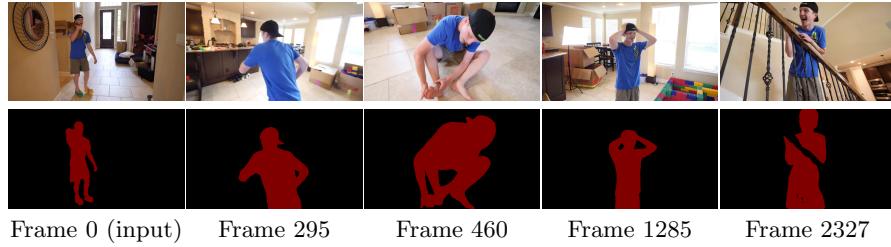


XMem: Long-Term Video Object Segmentation with an Atkinson-Shiffrin Memory Model

Ho Kei Cheng and Alexander G. Schwing

University of Illinois Urbana-Champaign
{hokeikc2, aschwing}@illinois.edu



Abstract. We present XMem, a video object segmentation architecture for long videos with unified feature memory stores inspired by the Atkinson-Shiffrin memory model. Prior work on video object segmentation typically only uses one type of feature memory. For videos longer than a minute, a single feature memory model tightly links memory consumption and accuracy. In contrast, following the Atkinson-Shiffrin model, we develop an architecture that incorporates **multiple independent yet deeply-connected feature memory stores**: a rapidly updated *sensory memory*, a high-resolution *working memory*, and a compact thus sustained *long-term memory*. Crucially, we develop a memory potentiation algorithm that routinely consolidates actively used working memory elements into the long-term memory, which avoids memory explosion and minimizes performance decay for long-term prediction. Combined with a new memory reading mechanism, XMem greatly exceeds state-of-the-art performance on long-video datasets while being on par with state-of-the-art methods (that do not work on long videos) on short-video datasets.¹

1 Introduction

Video object segmentation (VOS) highlights specified target objects in a given video. Here, we focus on the semi-supervised setting where a first-frame annotation is provided by the user, and the method segments objects in all other frames as accurately as possible while preferably running in real-time, online, and while having a small memory footprint even when processing long videos.

As information has to be propagated from the given annotation to other video frames, most VOS methods employ a *feature memory* to store relevant deep-net representations of an object. Online learning methods [3,49,42] use the weights of a network as their feature memory. This requires training at test-time, which slows down prediction. Recurrent methods propagate information often from the most recent frames, either via a mask [39] or via a hidden representation [20,47]. These methods are prone to drifting and struggle with occlusions.

¹ Code is available at hkchengrex.github.io/XMem

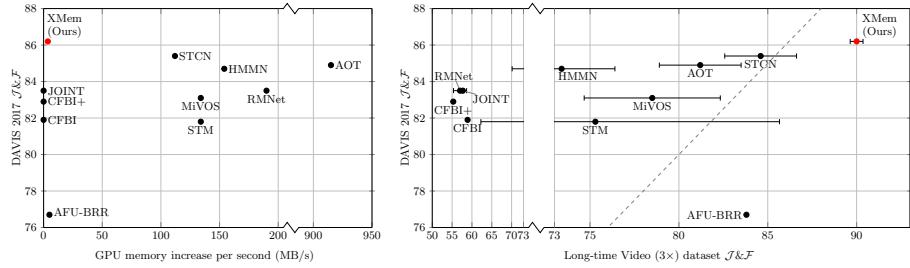


Fig. 1. Do state-of-the-art VOS algorithms scale well? **Left:** Memory scaling with respect to short-term segmentation quality. **Right:** Segmentation quality scaling from standard short videos (y-axis) to long videos (x-axis) – the dashed line indicates a 1:1 performance ratio. Error bars show standard deviations in memory sampling if applicable. See Section 4.1 for details.

Recent state-of-the-art VOS methods use attention [36,18,54,9,60] to link representations of past frames stored in the feature memory with features extracted from the newly observed query frame which needs to be segmented. Despite the high performance of these methods, they require a large amount of GPU memory to store past frame representations. In practice, they usually struggle to handle videos longer than a minute on consumer-grade hardware.

Methods that are specifically designed for VOS in long videos exist [29,27]. However, they often sacrifice segmentation quality. Specifically, these methods reduce the size of the representation during feature memory insertion by merging new features with those already stored in the feature memory. As high-resolution features are compressed right away, they produce less accurate segmentations. Figure 1 shows the relation between GPU memory consumption and segmentation quality in short/long video datasets (details are given in Section 4.1).

We think this undesirable connection of performance and GPU memory consumption is a direct consequence of using a single feature memory type. To address this limitation we propose a unified memory architecture, dubbed XMem. Inspired by the Atkinson–Shiffrin memory model [1] which hypothesizes that the human memory consists of three components, XMem maintains three independent yet deeply-connected feature memory stores: a rapidly updated *sensory memory*, a high-resolution *working memory*, and a compact thus sustained *long-term memory*. In XMem, the sensory memory corresponds to the hidden representation of a GRU [11] which is updated every frame. It provides temporal smoothness but fails for long-term prediction due to representation drift. To complement, the working memory is agglomerated from a subset of historical frames and considers them equally [36,9] without drifting over time. To control the size of the working memory, XMem routinely consolidates its representations into the long-term memory, inspired by the consolidation mechanism in the human memory [46]. XMem stores long-term memory as a set of highly compact prototypes. For this, we develop a memory potentiation algorithm that aggregates richer information into these prototypes to prevent aliasing due to sub-sampling. To read from the working and long-term memory, we devise a space-time memory

motivation

three memory blocks:

- sensory memory
- working memory
- long-term memory

reading operation. The three feature memory stores combined permit handling long videos with high accuracy while keeping GPU memory usage low.

We find XMem to greatly exceed prior state-of-the-art results on the Long-time Video dataset [29]. Importantly, XMem is also on par with current state-of-the-art (that cannot handle long videos) on short-video datasets [41,57]. In summary:

- We devise XMem. Inspired by the Atkinson–Shiffrin memory model [1], we introduce memory stores with different temporal scales and equip them with a memory reading operation for high-quality video object segmentation on both long and short videos.
- We develop a memory consolidation algorithm that selects representative prototypes from the working memory, and a memory potentiation algorithm that enriches these prototypes into a compact yet powerful representation for long-term memory storage.

2 Related Works

General VOS Methods. Most VOS methods employ a *feature memory* to store information given in the first frame and to segment any new frames. Online learning approaches either train or fine-tune their networks at test-time and are therefore typically slow in inference [3,49,32]. Recent improvements are more efficient [34,42,37,2], but they still require online adaptation which is sensitive to the input and has diminishing gains when more training data is available. In contrast, tracking-based approaches [39,52,10,22,5,35,63,19,47,20,56] perform frame-to-frame propagation and are thus efficient at test-time. They however lack long-term context and often lose track after object occlusions. While some methods [48,59,53,23,26,6] also include the first reference frame for global matching, the context is still limited and it becomes harder to match as the video progresses. To address the context limitation, recent state-of-the-art methods use more past frames as feature memory [36,13,64,21,28,58,16]. Particularly, Space-Time Memory (STM) [36] is popular and has been extended by many follow-up works [43,8,18,54,50,31,9,44,33]. Among these extensions, we use STCN [9] as our working memory backbone as it is simple and effective. However, most variants cannot handle long videos due to the ever-expanding feature memory bank of STM. AOT [60] is a recent work that extends the attention mechanism to transformers but does not solve the GPU memory explosion problem. Some methods [33,14] employ a local feature memory window that fails to consider long-term context outside of this window. In contrast, *XMem* uses multiple memory stores to capture different temporal contexts while keeping the GPU memory usage strictly bounded due to our long-term memory and consolidation.

Methods that Specialize in Handling Long Videos. Liang *et al.* [29] propose AFB-URR which selectively uses exponential moving averages to merge a given memory element with existing ones if they are close, or to add it as a new element otherwise. A least-frequently-used-based mechanism is employed to remove unused features when the feature memory reaches a predefined limit. Li *et al.* [27] propose the global context module. It averages all past memory

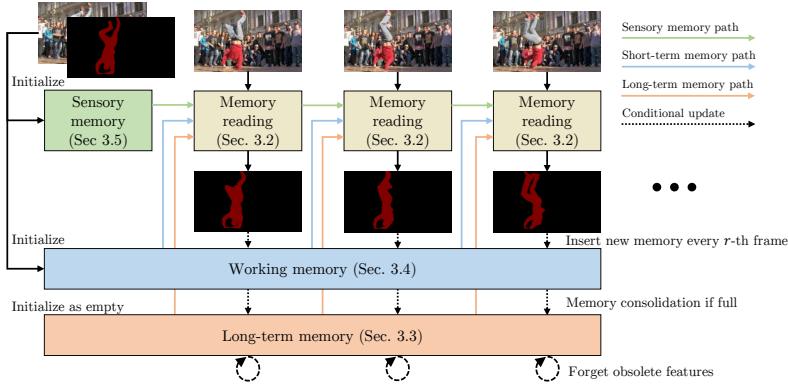


Fig. 2. Overview of XMem. The memory reading operation extracts relevant features from all three memory stores and uses those features to produce a mask. To incorporate new memory, the sensory memory is updated every frame while the working memory is only updated every r -th frame. The working memory is consolidated into the long-term memory in a compact form when it is full, and the long-term memory will forget obsolete features over time.

into a single representation, thus having zero GPU memory increase over time. However, both of these methods *eagerly* compress new high-resolution feature memory into a compact representation, thus sacrificing segmentation accuracy. Our multi-store feature memory avoids eager compression and achieves much higher accuracy in both short-term and long-term predictions.

3 XMem

3.1 Overview

Figure 2 provides an overview of XMem. For readability, we consider a single target object. However, note that XMem is implemented to deal with multiple objects, which is straightforward. Given the image and target object mask at the first frame (top-left of Figure 2), XMem tracks the object and generates corresponding masks for subsequent query frames. For this, we first initialize the different feature memory stores using the inputs. For each subsequent query frame, we perform memory reading (Section 3.2) from long-term memory (Section 3.3), working memory (Section 3.4), and sensory memory (Section 3.5) respectively. The readout features are used to generate a segmentation mask. Then, we update each of the feature memory stores at different frequencies. We update the sensory memory every frame and insert features into the working memory at every r -th frame. When the working memory reaches a pre-defined maximum of T_{\max} frames, we consolidate features from the working memory into the long-term memory in a highly compact form. When the long-term memory is also full (which only happens after processing thousands of frames), we discard obsolete features to bound the maximum GPU memory usage. These feature memory stores work in conjunction to provide high-quality features with low GPU memory usage even for very long videos.

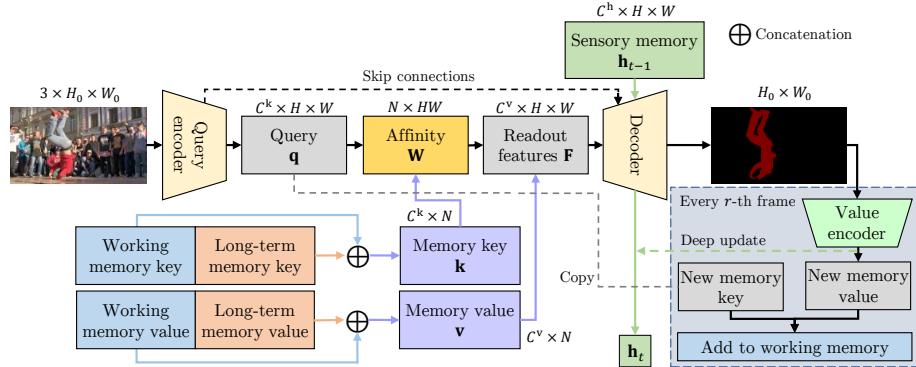


Fig. 3. Process of memory reading and mask decoding of a single query frame. We extract query \mathbf{q} from the image and perform attention-based memory reading from the working/long-term memory to obtain features F . Together with the sensory memory, it is fed into the decoder to generate a mask. For every r -th frame, we store new features into the working memory and perform a deep update to the sensory memory.

XMem consists of three end-to-end trainable convolutional networks as shown in Figure 3: a *query encoder* that extracts query-specific image features, a *decoder* that takes the output of the memory reading step to generate an object mask, and a *value encoder* that combines the image with the object mask to extract new memory features. See Section 3.6 for details of these networks. In the following, we will first describe the memory reading operation before discussing each feature memory store in detail.

3.2 Memory Reading

Figure 3 illustrates the process of memory reading and mask generation for a single frame. The mask is computed via the decoder which uses as input the short-term sensory memory $\mathbf{h}_{t-1} \in \mathbb{R}^{C^h \times H \times W}$ and a feature $\mathbf{F} \in \mathbb{R}^{C^v \times H \times W}$ representing information stored in both the long-term and the working memory.

The feature \mathbf{F} representing information stored in both the long-term and the working memory is computed via the readout operation

$$\mathbf{F} = \mathbf{vW}(\mathbf{k}, \mathbf{q}). \quad (1)$$

Here, $\mathbf{k} \in \mathbb{R}^{C^k \times N}$ and $\mathbf{v} \in \mathbb{R}^{C^v \times N}$ are C^k - and C^v -dimensional keys and values for a total of N memory elements which are stored in both the long-term and working memory. Moreover, $\mathbf{W}(\mathbf{k}, \mathbf{q})$ is an affinity matrix of size $N \times HW$, representing a readout operation that is controlled by the key \mathbf{k} and a query $\mathbf{q} \in \mathbb{R}^{C^k \times HW}$ obtained from the query frame through the query encoder. The readout operation maps every query element to a distribution over all N memory elements and correspondingly aggregates their values \mathbf{v} .

The affinity matrix $\mathbf{W}(\mathbf{k}, \mathbf{q})$ is obtained by applying a softmax on the memory dimension (rows) of a similarity matrix $\mathbf{S}(\mathbf{k}, \mathbf{q})$ which contains the pairwise similarity between every key element and every query element. For computing

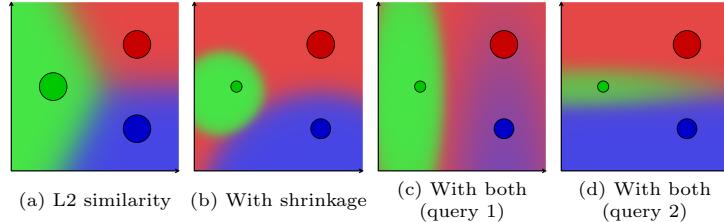


Fig. 4. Visualization of similarity functions in 2D with the background color showing the influence of each memory element (RGB). L2 similarity (a) [9] considers all memory elements uniformly. The shrinkage term (b) allows encoding element-level confidence (visualized by the size of dots) that accounts for the area of influence and sharpness of the mixing weights. The selection term allows query-specific interpretation of the memory – (c) and (d) show its effect with two different queries that focus on the vertical and horizontal dimension respectively. (b) can be seen as a case where the selection term is isotropic. When combined, we can model more complex similarity relations.

the similarity matrix we note that the L2 similarity proposed in STCN [9] is more stable than the dot product [36], but it is less expressive, e.g., it cannot encode the confidence level of a memory element. To overcome this, we propose a new similarity function (*anisotropic L2*) by introducing two new scaling terms that break the symmetry between key and query. Figure 4 visualizes their effects.

Concretely, the key is associated with a shrinkage term $\mathbf{s} \in [1, \infty)^N$ and the query is associated with a selection term $\mathbf{e} \in [0, 1]^{C^k \times HW}$. Then, the similarity between the i -th key element and the j -th query element is computed via

$$\mathbf{S}(\mathbf{k}, \mathbf{q})_{ij} = -\mathbf{s}_i \sum_c^{C^k} \mathbf{e}_{cj} (\mathbf{k}_{ci} - \mathbf{q}_{cj})^2, \quad (2)$$

which equates to the original L2 similarity [9] if $\mathbf{s}_i = \mathbf{e}_{cj} = 1$ for all i , j , and c . The shrinkage term \mathbf{s} directly scales the similarity and explicitly encodes confidence – a high shrinkage represents low confidence and leads to a more local influence. Note that even low-confidence keys can have a high contribution if the query happens to coincide with it – thus avoiding the **memory domination problem** of the dot product, as discussed in [9]. Differently, the selection term \mathbf{e} controls the relative importance of each channel in the key space such that attention is given to the more discriminative channels.

The selection term \mathbf{e} is generated together with the query \mathbf{q} by the query encoder. The shrinkage term \mathbf{s} is collected together with the key \mathbf{k} and the value \mathbf{v} from the working and the long-term memory.² The collection is simply implemented as a concatenation in the last dimension: $\mathbf{k} = \mathbf{k}^w \oplus \mathbf{k}^{lt}$ and $\mathbf{v} = \mathbf{v}^w \oplus \mathbf{v}^{lt}$, where superscripts ‘w’ and ‘lt’ denote working and long-term memory respectively. The working memory consists of key $\mathbf{k}^w \in \mathbb{R}^{C^k \times THW}$ and value $\mathbf{v}^w \in \mathbb{R}^{C^v \times THW}$, where T is the number of working memory frames. The long-term memory similarly consists of keys $\mathbf{k}^{lt} \in \mathbb{R}^{C^k \times L}$ and values $\mathbf{v}^{lt} \in \mathbb{R}^{C^v \times L}$,

² For brevity, we omit the handling of these two scaling terms in memory updates for the rest of the paper. They are updated in the same way as the value.

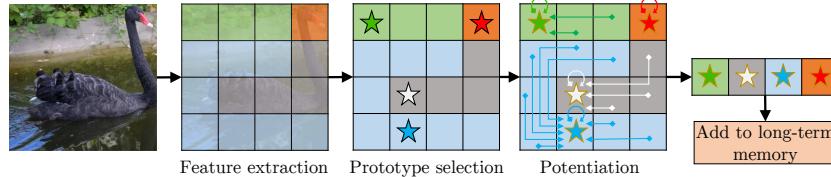


Fig. 5. Memory consolidation procedure. Given an image, we extract features as memory keys (image stride exaggerated). We visualize these features with colors. For memory consolidation, we first select prototype keys (stars) from the candidates (all grids). Then, we invoke potentiation which non-locally aggregates values from all the candidates to generate more representative prototype values (golden outline). The resultant prototype keys and values are added to the long-term memory. Only one frame is shown here – in practice multiple frames are used in a single consolidation.

where L is the number of long-term memory prototypes. Thus, the total number of elements in the working/long-term memory is $N = THW + L$.

Next, we discuss the feature memory stores in detail.

3.3 Long-Term Memory

Motivation. A long-term memory is crucial for handling long videos. With the goal of storing a set of compact (consume little GPU memory) yet representative (lead to high segmentation quality) memory features, we design a *memory consolidation* procedure that selects *prototypes* from the working memory and enriches them with a *memory potentiation* algorithm, as illustrated in Figure 5.

We perform memory consolidation when the working memory reaches a pre-defined size T_{\max} . The first frame (with user-provided ground-truth) and the most recent $T_{\min} - 1$ memory frames will be kept in the working memory as a high-resolution buffer while the remainder ($T_{\max} - T_{\min}$ frames) are *candidates* for being converted into long-term memory representations. We refer to the keys and values of these candidates as $\mathbf{k}^c \subset \mathbf{k}^w$ and $\mathbf{v}^c \subset \mathbf{v}^w$ respectively. In the following, we describe the prototype selection process that picks a compact set of prototype keys $\mathbf{k}^p \subset \mathbf{k}^c$, and the memory potentiation algorithm that generates enriched prototype values \mathbf{v}^p associated with these prototype keys. Finally, these prototype keys and values are appended to the long-term memory \mathbf{k}^{lt} and \mathbf{v}^{lt} .

Prototype Selection. In this step, we sample a small representative subset $\mathbf{k}^p \subset \mathbf{k}^c$ from the candidates as *prototypes*. It is essential to pick only a small number of prototypes, as their amount is directly proportional to the size of the resultant long-term memory. Inspired by human memory which moves frequently accessed or studied patterns to a long-term store, we pick candidates with high *usage*. Concretely, we pick the **top- P** frequently used candidates as prototypes. “Usage” of a memory element is defined by its cumulative total affinity (probability mass) in the affinity matrix \mathbf{W} (Eq. (1)), and **normalized** by the duration that each candidate is in the working memory. Note that the duration for each

candidate is at least $r \cdot (T_{\min} - 1)$, leading to stable usage statistics. We obtain the keys of these prototypes as $\mathbf{k}^P \in \mathbb{R}^{C^k \times P}$.

Memory Potentiation. Note that, so far, our sampling of prototype keys \mathbf{k}^P from the candidate keys \mathbf{k}^c is both *sparse* and *discrete*. If we were to sample the prototypes values \mathbf{v}^P in the same manner, the resultant prototypes would inevitably under-represent other candidates and would be prone to *aliasing*. The common technique to prevent aliasing is to apply an anti-aliasing (e.g., Gaussian) filter [15]. Similarly motivated, we perform filtering and aggregate more information into every sampled prototype. While standard filtering can be easily performed on the image plane (2D) or the spatial-temporal volume (3D), it leads to blurry features – especially near object boundaries. To alleviate, we instead construct the neighbourhood for the filtering in the high dimensional (C^k) key space, such that the highly expressive adjacency information given by the keys \mathbf{k}^P and \mathbf{k}^c is utilized. As these keys have to be computed and stored for memory reading anyway, it is also economical in terms of run-time and memory consumption.

Concretely, for each prototype, we aggregate values from all the value candidates \mathbf{v}^c via a weighted average. The weights are computed using a softmax over the key-similarity. For this, we conveniently re-use Eq. (2). By substituting the memory key \mathbf{k} with the candidate key \mathbf{k}^c , and the query \mathbf{q} with the prototype keys \mathbf{k}^P , we obtain the similarity matrix $\mathbf{S}(\mathbf{k}^c, \mathbf{k}^P)$. As before, we use a softmax to obtain the affinity matrix $\mathbf{W}(\mathbf{k}^c, \mathbf{k}^P)$ (where every prototype corresponds to a distribution over candidates). Then, we compute the prototype values \mathbf{v}^P via

$$\mathbf{v}^P = \mathbf{v}^c \mathbf{W}(\mathbf{k}^c, \mathbf{k}^P). \quad (3)$$

Finally, \mathbf{k}^P and \mathbf{v}^P are appended to the long-term memory \mathbf{k}^{lt} and \mathbf{v}^{lt} respectively – concluding the memory consolidation process. Note, similar prototypical approximations have been used in transformers [55,38]. Differently, our approach uses a novel prototype selection scheme suitable for video object segmentation.

Removing Obsolete Features. Although the long-term memory is extremely compact with a high ($> 6000\%$) compression ratio, memory can still overflow since we are continuously appending new features. Empirically, with a 6GB memory budget (e.g., a consumer-grade mid-end GPU), we can process up to 34,000 frames before running into any memory issues. To handle even longer videos, we introduce a least-frequently-used (LFU) eviction algorithm similar to [29]. Unlike [29], our “usage” (as defined in Section 3.3, Prototype Selection) is defined by the cumulative affinity after top- k filtering [8] which circumvents the introduction of an extra threshold hyperparameter. Long-term memory elements with the least usage will be evicted when a pre-defined memory limit is reached.

The long-term memory is key to enabling efficient and accurate segmentation of long videos. Next, we discuss the working memory, which is crucial for accurate short-term prediction. It acts as the basis for the long-term memory.

3.4 Working Memory

The working memory stores high-resolution features in a temporary buffer. It facilitates accurate matching in the temporal context of a few seconds. It also acts as a gateway into the long-term memory, as the importance of each memory element is estimated by their usage frequency in the working memory.

In our multi-store feature memory design, we find that a classical instantiation of the working memory is sufficient for good results. We largely employ a baseline STCN-style [9] feature memory bank as our working memory, which we will briefly describe for completeness. We refer readers to [9] for details. However, note that our memory reading step (Section 3.2) differs significantly. The working memory consists of keys $\mathbf{k}^w \in \mathbb{R}^{C^k \times THW}$ and values $\mathbf{v}^w \in \mathbb{R}^{C^v \times THW}$, where T is the number of working memory frames. The key is encoded from the image and resides in the same embedding space as the query \mathbf{q} while the value is encoded from both the image and the mask. Bottom-right of Figure 3 illustrates the working memory update process. At every r -th frame, we 1) copy the query as a new key; and 2) generate a new value by feeding the image and the predicted mask into the value encoder. The new key and value are appended to the working memory and are later used in memory reading for subsequent frames. To avoid memory explosion, we limit the number of frames in the working memory T : $T_{\min} \leq T < T_{\max}$ by consolidating extra frames into the long-term memory store as discussed in Section 3.3.

3.5 Sensory Memory

The sensory memory focuses on the short-term and retains low-level information such as object location which nicely complements the lack of temporal locality in the working/long-term memory. Similar to the working memory, we find a classical baseline to work well.

Concretely, the sensory memory stores a hidden representation $\mathbf{h}_t \in \mathbb{R}^{C^h \times H \times W}$, initialized as the zero vector, and propagated by a Gated Recurrent Unit (GRU) [11] as illustrated in Figure 6. This sensory memory is updated every frame using multi-scale features of the decoder. At every r -th frame, whenever a new working memory frame is generated, we perform a *deep update*. Features from the value encoder are used to refresh the sensory memory with another GRU. This allows the sensory memory to 1) discard redundant information that has already been saved to the working memory, and 2) receive updates from a deep network (i.e., the value encoder) with minimal overhead as we are reusing existing features.

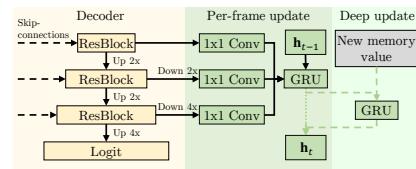


Fig. 6. Sensory memory update overview. Multi-scale features from the decoder are downsampled and concatenated as inputs to a GRU. In the deep update, a separate GRU is additionally used to refresh the sensory memory.

3.6 Implementation Details

Here, we describe some key implementation details. To fully reproduce both training and inference, please see our open-source implementation (footnote 1).

Networks. Following common practice [36,43,29,9], we adopt ResNets [17] as the feature extractor, removing the classification head and the last convolutional stage. This results in features with stride 16. The query encoder is based on a ResNet-50 and the value encoder is based on a ResNet-18, following [9]. To generate the query \mathbf{q} , the shrinkage term \mathbf{s} , and the selection term \mathbf{e} , we apply separate 3×3 convolutional projections to the query encoder feature output. Note that both the query and the shrinkage term are used for the current query frame, while the selection term is copied to the memory (along the copy path in Figure 3) for later use if and only if we are inserting new working memory. We set $C^k = 64$, $C^v = 512$ following [9], and $C^h = 64$. To control the range of the shrinkage factor to be in $[1, \infty)$, we apply $(\cdot)^2 + 1$, and to control the range of the selection factor to be in $[0, 1]$, we apply a sigmoid.

The decoder concatenates hidden representation \mathbf{h}_{t-1} and readout feature \mathbf{F} . It then iteratively upsamples by $2 \times$ at a time until stride 4 while fusing skip-connections from the query encoder at every level, following STM [36]. The stride 4 feature map is projected to a single channel logit via a 3×3 convolution, and is bilinearly upsampled to the input resolution. In the multi-object scenario, we use soft-aggregation [36] to fuse the final logits from different objects. Note that the bulk of the computation (i.e., query encoder, affinity \mathbf{W}) can be shared between different objects as they are only conditioned on the image [9].

Training. Following [36,43,29,9], we first pretrain our network on synthetic sequences of length three generated by deforming static images. We adopt the open-source implementation of STCN [9] without modification, which trains on [45,51,25,62,7]. Next, we perform the main training on YouTubeVOS [57] and DAVIS [41] with curriculum sampling [36]. We note that the default sequence length of three is insufficient to train the sensory memory as it would be heavily dependent on the initial state. Thus, we instead sample sequences of length eight. To reduce training time and for regularization, a maximum of three (instead of all) past frames are randomly selected to be the working memory for any query in training time. The entire training process takes around 35 hours on two RTX A6000 GPUs. Deep updates are performed with a probability of 0.2, which is $1/r$ as we use $r = 5$ by default following [9]. Optionally, we also pretrain on BL30K [8,12,4] which gives a further boost in accuracy. We label any method that uses BL30K with an asterisk (*).

We use bootstrapped cross entropy loss and dice loss with equal weighting following [60]. For optimization, we use AdamW [24,30] with a learning rate of 1e-5 and a weight decay of 0.05, for 150K iterations with batch size 16 in static image pretraining, and for 110K iterations with batch size 8 in main training. We drop the learning rate by a factor of 10 after the first 80K iterations. For a fair comparison, we also retrain the STCN [9] baseline with the above setting. There is no significant difference in performance for STCN (see appendix).

4 Experiments

Unless otherwise specified, we use $T_{\min} = 5$, $T_{\max} = 10$, and $P = 128$, resulting in a compression ratio of 6328% from working memory to long-term memory. We set the maximum number of long-term memory elements to be 10,000 which means XMem never consumes more than 1.4GB of GPU memory, possibly enabling applications even on mobile devices. We use top- k filtering [8] with $k = 30$. 480p videos are used by default. To evaluate we use standard metrics (higher is better) [40]: Jaccard index \mathcal{J} , contour accuracy \mathcal{F} , and their average $\mathcal{J}\&\mathcal{F}$. For YouTubeVOS [57], \mathcal{J} and \mathcal{F} are computed for “seen” and “unseen” classes separately, denoted by subscripts S and U respectively. \mathcal{G} is averaged $\mathcal{J}\&\mathcal{F}$ for both seen and unseen classes. For AOT [60], we compare with their R50 variant which has the same ResNet backbone as ours.

4.1 Long-Time Video Dataset

To evaluate long-term performance, we test models on the Long-time Video dataset [29] which contains three videos with more than 7,000 frames in total. We also synthetically extend it to even longer variants by playing the video back and forth. $n\times$ denotes a variant that has n times the number of frames. For comparison, we select state-of-the-art methods with available implementation as we need to re-run their models. Most SOTA methods cannot handle long videos natively. We first measure their GPU memory increase per frame by averaging the memory consumption difference between the 100-th and 200-th frame in 480p.³ Figure 1 (left) shows our findings, assuming 24FPS. For methods with prohibitive memory usage on long videos, we limit their feature memory insertion frequency accordingly, using 50 memory frames in STM as a baseline following [29]. Our method uses less memory than this baseline. We note that a low memory insertion frequency leads to high variances in performance, thus we run these experiments with 5 evenly-spaced offsets to the memory insertion routine and show “mean \pm standard deviation” if applicable. In this dataset, we use $r = 10$. We do not find BL30K [8] pretraining to help here.

Table 1 tabulates the quantitative results, and Figure 1 (right) plots the short-term performance against the long-term performance. Methods that use a temporally local feature window (CFBI(+) [59,61], JOINT [33]) have a constant memory cost but fail when they lose track of the context. Methods with a fast-growing memory bank (e.g., STM [36], AOT [60], STCN [9]) are forced to use a low feature memory insertion frequency and do not scale well to long videos. Figure 7 shows the scaling behavior of STCN vs. XMem in more detail.

AFB-UVR [29] is designed to handle long videos and scales well with no degradation – but due to eager feature compression it has relatively low performance in the short term compared to other methods. In contrast, XMem not only holds up well in scaling to longer videos but also performs well in the short-term as shown in the next section. We provide qualitative comparisons in the appendix.

³ We make sure to exclude any caching or input buffering overhead.

Table 1. Quantitative comparisons on the Long-time Video dataset [29].

Method	Long-time Video (1×)			Long-time Video (3×)			$\Delta_{1\times\rightarrow 3\times}$
	$\mathcal{J}\&\mathcal{F}$	\mathcal{J}	\mathcal{F}	$\mathcal{J}\&\mathcal{F}$	\mathcal{J}	\mathcal{F}	$\mathcal{J}\&\mathcal{F}$
CFBI+ [61]	50.9	47.9	53.8	55.3	54.0	56.5	4.4
RMNet [54]	59.8±3.9	59.7±8.3	60.0±7.5	57.0±1.6	56.6±1.5	57.3±1.8	-2.8
JOINT [33]	67.1±3.5	64.5±4.2	69.6±3.9	57.7±0.2	55.7±0.3	59.7±0.2	-9.4
CFBI [59]	53.5	50.9	56.1	58.9	57.7	60.1	5.4
HMMN [44]	81.5±1.8	79.9±1.2	83.0±1.5	73.4±3.3	72.6±3.1	74.3±3.5	-8.1
STM [36]	80.6±1.3	79.9±0.9	81.3±1.0	75.3±13.0	74.3±13.0	76.3±13.1	-5.3
MiVOS* [8]	81.1±3.2	80.2±2.0	82.0±3.1	78.5±4.5	78.0±3.7	79.0±5.4	-2.6
AOT [60]	84.3±0.7	83.2±3.2	85.4±3.3	81.2±2.5	79.6±3.0	82.8±2.1	-3.1
AFB-URR [29]	83.7	82.9	84.5	83.8	82.9	84.6	0.1
STCN [9]	87.3±0.7	85.4±1.1	89.2±1.1	84.6±1.9	83.3±1.7	85.9±2.2	-2.7
XMem (Ours)	89.8±0.2	88.0±0.2	91.6±0.2	90.0±0.4	88.2±0.3	91.8±0.4	0.2

4.2 Short Video Datasets

Table 2 and Table 3 tabulate our result on YouTubeVOS [57] 2018 validation, DAVIS [40] 2016/2017 validation, and DAVIS 2017 [41] test-dev. Results on YouTubeVOS [57] 2019 validation can be found in the appendix. The test set for YouTubeVOS is closed at the time of writing. We use $r = 5$ for these datasets. Following standard practice [36, 59, 9], we report single/multi-object FPS on DAVIS 2016/2017 validation. We additionally report FPS on YouTubeVOS 2018 validation which has longer videos on average. We measure FPS on a V100 GPU. For a fair comparison, we re-time prior works that report FPS on a slower GPU if possible and label this with a †. We note that some methods (not ours) are faster on a 2080Ti than on a V100. In these cases, we always give competing methods the benefit. Our speed-up solely comes from the use of long-term memory – a compact feature memory representation is faster to read from.

Table 2. Quantitative comparisons on three commonly used short-term datasets. * denotes BL30K [8] pretraining. Bold and underline denote the best and the second-best respectively in each column. † denotes FPS re-timed on our hardware. On YouTubeVOS, we re-run AOT with all input frames (improving its performance) for a fair comparison.

Method	YT-VOS 2018 val [57]						DAVIS 2017 val [41]				DAVIS 2016 val [40]			
	\mathcal{G}	\mathcal{J}_s	\mathcal{F}_s	\mathcal{J}_u	\mathcal{F}_u	FPS	$\mathcal{J}\&\mathcal{F}$	\mathcal{J}	\mathcal{F}	FPS	$\mathcal{J}\&\mathcal{F}$	\mathcal{J}	\mathcal{F}	FPS
STM [36]	79.4	79.7	84.2	72.8	80.9	-	81.8	79.2	84.3	11.1†	89.3	88.7	89.9	14.0†
AFB-URR [29]	79.6	78.8	83.1	74.1	82.6	-	76.9	74.4	79.3	6.8†	-	-	-	-
CFBI [59]	81.4	81.1	85.8	75.3	83.4	3.4	81.9	79.1	84.6	5.9	89.4	88.3	90.5	6.2
RMNet [54]	81.5	82.1	85.7	75.7	82.4	-	83.5	81.0	86.0	4.4†	88.8	88.9	88.7	11.9
HMMN [44]	82.6	82.1	87.0	76.8	84.6	-	84.7	81.9	87.5	9.3†	90.8	89.6	92.0	13.0†
MiVOS* [8]	82.6	81.1	85.6	77.7	86.2	-	84.5	81.7	87.4	11.2	91.0	89.6	92.4	16.9
STCN [9]	83.0	81.9	86.5	77.9	85.7	<u>13.2</u> †	85.4	82.2	88.6	<u>20.2</u> †	91.6	90.8	92.5	<u>26.9</u> †
JOINT [33]	83.1	81.5	85.9	78.7	86.5	-	83.5	80.8	86.2	6.8†	-	-	-	-
STCN* [9]	84.3	83.2	87.9	79.0	87.3	<u>13.2</u> †	85.3	82.0	88.6	<u>20.2</u> †	91.7	90.4	<u>93.0</u>	<u>26.9</u> †
AOT [60]	85.5	84.5	<u>89.5</u>	79.6	88.2	6.4	84.9	82.3	87.5	18.0	91.1	90.1	92.1	18.0
XMem (Ours)	<u>85.7</u>	<u>84.6</u>	89.3	<u>80.2</u>	<u>88.7</u>	22.6	<u>86.2</u>	<u>82.9</u>	<u>89.5</u>	22.6	91.5	90.4	92.7	29.6
XMem* (Ours)	86.1	85.1	89.8	80.3	89.2	22.6	87.7	84.0	91.4	22.6	92.0	<u>90.7</u>	93.2	29.6

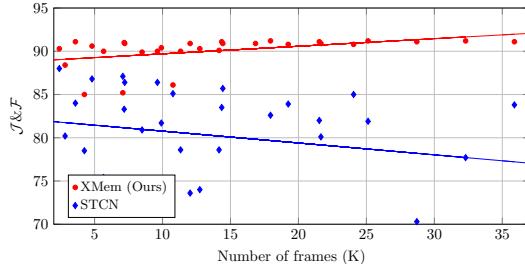


Fig. 7. Least-square fits of performance over video length for XMem and STCN [9] on variants of the Long-time Video dataset [29] from $1\times$ to $10\times$. In longer videos, STCN decays due to missing context while ours stabilizes as we gain sufficient context.

Table 4. Ablation on our memory stores. Standard deviations for $L_{1\times}$ are omitted.

Setting	Y_{18}	D_{17}	$L_{1\times}$	FPS	D_{17}	FPS	Y_{18}
All memory stores	85.7	86.2	89.8	22.6	22.6		
No sensory memory	84.4	85.1	87.9	23.1	23.1		
No working memory	72.7	77.6	38.7	31.8	28.1		
No long-term memory	85.9	86.3	n/a	17.6	10.0		

4.3 Ablations

We perform ablation studies on validation sets of YouTubeVOS 2018 [57] (Y_{18}), DAVIS 2017 [41] (D_{17}), and Long-time Video ($n\times$) [29] ($L_{n\times}$). We report the most representative metric (\mathcal{G} for YouTubeVOS, $\mathcal{J}\&\mathcal{F}$ for DAVIS/Long-time Video). FPS is measured on DAVIS 2017 validation unless otherwise specified. We highlight our final configuration with cyan.

Memory Stores. Table 4 tabulates the performance of XMem without any one of the memory stores. If the working memory is removed, long-term memory cannot function and it becomes “sensory memory only” with a constant memory cost. If the long-term memory is removed, all the memory frames are stored in the working memory. Although it has a slightly better performance due to its higher resolution feature, it cannot handle long videos and is slower.

Memory Reading. Table 5 shows the importance of the two scaling terms in the anisotropic L2 similarity. Interestingly, the selection term e alone does not help. We hypothesize that the selection term allows attention on a different subset of memory elements for every query, thus increasing the relative importance of each memory element. The shrinkage term s allows element-level modulation of confidence, thus avoiding too much emphasis on less confident elements. There is a synergy between the two terms, and our final model benefits from both.

Long-term Memory Strategies. Table 6 compares different prototype selection strategies and shows the importance of potentiation. We run all algorithms 5 times with evenly-spaced memory insertion offsets and show standard deviations. We choose the usage-based selection scheme with $P = 128$ for a balance between performance and memory compression. Table 7 compares additional strategies used by prior works, employed on our model. Eager compression is inspired by AFB-URR [29]. We set $T_{\min} = 1$ and $T_{\max} = 2$. Note, since we cannot

Table 3. Results on DAVIS 2017 test-dev.
‡: uses 600p videos.

Method	DAVIS 2017 td		
	$\mathcal{J}\&\mathcal{F}$	\mathcal{J}	\mathcal{F}
STM‡ [36]	72.2	69.3	75.2
RMNet [54]	75.0	71.9	78.1
STCN [9]	76.1	73.1	80.0
CFBI+‡ [61]	78.0	74.4	81.6
HMMN [44]	78.6	74.7	82.5
MiVOS* [8]	78.6	74.9	82.2
AOT [60]	79.6	75.9	83.3
STCN* [9]	79.9	76.3	83.5
XMem (Ours)	81.0	77.4	84.5
XMem* (Ours)	81.2	77.6	84.7
XMem*‡ (Ours)	82.5	79.1	85.8

Table 5. Ablation on the two scaling terms in memory reading.

Setting	Y_{18}	D_{17}
With both terms	85.7	86.2
With shrinkage s only	85.1	85.6
With selection e only	84.8	84.8
With neither	85.0	85.1

Table 6. Comparisons between different memory consolidation methods.

Setting		$L_{3\times}$	Compress ratio
Random	$P = 64$	89.5 ± 0.8	12625%
K-means centroid	$P = 64$	89.5 ± 0.5	12625%
Usage-based	$P = 64$	89.6 ± 0.4	12625%
Random	$P = 128$	89.7 ± 0.7	6328%
K-means centroid	$P = 128$	82.4 ± 10.3	6328%
Usage-based	$P = 128$	90.0 ± 0.4	6328%
Random	$P = 256$	89.8 ± 0.7	3164%
K-means centroid	$P = 256$	74.5 ± 17.0	3164%
Usage-based	$P = 256$	90.1 ± 0.4	3164%
No potentiation		87.9 ± 0.2	
With potentiation		90.0 ± 0.4	

Table 7. Comparisons between different strategies for handling long videos.

Setting	$L_{1\times}$	$L_{3\times}$	$\Delta_{1\times \rightarrow 3\times}$
Consolidation	89.8 ± 0.2	90.0 ± 0.4	0.2
Eager compression	87.8 ± 0.3	87.3 ± 1.3	-0.5
Sparse insertion	89.8 ± 0.4	87.3 ± 1.0	-2.5
Local window	86.2 ± 1.5	85.5 ± 0.9	-0.7

Table 8. Ablation on the deep update frequency of sensory memory.

Setting	Y_{18}	D_{17}	FPS
Every r -th frame	85.7	86.2	22.6
Every single frame	85.5	86.1	18.5
No deep update	85.3	85.4	22.6

compute usage statistics in this setting, we use random prototype selection with the same compression ratio. Sparse insertion follows our treatment to methods with a growing memory bank [36,9]. We set the maximum number of memory frames to be 50 following [29]. Local window follows [59,14,33], where we simply discard the oldest memory frame when the memory bank reaches its capacity. We always keep the first reference frame and set the memory bank capacity to be 50. Our memory consolidation algorithm is the most effective among these.

Deep Update. Table 8 shows different configurations of the deep update. Employing deep update every r -th frame results in a performance boost, with no noticeable speed drop (recall that we have to use the value encoder every r -th frame for our working memory anyway). However, using deep updates more often requires extra invocations of the value encoder and leads to a slowdown.

Pretraining. There are prior works that do not use static image pretraining [61,2,14,33]. We provide our results without pretraining in the appendix.

4.4 Limitations

Our method sometimes fails when the target object moves too quickly or has severe motion blur as even the fastest updating sensory memory cannot catch up. See the appendix for examples. We think a sensory memory with a large receptive field that is more powerful than our baseline instantiation could help.

5 Conclusion

We present XMem – to our best knowledge the first multi-store feature memory model used for video object segmentation. XMem achieves excellent performance with minimal GPU memory usage for both long and short videos. We believe XMem is a good step toward accessible VOS on mobile devices, and we hope to draw attention to the more widely-applicable long-term VOS task.

Acknowledgment. Work supported in part by NSF under Grants 1718221, 2008387, 2045586, 2106825, MRI 1725729, and NIFA award 2020-67021-32799.

References

1. Atkinson, R.C., Shiffrin, R.M.: Human memory: A proposed system and its control processes. In: *Psychology of learning and motivation*, vol. 2, pp. 89–195. Elsevier (1968)
2. Bhat, G., Lawin, F.J., Danelljan, M., Robinson, A., Felsberg, M., Van Gool, L., Timofte, R.: Learning what to learn for video object segmentation. In: *ECCV* (2020)
3. Caelles, S., Maninis, K.K., Pont-Tuset, J., Leal-Taixé, L., Cremers, D., Van Gool, L.: One-shot video object segmentation. In: *CVPR* (2017)
4. Chang, A.X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L., Yu, F.: ShapeNet: An Information-Rich 3D Model Repository. In: *arXiv:1512.03012* (2015)
5. Chen, X., Li, Z., Yuan, Y., Yu, G., Shen, J., Qi, D.: State-aware tracker for real-time video object segmentation. In: *CVPR* (2020)
6. Chen, Y., Pont-Tuset, J., Montes, A., Van Gool, L.: Blazingly fast video object segmentation with pixel-wise metric learning. In: *CVPR* (2018)
7. Cheng, H.K., Chung, J., Tai, Y.W., Tang, C.K.: Cascadepsp: Toward class-agnostic and very high-resolution segmentation via global and local refinement. In: *CVPR* (2020)
8. Cheng, H.K., Tai, Y.W., Tang, C.K.: Modular interactive video object segmentation: Interaction-to-mask, propagation and difference-aware fusion. In: *CVPR* (2021)
9. Cheng, H.K., Tai, Y.W., Tang, C.K.: Rethinking space-time networks with improved memory coverage for efficient video object segmentation. In: *NeurIPS* (2021)
10. Cheng, J., Tsai, Y.H., Hung, W.C., Wang, S., Yang, M.H.: Fast and accurate online video object segmentation via tracking parts. In: *CVPR* (2018)
11. Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: Encoder-decoder approaches. In: *arXiv* (2014)
12. Denninger, M., Sundermeyer, M., Winkelbauer, D., Zidan, Y., Olefir, D., Elbadrawy, M., Lodhi, A., Katam, H.: Blenderproc. In: *arXiv:1911.01911* (2019)
13. Duarte, K., Rawat, Y.S., Shah, M.: Capsulevos: Semi-supervised video object segmentation using capsule routing. In: *ICCV* (2019)
14. Duke, B., Ahmed, A., Wolf, C., Aarabi, P., Taylor, G.W.: Sstvos: Sparse spatiotemporal transformers for video object segmentation. In: *CVPR* (2021)
15. Forsyth, D., Ponce, J.: Computer vision: A modern approach. Prentice hall (2011)
16. Ge, W., Lu, X., Shen, J.: Video object segmentation using global and instance embedding learning. In: *CVPR* (2021)
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: *CVPR* (2016)
18. Hu, L., Zhang, P., Zhang, B., Pan, P., Xu, Y., Jin, R.: Learning position and target consistency for memory-based video object segmentation. In: *CVPR* (2021)
19. Hu, P., Wang, G., Kong, X., Kuen, J., Tan, Y.P.: Motion-guided cascaded refinement network for video object segmentation. In: *CVPR* (2018)
20. Hu, Y.T., Huang, J.B., Schwing, A.: Maskrnn: Instance level video object segmentation. In: *NIPS* (2017)
21. Huang, X., Xu, J., Tai, Y.W., Tang, C.K.: Fast video object segmentation with temporal aggregation network and dynamic template matching. In: *CVPR* (2020)
22. Jang, W.D., Kim, C.S.: Online video object segmentation via convolutional trident network. In: *CVPR* (2017)

23. Johnander, J., Danelljan, M., Brissman, E., Khan, F.S., Felsberg, M.: A generative appearance model for end-to-end video object segmentation. In: CVPR (2019)
24. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: ICLR (2015)
25. Li, X., Wei, T., Chen, Y.P., Tai, Y.W., Tang, C.K.: Fss-1000: A 1000-class dataset for few-shot segmentation. In: CVPR (2020)
26. Li, X., Loy, C.C.: Video object segmentation with joint re-identification and attention-aware mask propagation. In: ECCV (2018)
27. Li, Y., Shen, Z., Shan, Y.: Fast video object segmentation using the global context module. In: ECCV (2020)
28. Liang, S., Shen, X., Huang, J., Hua, X.S.: Video object segmentation with dynamic memory networks and adaptive object alignment. In: ICCV (2021)
29. Liang, Y., Li, X., Jafari, N., Chen, J.: Video object segmentation with adaptive feature bank and uncertain-region refinement. In: NeurIPS (2020)
30. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. ICLR (2019)
31. Lu, X., Wang, W., Martin, D., Zhou, T., Shen, J., Luc, V.G.: Video object segmentation with episodic graph memory networks. In: ECCV (2020)
32. Maninis, K.K., Caelles, S., Chen, Y., Pont-Tuset, J., Leal-Taixé, L., Cremers, D., Van Gool, L.: Video object segmentation without temporal information. In: PAMI (2018)
33. Mao, Y., Wang, N., Zhou, W., Li, H.: Joint inductive and transductive learning for video object segmentation. In: ICCV (2021)
34. Meinhardt, T., Leal-Taixé, L.: Make one-shot video object segmentation efficient again. In: NeurIPS (2020)
35. Oh, S.W., Lee, J.Y., Sunkavalli, K., Joo Kim, S.: Fast video object segmentation by reference-guided mask propagation. In: CVPR (2018)
36. Oh, S.W., Lee, J.Y., Xu, N., Kim, S.J.: Video object segmentation using space-time memory networks. In: ICCV (2019)
37. Park, H., Yoo, J., Jeong, S., Venkatesh, G., Kwak, N.: Learning dynamic network using a reuse gate function in semi-supervised video object segmentation. In: CVPR (2021)
38. Patrick, M., Campbell, D., Asano, Y.M., Metze, I.M.F., Feichtenhofer, C., Vedaldi, A., Henriques, J., et al.: Keeping your eye on the ball: Trajectory attention in video transformers. In: NeurIPS (2021)
39. Perazzi, F., Khoreva, A., Benenson, R., Schiele, B., Sorkine-Hornung, A.: Learning video object segmentation from static images. In: CVPR (2017)
40. Perazzi, F., Pont-Tuset, J., McWilliams, B., Van Gool, L., Gross, M., Sorkine-Hornung, A.: A benchmark dataset and evaluation methodology for video object segmentation. In: CVPR (2016)
41. Pont-Tuset, J., Perazzi, F., Caelles, S., Arbeláez, P., Sorkine-Hornung, A., Van Gool, L.: The 2017 davis challenge on video object segmentation. In: arXiv:1704.00675 (2017)
42. Robinson, A., Lawin, F.J., Danelljan, M., Khan, F.S., Felsberg, M.: Learning fast and robust target models for video object segmentation. In: CVPR (2020)
43. Seong, H., Hyun, J., Kim, E.: Kernelized memory network for video object segmentation. In: ECCV (2020)
44. Seong, H., Oh, S.W., Lee, J.Y., Lee, S., Lee, S., Kim, E.: Hierarchical memory matching network for video object segmentation. In: ICCV (2021)
45. Shi, J., Yan, Q., Xu, L., Jia, J.: Hierarchical image saliency detection on extended cssd. In: TPAMI (2015)

46. Squire, L.R., Genzel, L., Wixted, J.T., Morris, R.G.: Memory consolidation. In: Cold Spring Harbor perspectives in biology. Cold Spring Harbor Lab (2015)
47. Ventura, C., Bellver, M., Girbau, A., Salvador, A., Marques, F., Giro-i Nieto, X.: Rvos: End-to-end recurrent network for video object segmentation. In: CVPR (2019)
48. Voigtlaender, P., Chai, Y., Schroff, F., Adam, H., Leibe, B., Chen, L.C.: Feelvos: Fast end-to-end embedding learning for video object segmentation. In: CVPR (2019)
49. Voigtlaender, P., Leibe, B.: Online adaptation of convolutional neural networks for video object segmentation. In: BMVC (2017)
50. Wang, H., Jiang, X., Ren, H., Hu, Y., Bai, S.: Swiftnet: Real-time video object segmentation. In: CVPR (2021)
51. Wang, L., Lu, H., Wang, Y., Feng, M., Wang, D., Yin, B., Ruan, X.: Learning to detect salient objects with image-level supervision. In: CVPR (2017)
52. Wang, Q., Zhang, L., Bertinetto, L., Hu, W., Torr, P.H.: Fast online object tracking and segmentation: A unifying approach. In: CVPR (2019)
53. Wang, Z., Xu, J., Liu, L., Zhu, F., Shao, L.: Ranet: Ranking attention network for fast video object segmentation. In: ICCV (2019)
54. Xie, H., Yao, H., Zhou, S., Zhang, S., Sun, W.: Efficient regional memory network for video object segmentation. In: CVPR (2021)
55. Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., Singh, V.: Nyströmformer: A nyström-based algorithm for approximating self-attention. In: AAAI (2021)
56. Xu, K., Wen, L., Li, G., Bo, L., Huang, Q.: Spatiotemporal cnn for video object segmentation. In: CVPR (2019)
57. Xu, N., Yang, L., Fan, Y., Yue, D., Liang, Y., Yang, J., Huang, T.: Youtube-vos: A large-scale video object segmentation benchmark. In: ECCV (2018)
58. Xu, X., Wang, J., Li, X., Lu, Y.: Reliable propagation-correction modulation for video object segmentation. In: AAAI (2022)
59. Yang, Z., Wei, Y., Yang, Y.: Collaborative video object segmentation by foreground-background integration. In: ECCV (2020)
60. Yang, Z., Wei, Y., Yang, Y.: Associating objects with transformers for video object segmentation. In: NeurIPS (2021)
61. Yang, Z., Wei, Y., Yang, Y.: Collaborative video object segmentation by multi-scale foreground-background integration. In: PAMI (2021)
62. Zeng, Y., Zhang, P., Zhang, J., Lin, Z., Lu, H.: Towards high-resolution salient object detection. In: ICCV (2019)
63. Zhang, L., Lin, Z., Zhang, J., Lu, H., He, Y.: Fast video object segmentation via dynamic targeting network. In: ICCV (2019)
64. Zhang, Y., Wu, Z., Peng, H., Lin, S.: A transductive approach for video object segmentation. In: CVPR (2020)

Appendix

The appendix is structured as follows:

- We first provide a more detailed analysis of the memory consolidation process (Sec. A).
- We then provide qualitative results, comparing the proposed XMem to baselines (Sec. B).
- We demonstrate failure cases (Sec. C).
- We compare different limits on the size of the long-term memory and illustrate the processing rate over the number of processed frames (Sec. D).
- We give quantitative results when retraining the STCN baseline in our training setting (Sec. E).
- We provide results on the YouTubeVOS 2019 validation set (Sec. F).
- We provide detailed quantitative results when XMem is trained on different datasets (Sec. G).
- We explain our multi-scale (MS) evaluation methodology and provide the corresponding enhanced performance of XMem (Sec. H).
- We outline an efficient implementation of the proposed anisotropic L2 similarity function (Sec. I).

A Visualizing Memory Consolidation

Here, we visualize the memory consolidation process (Section 3.3) by showing the candidate frames, some of the selected prototypes, and the corresponding aggregation weights (columns of $\mathbf{W}(\mathbf{k}^c, \mathbf{k}^p)$, each mapping to a distribution over all the candidates). Recall that $\mathbf{W}(\mathbf{k}^c, \mathbf{k}^p)$ is used to aggregate candidate values v^c into prototype values v^p . Figure S1 and Figure S2 show two examples. As illustrated in Figure 5 of the main paper we observe semantically meaningful regions to be grouped.

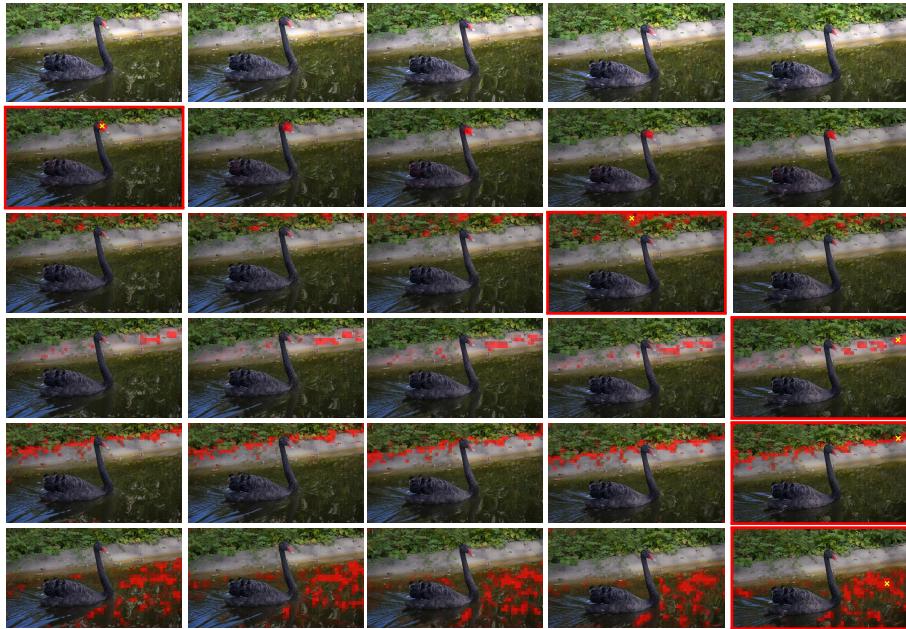


Fig. S1. Visualization of memory consolidation. The first row shows the candidate frames to be converted into long-term memory. Each of the following rows show a prototype position (indicated by a yellow cross), and the corresponding aggregation weights (visualized as a red overlay). Frames that contain a prototype are framed in red. The consolidation process aggregates information from semantically meaningful regions (top-to-bottom): the swan’s beak, part of the vegetation, part of the riverbank, the transition between vegetation and river bank, and part of the water surface.

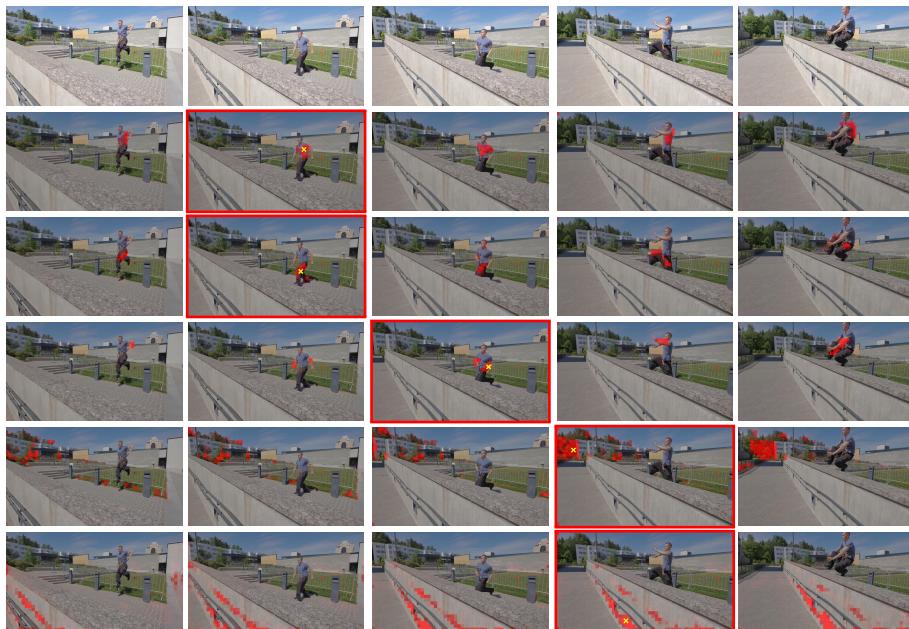


Fig. S2. Visualization of memory consolidation. The first row shows the candidate frames to be converted into long-term memory. Each of the following rows show a prototype position (indicated by a yellow cross), and the corresponding aggregation weights (visualized as a red overlay). Frames that contain a prototype are framed in red. The consolidation process aggregates information from semantically meaningful regions (top-to-bottom): torso, legs, arms, trees, and part of the wall.

B Qualitative Results

Here, we compare qualitatively to JOINT [33], AFB-URR [29], and STCN [9] using several long videos and using the same setting as in the paper. We show results on the **dressage** sequence (10,767 frames) which is part of the Long-time Video ($3\times$) dataset [29], and two additional in-the-wild videos. **breakdance** contains a single foreground object with large and fast motion, and has 18,187 frames. **cans** is very challenging, contains five different objects, two of which (Dr. Pepper and Coca-Cola cans) are similar. The two cans are completely occluded for more than 2,000 frames, and our method can successfully capture them when they reappear. Figure S3, S4, and S5 compare results on these videos respectively. We show one potential application where an image layer is inserted between the foreground and the background using the predicted object mask on a snippet of the **breakdance** sequence.



Fig. S3. Results on the **dressage** sequence. JOINT [33] uses a temporally local feature window and loses track over time. AFB-URR [29] is stable but produces overall less accurate segmentations. STCN [9] uses a low memory insertion frequency to avoid memory explosion, and thus misses fast changes (2nd and 4th column). Ours is sometimes better than the provided ground-truth (last column, the horse’s front legs).

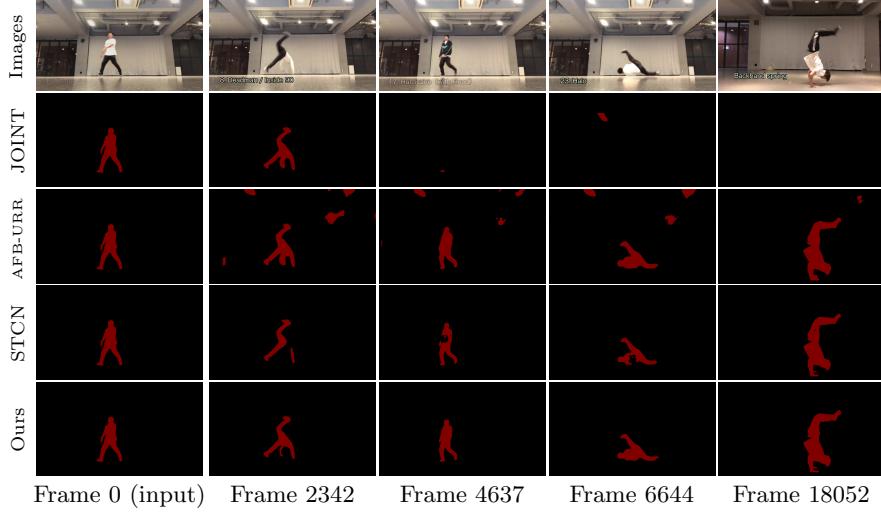


Fig. S4. Results on the **breakdance** sequence. We manually annotated the first frame as input. Similarly to the **dressage** sequence (Figure S3), JOINT [33] loses track over time, AFB-URR [29] is overall less accurate, STCN [9] struggles with fast motion, and our method performs well on this sequence.

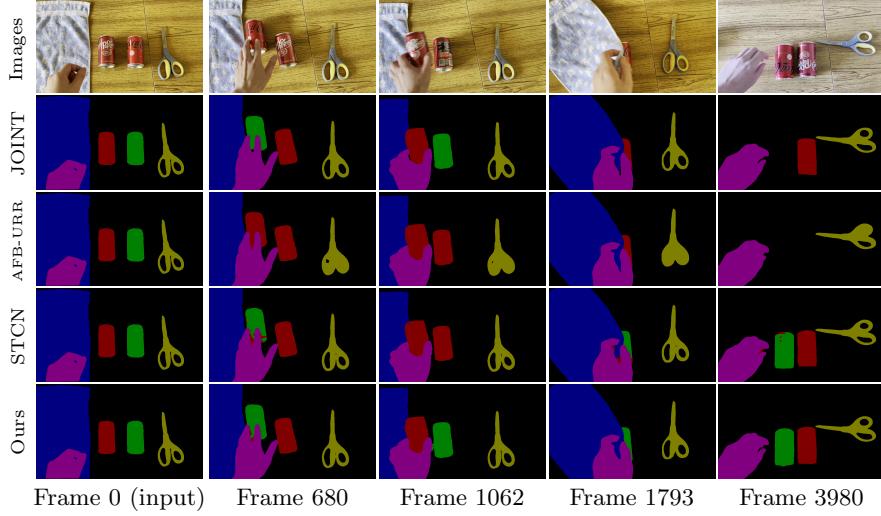


Fig. S5. Results on the **cans** sequence. We manually annotated the first frame as input. The Dr. Pepper can is labeled with red, and the Coca-Cola can is labeled with green. The two cans are completely occluded with a towel after frame 1,793, and reappear about 2,000 frames later. The color tone change is due to the camera's auto white balance. JOINT [33] misses the Coca-Cola can after occlusion. It still captures the Dr. Pepper can as the available first reference frame helps. AFB-URR [29] mixes up the two cans early on, and fails to capture them after they reappear. This is due to its eager feature compression and thus lower modeling capability. STCN [9] uses a low memory insertion frequency to avoid memory explosion which causes it to be less accurate when changes happen. Our method is the most accurate overall.

C Failure Cases

As mentioned in the limitation section of the main text, our method struggles with very fast moving objects. This is because even the fastest-updating sensory memory fails to track such objects, and the working memory fails to model objects with large motion blur. Figure S6 visualizes some failure cases.



Fig. S6. Failure cases. We point to objects of interest with an arrow. First row: multiple birds with similar appearances are flying. We fail to discriminate between some birds that are close to each other. Second row: a frisbee is being thrown. We cannot catch up as it is moving quickly with a large motion blur. Third row: two flags are being waved quickly. We fail to segment the whole left flag due to fast motion.

D Long-Term Memory Size and FPS Scaling

By default, we use a maximum long-term memory size of 10,000 which consumes a small amount of GPU memory and is reasonably capable – it can store information from around 3,900 frames after memory consolidation ($r = 10$). In practice, users might opt for a different upper limit of the long-term memory (LT_{max}), in consideration of any memory constraints, speed, and the complexity of the video. Here, we test the performance of different LT_{max} settings on the Long-time Video (3 \times) dataset [29] and show the results in Table S1. There is significant memory saving and speed-up when LT_{max} is decreased. While a smaller LT_{max} seems to be sufficient for this dataset, we expect using a higher LT_{max} can benefit more challenging videos with long-term occlusions.

We also plot the single-object FPS (1/time required to process a new frame) against the total number of processed frames for STCN [9] and different LT_{max} settings of XMem in Figure S7. We use a high-capacity 32GB V100 GPU in this experiment such that STCN can be run without out-of-memory errors. FPSs for XMem plateau after reaching LT_{max} .

Table S1. Performance of XMem with different upper limits of the long-term memory LT_{max} on the Long-time Video ($3\times$) dataset [29].

LT_{max}	$\mathcal{J}\&\mathcal{F}$	Max. GPU memory	FPS
500	87.2 ± 4.7	1168 MB	35.3
1,000	89.5 ± 0.3	1186 MB	34.1
2,500	89.8 ± 0.2	1243 MB	31.1
5,000	89.9 ± 0.2	1332 MB	27.5
10,000	90.0 ± 0.4	1515 MB	23.4
20,000	90.0 ± 0.4	1632 MB	21.1
30,000	90.0 ± 0.4	1632 MB	20.9

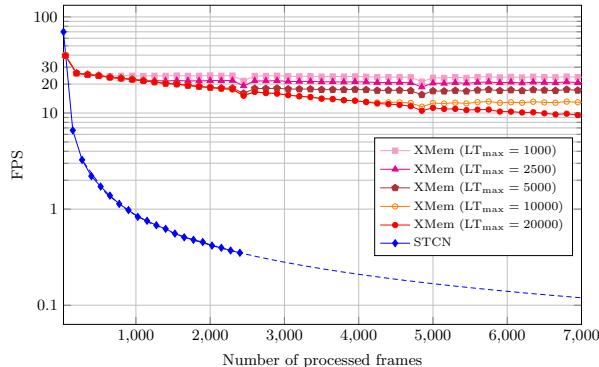


Fig. S7. FPS scaling of STCN [9] vs. variants of XMem. STCN starts off faster due to its simpler construction but slows down drastically as its memory bank expands. As STCN soon becomes too slow for practical use, we estimate its FPS by fitting a linear function to its processing time (i.e., inverse linear to FPS). This linear function is illustrated with a dashed line. XMem maintains a relatively stable and fast FPS throughout, thanks to our memory consolidation algorithm.

E Re-training STCN

We have changed the training schedule (see Section 3.6, Implementation Details) and adjusted parts of the network (including removing some convolutional layers and adding a feature fusion block [9] to the decoder for incorporating the sensory memory). For a fair comparison, we re-train the STCN [9] baseline under our setting. This is equivalent to removing the sensory memory, long-term memory, and both scaling terms. BL30K [8] is not used. Table S2 tabulates the results on the YouTubeVOS 2018 validation set [57] and the DAVIS 2017 validation set [41]. The re-trained method achieves a 1.2 higher $\mathcal{J}\&\mathcal{F}$ on DAVIS and a 1.0 higher \mathcal{G} on YouTubeVOS. On average, the change is insignificant, i.e., our training schedule is not a sufficient condition for improved results.

Table S2. We compare the performance of STCN [9] to the re-trained version with our training setup. On average, there is no significant difference. The performance of XMem is provided as a reference.

Method	YouTubeVOS 2018 val \mathcal{G}	DAVIS 2017 val $\mathcal{J} \& \mathcal{F}$
STCN [9] (original)	83.0	85.4
STCN [9] (re-trained)	84.0	84.2
XMem (Ours)	85.7	86.2

F Results on YouTubeVOS 2019 validation

Table S3 tabulates our results on the YouTubeVOS [57] 2019 validation set. We compare the measured FPS on the 2018 version. The FPS on these two versions are highly correlated as their average number of objects and video length are similar.

Table S3. Quantitative comparisons on YouTubeVOS 2019 validation.

Method	YouTubeVOS 2019 val [57]					
	\mathcal{G}	\mathcal{J}_s	\mathcal{F}_s	\mathcal{J}_u	\mathcal{F}_u	FPS _{Y18}
CFBI [59]	81.0	80.6	85.1	75.2	83.0	3.4
SST [14]	81.8	80.9	-	76.7	-	-
MiVOS* [8]	82.4	80.6	84.7	78.1	86.4	-
HMMN [44]	82.5	81.7	86.1	77.3	85.0	-
CFBI+ [61]	82.6	81.7	86.2	77.1	85.2	4.0
STCN [9]	82.7	81.1	85.4	78.2	85.9	13.2
JOINT [33]	82.8	80.8	84.8	79.0	86.6	-
STCN* [9]	84.2	82.6	87.0	79.4	87.7	13.2
AOT [60]	85.3	83.9	88.8	79.9	88.5	6.4
XMem (Ours)	85.5	84.3	88.6	80.3	88.6	22.6
XMem* (Ours)	85.8	84.8	89.2	80.3	88.8	22.6

G Results with Different Training Datasets

Following prior works [36], we first pretrain our network on static images. As in the implementation of [8,9], we use a mix of single object datasets [45,51,25,62,7]. We compare with prior works that do not use pretraining in Table S4. We additionally present detailed results of our method when it is trained on 1) DAVIS 2017 [40] only, 2) YouTubeVOS 2019 [57] only, and 3) a mix of both, in the followings tables.

Table S4. Comparisons with methods without static image pretraining.

Method	Y ₁₈	Y ₁₉	D ₁₆	D ₁₇	D _{17<td>FPS_{D17}</td>}	FPS _{D17}
LWL [2]	81.5	81.0	-	81.6	-	
SST [14]	81.7	81.8	-	82.5	-	
CFBI+ [61]	82.0	82.6	89.9	82.9	78.0 5.6	
JOINT [33]	83.1	82.8	-	83.5	- 6.8	
Ours ⁻	84.3	84.2	90.8	84.5	79.8 20.2	

Table S5. Performance of XMem on DAVIS 2016 with different training data.

Training data	$\mathcal{J} \& \mathcal{F}$	\mathcal{J}	\mathcal{F}
DAVIS only	87.8	86.7	88.9
DAVIS+YouTubeVOS only	90.8	89.6	91.9
Static+DAVIS+YouTubeVOS	91.5	90.4	92.7
Static+BL30K+DAVIS+YouTubeVOS	92.0	90.7	93.2

Table S6. Performance of XMem on DAVIS 2017 validation with different training data.

Training data	$\mathcal{J} \& \mathcal{F}$	\mathcal{J}	\mathcal{F}
DAVIS only	76.7	74.1	79.3
DAVIS+YouTubeVOS only	84.5	81.4	87.6
Static+DAVIS+YouTubeVOS	86.2	82.9	89.5
Static+BL30K+DAVIS+YouTubeVOS	87.7	84.0	91.4

Table S7. Performance of XMem on DAVIS 2017 test-dev with different training data.

Training data	$\mathcal{J} \& \mathcal{F}$	\mathcal{J}	\mathcal{F}
DAVIS only	64.8	61.4	68.1
DAVIS+YouTubeVOS only	79.8	76.3	83.4
Static+DAVIS+YouTubeVOS	81.0	77.4	84.5
Static+BL30K+DAVIS+YouTubeVOS	81.2	77.6	84.7

Table S8. Performance of XMem on YouTubeVOS 2018 validation with different training data.

Training data	\mathcal{G}	\mathcal{J}_s	\mathcal{F}_s	\mathcal{J}_u	\mathcal{F}_u
YouTubeVOS only	84.4	83.7	88.5	78.2	87.2
DAVIS+YouTubeVOS only	84.3	83.9	88.8	77.7	86.7
Static+DAVIS+YouTubeVOS	85.7	84.6	89.3	80.2	88.7
Static+BL30K+DAVIS+YouTubeVOS	86.1	85.1	89.8	80.3	89.2

Table S9. Performance of XMem on YouTubeVOS 2019 validation with different training data.

Training data	\mathcal{G}	\mathcal{J}_s	\mathcal{F}_s	\mathcal{J}_u	\mathcal{F}_u
YouTubeVOS only	84.3	83.6	88.0	78.5	87.1
DAVIS+YouTubeVOS only	84.2	83.8	88.3	78.1	86.7
Static+DAVIS+YouTubeVOS	85.5	84.3	88.6	80.3	88.6
Static+BL30K+DAVIS+YouTubeVOS	85.8	84.8	89.2	80.3	88.8

H Multi-scale Evaluation

Multi-scale evaluation is a general trick used in segmentation tasks to boost accuracy by combining results from augmented inputs. Common augmentations include scale-change or vertical mirroring. Here, we show XMem’s results with multi-scale evaluation as an attempt to achieve the best performance with a single model without retraining or using a better backbone. For these results, we use $P = 512$ for a relaxed compression. Vertical mirroring is used. Different augmentations are processed independently and the output probability maps are simply averaged.

On DAVIS, we note that a single large scale (720p) is better than merging multiple smaller scales. We use $r = 3$ for better results which has also been noted in STCN [9]. In the test-dev set, we additionally include results with $r = 5$ (i.e., multi-temporal-scale) in the merge. Table S10 tabulates our results.

Table S10. XMem with/without multi-scale evaluation on DAVIS. \ddagger : 600p evaluation.

Method	DAVIS 2017 val			DAVIS 2016 val			DAVIS 2017 test-dev		
	$\mathcal{J} \& \mathcal{F}$	\mathcal{J}	\mathcal{F}	$\mathcal{J} \& \mathcal{F}$	\mathcal{J}	\mathcal{F}	$\mathcal{J} \& \mathcal{F}$	\mathcal{J}	\mathcal{F}
XMem	86.2	82.9	89.5	91.5	90.4	92.7	81.0	77.4	84.5
XMem*	87.7	84.0	91.4	92.0	90.7	93.2	81.2	77.6	84.7
XMem* \ddagger	-	-	-	-	-	-	82.5	79.1	85.8
XMem (MS)	88.2	85.4	91.0	92.7	92.0	93.5	83.1	79.7	86.4
XMem* (MS)	89.5	86.3	92.6	93.3	92.2	94.4	83.7	80.5	87.0

On YouTubeVOS, we adopt multiple scales: $\{480, 528, 576, 624\}$. Unlike on DAVIS, we find larger scales to be unhelpful – which might be due to the overall less accurate annotation of YouTubeVOS. We do not adopt multiple temporal scales here. Table S11 tabulates our results.

Table S11. XMem with/without multi-scale evaluation on YouTubeVOS.

Method	YouTubeVOS 2018 val					YouTubeVOS 2019 val				
	\mathcal{G}	\mathcal{J}_s	\mathcal{F}_s	\mathcal{J}_u	\mathcal{F}_u	\mathcal{G}	\mathcal{J}_s	\mathcal{F}_s	\mathcal{J}_u	\mathcal{F}_u
XMem	85.7	84.6	89.3	80.2	88.7	85.5	84.3	88.6	80.3	88.6
XMem*	86.1	85.1	89.8	80.3	89.2	85.8	84.8	89.2	80.3	88.8
XMem (MS)	86.7	85.3	89.9	81.7	89.9	86.4	84.9	89.2	81.8	89.8
XMem* (MS)	86.9	85.6	90.3	81.7	90.2	86.8	85.5	89.8	81.8	89.9

I Implementation of the Anisotropic L2 Similarity

STCN [9] decomposes the L2 similarity into a sequence of tensor operations for a memory- and compute-efficient implementation. For the proposed similarity function to be practical, a similar decomposition is required. Here, we derive and outline our implementation. Recall the definition of the anisotropic L2 similarity:

We are given key $\mathbf{k} \in \mathbb{R}^{C^k \times N}$, value $\mathbf{v} \in \mathbb{R}^{C^v \times N}$, and query $\mathbf{q} \in \mathbb{R}^{C^q \times HW}$. The key is associated with a shrinkage term $\mathbf{s} \in [1, \infty)^N$ and the query is associated with a selection term $\mathbf{e} \in [0, 1]^{C^k \times HW}$. Then, the similarity between the i -th key element and the j -th query element is computed via

$$\mathbf{S}(\mathbf{k}, \mathbf{q})_{ij} = -\mathbf{s}_i \sum_c^{C^k} \mathbf{e}_{cj} (\mathbf{k}_{ci} - \mathbf{q}_{cj})^2, \quad (\text{S1})$$

which equates to the original L2 similarity [9] if $\mathbf{s}_i = \mathbf{e}_{cj} = 1$ for all i , j , and c .

We use “:” to denote all the elements in a dimension and “@” to denote a singleton dimension to be broadcasted.⁴ \odot denotes the Hadamard (element-wise) product. $\mathbf{1}$ is an all-ones row vector with length C^k .

$$\begin{aligned} \mathbf{S}(\mathbf{k}, \mathbf{q})_{ij} &= -\mathbf{s}_i \sum_c^{C^k} \mathbf{e}_{cj} (\mathbf{k}_{ci} - \mathbf{q}_{cj})^2 \\ &= -\mathbf{s}_i \left(\sum_c^{C^k} \mathbf{e}_{cj} \mathbf{k}_{ci}^2 - \sum_c^{C^k} 2\mathbf{e}_{cj} \mathbf{k}_{ci} \mathbf{q}_{cj} + \sum_c^{C^k} \mathbf{e}_{cj} \mathbf{q}_{cj}^2 \right) \\ &= -\mathbf{s}_i ((\mathbf{k}_{:i} \odot \mathbf{k}_{:i})^T \mathbf{e}_{:j} - 2\mathbf{k}_{:i}^T (\mathbf{e}_{:j} \odot \mathbf{q}_{:j}) + \mathbf{1}(\mathbf{e}_{:j} \odot \mathbf{q}_{:j} \odot \mathbf{q}_{:j})) \\ \Rightarrow \mathbf{S}(\mathbf{k}, \mathbf{q}) &= \mathbf{s}_{:@} (-(\mathbf{k} \odot \mathbf{k})^T \mathbf{e} + 2\mathbf{k}^T (\mathbf{e} \odot \mathbf{q}) - \mathbf{1}(\mathbf{e} \odot \mathbf{q} \odot \mathbf{q})). \end{aligned} \quad (\text{S2})$$

This gives a fully vectorized implementation consisting of only element-wise operations and matrix multiplications with broadcasting.

⁴ Broadcasting as in numpy.