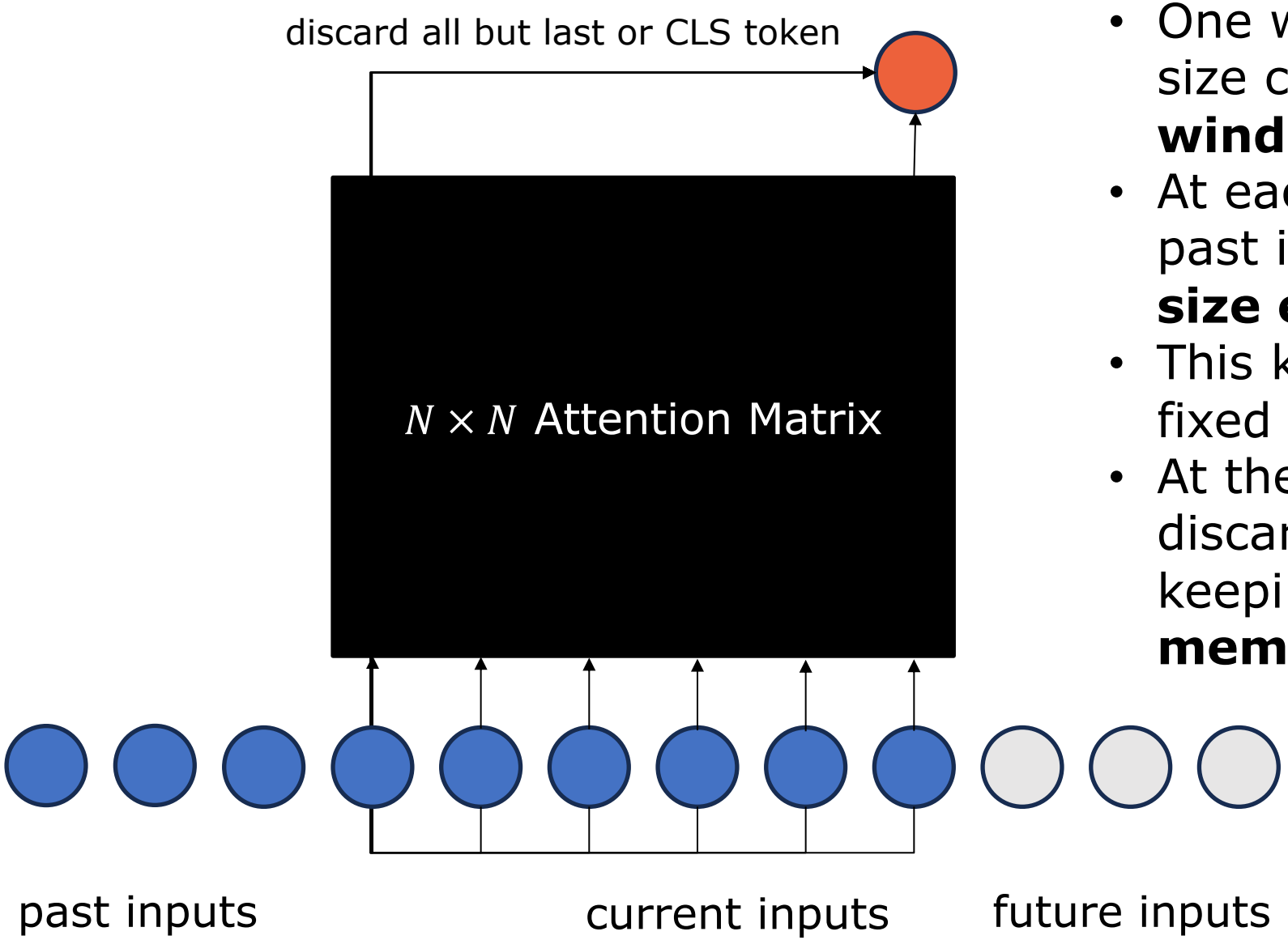


Online Processing in Practice (KV-Cache)



Problem: KV cache size and computation increase with time.

Sliding Window Transformer



- One way to keep the KV cache's size constant is to use a **sliding window**;
- At each time-step, we discard past inputs, keeping a **fixed size equal to N**;
- This keeps the attention matrix fixed to **N x N**;
- At the same time, however, it discards past inputs, practically keeping only a «**short memory**» of past observations.

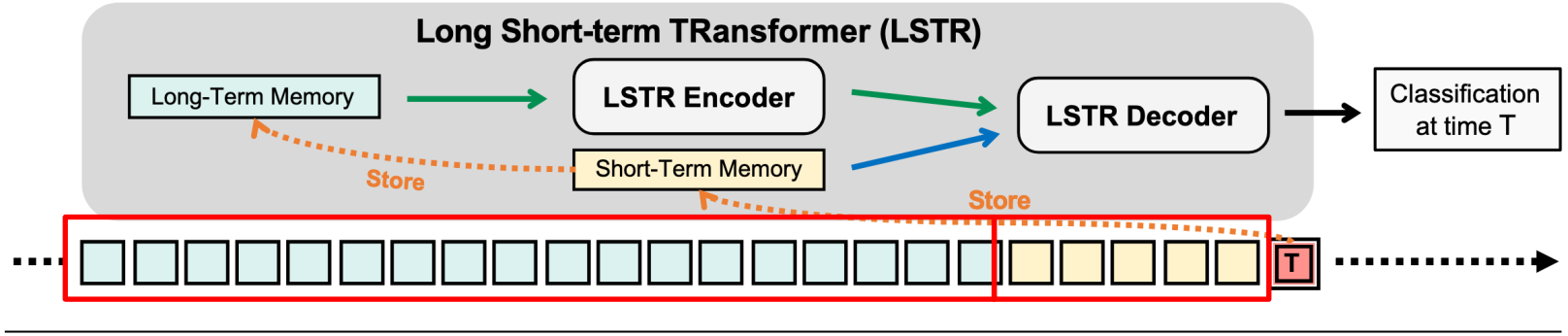
Long Short-Term Transformer for Online Action Detection

Mingze Xu **Yuanjun Xiong** **Hao Chen** **Xinyu Li**
Wei Xia **Zhuowen Tu** **Stefano Soatto**
Amazon/AWS AI
{xumingze,yuanjx,hxen,xxnl,wxia,ztu,soattos}@amazon.com

Abstract

We present Long Short-term TRansformer (LSTR), a temporal modeling algorithm for online action detection, which employs a long- and short-term memory mechanism to model prolonged sequence data. It consists of an LSTR encoder that dynamically leverages coarse-scale historical information from an extended temporal window (*e.g.*, 2048 frames spanning of up to 8 minutes), together with an LSTR decoder that focuses on a short time window (*e.g.*, 32 frames spanning 8 seconds) to model the fine-scale characteristics of the data. Compared to prior work, LSTR provides an effective and efficient method to model long videos with fewer heuristics, which is validated by extensive empirical analysis. LSTR achieves state-of-the-art performance on three standard online action detection benchmarks, THUMOS'14, TVSeries, and HACS Segment. Code has been made available at: https://xumingze0308.github.io/projects/lstr.

Long Short-Term Transformer (LSTR)

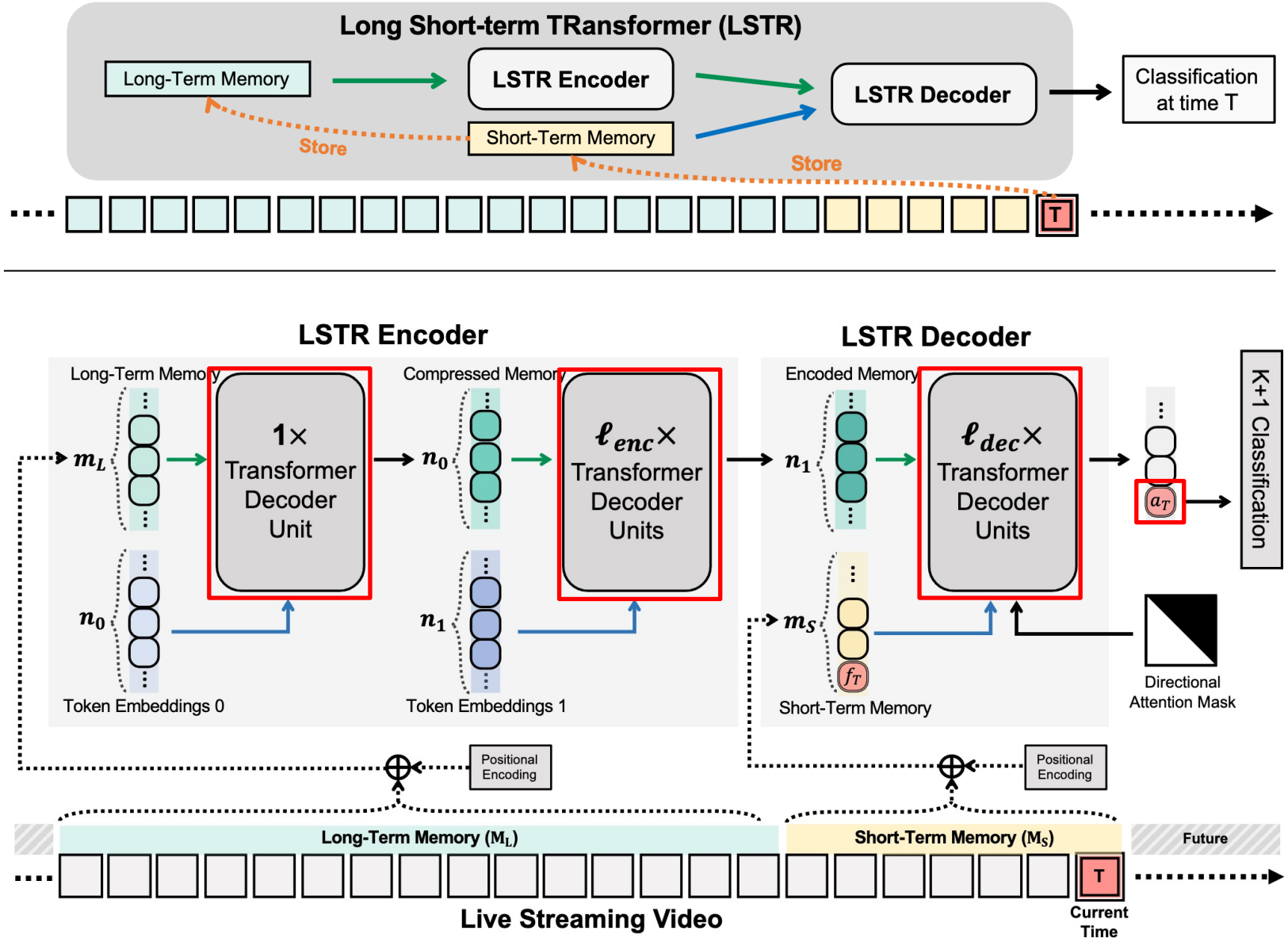


LSTR tries to address this problem by using **two memories**:

- **Long-term**: all past observations, compressed with an encoder to reduce size
- **Short-term**: within a small sliding window, used to «index» content from the long-term memory

Xu, Mingze, et al. "Long short-term transformer for online action detection." *Advances in Neural Information Processing Systems* 34 (2021): 1086-1099.

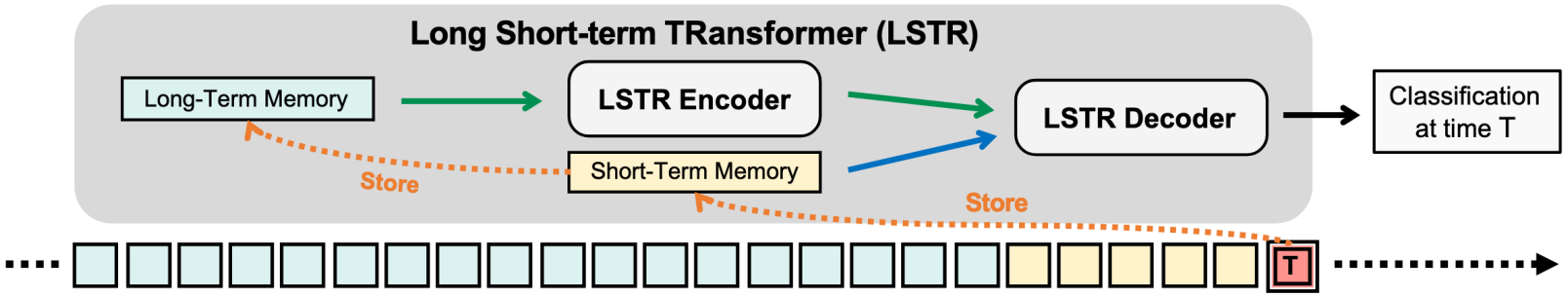
Long Short-Term Transformer (LSTR)

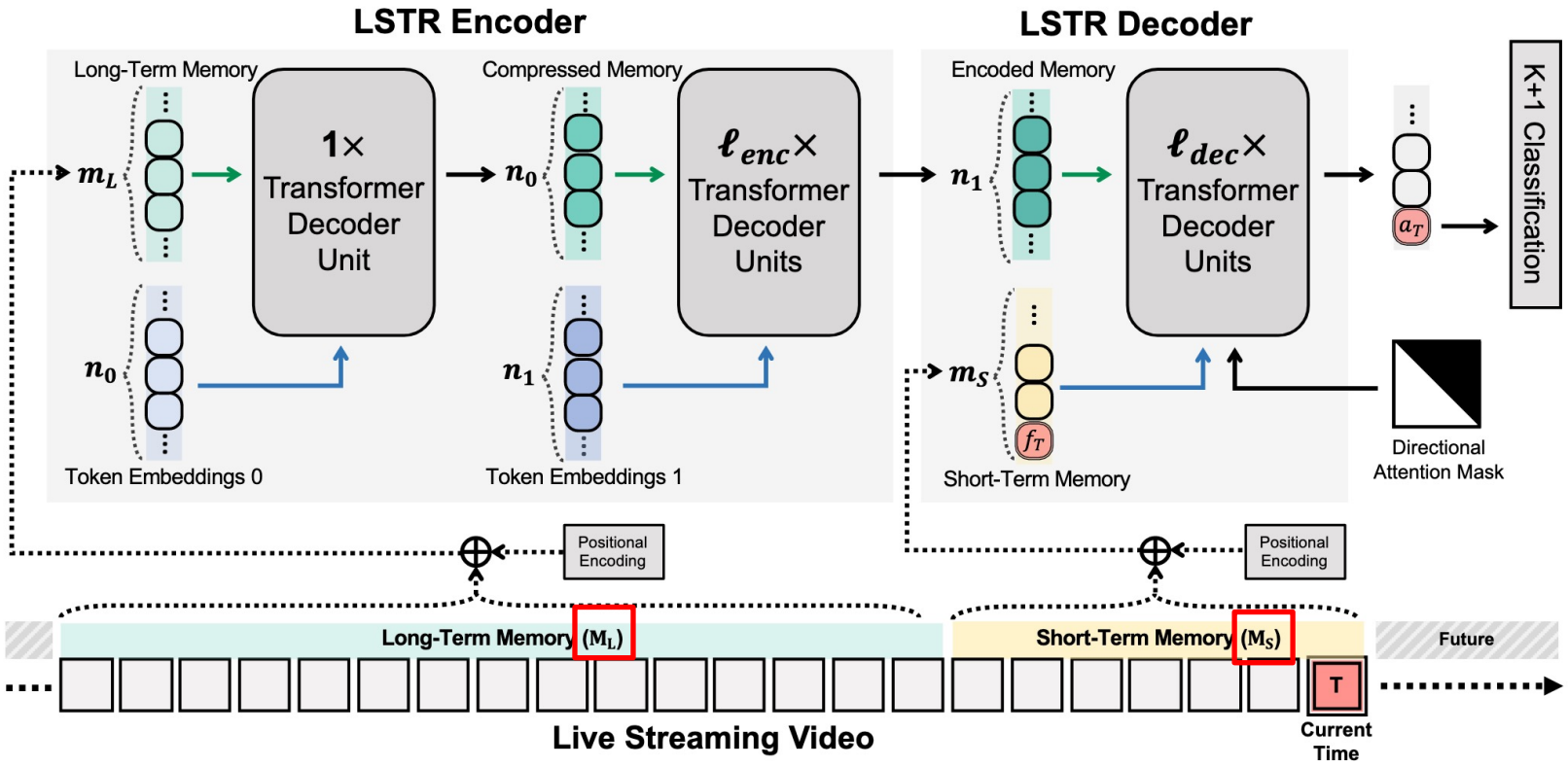


- **Compression** is achieved using a **single decoder layer** and a set of learned token embeddings
- The compressed memory is further transformerd with a **multi-layer decoder**
- The short-term memory is then used to **retrieve information from the encoded memory** using another multi-layer decoder
- The prediction corresponding to the current time is taken as **current prediction**

Xu, Mingze, et al. "Long short-term transformer for online action detection." *Advances in Neural Information Processing Systems* 34 (2021): 1086-1099.

Long Short-Term Transformer (LSTR)





- Long and short memory **sizes are fixed to L and S** respectively
- Compression is optimized by **caching previous keys and values** to save computation
- This approach allows the model to work in a **«streaming» fashion, where elements from the short-term memory progressively transit into the long-term memory**, which is compressed and kept as a reference.

Still discards old values

Xu, Mingze, et al. "Long short-term transformer for online action detection." *Advances in Neural Information Processing Systems* 34 (2021): 1086-1099.

Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention

Angelos Katharopoulos [1 2] **Apoorv Vyas** [1 2] **Nikolaos Pappas** [3] **François Fleuret** [2 4 *]

Abstract

Transformers achieve remarkable performance in several tasks but due to their quadratic complexity, with respect to the input's length, they are prohibitively slow for very long sequences. To address this limitation, we express the self-attention as a linear dot-product of kernel feature maps and make use of the associativity property of matrix products to reduce the complexity from $\mathcal{O}\left(N^2\right)$ to $\mathcal{O}\left(N\right)$, where N is the sequence length. We show that this formulation permits an iterative implementation that dramatically accelerates autoregressive transformers and reveals their relationship to recurrent neural networks. Our *linear transformers* achieve similar performance to vanilla transformers and they are up to 4000x faster on autoregressive prediction of very long sequences.

by the global receptive field of self-attention, which processes contexts of N inputs with a quadratic memory and time complexity $\mathcal{O}\left(N^2\right)$. As a result, in practice transformers are slow to train and their context is *limited*. This disrupts temporal coherence and hinders the capturing of long-term dependencies. Dai et al. (2019) addressed the latter by attending to memories from previous contexts albeit at the expense of computational efficiency.

Lately, researchers shifted their attention to approaches that increase the context length without sacrificing efficiency. Towards this end, Child et al. (2019) introduced sparse factorizations of the attention matrix to reduce the self-attention complexity to $\mathcal{O}\left(N\sqrt{N}\right)$. Kitaev et al. (2020) further reduced the complexity to $\mathcal{O}\left(N \log N\right)$ using locality-sensitive hashing. This made scaling to long sequences possible. Even though the aforementioned models can be efficiently trained on large sequences, they do not speed-up autoregressive inference.

Transformers are RNNs

Similarity-Based Attention

Standard Attention

$$
\begin{aligned}
Q &= xW_Q, \\
K &= xW_K, \\
V &= xW_V,
\end{aligned}
$$

$$
A_l(x) = V' = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right) V.
$$

Equivalent To

$$
V_i' = \frac{\sum_{j=1}^{N} \text{sim}\left(Q_i, K_j\right) V_j}{\sum_{j=1}^{N} \text{sim}\left(Q_i, K_j\right)}.
$$

$$
\text{sim}\left(q, k\right) = \exp\left(\frac{q^T k}{\sqrt{D}}\right)
$$

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020, November). Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning* (pp. 5156-5165). PMLR.

Transformers are RNNs

Linearized Attention and Kernel View

Equivalent To

$$V_i' = \frac{\sum_{j=1}^{N} \text{sim}\left(Q_i, K_j\right) V_j}{\sum_{j=1}^{N} \text{sim}\left(Q_i, K_j\right)}.$$

$$\text{sim}\left(q, k\right) = \exp\left(\frac{q^T k}{\sqrt{D}}\right)$$

In practice, we obtain a valid attention choosing sim as any valid kernel function $k(x, y) = \Re^{2 \times F} \rightarrow R_+$ (F feature size).

Siven such a kernel with feature representation ϕ (i.e., such that $k(x, y) = \phi(x)\phi(y)$), we can re-write attention as follows:

$$V_i' = \frac{\sum_{j=1}^{N} \phi\left(Q_i\right)^T \phi\left(K_j\right) V_j}{\sum_{j=1}^{N} \phi\left(Q_i\right)^T \phi\left(K_j\right)},$$

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020, November). Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning* (pp. 5156-5165). PMLR.

Transformers are RNNs

Linear Complexity and Elu Function

Given such a kernel with a feature representation $\phi(x)$ we can rewrite equation 2 as follows,

$$V_i' = \frac{\sum_{j=1}^{N} \phi(Q_i)^T \phi(K_j) V_j}{\sum_{j=1}^{N} \phi(Q_i)^T \phi(K_j)}, \tag{4}$$

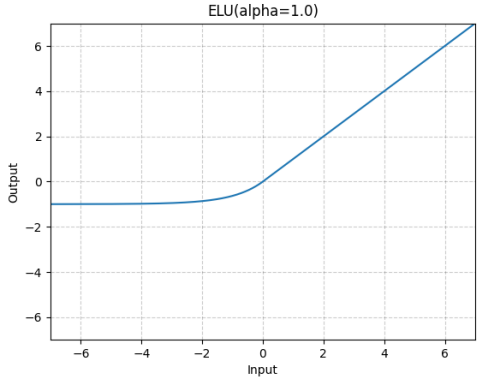
and then further simplify it by making use of the associative property of matrix multiplication to

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j)}. \tag{5}$$

This is much more efficient ($O(N)$ vs $O(N^2)$) as we can compute $\sum_j \phi(K_j) V_j^T$ and $\sum_j \phi(K_j)$ once and re-use it for every query.

Authors use:

$$\phi(x) = \text{elu}(x) + 1,$$



Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020, November). Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning* (pp. 5156-5165). PMLR.

Causal Masking

Causal masking for online processing changes attention form this form:

$$V_i' = \frac{\sum_{j=1}^{N} \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{N} \text{sim}(Q_i, K_j)}.$$

To this form:

$$V_i' = \frac{\sum_{j=1}^{i} \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{i} \text{sim}(Q_i, K_j)}.$$

Following the kernel view:

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^{i} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{i} \phi(K_j)}$$

Or Alternatively:

$$V_i' = \frac{\phi(Q_i)^T S_i}{\phi(Q_i)^T Z_i} \quad \text{with} \quad S_i = \sum_{j=1}^{i} \phi(K_j) V_j^T, \quad Z_i = \sum_{j=1}^{i} \phi(K_j),$$

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020, November). Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning* (pp. 5156-5165). PMLR.

Transformers are RNNs

Linear transformers with causal masking

$$V_i' = \frac{\phi\left(Q_i\right)^T S_i}{\phi\left(Q_i\right)^T Z_i}$$

with

$$S_i = \sum_{j=1}^{i} \phi\left(K_j\right) V_j^T,$$

$$Z_i = \sum_{j=1}^{i} \phi\left(K_j\right),$$

Recurrent View

$$s_0 = 0,$$

$$z_0 = 0,$$

$$s_i = s_{i-1} + \phi\left(x_i W_K\right)\left(x_i W_V\right)^T,$$

$$z_i = z_{i-1} + \phi\left(x_i W_K\right),$$

$$y_i = f_l\left(\frac{\phi\left(x_i W_Q\right)^T s_i}{\phi\left(x_i W_Q\right)^T z_i} + x_i\right).$$

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020, November). Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning* (pp. 5156-5165). PMLR.

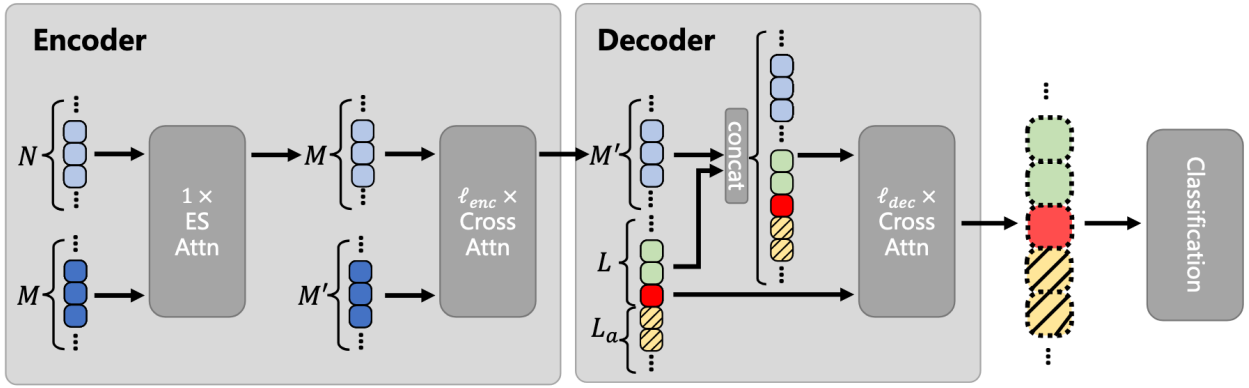
Real-time Online Video Detection with Temporal Smoothing Transformers

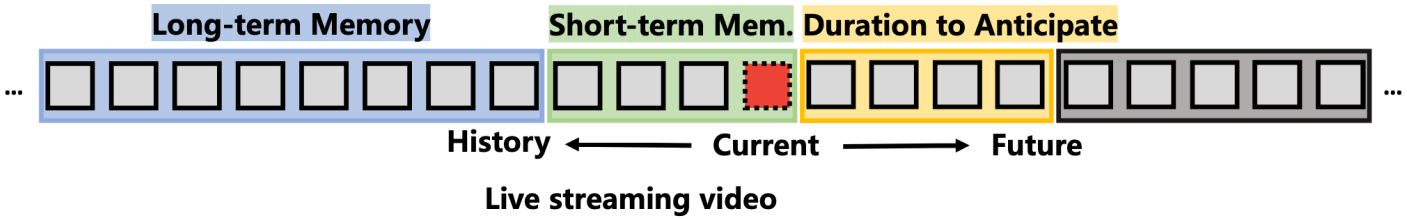
Yue Zhao[1] and Philipp Krähenbühl[1]

University of Texas at Austin, Austin TX 78712, USA
{yzhao,philkr}@cs.utexas.edu

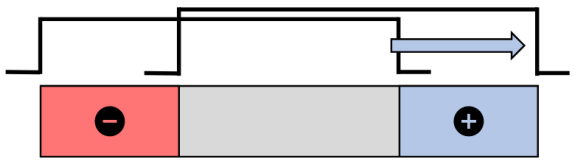
Abstract. Streaming video recognition reasons about objects and their actions in every frame of a video. A good streaming recognition model captures both long-term dynamics and short-term changes of video. Unfortunately, in most existing methods, the computational complexity grows linearly or quadratically with the length of the considered dynamics. This issue is particularly pronounced in transformer-based architectures. To address this issue, we reformulate the cross-attention in a video transformer through the lens of kernel and apply two kinds of temporal smoothing kernel: A box kernel or a Laplace kernel. The resulting streaming attention reuses much of the computation from frame to frame, and only requires a constant time update each frame. Based on this idea, we build TeSTra, a Temporal Smoothing Transformer, that takes in arbitrarily long inputs with constant caching and computing overhead. Specifically, it runs 6× faster than equivalent sliding-window based transformers with 2,048 frames in a streaming setting. Furthermore, thanks to the increased temporal span, TeSTra achieves state-of-the-art results on THUMOS'14 and EPIC-Kitchen-100, two standard online action detection and action anticipation datasets. A real-time version of TeSTra outperforms all but one prior approaches on the THUMOS'14 dataset.

Temporal Smoothing Transformers (TeSTra)

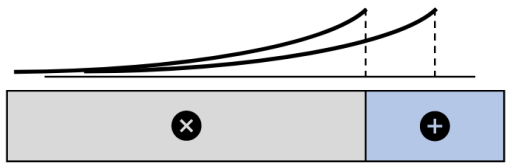




$$K_B(\omega_t, \omega_n) = 1_{[t-n<N]}$$



$$K_L(\omega_t, \omega_n) = e^{-\lambda(t-n)}$$



$$\psi(t) = \psi(t-1) + \kappa(\mathbf{q}_m, \mathbf{k}_t) - \kappa(\mathbf{q}_m, \mathbf{k}_{t-N})$$
$$\phi(t) = \phi(t-1) + \kappa(\mathbf{q}_m, \mathbf{k}_t)\mathbf{v}_t - \kappa(\mathbf{q}_m, \mathbf{k}_{t-N})\,\mathbf{v}_{t-N}$$

$$\psi(t) = e^{-\lambda} \cdot \psi(t-1) + \kappa(\mathbf{q}_m, \mathbf{k}_t)$$
$$\phi(t) = e^{-\lambda} \cdot \phi(t-1) + \kappa(\mathbf{q}_m, \mathbf{k}_t)\,\mathbf{v}_t$$

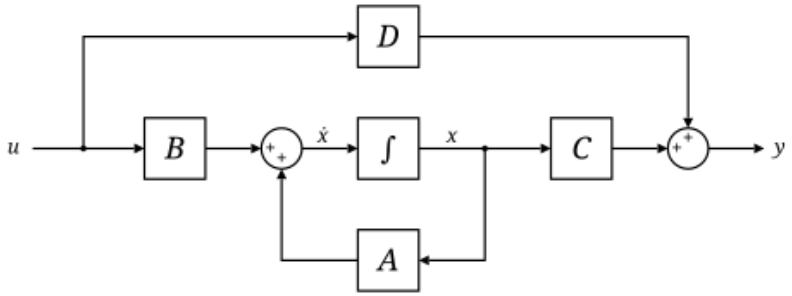
- Similar as LSTR, but **optimizes long-term memory computation** and also tackles anticipation
- To make long-term memory encoding more efficient, implements a **«streaming attention» which follows the online attention fomulation** using a kernel. Two kernels are explored:
  - A **box kernel** only considers the last N elements
  - The **Laplace kernel** uses an exponential function to weigh distant elements less
- In both cases, the computation of long-term memory can be perfomed in a recurrent fashion

Zhao, Yue, and Philipp Krähenbühl. "Real-time online video detection with temporal smoothing transformers." *European Conference on Computer Vision*. 2022.

State Space Models and Mamba (or the return of RNNs)

State Space Models

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)$$
$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t)$$



Where:

$\mathbf{x}(\cdot)$ is called the "state vector", $\mathbf{x}(t) \in \mathbb{R}^n$;
$\mathbf{y}(\cdot)$ is called the "output vector", $\mathbf{y}(t) \in \mathbb{R}^q$;
$\mathbf{u}(\cdot)$ is called the "input (or control) vector", $\mathbf{u}(t) \in \mathbb{R}^p$;
$\mathbf{A}(\cdot)$ is the "state (or system) matrix", $\dim[\mathbf{A}(\cdot)] = n \times n$,
$\mathbf{B}(\cdot)$ is the "input matrix", $\dim[\mathbf{B}(\cdot)] = n \times p$,
$\mathbf{C}(\cdot)$ is the "output matrix", $\dim[\mathbf{C}(\cdot)] = q \times n$,
$\mathbf{D}(\cdot)$ is the "feedthrough (or feedforward) matrix" (in cases where the system model does not have a direct feedthrough, $\mathbf{D}(\cdot)$ is the zero matrix), $\dim[\mathbf{D}(\cdot)] = q \times p$,
$\dot{\mathbf{x}}(t) := \frac{d}{dt}\mathbf{x}(t)$.

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)$$
$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t)$$

- In **control engineering**, a state space model describes how a system evolves over time by defining (two equations):
  - **State evolution**: how the internal state changes
  - **Observation model**: how the output (what we observe) relates to the internal state
- The internal state of the state space model is very similar to the concept of «**hidden state**» in an RNN

A, B, C, and D are usually set by hand depending on our knowledge of the system.

https://en.wikipedia.org/wiki/State-space_representation

State Space Models

Ideally, if we know $\dot{x}(t)$, we can compute the state at future times

$$x(t') = x(t) + \int_t^{t'} \dot{x}(\tau)\, d\tau$$

Derivative of x with respect to time «how the state evolves in time»

Current state

Current input

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)$$

$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t)$$

Current state

Current input

- $A(t)$, $B(t)$, $C(t)$, and $D(t)$ are generally **time-dependent**
- However, we can simplify and **drop the dependence on time**, considering these as constant matrices
- We hence obtain a **Linear Time Invariant (LTI)** system where matrices do not evolve with time

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$$

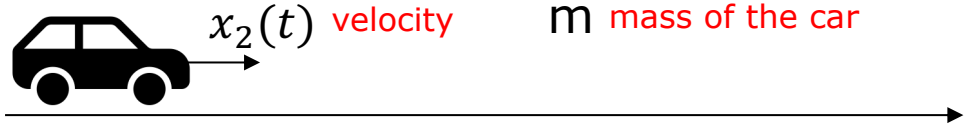
$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)$$

https://en.wikipedia.org/wiki/State-space_representation

Continuous State Space model Example

force applied with the gas pedal to make the car move

$u(t)$



$$F_f(t) = -b \cdot x_2(t)$$

Friction force (proportional to velocity)

$$x(t) = \begin{bmatrix} x_{1(t)} \\ x_{2(t)} \end{bmatrix} \begin{matrix} \leftarrow \text{position} \\ \leftarrow \text{velocity} \end{matrix}$$

System state

To build a state space model, we now need to find the derivatives of the state. For $\dot{x}_1(t)$ it's easy, as velocity is the derivative of position:

$$\underbrace{\dot{x}_1(t) = x_2(t)}_{\text{how position evolves with time}}$$

For $\dot{x}_2(t)$, recall $F = m \cdot a$ (mass times acceleration), so:

$$\underbrace{m \cdot \dot{x}_2(t)}_{\text{acceleration (derivative of velocity)}} = \underbrace{u(t)}_{\text{gas}} - \underbrace{b \cdot x_2(t)}_{\text{friction}} \quad \Rightarrow \underbrace{\dot{x}_2(t) = \frac{u(t)}{m} - \frac{b}{m} x_2(t)}_{\text{how velocity evolves with time and input } u(t)}$$

We finally have this sytem describing the dynamics of the car:

$$\begin{cases} \dot{x}_1(t) = x_2(t) \\ \dot{x}_2(t) = \frac{u(t)}{m} - \frac{b}{m} x_2(t) \end{cases}$$

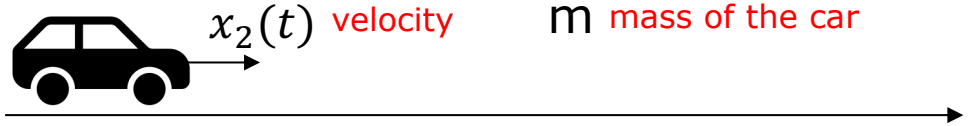
- Imagine a car moving in a straight line.
- We want to model its position and velocity over time.
- The system state $x(t)$ will be made of:
  - $x_1(t)$: position at each moment in time
  - $x_2(t)$: velocity at each moment in time
- We also assume that the car has friction (air or road) proportional to the car's velocity with constant $b > 0$
- The car's state is controlled by the input acceleration $u(t)$
  - Note that the position and velocity at each time step are not just determined by the acceleration, but also by the state of the system (past velocity and position) which act as a sort of «memory» of the system.

https://www.flaticon.com/free-icon/gas-pedal_3409879

Continuous State Space model Example

force applied with the gas pedal to make the car move

$u(t)$



$$F_f(t) = -b \cdot x_2(t)$$

Friction force (proportional to velocity)

$$x(t) = \begin{bmatrix} x_{1(t)} \\ x_{2(t)} \end{bmatrix} \begin{matrix} \leftarrow \text{position} \\ \leftarrow \text{velocity} \end{matrix}$$

System state

To build a state space model, we now need to find the derivatives of the state. For $\dot{x}_1(t)$ it's easy, as velocity is the derivative of position:

$$\underbrace{\dot{x}_1(t) = x_2(t)}$$

how position evolves with time

For $\dot{x}_2(t)$, recall $F = m \cdot a$ (mass times acceleration), so:

$$\underbrace{m \cdot \dot{x}_2(t)}_{\text{acceleration (derivative of velocity)}} = \underbrace{u(t)}_{\text{gas}} - \underbrace{b \cdot x_2(t)}_{\text{friction}} \quad \Rightarrow \underbrace{\dot{x}_2(t) = \frac{u(t)}{m} - \frac{b}{m} x_2(t)}_{\text{how velocity evolves with time and input } u(t)}$$

We finally have this sytem describing the dynamics of the car:

$$\begin{cases} \dot{x}_1(t) = x_2(t) \\ \dot{x}_2(t) = \frac{u(t)}{m} - \frac{b}{m} x_2(t) \end{cases}$$

We can rewrite the system as follows:

$$\dot{x}(t) = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & -\frac{b}{m} \end{bmatrix}}_{A} x(t) + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}}_{B} u(t)$$

System of equations in matrix form

Also, since we are interested in knowing only the position as output, our output equation will be:

$$y(t) = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C} x(t)$$

Selects position and discards velocity

In practice, we defined the following time-invariant state space model:

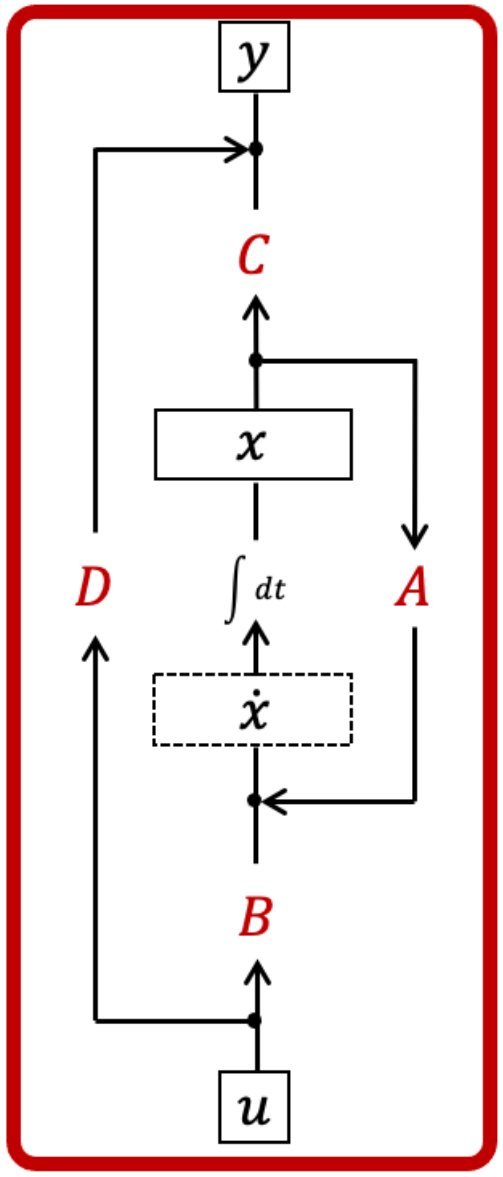
$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

Where:

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{b}{m} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = 0$$

https://www.flaticon.com/free-icon/gas-pedal_3409879

Continuous State Space Models



$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)$$

$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t)$$

- We can see continuous state space models as shown on the left.
- The input u is multiplied by the matrix B
- The current state x is multiplied by A
- These are added to compute the derivative $\dot{x}$
- Integration allows us to compute a new value of x
- The state x is multiplied by C to obtain the output
- Finally, u is multiplied by D and the result is added to the output
- The last term can be seen as a skip connection from input to output (similar to ResNet)

https://hazyresearch.stanford.edu/blog/2022-01-14-s4-3

Discretized State Space Model

$u(t)$ force applied with the gas pedal to make the car move

$$\dot{x}(t) = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & -\frac{b}{m} \end{bmatrix}}_{A} x(t) + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}}_{B} u(t) \qquad y(t) = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C} x(t)$$

The state space model defines the dynamics of the car with respect to the input force in a continuous space. However, we are treating t as a continuous quantity, while for practical applications (e.g., simulations), we need to reason in discrete terms. For instance, we may want to compute the position of the car every Δt seconds.

To do so, we need to discretize our state space model. This can be done in different ways. One easy way is to use the Euler method.

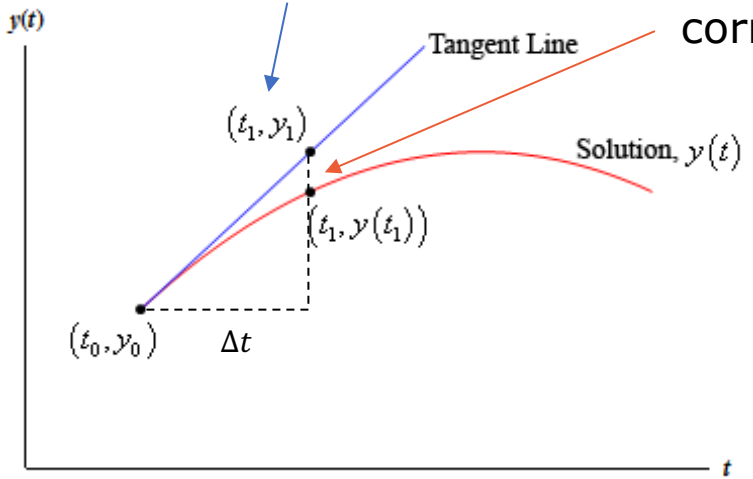
In general, let's assume:

$$\frac{dy}{dt} = f(t, y), \qquad y(t_o) = 0$$

We want to compute

$$y(t_1 = t_0 + \Delta t)$$

Euler approximation



correct, analytical solution

$$y(t_1 = t_0 + \Delta t) = y(t_0) + f(t_0, y_0)(t_1 - t_0)$$
$$= y(t_0) + f(t_0, y_0)\Delta t$$

If Δt is small enough, the approximation is not too bad

https://www.flaticon.com/free-icon/gas-pedal_3409879

https://tutorial.math.lamar.edu/classes/de/eulersmethod.aspx

Discretized State Space Model

$u(t)$

force applied with the gas pedal to make the car move

$$\dot{x}(t) = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & -\frac{b}{m} \end{bmatrix}}_{A} x(t) + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}}_{B} u(t) \qquad y(t) = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C} x(t)$$

We now assume discretized steps k happening at a distance of Δt. Applying Euler's rule to our system, we obtain:

$$\dot{x}_1(t) = x_2(t)$$
$$\dot{x}_2(t) = \frac{u(t)}{m} - \frac{b}{m} x_2(t)$$

$$x_1[k+1] = x_1[k] + \Delta t \cdot x_2[k]$$
$$x_2[k+1] = x_2[k] + \Delta t \cdot \left(-\frac{b}{m} x_2[k] + \frac{1}{m}\, u[k]\right)$$

Equivalently:

$$x[k+1] = A_d x[k] + B_d u[k]$$

$$A_d = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 - \frac{b}{m}\Delta t \end{bmatrix}, \quad B_d = \begin{bmatrix} 0 \\ \frac{\Delta t}{m} \end{bmatrix}$$

$$y[k] = Cx[k], \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

In general terms, we convert a continuous state space model discretizing the matrices with some formula:

$$A_d = \text{discretize}_A(A, \Delta t), \quad B_d = \text{discretize}_B(B, A, \Delta t)$$

If we use the forward Euler rule (but we could use other rules):

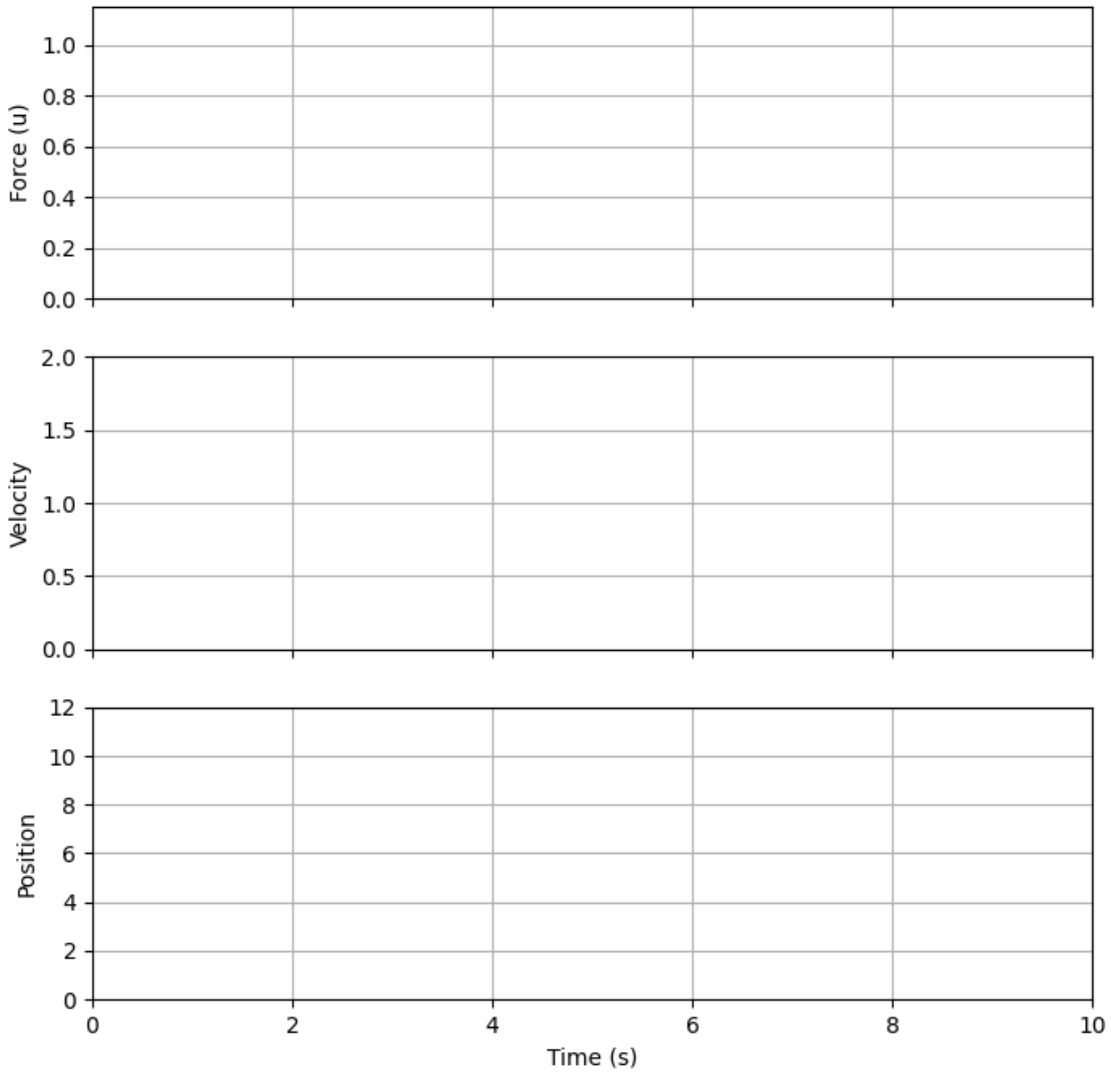
$$A_d \approx I + \Delta t \cdot A, \quad B_d \approx \Delta t \cdot B$$

https://www.flaticon.com/free-icon/gas-pedal_3409879

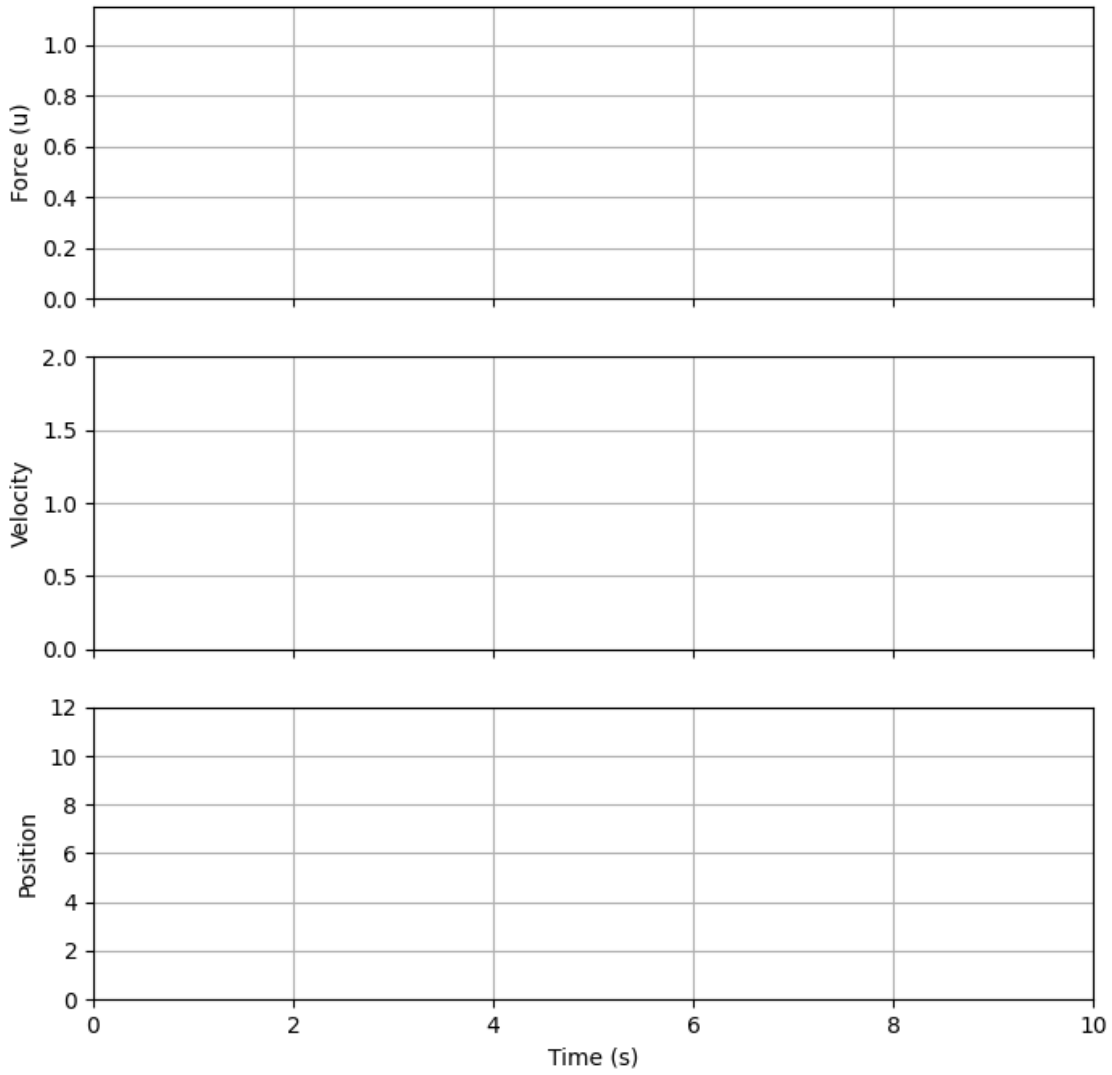
https://tutorial.math.lamar.edu/classes/de/eulersmethod.aspx

Discretized State Space Model - Simulation

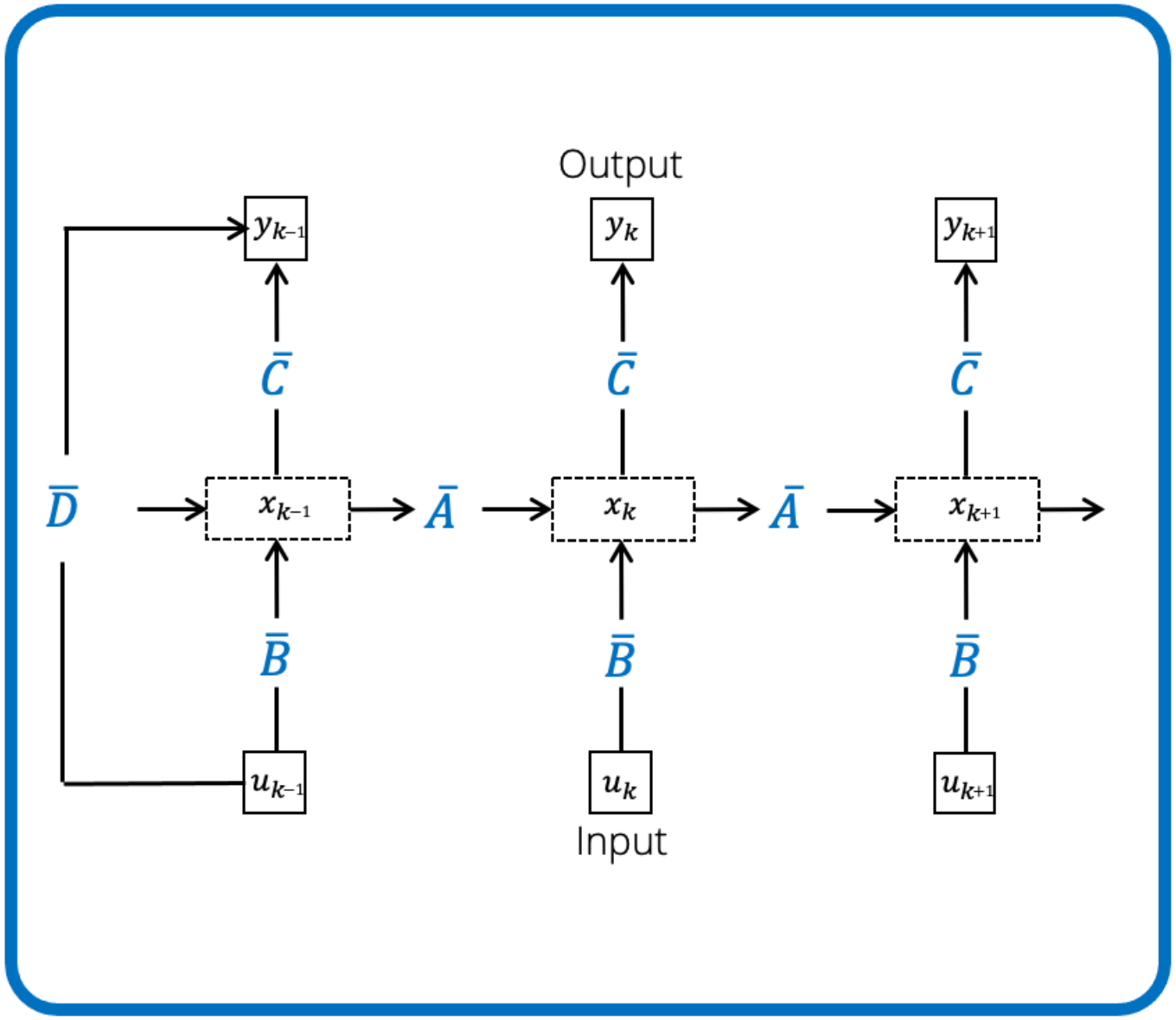
$\Delta t = 0.1$



$\Delta t = 0.5$



Discretized State Space Models



$$x_{k+1} = A_d x_k + B_d u_k \qquad \text{(state update)}$$
$$y_k = C_d x_k + D_d u_k \qquad \text{(output)}$$

- Discretized state space models can be seen as shown on the left
- This is similar to the continuous view with A and B used to compute the next state, C used to compute the output, and D allowing to implement a skip connection

https://hazyresearch.stanford.edu/blog/2022-01-14-s4-3

A. Gu *et al.*, «Combining Recurrent, Convolutional, and Continuous-time Models with Linear State-Space Layers», 2021, *arXiv*: arXiv:2110.13985.

State Space Models vs RNNs

State Space Models

$$x_{k+1} = A_d x_k + B_d u_k \qquad \text{(state update)}$$
$$y_k = C_d x_k + D_d u_k \qquad \text{(output)}$$

This is usually omitted

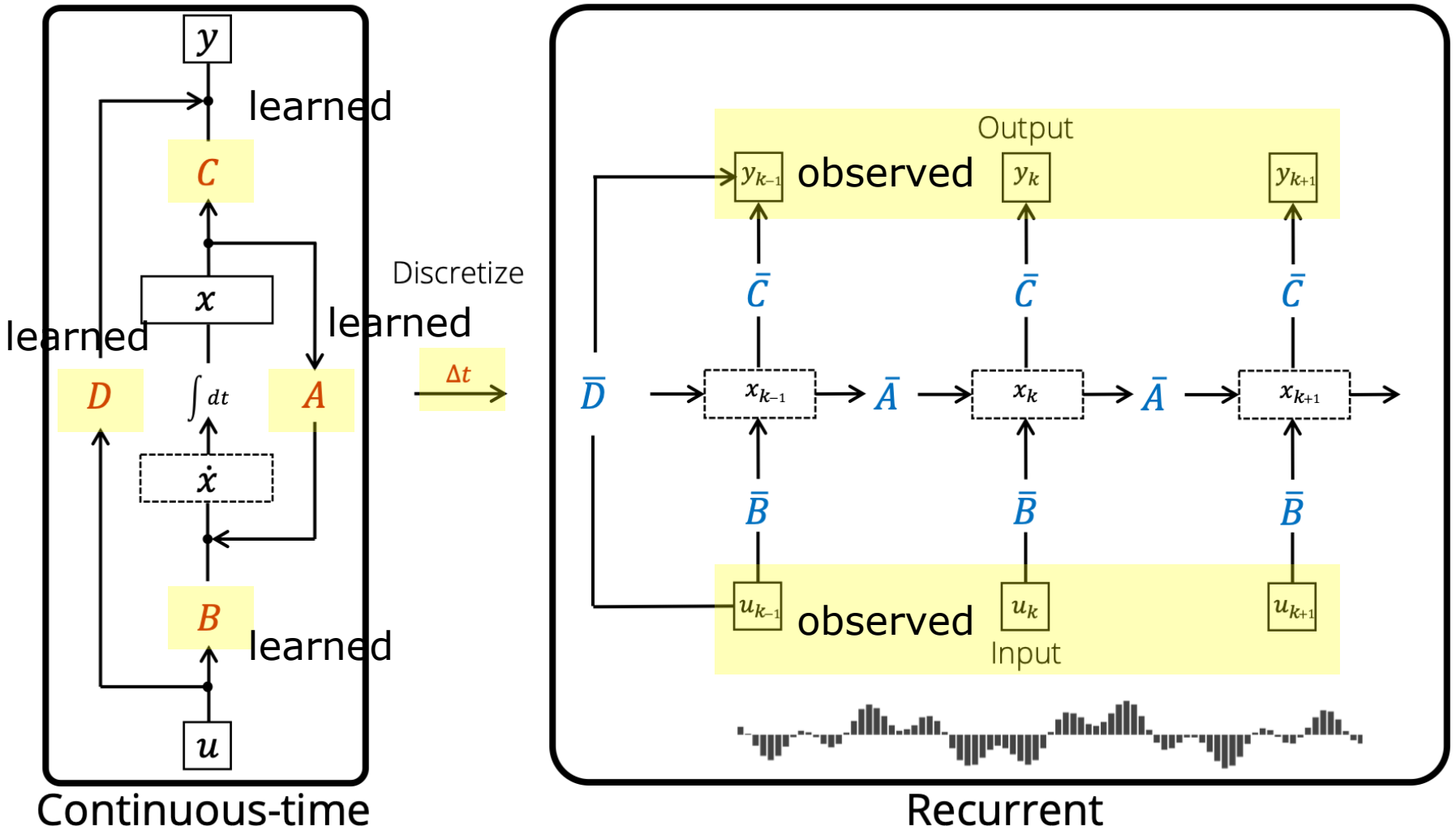
VS

Recurrent Neural Networks

$$h_{k+1} = \sigma(W_h h_k + W_x u_k + b_h) \qquad \text{(hidden state)}$$
$$y_k = W_y h_k + b_y \qquad \text{(output)}$$

Very similar, apart from bias terms, **non-linearity**, and a skip-connection

Neural State Space Models



- Similar to RNNs, **rather than setting the A, B, C, and D matrices by hand, we can learn them**!
- Since the **discrete matrices can be computed from the continuous ones** (e.g., A vs $\overline{A}$) with known differentiable formulas, we can actually learn the continuous versions
- Note that different SSMs can use **different discretizations**

A. Gu *et al.*, «Combining Recurrent, Convolutional, and Continuous-time Models with Linear State-Space Layers», 2021, *arXiv*: arXiv:2110.13985.

State Space Model Computation

Algorithm 1 SSM (S4)

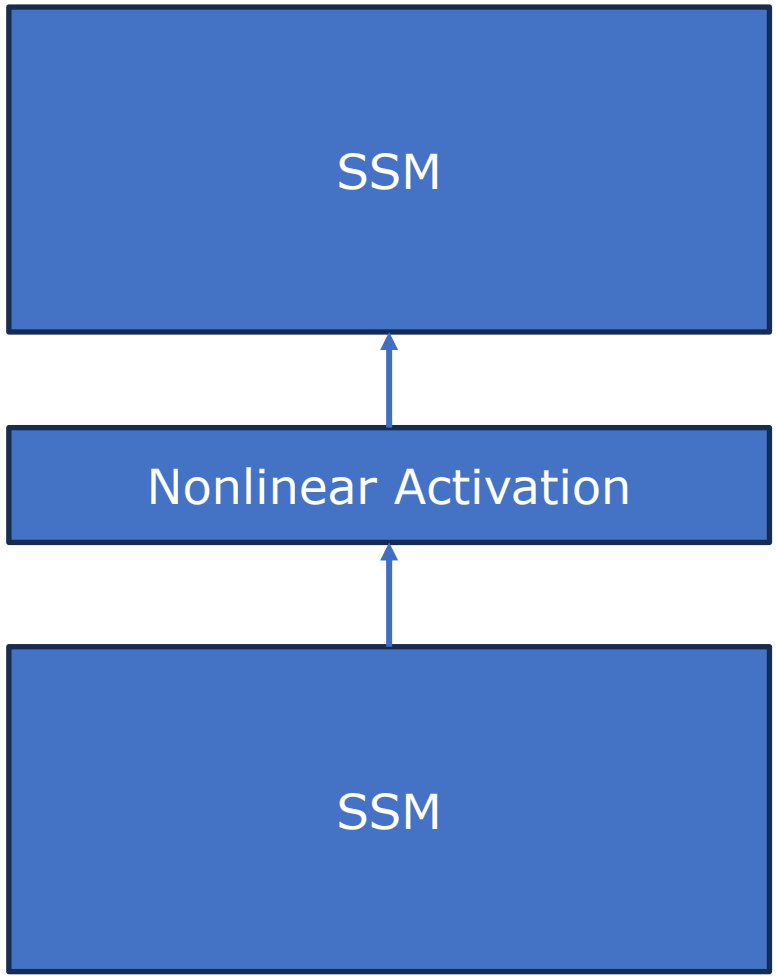
Input: $x : (\mathsf{B}, \mathsf{L}, \mathsf{D})$
Output: $y : (\mathsf{B}, \mathsf{L}, \mathsf{D})$

1: $\boldsymbol{A} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
2: $\boldsymbol{B} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
3: $\boldsymbol{C} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
4: $\Delta : (\mathsf{D}) \leftarrow \tau_\Delta(\text{Parameter})$
5: $\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathsf{D}, \mathsf{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
6: $y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-invariant: recurrence or convolution
7: **return** y

- An SSM learns continuous A, B, and C matrices, as well as a discretization parameter Δ;
- A and B are discretized using known formulas and used to compute the SSM;
- This can be computed recurrently (tipically at test time), but it can also computed with a convolution to speed up training.

A. Gu e T. Dao, «Mamba: Linear-Time Sequence Modeling with Selective State Spaces», 2024, *arXiv*: arXiv:2312.00752.

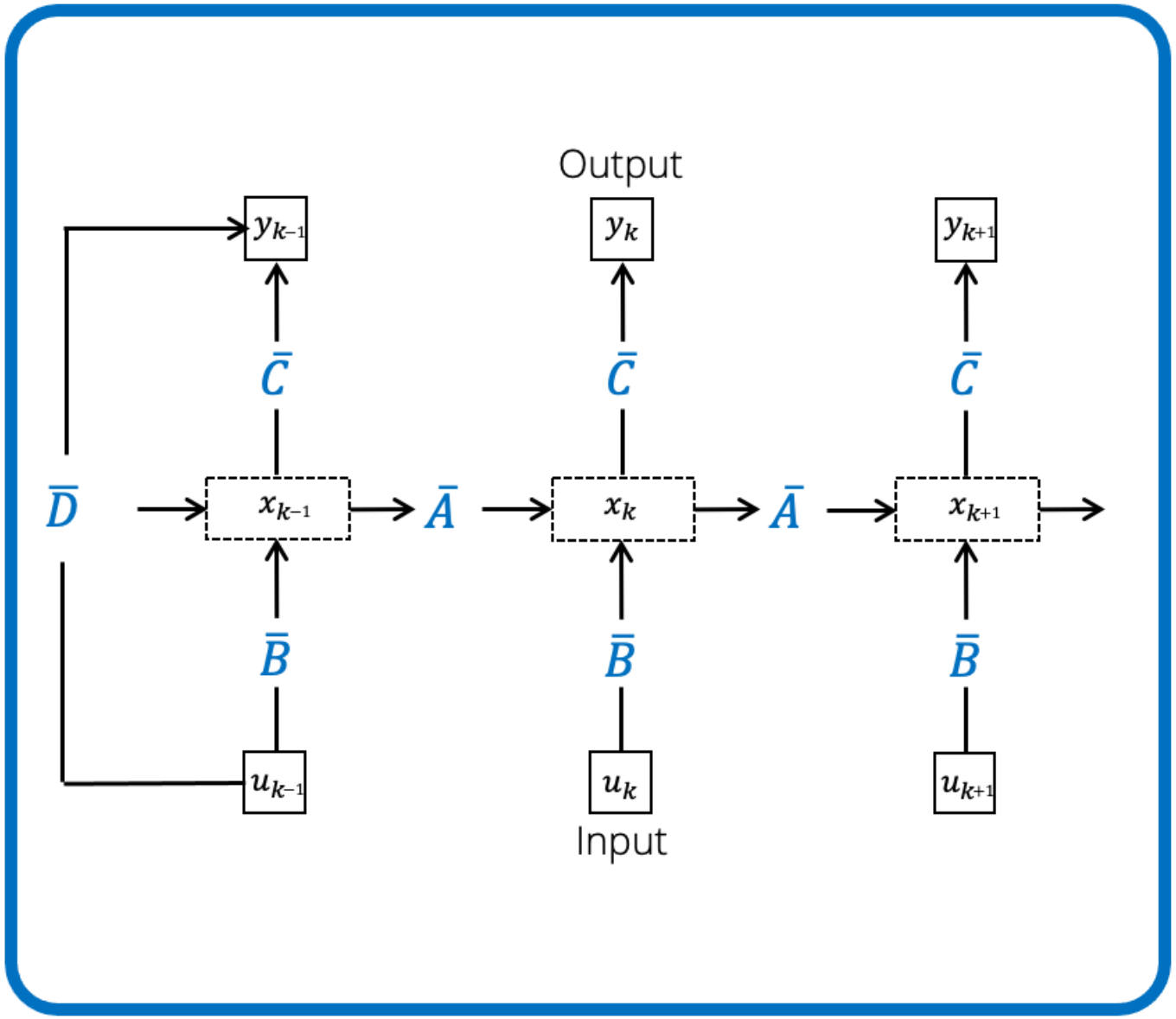
So… are SSMs just linear models?



$$x_{k+1} = A_d x_k + B_d u_k \qquad \text{(state update)}$$
$$y_k = C_d x_k + D_d u_k \qquad \text{(output)}$$

- While SSMs are linear, we can easily make them more expressive by stacking SMM layers and putting non-linearities in between
- This is the same approach followed by Transformers
- Indeed, attention is a linear operation at its core, with final MLPs adding non-linearities
- By stacking multiple layers in a deep architectures, SSMs can model nonlinear relationships

Recurrence, isn't this slow?



- We know from RNNs, that they are hard to parallelize and hence slow at training time;
- Indeed, in order to compute the output of the model at time k, we need the output obtained at time k-1 due to the recurrence;
- Don't we have the same problem with SSMs?

https://hazyresearch.stanford.edu/blog/2022-01-14-s4-3

A. Gu *et al.*, «Combining Recurrent, Convolutional, and Continuous-time Models with Linear State-Space Layers», 2021, *arXiv*: arXiv:2110.13985.

Convolutional View of SSMs

It actually turns out that SSMs can be computed using a convolution:

$$x_k = \bar{\mathbf{A}} x_{k-1} + \bar{\mathbf{B}} u_k$$
$$y_k = \bar{\mathbf{C}} x_k$$

In general

$$y_k = \overline{CA}^k \overline{B} u_0 + \overline{CA}^{k-1} \overline{B} u_1 + \cdots + \overline{CAB} u_{k-1} + \overline{CB} u_k$$

$$y = \overline{K} * u \qquad \overline{K} \in \mathbb{R}^L = (\overline{CB}, \overline{CAB}, \ldots, \overline{CA}^{L-1} \overline{B})$$

Efficient if we pre-compute the kernel!

$$\text{Step 0: } x_0 = \bar{\mathbf{B}} u_0$$
$$\text{Step 0: } y_0 = \bar{\mathbf{C}} x_0 = \bar{\mathbf{C}} \bar{\mathbf{B}} u_0$$

$$\text{Step 1: } x_1 = \bar{\mathbf{A}} x_0 + \bar{\mathbf{B}} u_1 = \bar{\mathbf{A}} \bar{\mathbf{B}} u_0 + \bar{\mathbf{B}} u_1$$
$$\text{Step 1: } y_1 = \bar{\mathbf{C}} x_1 = \bar{\mathbf{C}} (\bar{\mathbf{A}} \bar{\mathbf{B}} u_0 + \bar{\mathbf{B}} u_1) = \bar{\mathbf{C}} \bar{\mathbf{A}} \bar{\mathbf{B}} u_0 + \bar{\mathbf{C}} \bar{\mathbf{B}} u_1$$

$$\text{Step 2: } x_2 = \bar{\mathbf{A}} x_1 + \bar{\mathbf{B}} u_2 = \bar{\mathbf{A}} (\bar{\mathbf{A}} \bar{\mathbf{B}} u_0 + \bar{\mathbf{B}} u_1) + \bar{\mathbf{B}} u_2 = \bar{\mathbf{A}}^2 \bar{\mathbf{B}} u_0 + \bar{\mathbf{A}} \bar{\mathbf{B}} u_1 + \bar{\mathbf{B}} u_2$$
$$\text{Step 2: } y_2 = \bar{\mathbf{C}} x_2 = \bar{\mathbf{C}} (\bar{\mathbf{A}}^2 \bar{\mathbf{B}} u_0 + \bar{\mathbf{A}} \bar{\mathbf{B}} u_1 + \bar{\mathbf{B}} u_2) = \bar{\mathbf{C}} \bar{\mathbf{A}}^2 \bar{\mathbf{B}} u_0 + \bar{\mathbf{C}} \bar{\mathbf{A}} \bar{\mathbf{B}} u_1 + \bar{\mathbf{C}} \bar{\mathbf{B}} u_2$$

https://huggingface.co/blog/lbourdois/get-on-the-ssm-train

Copying, Selective Copying, and Induction Heads

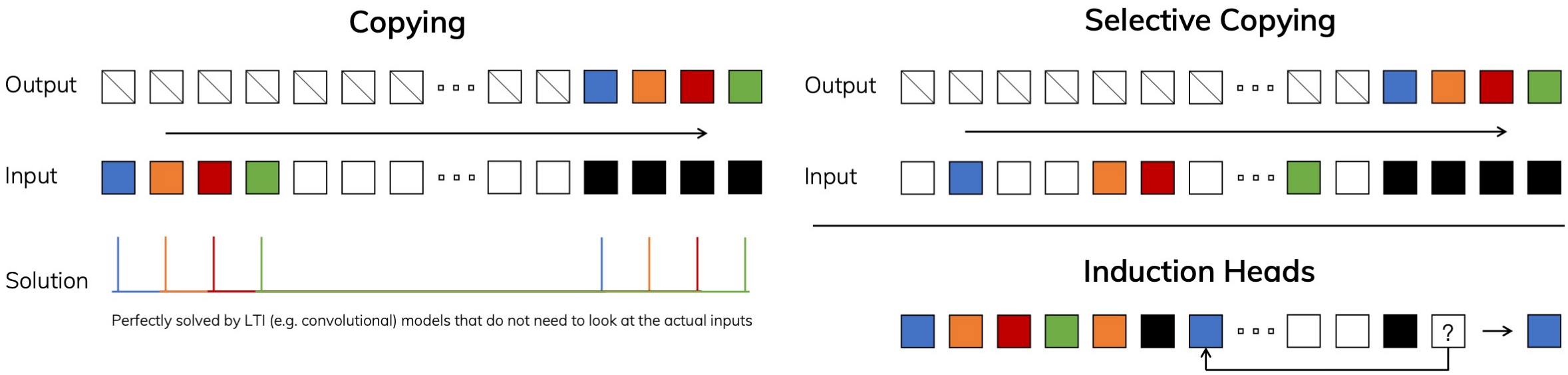
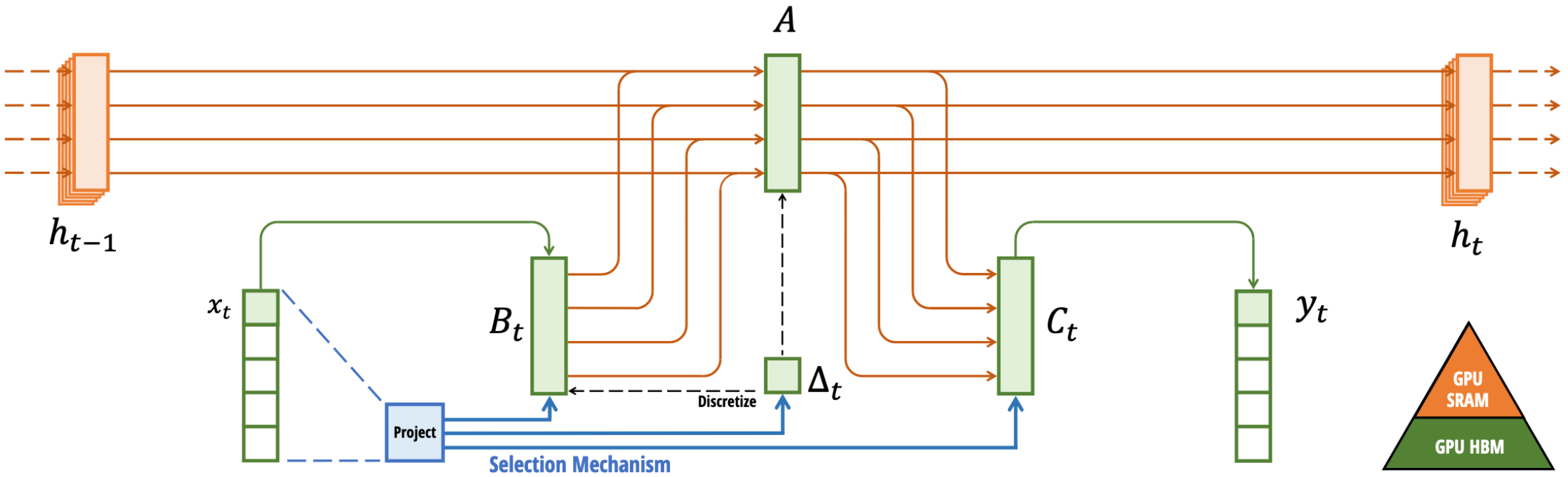


Figure 2: (*Left*) The standard version of the Copying task involves constant spacing between input and output elements and is easily solved by time-invariant models such as linear recurrences and global convolutions. (*Right Top*) The Selective Copying task has random spacing in between inputs and requires time-varying models that can *selectively* remember or ignore inputs depending on their content. (*Right Bottom*) The Induction Heads task is an example of associative recall that requires retrieving an answer based on context, a key ability for LLMs.

A. Gu e T. Dao, «Mamba: Linear-Time Sequence Modeling with Selective State Spaces», 2024, *arXiv*: arXiv:2312.00752.

MAMBA: Selective SSM computed with a scan (S6)

Main idea: make B and C time-varying and depending on the inputs. These matrices are multiplied by the input, so making them time-varying allows to use them as gates filtering out irrelevant information (e.g., white spaces in selective copying).



A. Gu e T. Dao, «Mamba: Linear-Time Sequence Modeling with Selective State Spaces», 2024, *arXiv*: arXiv:2312.00752.

The selection mechanism

Algorithm 1 SSM (S4)

Input: $x : (\mathsf{B}, \mathsf{L}, \mathsf{D})$
Output: $y : (\mathsf{B}, \mathsf{L}, \mathsf{D})$

$\boldsymbol{A} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
$\boldsymbol{B} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
$\boldsymbol{C} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
$\Delta : (\mathsf{D}) \leftarrow \tau_\Delta(\text{Parameter})$
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathsf{D}, \mathsf{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
$y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-invariant: recurrence or convolution
return y

Algorithm 2 SSM + Selection (S6)

Input: $x : (\mathsf{B}, \mathsf{L}, \mathsf{D})$
Output: $y : (\mathsf{B}, \mathsf{L}, \mathsf{D})$

$\boldsymbol{A} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
$\boldsymbol{B} : (\mathsf{B}, \mathsf{L}, \mathsf{N}) \leftarrow s_B(x)$
$\boldsymbol{C} : (\mathsf{B}, \mathsf{L}, \mathsf{N}) \leftarrow s_C(x)$
$\Delta : (\mathsf{B}, \mathsf{L}, \mathsf{D}) \leftarrow \tau_\Delta(\text{Parameter} + s_\Delta(x))$
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathsf{B}, \mathsf{L}, \mathsf{D}, \mathsf{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
$y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-varying: recurrence (*scan*) only
return y

- Differently from S4, here B, C and Δt are time-varying (they gained an L dimension);
- We obtain time varying discretized $\overline{A}_t$ (A is one, but discretized many times), $\overline{B}_t$, $\overline{C}_t$
- In practice, MAMBA uses a selective scan operation to make this efficient

A. Gu e T. Dao, «Mamba: Linear-Time Sequence Modeling with Selective State Spaces», 2024, *arXiv*: arXiv:2312.00752.

No closed-form kernel

- Since matrices are now all time-dependent, we cannot find a nice closed-form solution for the kernel;
- We can still see the operation as a convlution, but now the kernel is not static, but dynamic;
- One way to compute the output is simply through recurrence:

$$x_0 = 0$$
$$x_{t+1} = A_t \cdot x_t + B_t \cdot u_t$$
$$y_t = C_t \cdot x_t$$

- In practice, MAMBA uses a **selective scan operation** to make this parallizable;
- This is not fully parallel, but fast in practice on CUDA GPUS;

A. Gu e T. Dao, «Mamba: Linear-Time Sequence Modeling with Selective State Spaces», 2024, *arXiv*: arXiv:2312.00752.

Slective Scan – Parallelizing Sequential Associative Operations

A selective scan is an algorithm which can parallelize sequential operations provided that the operator is **associative**.

A classic example is the all-prefix-sums operation, which takes a binary associative operator + with identity i and, given an array:

$$[a_0, a_1, \ldots, a_n]$$

Returns:

$$[i, a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + \cdots + a_n]$$

This is sometimes called a cumulative sum operation.

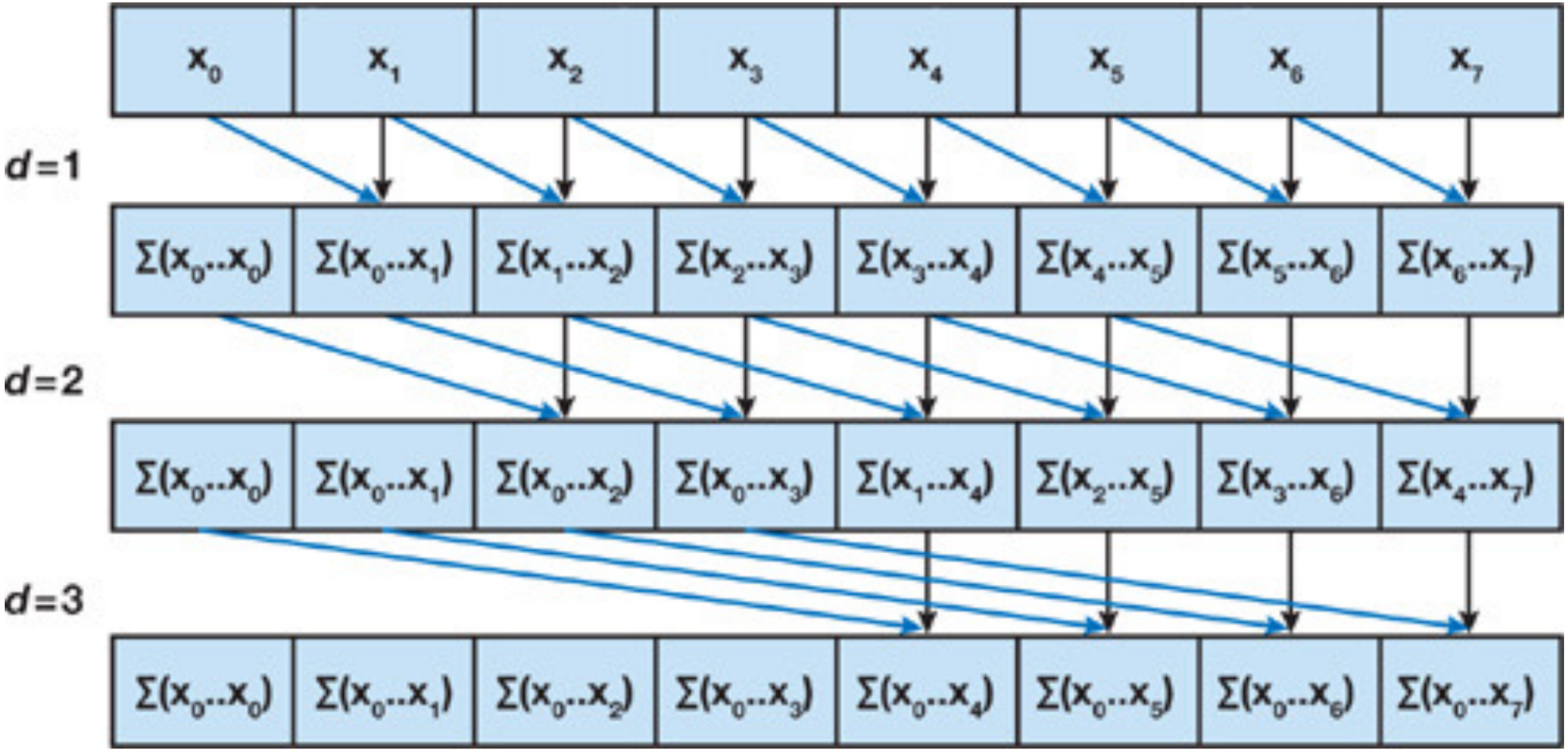
While the operation is sequential, due to the associative nature of the operator, we can pre-compute different sub-sums in parallel and proceed in multiple steps.

This allows to speed up computation is we have parallel hardware (e.g., GPU).

See examples in the following slides.

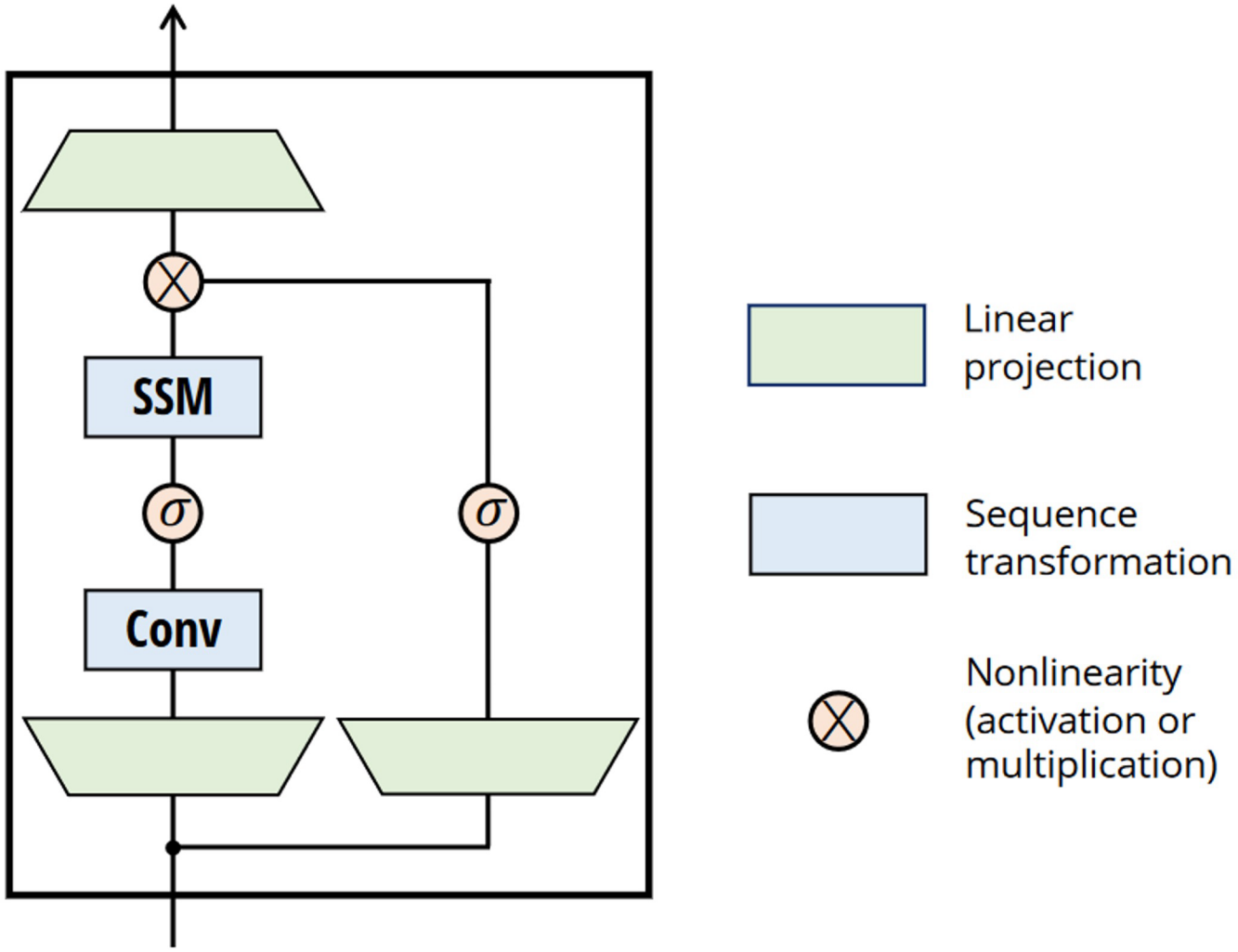
Example of Parallel Scan

A first solution is as follows. At each step, we have a set of parallel workers. However, this is not worl-efficient as it requires $O(n \log n)$ operations, while a sequential sum requires $O(n)$. In practice, work efficient versions requiring $O(n)$ operations (but being much faster) exist.



https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

Mamba Block



- A MAMBA block is hence made as highlighted on the left;
- The input is up-projected into two branches, one for the memory update (left), and one for gating (right);
- A causal depthwise convolution is applied on the left branch to introduce local context (no mixing across channels);
- A non-linearity is added before the SSM (S6) layer;
- The output is multiplied by the output of the right branch for gating;
- The final result is linearly projected

A. Gu e T. Dao, «Mamba: Linear-Time Sequence Modeling with Selective State Spaces», 2024, *arXiv*: arXiv:2312.00752.

MAMBA: interpretation of parameters

- Δ_t controls the balance between how much to focus or ignore the current input x_t. It generalizes RNN gates:
  - A large Δ_t resets the state h_t and focuses on the current input x_t, while a small Δ_t persists the state and ignores the current input;
- A is selective only through the discretization induced by Δ_t (which leads to $\overline{A}_t$). It controls how the hidden state evloves with time;
- B_t and C_t act as gates, filtering out irrelevant information. Modifying B and C allows to gain fine-grained control into whether let an input x_t into the state h_t, or the state into the output y_t

$$\mathbf{h}_{t+1} = \bar{\mathbf{A}}_t \mathbf{h}_t + \bar{\mathbf{B}}_t \mathbf{x}_t$$

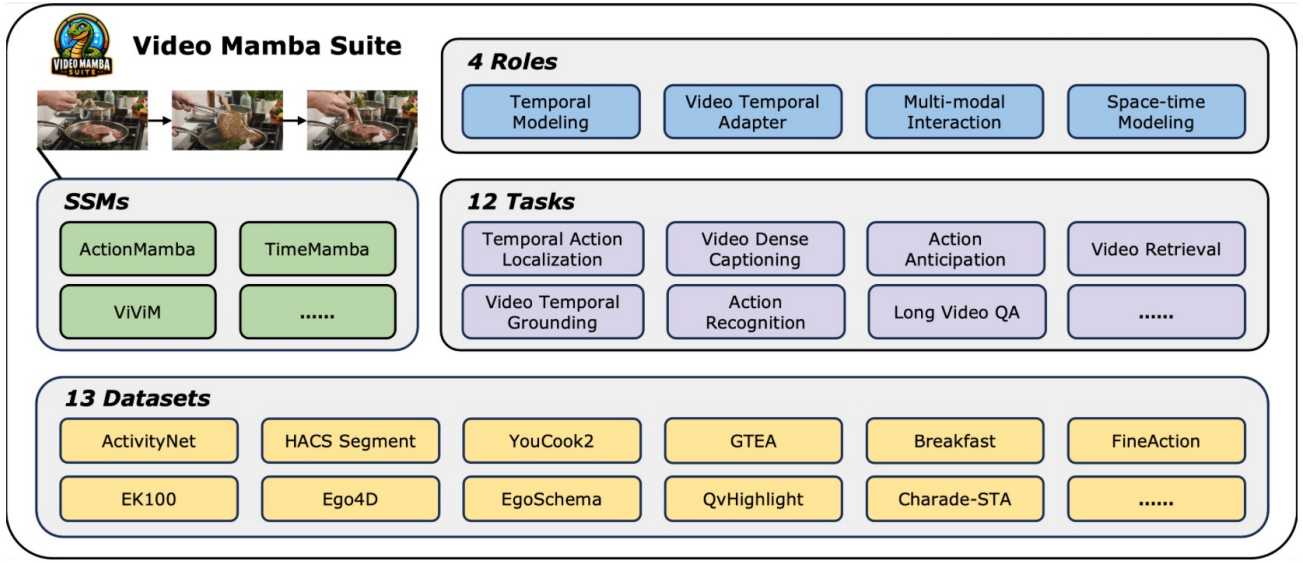
$$\hat{\mathbf{y}}_t = \bar{\mathbf{C}}_t \mathbf{h}_t$$

Video-Mamba Suite

Video Mamba Suite: State Space Model as a Versatile Alternative for Video Understanding

Guo Chen, Yifei Huang, Jilan Xu, Baoqi Pei, Zhe Chen, Zhiqi Li, Jihao Wang, Kunchang Li, Tong Lu and Limin Wang.

Introduction



- Evaluates the effectiveness of Mamba for video understanding tasks
- Proposes different vanilla methods to tackle a variety of tasks on different datasets
- Results are competitive or superior to SOTA
- Mamba proves efficient and versatile, suitable for downstream tasks
- Repository with code available: https://github.com/OpenGVLab/video-mamba-suite

Chen, G., Huang, Y., Xu, J., Pei, B., Chen, Z., Li, Z., ... & Wang, L. (2024). Video mamba suite: State space model as a versatile alternative for video understanding. *arXiv preprint arXiv:2403.09626*.

MAMBA Recap

- Mamba implements selective **state** updates, letting the model focus computation on relevant tokens while keeping a compact latent state instead of a full attention map.
- It **scales to long contexts** with $O(L)$ time and memory, which is attractive for very long sequences where Transformers' $O(L^2)$ attention becomes a bottleneck.
- It offers **improved hardware efficiency** and **throughput** on modern accelerators compared to standard Transformers at long sequence lengths.
- Applied in **language modeling**, **code**, **speech**, **vision backbones** (e.g., Spatial-Mamba) and **multimodal or foundation models**.
- **Compared to LSTMs, handle much longer-range dependencies**, train more stably at scale, and integrate better with modern deep learning stacks.
- **Compared to Transformers, reduce quadratic cost**, can be **easier to deploy in streaming/online scenarios**, and naturally **support recurrent inference over unbounded streams**.

What About Plain old LSTMs?

xLSTM: Extended Long Short-Term Memory

Maximilian Beck[*] [1,2,3] **Korbinian Pöppel**[*] [1,2,3] **Markus Spanring** [1]
Andreas Auer [1,2] **Oleksandra Prudnikova** [1] **Michael Kopp**
Günter Klambauer [1,2,3] **Johannes Brandstetter** [1,2,3] **Sepp Hochreiter** [1,2,3]
[*]Equal contribution
[1]ELLIS Unit, LIT AI Lab, Institute for Machine Learning, JKU Linz, Austria
[2]NXAI Lab, Linz, Austria, [3]NXAI GmbH, Linz, Austria

Abstract

In the 1990s, the constant error carousel and gating were introduced as the central ideas of the Long Short-Term Memory (LSTM). Since then, LSTMs have stood the test of time and contributed to numerous deep learning success stories, in particular they constituted the first Large Language Models (LLMs). However, the advent of the Transformer technology with parallelizable self-attention at its core marked the dawn of a new era, outpacing LSTMs at scale. We now raise a simple question: How far do we get in language modeling when scaling LSTMs to billions of parameters, leveraging the latest techniques from modern LLMs, but mitigating known limitations of LSTMs? Firstly, we introduce exponential gating with appropriate normalization and stabilization techniques. Secondly, we modify the LSTM memory structure, obtaining: (i) sLSTM with a scalar memory, a scalar update, and new memory mixing, (ii) mLSTM that is fully parallelizable with a matrix memory and a covariance update rule. Integrating these LSTM extensions into residual block backbones yields xLSTM blocks that are then residually stacked into xLSTM architectures. Exponential gating and modified memory structures boost xLSTM capabilities to perform favorably when compared to state-of-the-art Transformers and State Space Models, both in performance and scaling.
Code available at: https://github.com/NX-AI/xlstm

Recap of LSTM

$$c_t = \mathrm{f}_t \; c_{t-1} + \mathrm{i}_t \; z_t \qquad \text{cell state} \qquad (2)$$

$$h_t = \mathrm{o}_t \; \tilde{h}_t \,, \qquad \tilde{h}_t = \psi\left(c_t\right) \qquad \text{hidden state} \qquad (3)$$

$$z_t = \varphi\left(\tilde{z}_t\right) \,, \qquad \tilde{z}_t = \boldsymbol{w}_z^\top \boldsymbol{x}_t + r_z h_{t-1} + b_z \qquad \text{cell input} \qquad (4)$$

$$\mathrm{i}_t = \sigma\left(\tilde{\mathrm{i}}_t\right) \,, \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_\mathrm{i}^\top \boldsymbol{x}_t + r_\mathrm{i} \, h_{t-1} + b_\mathrm{i} \qquad \text{input gate} \qquad (5)$$

$$\mathrm{f}_t = \sigma\left(\tilde{\mathrm{f}}_t\right) \,, \qquad \tilde{\mathrm{f}}_t = \boldsymbol{w}_\mathrm{f}^\top \boldsymbol{x}_t + r_\mathrm{f} \, h_{t-1} + b_\mathrm{f} \qquad \text{forget gate} \qquad (6)$$

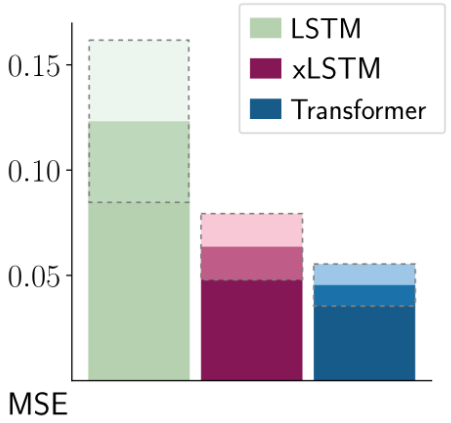
$$\mathrm{o}_t = \sigma\left(\tilde{\mathrm{o}}_t\right) \,, \qquad \tilde{\mathrm{o}}_t = \boldsymbol{w}_\mathrm{o}^\top \boldsymbol{x}_t + r_\mathrm{o} \, h_{t-1} + b_\mathrm{o} \qquad \text{output gate} \qquad (7)$$

- Weights $\boldsymbol{w}$ regulate the contribution of input $\boldsymbol{x_t}$;
- Weights r correspond to the recurrent weights with the hidden state;
- Bias terms are denoted with b;
- i, f, o are input, forget, and output gates;
- z_t is the input to the cell;
- Hidden state is computed from cell;
- Cell update is linear;

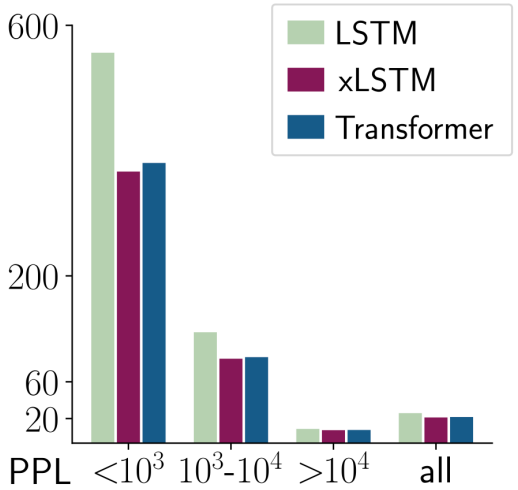
Three Limitations of LSTMs

Authors highlight three main limitations of LSTMs:

- **Inability to revise storage decisions**: once some value is stored in the hidden vector, the LSTM struggles to revise it and replace with something else;
- **Limited storage capabilities**: LSTM has to store information in a very compressed space, leading to inabilities in predicting rare tokens;
- **Lack of parallelization**: this is due to memory mixing (fully connected) hidden-hidden connections between hidden states from one time step to the next, which enforces sequential processing.



Given a reference vector, a sequence is scanned sequentially to identify the most similar vector, with the goal of returning its associated value at the end of the sequence. Standard LSTMs struggle with this task because they cannot reliably update a stored value when a more similar vector appears later in the sequence.



Perplexity (PPL) on Wikitext-103 grouping tokens according to their frequency. Standard LSTMs perform poorly on rare tokens due to their limited capacity to store and retrieve infrequent information.

Revising storage decisions with sLSTM (scalar memory)

To enable revising storage decisions, authors introduce exponantial gates (red) in sLSTMs:

$$c_t = \mathrm{f}_t \; c_{t-1} + \mathrm{i}_t \; z_t \qquad \text{cell state} \qquad (8)$$

$$n_t = \mathrm{f}_t \; n_{t-1} + \mathrm{i}_t \qquad \text{normalizer state} \qquad (9)$$

$$h_t = \mathrm{o}_t \; \tilde{h}_t \,, \qquad \tilde{h}_t = c_t \,/\, n_t \qquad \text{hidden state} \qquad (10)$$

$$z_t = \varphi\left(\tilde{z}_t\right) \,, \qquad \tilde{z}_t = \boldsymbol{w}_z^\top \boldsymbol{x}_t + r_z h_{t-1} + b_z \qquad \text{cell input} \qquad (11)$$

$$\mathrm{i}_t = \exp\left(\tilde{\mathrm{i}}_t\right) \,, \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_\mathrm{i}^\top \boldsymbol{x}_t + r_\mathrm{i} \, h_{t-1} + b_\mathrm{i} \qquad \text{input gate} \qquad (12)$$

$$\mathrm{f}_t = \sigma\left(\tilde{\mathrm{f}}_t\right) \text{ OR } \exp\left(\tilde{\mathrm{f}}_t\right) \,, \qquad \tilde{\mathrm{f}}_t = \boldsymbol{w}_\mathrm{f}^\top \boldsymbol{x}_t + r_\mathrm{f} \, h_{t-1} + b_\mathrm{f} \qquad \text{forget gate} \qquad (13)$$

$$\mathrm{o}_t = \sigma\left(\tilde{\mathrm{o}}_t\right) \,, \qquad \tilde{\mathrm{o}}_t = \boldsymbol{w}_\mathrm{o}^\top \boldsymbol{x}_t + r_\mathrm{o} \, h_{t-1} + b_\mathrm{o} \qquad \text{output gate} \qquad (14)$$

Since gates are now unbounded (not a sigmoid anymore), n_t is used for normalization.

In practice, the authors use stabilized versions for i and f to avoid overflow.

Associative Matrix-Like Memories

To understand how mLSTM extends LSTM from a scalar to a matrix memory, we first need to understand how a matrix-like associative memory works. Let C_t be the memory at time t, the following rule writes vector v_t with key k_t (**covariance update rule**):

$$C_t = C_{t-1} + v_t k_t^T$$

Value v_t can be extracted from the memory with an appropriate query q_t as follows:

$$\hat{v}_t = C_t q_t$$

To understand why this makes sense, let's replace the write rule into the read rule:

$$\hat{v}_t = (C_{t-1} + v_t k_t^T) q_t = C_{t-1} q_t + v_t (k_t^T q_t)$$

Which shows that we retrieve the original value v_t scaled by the similarity between key and query $(k_t^T q_t)$, plus some noise/superposition $C_{t-1} q_t$ coming from the old memory.

In practice, if the key matches the query well and if no old value was stored (imagine the matrix initialized with all zeros), we will retrieve the correct vector.

Enhancing LSTM with matrix memory (mLSTM)

mLSTM uses a matrix memory, where each cell writes and reads information from a matrix.

$$\boldsymbol{C}_t = \mathrm{f}_t \, \boldsymbol{C}_{t-1} + \mathrm{i}_t \, \boldsymbol{v}_t \, \boldsymbol{k}_t^\top \quad \text{cell state} \quad (19)$$

$$\boldsymbol{n}_t = \mathrm{f}_t \, \boldsymbol{n}_{t-1} + \mathrm{i}_t \, \boldsymbol{k}_t \quad \text{normalizer state} \quad (20)$$

$$\boldsymbol{h}_t = \mathrm{o}_t \odot \tilde{\boldsymbol{h}}_t \,, \qquad \tilde{\boldsymbol{h}}_t = \boldsymbol{C}_t \, \boldsymbol{q}_t \; / \; \max\left\{ \left| \boldsymbol{n}_t^\top \, \boldsymbol{q}_t \right|, 1 \right\} \quad \text{hidden state} \quad (21)$$

$$\boldsymbol{q}_t = \boldsymbol{W}_q \, \boldsymbol{x}_t + \boldsymbol{b}_q \quad \text{query input} \quad (22)$$

$$\boldsymbol{k}_t = \frac{1}{\sqrt{d}} \boldsymbol{W}_k \, \boldsymbol{x}_t + \boldsymbol{b}_k \quad \text{key input} \quad (23)$$

$$\boldsymbol{v}_t = \boldsymbol{W}_v \, \boldsymbol{x}_t + \boldsymbol{b}_v \quad \text{value input} \quad (24)$$

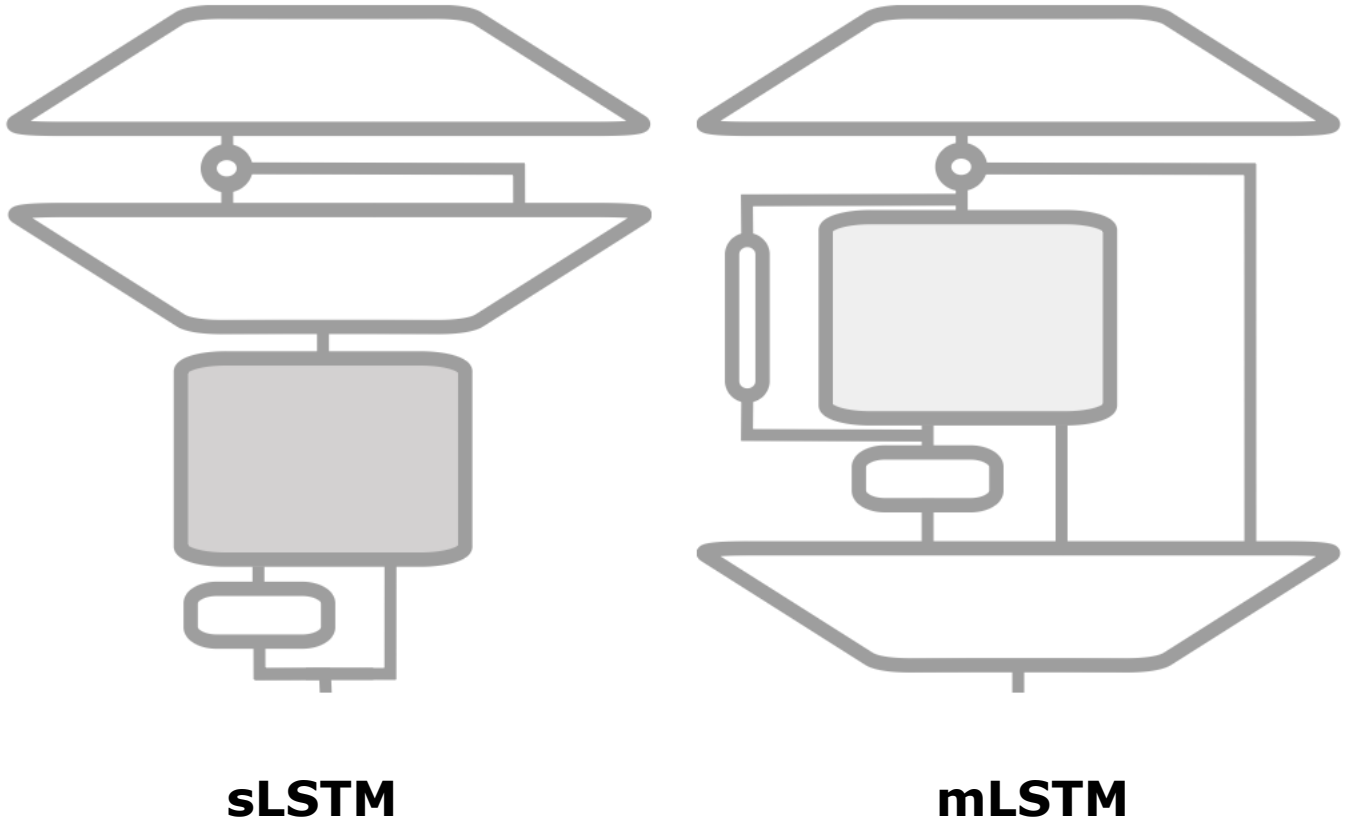
$$\mathrm{i}_t = \exp\left(\tilde{\mathrm{i}}_t\right) \,, \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_\mathrm{i}^\top \, \boldsymbol{x}_t + b_\mathrm{i} \quad \text{input gate} \quad (25)$$

$$\mathrm{f}_t = \sigma\left(\tilde{\mathrm{f}}_t\right) \text{ OR } \exp\left(\tilde{\mathrm{f}}_t\right), \quad \tilde{\mathrm{f}}_t = \boldsymbol{w}_\mathrm{f}^\top \, \boldsymbol{x}_t + b_\mathrm{f} \quad \text{forget gate} \quad (26)$$

$$\mathrm{o}_t = \sigma\left(\tilde{\mathrm{o}}_t\right) \,, \qquad \tilde{\mathrm{o}}_t = \boldsymbol{W}_\mathrm{o} \, \boldsymbol{x}_t + \boldsymbol{b}_\mathrm{o} \quad \text{output gate} \quad (27)$$

No explicit dependence on h_{t-1}. **The only recurrence is a lightweight sum on C_{t-1}.**

sLSTM and mLSTM blocks



sLSTM **mLSTM**

- Dedicated sLSTM and mLSTM blocks are formed using activations, up-projections and down-projections;
- Details are in the paper, but we are not particularly interested in them, as they are architectural details;
- This follows similar concepts as Transformers and State Space models, where the mathematical formulations (e.g., attention) are enclosed into learnable blocks.

The xLSTM Architecture

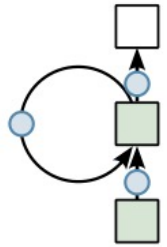
LSTM

Memory Cells

- → Constant Error Carousel
- → Sigmoid Gating
- → Recurrent Inference
- → Recurrent Training

$$c_t = f_t\, c_{t-1} + i_t\, z_t$$

$$h_t = o_t\, \psi(c_t)$$



Memory Cells

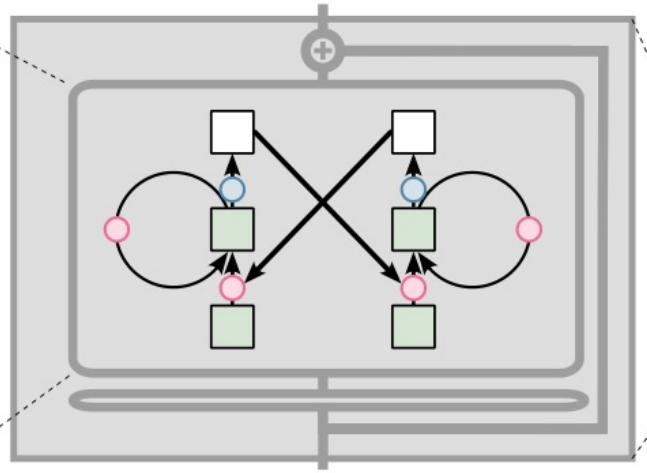
sLSTM

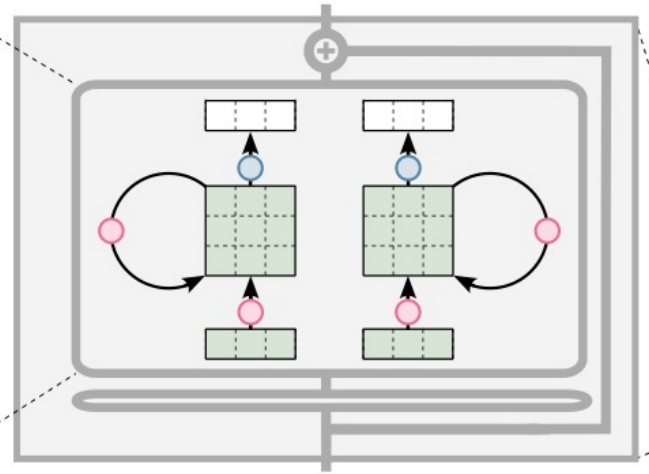
+ Exponential Gating
+ New Memory Mixing

mLSTM

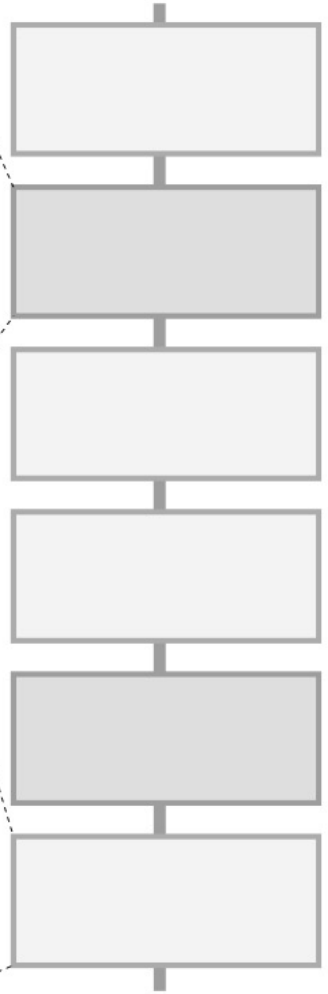
+ Exponential Gating
+ Matrix Memory
+ Parallel Training
+ Covariance Update Rule

xLSTM Blocks





xLSTM



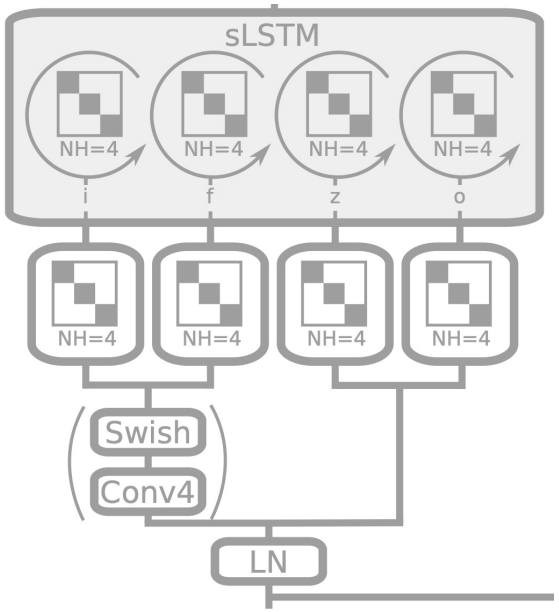
xLSTM heavily skews the architecture using more mLSTMs than sLSTMs (typical ratio 7:1)

What about parallelism?

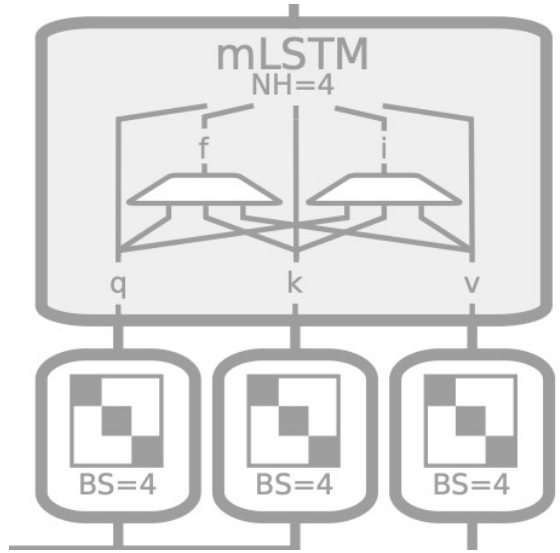
In practice, xLSTM is not strictly parallelizable. Indeed:

- sLSTM still has a recurrence (dependence on h_{t-1});
- mLSTM removes that recurrence but still has a recurrence to update C_t.

However, in practice, the parallel load of xLSTM is reduced as follows:



Since mLSTM is more parallelizable, xLSTM architectures are usually skewed towards mLSTM with typical ratios of 7:1.

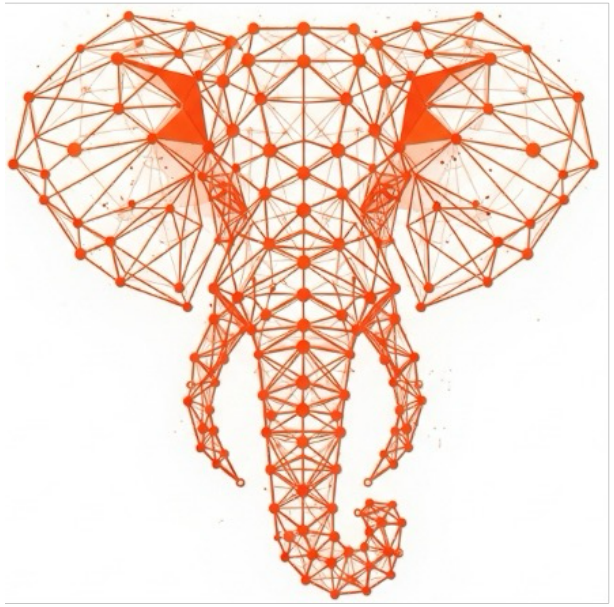


sLSTM uses block-connections to implement heads (as in trasformers) when computing h_t from h_{t-1} (in practice the linear projection is performed head-wise). This does not remove recurrence, but makes it much lighter.

In mLSTM, the computation of keys, keys and values happen in parallel. The only bottleneck is in the update of C_t. However, this is lightweight and implemented with a parallel scan, making it very efficient.

xLSTM summary

- xLSTMs revamp the classic LSTM architecture, drawing from progress in transformers and state space models;
- This is done by:
  - Implementing exponential gates to avoid «saturate» storage, allowing to revising storage decisions;
  - Uses matrix-based memory and key-query-value processing to store values, reducing recurrent dependencies;
  - Engineers training in a way that parallel dependence is minimized and training is efficient and fast.



Deep Learning
Advanced Models & Methods

Prof. Antonino Furnari (antonino.furnari@unict.it)
Corso di Laurea Magistrale in Informatica
Dip. di Matematica e Informatica
Università di Catania

The End