System-status-aware Adaptive Network for Online Streaming Video Understanding (Supplementary Material)

Lin Geng Foo[1†] Jia Gong[1†] Zhipeng Fan[2§] Jun Liu[1‡]
[1]Singapore University of Technology and Design [2]New York University
{lingeng_foo,jia_gong}@mymail.sutd.edu.sg, zf606@nyu.edu, jun_liu@sutd.edu.sg

In this Supplementary, we first introduce more details of our experimental setup in Sec. 1. Then in Sec. 2, we present more experiments, including performance on other platforms in Sec. 2.2 and additional analysis in Sec. 2.1. Next, we describe more details of our dynamic main network and the agent module in Sec. 3. Finally, we have some further discussions regarding related work in Sec. 4.

1. More Details of Experimental Setup

Here, we present the details of how we build the dynamic system and test the online models. We use several computation programs (e.g., matrix calculation and deep model inference) to act as background processes to occupy computation resources, which leads to a dynamic and fluctuating system status.

Specifically, to generate various system statuses via background processes, we prepare three kinds of controllable programs: 1) Matrix calculation, where three types of matrix operations are used in our experiments, e.g., matrix addition/subtraction, matrix scalar multiplication, and matrix multiplication. The matrix size varies from $[4, 64, 64]$ to $[16, 512, 512]$. 2) Video compression, where we use python to call FFmpeg [2], to convert videos to MP4 format. 3) Deep model inference, where we run various deep learning models, including ResNet101 [5], HrNet-w32 [12] and U-Net [10]. Note that the same program can have multiple parallel processes, e.g. multiple matrix calculations can be performed simultaneously to process different matrices.

We write a script to control the above-mentioned background processes to generate the dynamic system status. By activating different background processes at different time steps according to a generated schedule, we can create various system status conditions, as shown in Fig. 1. During training, we randomly generate various schedules (with each schedule using different processes with different activation and sleep periods) for training the models. For testing, we first randomly generate three schedules with dynamic system statuses, which have not been seen during

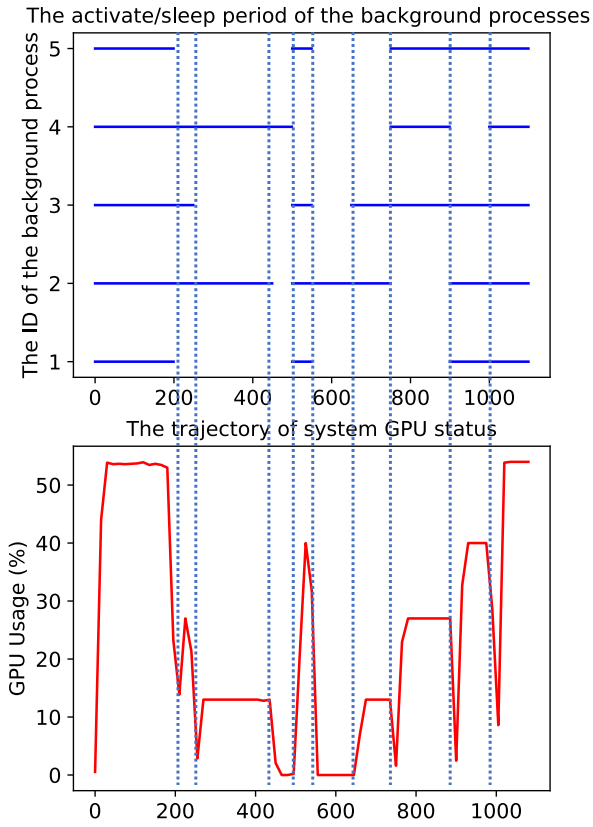


Figure 1. A schedule of 5 background process' activation/sleep periods (top) with the corresponding system status trajectory (bottom). Note that, in our background process schedule (top), the y axis corresponds to the ID of the background process (which ranges from 1 to 5 in this figure), and a solid line means that the specific process is activated for the corresponding duration indicated on the x axis. For the sake of simplicity, in the bottom plot, we only plot the GPU Usage rates, which is a part of our system status sys_t. The GPU Usage rates fluctuate up and down corresponding to the activated background processes, which shows that we can generate different system status trajectories by activating or deactivating background processes.

† Equal contribution; § Currently at Meta; ‡ Corresponding author

training. These three sets of background process schedules (with each leading to a different system status trajectory) are used to test the performance of all models, and the final performance is obtained by averaging the results of each model on all three sets. Notably, we observe that, by controlling the background processes via our script, we can get nearly the same system status trajectory in multiple runs (for a given schedule), and since we average over multiple sets of trajectories, this allows us to have fair comparisons between different models.

Implementation Details. In Sec. 4.1 of the main paper, we conduct our experiments using a laptop consisting of an AMD Ryzen Threadripper 2950X CPU and an NVIDIA Geforce GTX 1080 GPU (11 GB). On this platform, we prepare four matrix calculators, four deep learning models, and two video compressors as the background processes to generate a dynamic system for online models' evaluation. The activate/sleep period length of each process has a duration of $[30, 300]$ frames. In Sec. 4.2 of the main paper, similarly, we set up 10 background processes for all devices. In order to sweep the full range of system status in each device, the configurations of the processes on each device are different.

2. Additional Experiments

2.1. Additional analysis

The performance of each resolution-depth pair. To further observe and analyze the behavior of our learned policy, we first report the performance of each subnetwork (i.e., resolution-depth pair), by testing each individual "static" subnetwork for online action recognition on 50Salads. We present the results in Tab. 1. It shows that all subnetworks can achieve reasonable performance on online action recognition task with different computational complexity.

Table 1. Results of each resolution-depth pair for online action recognition.

Resolution-Depth Pair	Acc. (%)	Max Delay (ms)	Mean Delay (ms)
{112,1}	36.6	31.1	18.3
{112,2}	39.6	42.7	24.8
{112,3}	45.2	66.5	42.1
{168,1}	41.9	45.5	27.2
{168,2}	47.7	77.3	47.8
{168,3}	54.3	98.7	70.6
{224,1}	43.6	59.3	40.5
{224,2}	55.1	101.9	70.6
{224,3}	56.0	174.2	125.3
Our SAN	53.8	49.4	29.9

The selection frequency of each resolution-depth pair. Then, we record the frequency that each subnetwork (i.e., resolution-depth pair) is selected by our agent during the testing phase on the online action recognition task, and present the results in Tab. 2. We observe that each resolution-depth pair is selected at least occasionally, which shows their usefulness, and also the overall efficacy of our design with dynamic resolution and dynamic depth.

Table 2. Selection frequency of each subnetwork (i.e., resolution-depth pair) by our agent for online action recognition.

Subnetwork:	{112,1}	{112,2}	{112,3}	{168,1}	{168,2}
Frequency:	21.4 %	16.7 %	2.5 %	19.8 %	26.0 %
Subnetwork:	{168,3}	{224,1}	{224,2}	{224,3}	
Frequency:	3.5 %	1.6 %	2.8 %	5.7 %	

2.2. More results on other portable devices

To further investigate the effectiveness of our SAN and MSA, we additionally set up experiments on an NVIDIA Jetson TX2 platform with an ARM-based Cortex-A57 CPU and 256-core NVIDIA Pascal GPU (**device d**), a Windows laptop with an Intel Core i5-11400H CPU (**device e**), and an ARM-based M1 macbook (**device f**). These experiments are conducted on the 50Salads dataset, and the results are reported in Tab. 3 and Tab. 4. Tab. 3 shows that, on these platforms, our SAN still obviously outperforms (main network + random policy) on all metrics. It also achieves comparable accuracy to (main network + stream aware), yet the delay of our method is much lower. Note that (main network + stream aware) is similar to our method (SAN), and the only difference is that the system status is not considered in (main network + stream aware). Tab. 4 reports the result of adapting the agent pre-trained on two x86 platforms (Device a and Device b as described in the paper) to an x86 unseen device (Device e) and two ARM-based unseen devices (Device d and f). Results show that our MSA algorithm significantly improves the accuracy from the baseline (Direct Transfer) on all metrics.

3. More Details on Architecture of SAN

Below, we provide more details on our SAN network.

3.1. Our dynamic main network

Our dynamic main network adjusts its computation complexity by dynamically changing the execution depth and the input resolution, as shown in Fig. 2. Below, we introduce more details of how our network achieves dynamic depth and resolution.

Network for online action recognition. For online action recognition, we build a dynamic encoder based on ResNet50, in which three exit ports are added at the end of specific convolution blocks $\{conv3\_x, conv4\_x, conv5\_x\}$ [5] to achieve dynamic depth, as illustrated in Fig. 2. By producing predictions using only the early layers of the encoder (i.e., early-exit), we can directly reduce the computational complexity and the time consumption. Furthermore, because we select an encoder that is fully-convolutional

Table 3. Results of our SAN on other platforms (**device d, e and f**).

Method	d			e			f		
	Accuracy	Max delay	Mean Delay	Accuracy	Max delay	Mean Delay	Accuracy	Max delay	Mean Delay
main network + random policy	47.1 %	251.7 ms	93.3 ms	45.9 %	174.6 ms	78.6 ms	46.5 %	177.0 ms	82.2 ms
main network + stream aware	55.4 %	261.3 ms	112.6 ms	55.4 %	161.9 ms	82.0 ms	55.4 %	194.7 ms	110.4 ms
main network + our SAN	54.1 %	89.7 ms	60.4 ms	55.1 %	67.5 ms	59.1 ms	54.6 %	91.3 ms	73.4 ms

Table 4. Results of our SAN on other platforms (**device d, e and f**).

Method	a+b $\rightarrow$ d			a+b $\rightarrow$ e			a+b $\rightarrow$ f		
	Accuracy	Max Delay	Mean Delay	Accuracy	Max Delay	Mean Delay	Accuracy	Max Delay	Mean Delay
Direct Transfer	41.5 %	94.7 ms	79.0 ms	37.9 %	85.9 ms	59.4 ms	46.8 %	114.3 ms	89.5 ms
Our MSA	51.2 %	91.5 ms	68.4 ms	50.2 %	82.3 ms	55.1 ms	53.4 %	95.8 ms	76.3 ms

(ResNet50), it is also naturally able to process inputs with varying resolutions. Specifically, the inputs are sized to one of three resoluion candidates: $[112, 168, 224]$.

Due to the varying size of the dynamic encoder's output at different exits and using different input resolutions, we design a unification module for each resolution-depth pair in order to unify the shapes of the dynamic encoder's output features. The unification module consists of a pooling/upsampling layer and a 1×1 convolution layer, as shown in Fig. 2. Specifically, we first utilize a pooling/upsampling layer to resize the resolution of the encoder output to 7×7 and then expand the channel number of the resized output to 2048 to obtain the final unified feature $2048 \times 7 \times 7$ via a 1×1 convolutional layer. We remark that the unification module not only resizes the shape of the encoder's output but also helps to map features from various resolution-depth pairs to a common space. Lastly, we perform global average pooling on the unified feature to obtain the 2048-dimensional vector and feed this vector into an LSTM, whose output is fed into the task head to predict the class of the action. Here, the task head is a single fully connected layer.

Network for online pose estimation. For online pose estimation, the structure of the dynamic encoder is similar to that of action recognition, although the resolution candidates are instead: $[128, 192, 256]$. Also, differently, we utilize the unification module to resize the output of the encoder to $8 \times 8 \times 2048$ and feed the unified feature into a ConvLSTM layer [11] for processing the dense spatial-temporal information. Moreover, our pose estimation task head, which receives the output of ConvLSTM and outputs heatmaps for pose estimation, consists of three deconvolutional layers. The structure of the deconvolutional layers is the same as [18].

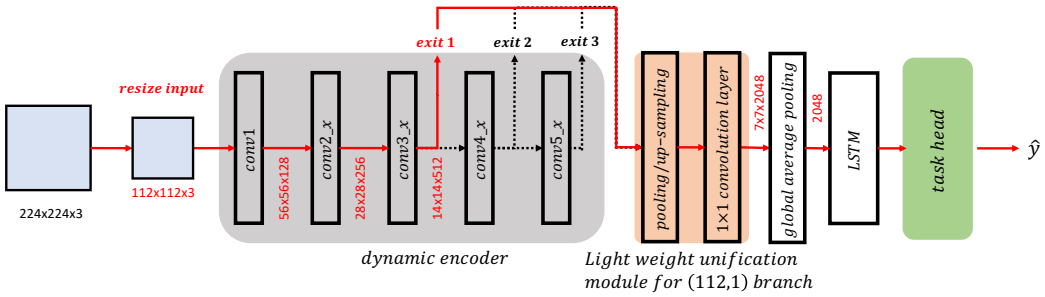
3.2. Our agent module

For our agent module, as shown in Fig. 3, we first utilize two fully connected layers to process the system information sys_t and the hidden state h_{t-1} in parallel. Next, we combine the outputs of these layers with additional useful information g_t (i.e., the previous step's action a_{t-1} and delay d_{t-1}) to build the intermediate feature f_{im}, which is then fed into two linear layers with a ReLU activation layer in between to generate the action distribution P_t to decide the action a_t. After that, in order to facilitate MSA, the intermediate feature f_{im} is combined with a_t and sent to the auxiliary head to predict the delay $\hat{d}_t$.

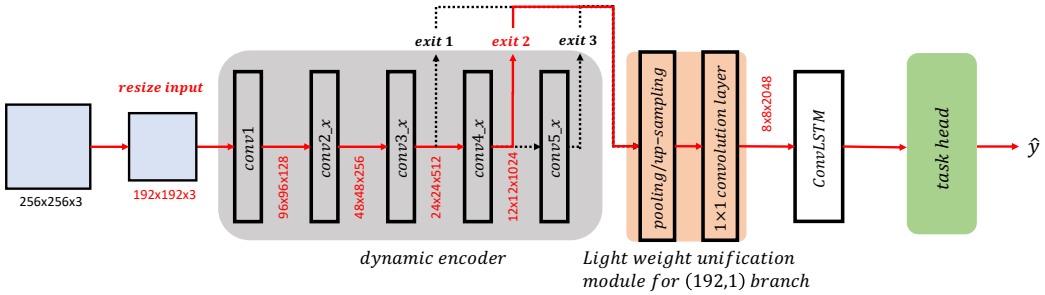
Here, we utilize the nvitop [19] and psutil [4] tools to capture the list of system status information to build the system status $sys_t \in \mathbb{R}^{27}$. We display the list below.

- **GPU status:** GPU's current usage rate, GPU's occupied memory, GPU's available memory, GPU's fan speed, GPU's power, GPU's temperature, and the number of running processes on GPU.
- **CPU status**: the number of CPU threads, the CPU's current load, the CPU's average load in the last 1 minute, 5 minutes, and 15 minutes, and the average temperature of the CPU.
- **Virtual Memory status**: the percentage of used virtual memory, the percentage of the virtual memory not being used at all (zeroed) that is readily available, the active virtual memory, and the inactive virtual memory.
- **Swap Memory status**: the total swap memory, the used swap memory, the free swap memory, the percentage of swap memory's usage, the number of bytes the system has swapped in from disk, and the number of bytes the system has swapped out from the disk.
- **I/O status:** the number of reads, the number of writes, the number of bytes read, and the number of bytes written.

Next, we collect two pieces of information that are generated in the previous step: the delay d_{t-1} and the selected action a_{t-1} to build the extra information vector $g_t \in \mathbb{R}^2$. Moreover, for action recognition, we directly



(a) The architecture of the main network for online action recognition



(b) The architecture of the main network for online pose estimation

Figure 2. The architecture of our dynamic main network. (a) The main network for online action recognition. The red line illustrates an example of the flow of the inputs based on the agent's policy, where the input resolution is resized to 112 and the 1^{st} exit port is activated. (b) The main network for online pose estimation. The red line illustrates an example of the flow of the inputs based on the agent's policy, where the input is resized to 192 and the 2^{nd} exit port is activated.

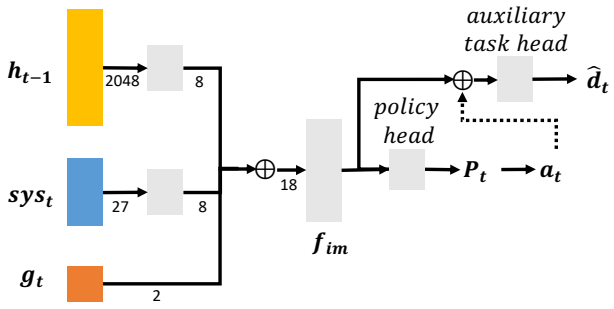


Figure 3. Illustration of our agent's architecture. The system status sys_t and the hidden state h_{t-1} are first separately fed into linear layers with ReLU activation. Then, the outputs of these linear layers are concatenated with the other useful information g_t to generate the intermediate feature $f_{im} \in \mathbb{R}^{18}$. Next, this feature f_{im} is fed into the policy head to generate the action probability distribution P_t to select action a_t. To facilitate MSA, the feature f_{im} is concatenated with a_t, and sent to the auxiliary task head to predict the delay $\hat{d}_t$. Both the policy head and the auxiliary task head consists of two fully-connected layers with a ReLU activation layer in between.

use the output of the LSTM in the previous step to build $h_{t-1} \in \mathbb{R}^{2048}$. For pose estimation, we pass the ConvLSTM's output through a global average pooling layer to generate h_{t-1}.

4. More Discussion on Related Work

Dynamic Networks [1, 3, 6–9, 13, 14, 17, 20] generally adapt their structure or parameters according to the input, i.e., they are input-aware. Dynamic networks are mainly adopted with the aim of computational efficiency [1, 6, 8, 9, 14–17], although they can also be used to improve accuracy [3, 7]. Existing dynamic networks can improve efficiency by dynamically selecting a cheaper subnetwork (e.g., skipping layers [14, 17], reducing channels [13]) when it is sufficient to provide a good result for the given input, thus reducing the number of redundant computations. However, existing input-aware dynamic networks have the *same policy under all different system conditions and platforms*. Therefore, since system status conditions can fluctuate, these input-aware dynamic networks can make sub-optimal decisions in real environments. Compared to existing dynamic networks, our contribution lies in 2 aspects: (1) Our SAN is the first to be both input-aware and *system-status-aware*, which considers the system status conditions when making dynamic decisions. (2) We are also the first to consider the challenging *cross-platform adaptation for dy-*

namic networks. In order to conveniently deploy our SAN for different platforms, we propose a novel MSA algorithm to facilitate self-supervised adaptation to a target deployment device, without the need for the original labeled data.

References

[1] Zhipeng Fan, Jun Liu, and Yao Wang. Adaptive computationally efficient network for monocular 3d hand pose estimation. In *European Conference on Computer Vision*, pages 127–144. Springer, 2020. 4

[2] FFmpeg. Ffmpeg/ffmpeg. 1

[3] Lin Geng Foo, Tianjiao Li, Hossein Rahmani, Qiuhong Ke, and Jun Liu. Era: Expert retrieval and assembly for early action prediction. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXIV*, pages 670–688. Springer, 2022. 4

[4] giampaolo. psutil. https://github.com/giampaolo/psutil. Accessed: 2022-4-25. 3

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1, 2

[6] Hengduo Li, Zuxuan Wu, Abhinav Shrivastava, and Larry S Davis. 2d or not 2d? adaptive 3d convolution selection for efficient video recognition. In *CVPR*, 2021. 4

[7] Tianjiao Li, Lin Geng Foo, Qiuhong Ke, Hossein Rahmani, Anran Wang, Jinghua Wang, and Jun Liu. Dynamic spatio-temporal specialization learning for fine-grained action recognition. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part IV*, pages 386–403. Springer, 2022. 4

[8] Yue Meng, Chung-Ching Lin, Rameswar Panda, Prasanna Sattigeri, Leonid Karlinsky, Aude Oliva, Kate Saenko, and Rogerio Feris. Ar-net: Adaptive frame resolution for efficient action recognition. In *European Conference on Computer Vision*, pages 86–104. Springer, 2020. 4

[9] Xuecheng Nie, Yuncheng Li, Linjie Luo, Ning Zhang, and Jiashi Feng. Dynamic kernel distillation for efficient pose estimation in videos. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6942–6950, 2019. 4

[10] O. Ronneberger, P.Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]). 1

[11] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015. 3

[12] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang. Deep high-resolution representation learning for human pose estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5693–5703, 2019. 1

[13] Yehui Tang, Yunhe Wang, Yixing Xu, Yiping Deng, Chao Xu, Dacheng Tao, and Chang Xu. Manifold regularized dynamic network pruning. In *CVPR*, 2021. 4

[14] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *ECCV*, 2018. 4

[15] Yawen Wu, Zhepeng Wang, Zhenge Jia, Yiyu Shi, and Jingtong Hu. Intermittent inference with nonuniformly compressed multi-exit neural network for energy harvesting powered devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 4

[16] Zuxuan Wu, Hengduo Li, Caiming Xiong, Yu-Gang Jiang, and Larry S Davis. A dynamic frame selection framework for fast video recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(4):1699–1711, 2020. 4

[17] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *CVPR*, 2018. 4

[18] Bin Xiao, Haiping Wu, and Yichen Wei. Simple baselines for human pose estimation and tracking. In *Proceedings of the European conference on computer vision (ECCV)*, pages 466–481, 2018. 3

[19] XuehaiPan. nvitop. https://github.com/XuehaiPan/nvitop. Accessed: 2022-4-25. 3

[20] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. Condconv: Conditionally parameterized convolutions for efficient inference. 2019. 4