Supplementary Material for:
Memory-augmented Dense Predictive Coding for Video Representation Learning

Tengda Han[0000−0002−1874−9664], Weidi Xie[0000−0003−3804−2639], and Andrew Zisserman[0000−0002−8945−8573]

Visual Geometry Group, Department of Engineering Science, University of Oxford
{htd,weidi,az}@robots.ox.ac.uk

Table of Contents

1 Architectures in detail

This section gives the architectural details of the MemDPC components, including the encoder $f(.)$ and temporal aggregator $g(.)$.

Architecture of encoder $f(.)$**.** The detailed architecture of the encoder function $f(.)$ is shown in Table S1. The size of the convolutional kernel is denoted by [temporal $\times$ spatial^2, channel]. The strides are denoted by [temporal, spatial^2]. We assume the input is 8 video blocks, 5 frames per video block, and the frame has a resolution of 128×128. The column of 'output size' shows the tensor dimension *after* the current stage. Some pooling layers are omitted for clarity. We will release all the source code after the paper decision.

Architecture of temporal aggregator $g(.)$**.** The detailed architecture of the temporal aggregation function $g(.)$ is shown in Table S2. It aggregates the feature maps over the past T time steps. Table S2 shows the case where $g(.)$ aggregates the feature maps over the past 5 steps. We use the same convention as above to denote convolutional kernel size. The temporal aggregator is an one-layer ConvGRU that returns context feature with same number of channels as its input.

Table S1. The 2D+3D ResNet18 structure of the encoding function $f(.)$. The 2D+3D ResNet34 structure replaces the depth of res_2 to res_5 from $\mathbf{2}, \mathbf{2}, \mathbf{2}, \mathbf{2}$ to $\mathbf{3}, \mathbf{4}, \mathbf{6}, \mathbf{3}$.

stage	detail	output size $T \times HW \times C$
input data	-	$5 \times 128^2 \times 3$
conv_1	$1 \times 7^2, 64$ stride $1, 2^2$	$5 \times 64^2 \times 64$
pool_1	$1 \times 3^2, 64$ stride $1, 2^2$	$5 \times 32^2 \times 64$
res_2	$\begin{bmatrix} 1 \times 3^2, 64 \\ 1 \times 3^2, 64 \end{bmatrix} \times \mathbf{2}$	$5 \times 32^2 \times 64$
res_3	$\begin{bmatrix} 1 \times 3^2, 128 \\ 1 \times 3^2, 128 \end{bmatrix} \times \mathbf{2}$	$5 \times 16^2 \times 128$
res_4	$\begin{bmatrix} 3 \times 3^2, 256 \\ 3 \times 3^2, 256 \end{bmatrix} \times \mathbf{2}$	$3 \times 8^2 \times 256$
res_5	$\begin{bmatrix} 3 \times 3^2, 256 \\ 3 \times 3^2, 256 \end{bmatrix} \times \mathbf{2}$	$2 \times 4^2 \times 256$
pool_2	$2 \times 1^2, 256$ stride $1, 1^2$	$1 \times 4^2 \times 256$

Table S2. The structure of aggregation function $g(.)$.

stage	detail	output sizes $T \times t \times d^2 \times C$
input data	-	$5 \times 1 \times 4^2 \times 256$
ConvGRU	$[1^2, 256] \times 1$ layer	$1 \times 1 \times 4^2 \times 256$

2 Details of unintentional action classification

Figure S1 shows the architecture of the unintentional action classification task. In detail, at each time step t, the single-directional `MemDPC` framework produces a feature z_t from the current input x_t, and also a predicted feature $\hat{z}_t$ from the past input $x_1, ..., x_{t-1}$. For each time step, two features z_t and $\hat{z}_t$ are concatenated and passed to a linear classifier $\xi(.)$ to classify into one of three categories (intentional, transitioning or unintentional actions).

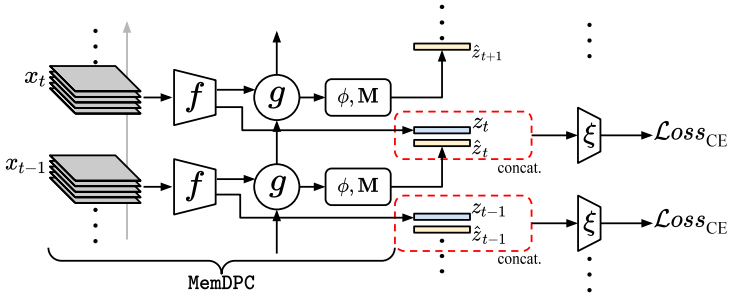

Fig. S1. Architecture of the unintentional action classification framework.

Naturally in the Oops dataset the distribution of the three classes is very unbalanced, *e.g.* transitioning actions are very rare comparing with the other two categories. To handle this we oversample transitioning actions during training. For testing, we take the same sequence length as training from the video with a temporal moving window, and summarize the prediction.

The classifier is trained with cross entropy loss and optimized by the Adam optimizer with a 10^{-3} learning rate. The learning rate is decayed once to 10^{-4} when the validation loss plateaus.

3 Video retrieval results

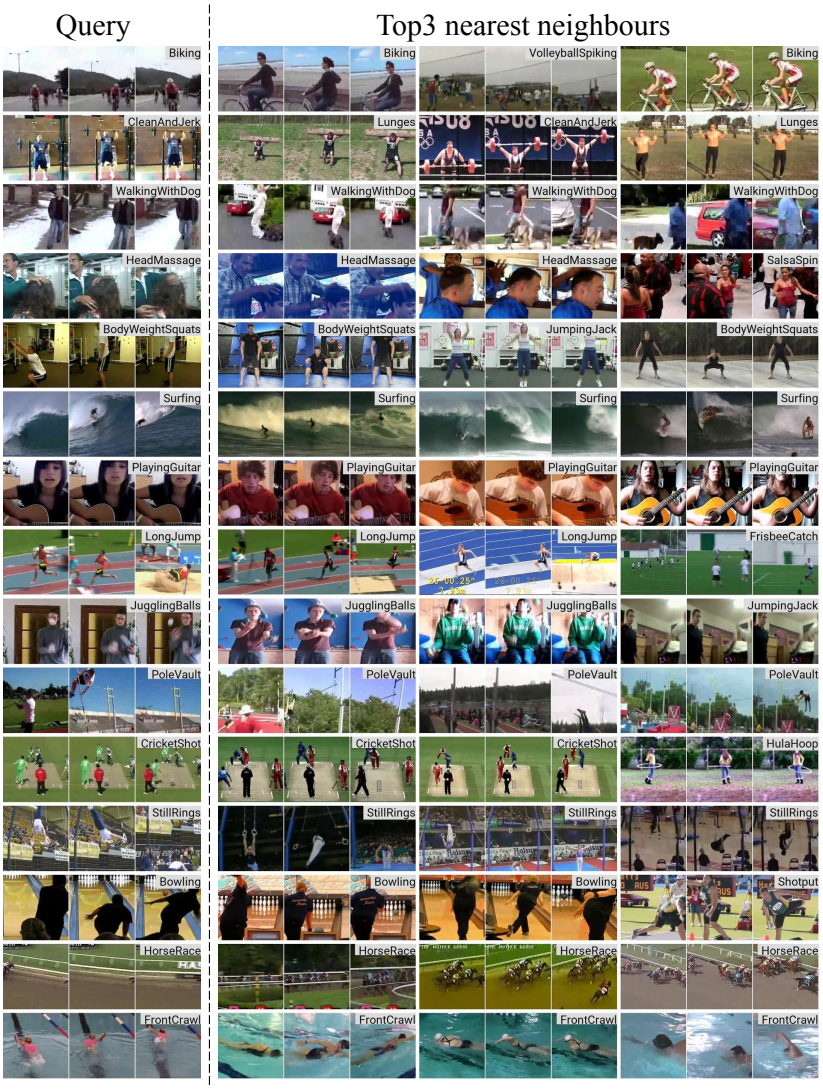

Fig. S2. More nearest neighbour retrieval results with `MemDPC` representations. The left side is the query video from the UCF101 testing set, and the right side are the top 3 nearest neighbours from the UCF101 training set. The `MemDPC` is trained only on UCF101 with optical flow input and we visualize their raw RGB frames for clarity. The action label for each video is shown in the upper right corner.

4 Pseudocode of MemDPC

In this section, we give the core implementation of `MemDPC` in PyTorch-like style, including the compressive memory bank and the computation of the contrastive loss. We will release all the source code.

Algorithm 1 Pseudocode for `MemDPC` in PyTorch-like style.

```
# f: feature extractor, 2d3d-ResNet
# g: aggregator, ConvGRU
# phi: future prediction function, MLP
# x: video input, size = [BatchSize,NumBlock,BlockSize,Channel,Height,Width],
                  # e.g. [16,8,5,3,128,128]
# pred_step: prediction step, e.g. predict 3 steps into the future
# MB: the compressive memory bank, has size [k,C], e.g. [1024, 256]

z_hat_all = []
feature_z = f(x) # extract feature,
                 # size=[B,N,C,H,W], e.g. [16,8,256,4,4]

for i in range(pred_step): # sequentially predict into the future
    if i == 0:
        feature_z_tmp = feature_z[:,0:(N-pred_step),:] # get past features
        hidden = g(feature_z_tmp) # temporal aggregation, hidden state == context,
                                  # size=[B,C,H,W], e.g. [16,256,4,4]
    else:
        hidden = g(feature_z_hat, hidden) # aggregate one more step,
                                          # size=[B,C,H,W]

    prob = Softmax(phi(hidden), dim=1) # probability over MB,
                                       # size=[B,k,H,W], e.g. [16,1024,4,4]
    feature_z_hat = torch.einsum('bkhw,kc->bchw', prob, MB) # weighted sum over MB,
                                                            # size=[B,C,H,W]
    z_hat_all.append(feature_z_hat)

z_hat = torch.stack(z_hat_all, dim=1) # predicted feature, size=[B,pred_step,C,H,W],
                                      # e.g. [16,3,256,4,4]
z = feature_z[:,-pred_step::,:] # desired feature, size=[B,pred_step,C,H,W]

# dot product over 'C', and flatten as 2D matrix, size=[B*pred_step*H*W, B*pred_step*H*W]
similarity = torch.einsum('abcde,fgchi->abdefghi', z_hat, z)\
                .view(B*pred_step*H*W, B*pred_step*H*W)
target = torch.arange(B*pred_step*H*W) # diagonal of similarity matrix is positive

loss = CrossEntropyLoss(similarity, target)
loss.backward()
```

5 Visualization of learned memory

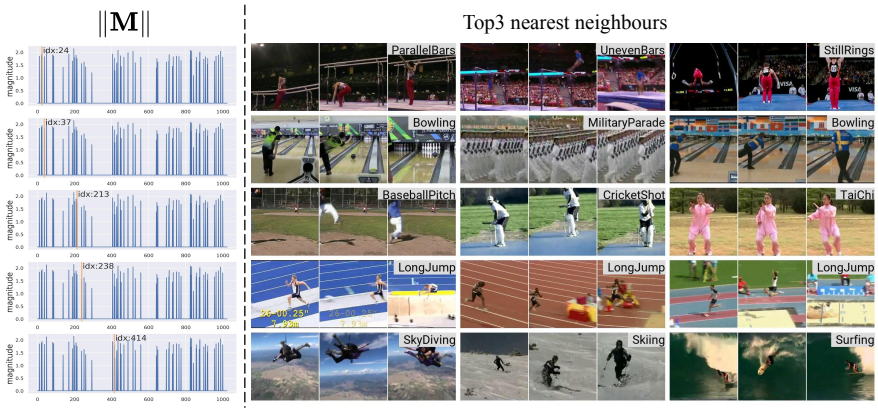

Fig. S3. Nearest-neighbour retrieval for $\{24, 37, 213, 238, 414\}$-th memory entries with UCF101 training videos. The left shows the magnitude of memory bank, $\|\mathbf{M}\| \in \mathbb{R}^{1024}$, with the chosen memory entry highlighted in orange. The right shows the top-3 nearest neighbour in the UCF101 training set retrieved by the corresponding memory entry.

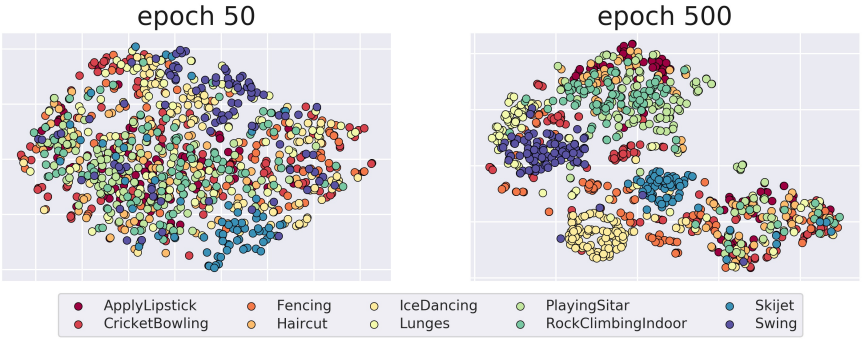

Fig. S4. t-SNE visualization of p_{t+1} from paper Eq. 3 at different training stages. For clarity, 10 action classes are randomly chosen from UCF101 training set and visualized.

This section visualizes the memory learned by `MemDPC`. In Figure S3, we use single memory entry as the query to retrieve videos in the feature space. It shows the memory may have captured certain features in the video like repetitive textures or wide background. Figure S4 shows the t-SNE clustering results of the memory addressing probability p_{t+1} from Equation 3. It shows that as the training progresses, the network attends to different memory entries to predict futures for different action categories, although category information is *not* involved during training.