DEEPEYE: A Deeply Tensor-Compressed Neural Network Hardware Accelerator

Invited Paper

Yuan Cheng*, Guangya Li†, Ngai Wong‡, Hai-Bao Chen*, Hao Yu†
*Department of Micro/Nano Electronics, Shanghai Jiao Tong University, Shanghai, China
†School of Microelectronics, Southern University of Science and Technology, Shenzhen, China
‡Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong

***Abstract*—Video detection and classification constantly involve high dimensional data that requires a deep neural network (DNN) with huge number of parameters. It is thereby quite challenging to develop a DNN video comprehension at terminal devices. In this paper, we introduce a deeply tensor compressed video comprehension neural network called DEEPEYE for inference at terminal devices. Instead of building a Long Short-Term Memory (LSTM) network directly from raw video data, we build a LSTM-based spatio-temporal model from tensorized time-series features for object detection and action recognition. Moreover, a deep compression is achieved by tensor decomposition and trained quantization of the time-series feature-based spatio-temporal model. We have implemented DEEPEYE on an ARM-core based IOT board with only 2.4W power consumption. Using the video datasets MOMENTS and UCF11 as benchmarks, DEEPEYE achieves a $228.1\times$ model compression with only 0.47% mAP deduction; as well as $15k\times$ parameter reduction yet 16.27% accuracy improvement.**

***Index Terms*—Video comprehension network, terminal devices, time-series features, tensorized compression**

I. Introduction

Various computer vision applications have witnessed outstanding progress in recent years. However, application such as an advanced driver assistance system (ADAS) requires a real-time sequence-to-sequence video object detection and action recognition (video comprehension), which still remains a challenging task especially for edge/terminal devices [1], [2]. The success of convolutional neural networks (CNNs) has resulted in a generic feature extraction engine for single image data which, however, cannot directly identify the temporal relationship of objects or actions in the video. To further understand the sequence information in video data, recurrent neural networks (RNNs) have been applied to spatio-temporal modeling [3], [4], [5]. It is unclear how to develop a high-throughput yet low-power video comprehension system in resource-limited terminal devices properly, given the deeper layer of neural networks and larger dimension of video data.

Existing approaches for video comprehension are mostly using CNNs to process each frame as an image. Recent works in [6], [7] build models with local features extracted from low-level motion cues, such as color, texture and optical flow. The two-stream CNN is applied in [8] to separate spatial features and temporal computation. However, no sequence-to-sequence image data is analysed with correlation in these models and hence may lose accuracy when performing video comprehension. This use of both CNN and RNN has become an appealing scheme for processing video data, e.g., pre-trained CNNs are used in [9], [10] where raw features are extracted to construct RNNs. 3D-CNNs are proposed in [11], [12] to learn video representations by replacing the 2D convolutions with 3D spatial convolutions. These approaches, however, increase the computational complexity without a convincing accuracy, and result in limited performance at terminal devices.

It is widely perceived that the sequence-to-sequence modeling efficiency of RNNs and their variants is low. The improvements on the dense connections in RNNs are discussed with several alternative structures. The low-rank approximation of the weight matrices was proposed in [13], namely, $\boldsymbol{W} = \boldsymbol{U} \times \boldsymbol{V}$, where $\boldsymbol{W} \in \mathbb{R}^{m \times n}$, $\boldsymbol{U} \in \mathbb{R}^{m \times r}$ and $\boldsymbol{V} \in \mathbb{R}^{r \times n}$. Therefore, the weight matrix can be compressed from mn to $(m+n)r$ parameters given a small rank r. However, with the huge amount of connections in RNNs for video data, compression using matrix decomposition is not effective and scalable. To this end, tensor decomposition methods are applied to deep neural networks [14], [15]. A tensor decomposition generalizes low-rank matrix decomposition to high dimensional data (tensors), usually resulting in higher compression ratio than its matrix counterpart. The recent works in [16], [17] propose the canonical polyadic (CP), Tucker and tensor-train decompositions to significantly reduce the parameters and computation in RNNs. Another work in [18] uses tensor trains to forecast long-term information. Although these tensor works have made a significant advance in compressing RNNs and their variants like LSTM, their primary limitation is the direct dealing with the raw input data but not the features in the sequence.

In this work, instead of inputting the direct video frames or raw feature data, we build a LSTM-based spatio-temporal model from *tensorized time-series features* aggregated from a CNN-based *feature extractor*. It allows a large flexibility for optimizing the subsequent tensor cores in such a tensorized feature format. To facilitate lightweight video comprehension deployment, we further develop a trained quantization (8-bit) for both feature extractor and LSTM. The proposed tensorization and quantization significantly reduce the network dimension yet with maintained accuracy, called DEEPEYE. DEEPEYE seamlessly leads into a video comprehension sys-

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

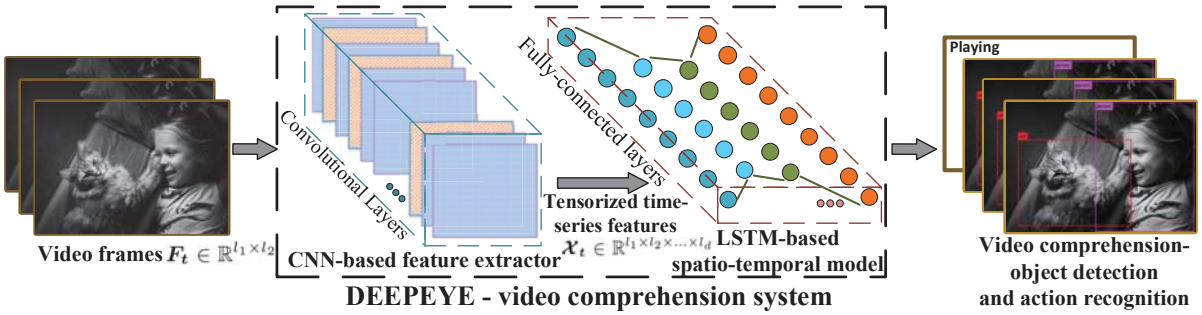

Fig. 1. DEEPEYE: an embedded video comprehension system for object detection and action recognition.

tem that can be realized at terminal devices.

The contributions of this paper come from threefold: **1)** We make the novel use of feature extractor and pool the time-series features into a tensor serving as input to the next-stage tensorized LSTM, instead of the conventional direct video frames or raw feature data into an RNN; **2)** We adopt an 8-bit quantized training strategy, for the first time, on the large-scale RNN networks subject to accuracy control; **3)** The realization of a challenging deep neural network for video comprehension on an ARM-core based IOT device. Experimental results on challenging video datasets show that DEEPEYE achieves $228.10\times$ compression with only 0.47% mAP decrease; and $15k\times$ parameter reduction and 16.27% accuracy improvement, while consuming only 2.4W power.

The rest of the paper is organized as follows. Section II describes the LSTM-based spatio-temporal model for DEEPEYE as well as the definition of time-series features. Section III introduces the proposed tensorized decomposition and further trained quantization of DEEPEYE. Section IV presents the overall workflow of DEEPEYE and its realization on an ARM-core based IOT board. Section V shows experimental results on both GPU and IOT board with conclusion in Section VI.

II. DEEPEYE Framework

Here we present the design of proposed DEEPEYE framework for video comprehension, which combining the object detection and action recognition, depicted in Fig. 1[1] The design of DEEPEYE incorporates the feedback from both the network algorithm and hardware implementation, which can be summarized into 3 aspects: **1)** Both object detection and action recognition should be realized in the proposed DEEPEYE framework with maintained accuracy. **2)** DEEPEYE should not contain too many network parameters or cost too many computation and storage resources. **3)** DEEPEYE should be highly integrated and compressed to work efficiently and hardware-friendly at terminal devices. To satisfy these requirements, first we propose the novel embedded video comprehension system in this section, next we systematically develop the tensorized and quantized compression on the whole system in the following sections.

In practice, we construct a spatio-temporal model based on an LSTM network using the tensorized time-series features aggregated from a CNN-based feature extractor. Assuming that $\boldsymbol{F_t} \in \mathbb{R}^{l_1 \times l_2}$ are the video frames, $\boldsymbol{x_t} \in \mathbb{R}^{l}$ are the raw features and $\boldsymbol{\mathcal{X}_t} \in \mathbb{R}^{l_1 \times l_2 \times \ldots \times l_d}$ are the tensorized time-series features, where subscript t denotes the time sequence, l represents the mode size of dimension and d is the dimensionality of the tensor. As shown in Fig. 1, the feature extractor uses several convolutional layers to learn tensorized time-series features from video frames:

$$extract(\boldsymbol{F_t}) = \boldsymbol{\mathcal{X}_t}, \tag{1}$$

the $extract$ operation represents the corresponding extraction method in the proposed feature extractor. Then the LSTM cells (consist of fully-connected layers) in LSTM network take $\boldsymbol{\mathcal{X}_t}$ as inputs, instead of direct video frames $\boldsymbol{F_t}$ or raw features $\boldsymbol{x_t}$, to learn the spatio-temporal information. Each LSTM cell keeps track of an internal state that represents its memory and learns to update its state over time based on the current input and past states. The computation within the proposed spatio-temporal model are as follows:

$$\boldsymbol{S_t} = \sigma(\boldsymbol{W_s}\boldsymbol{\mathcal{X}_t} + \boldsymbol{U_s}\boldsymbol{H_{t-1}} + \boldsymbol{B_s}), \tag{2a}$$
$$\boldsymbol{Z_t} = \sigma(\boldsymbol{W_z}\boldsymbol{\mathcal{X}_t} + \boldsymbol{U_z}\boldsymbol{H_{t-1}} + \boldsymbol{B_z}), \tag{2b}$$
$$\boldsymbol{D_t} = \sigma(\boldsymbol{W_d}\boldsymbol{\mathcal{X}_t} + \boldsymbol{U_d}\boldsymbol{H_{t-1}} + \boldsymbol{B_d}), \tag{2c}$$
$$\tilde{\boldsymbol{C}}_t = tanh(\boldsymbol{W_c}\boldsymbol{\mathcal{X}_t} + \boldsymbol{U_c}\boldsymbol{H_{t-1}} + \boldsymbol{B_c}), \tag{2d}$$
$$\boldsymbol{C_t} = \boldsymbol{S_t} \otimes \boldsymbol{C_{t-1}} + \boldsymbol{Z_t} \otimes \tilde{\boldsymbol{C}}_t, \tag{2e}$$
$$\boldsymbol{H_t} = \boldsymbol{D_t} \otimes tanh(\boldsymbol{C_t}), \tag{2f}$$

where $\otimes$ denotes the element-wise product, $\sigma(\circ)$ represents the sigmoid function and $tanh(\circ)$ represents the hyperbolic tangent function. $\boldsymbol{H_{t-1}}$ and $\boldsymbol{C_{t-1}}$ are the previous hidden state and previous update factor, $\boldsymbol{H_t}$ and $\boldsymbol{C_t}$ are the current hidden state and current update factor, respectively. The weight matrices $\boldsymbol{W_*}$ and $\boldsymbol{U_*}$ weigh the input $\boldsymbol{\mathcal{X}_t}$ and the previous hidden state $\boldsymbol{H_{t-1}}$ to update factor $\tilde{\boldsymbol{C}}_t$ and three sigmoid gates, namely, $\boldsymbol{S_t}$, $\boldsymbol{Z_t}$ and $\boldsymbol{D_t}$.

For each frame in video comprehension, the LSTM cell calculate their information by combining previous states and current features. Therefore, all video information can be captured from the beginning till the current frame. It should be noted that we make novel use of *tensorized time-series features* instead of the direct video frames [6] or raw feature data [9] as inputs to LSTM, as shown in Fig. 2. This way, the LSTM is fed with distilled information to deliver high accuracy and performance.

However, the dense fully-connected computation in proposed spatio-temporal model suffers from high dimensional inputs and huge number of parameters, making it hard to train

[1]All videos and images for illustrations in this paper are from MOMENTS, UCF11 and COCO.

and susceptible to overfitting even when it fully converges. To address this issue, a tensor decomposition strategy and an further 8-bit quantization is applied to DEEPEYE (see Section III).

III. Tensorized Compression of DEEPEYE

Herein, we present a tensorized LSTM (T-LSTM) to compress the LSTM-based spatio-temporal model and reduce the cost of computation and storage resources.

A. Tensor Decomposition

Tensors are multi-dimensional generalization of matrices. Specifically, 1-dimensional data arrays are vectors, represented in this paper by lowercase letters $\boldsymbol{x} \in \mathbb{R}^l$, $\boldsymbol{x}(h_1)$ refers to a specific element by the index h_1; 2-dimensional data arrays are matrices, represented by uppercase letters, denoted as $\boldsymbol{X} \in \mathbb{R}^{l_1 \times l_2}$, $\boldsymbol{X}(h_1, h_2)$ refers to a specific element by the 2-dimensional index h_1, h_2. And the higher dimensional data arrays are tensors, represented by calligraphic letters $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{l_1 \times l_2 \times \ldots \times l_d}$, $\boldsymbol{\mathcal{X}}(h_1, h_2, \ldots, h_d)$ refers to a specific element by the d-dimensional index $h_1, h_2, \ldots, h_d$.

The number of parameters in a d-dimensional tensor is $l_1 l_2 \cdots l_d$, which grows exponentially as d increases. In practice, tensor decomposition is used to find the low-rank approximation of a high-dimensional array, expressed as a series of d smaller-size factors. This reduces the computation complexity from exponential to only linear, thereby eluding the curse of dimensionality. Furthermore, in the low-dimensional but large-scale problems, to maintain a higher compression ratio and exploit the spatial information, we can tensorize an input vector $\boldsymbol{x}$ or matrix $\boldsymbol{X}$ to a high-dimensional tensor $\boldsymbol{\mathcal{X}}$ using the $reshape$ operation (see Fig. 3) followed by tensor decomposition. In particular, the tensor train [14], [15] is a promising decomposition that scales well to an arbitrary number of dimensions. Given a d-dimensional tensor $\boldsymbol{\mathcal{X}}$, the tensor train decomposition reads

$$\begin{aligned}\boldsymbol{\mathcal{X}}(h_1, h_2, \ldots, h_d) = \sum_{\alpha_0, \alpha_1, \ldots, \alpha_d}^{r_0, r_1, \ldots, r_d} \boldsymbol{\mathcal{G}}_1(\alpha_0, h_1, \alpha_1) \\ \times \boldsymbol{\mathcal{G}}_2(\alpha_1, h_2, \alpha_2) \ldots \boldsymbol{\mathcal{G}}_d(\alpha_{d-1}, h_d, \alpha_d),\end{aligned} \tag{3}$$

where $\boldsymbol{\mathcal{G}_k} \in \mathbb{R}^{r_{k-1} \times l_k \times r_k}$ is called the tensor core, r_k is the tensor core rank, α_k is the summation index which starts from

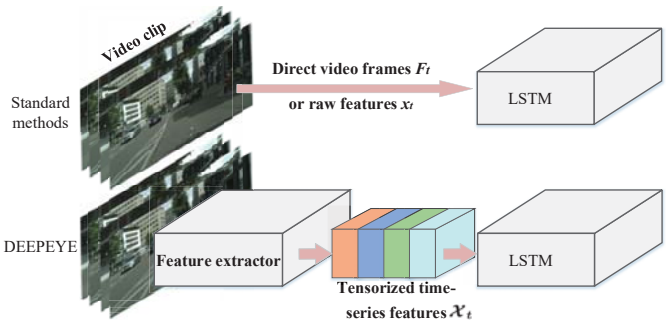

Fig. 2. Comparison between the standard methods with raw data inputs and the DEEPEYE with tensorized time-series features.

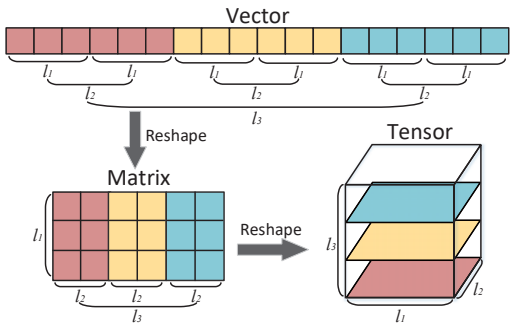

Fig. 3. Reshaping a vector into a matrix and then into a 3-dimensional tensor.

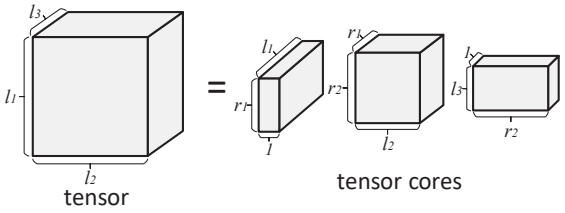

Fig. 4. Tensor train decomposition of a 3-dimensional tensor.

1 to r_k, and $l_1, l_2, \ldots, l_d$ are known as mode sizes of the d-dimensional tensor. Also, $r_0 = r_d = 1$ so that the right hand side of Eq. (3) is a scalar. For a more compact expression, using the notation of $\boldsymbol{\mathcal{G}_k}(h_k) \in \mathbb{R}^{r_k \times r_{k-1}}$, namely, a 2-dimensional slice from the 3-dimensional tensor $\boldsymbol{\mathcal{G}_k}$, Eq. (3) can be written as

$$\boldsymbol{\mathcal{X}}(h_1, h_2, \ldots, h_d) = \boldsymbol{\mathcal{G}}_1(h_1)\boldsymbol{\mathcal{G}}_2(h_2) \ldots \boldsymbol{\mathcal{G}}_d(h_d). \tag{4}$$

The decomposition of a 3-dimensional tensor is graphically shown in Fig. 4. Imposing the constraint that each integer l_k in Eq. (4) can be further decomposed as $l_k = n_k m_k$, so that each tensor core $\boldsymbol{\mathcal{G}}_k$ can be permuted and reshaped into $\boldsymbol{\mathcal{G}}_k^t \in \mathbb{R}^{n_k \times m_k \times r_k \times r_{k-1}}$

$$\boldsymbol{\mathcal{G}}_k(h_k) = \boldsymbol{\mathcal{G}}_k^t(j_k, i_k), \tag{5}$$

where $\boldsymbol{\mathcal{G}}_k^t(j_k, i_k) \in \mathbb{R}^{r_{k-1} \times r_k}$. Therefore, the decomposition for the tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{(n_1 \times m_1) \times (n_2 \times m_2) \times \ldots \times (n_d \times m_d)}$ can be correspondingly reformulated as

$$\begin{aligned}\boldsymbol{\mathcal{X}}((j_1, i_1), (j_2, i_2), \ldots, (j_d, i_d)) \\ = \boldsymbol{\mathcal{G}}_1^t(j_1, i_1)\boldsymbol{\mathcal{G}}_2^t(j_2, i_2) \ldots \boldsymbol{\mathcal{G}}_d^t(j_d, i_d).\end{aligned} \tag{6}$$

Such double-index trick is utilized to decompose the fully-connected computation as will be discussed next.

B. Extending to LSTM

The most expensive computation in an RNN is the large-scale matrix-vector multiplication:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \tag{7}$$

where $\boldsymbol{W} \in \mathbb{R}^{N \times M}$ is the weight matrix, $\boldsymbol{x} \in \mathbb{R}^M$ is the feature vector, $\boldsymbol{b} \in \mathbb{R}^N$ is the bias vector and $\boldsymbol{y} \in \mathbb{R}^N$ is the output vector. For a more specific expression with indices

$$\boldsymbol{y}(j) = \sum_{i=1}^{M} \boldsymbol{W}(j, i)\boldsymbol{x}(i) + \boldsymbol{b}(j). \tag{8}$$

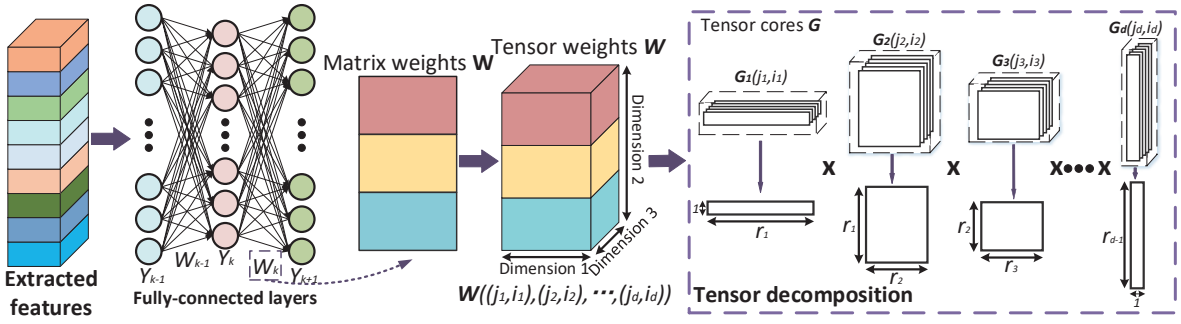

Fig. 5. Tensorizing weights of fully-connected layers for parameter compression. The center 3-dimensional tensor is conceptual and can be d-dimensional.

A tensorized model implements Eq. (8) with much fewer parameters by approximating $\boldsymbol{W}\boldsymbol{x}$. To do so, the weight matrix $\boldsymbol{W}$ is cast into a tensor

$$reshape(\boldsymbol{W}) = \boldsymbol{\mathcal{W}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d \times m_1 \times m_2 \times \cdots \times m_d}, \quad (9)$$

where $N = \Pi_{k=1}^{d} n_k$ and $M = \Pi_{k=1}^{d} m_k$. Following the previously discussed tensor decomposition in Eq. (6), the expression of the tensorized weight can be rewritten as

$$\begin{aligned} \boldsymbol{W}(j,i) \xrightarrow{reshape} & \boldsymbol{\mathcal{W}}((j_1,i_1),(j_2,i_2),\cdots,(j_d,i_d)) \\ = & \boldsymbol{\mathcal{G}}_1^t(j_1,i_1)\boldsymbol{\mathcal{G}}_2^t(j_2,i_2)\ldots\boldsymbol{\mathcal{G}}_d^t(j_d,i_d). \end{aligned} \quad (10)$$

Similarly, we can reshape the tensors $\boldsymbol{x} \in \mathbb{R}^M$, $\boldsymbol{b} \in \mathbb{R}^N$ into d-dimensional, namely, $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_d}$, $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$. As a result, the $\boldsymbol{y} \in \mathbb{R}^N$ also becomes a d-dimensional tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$. Therefore, we can reformulate Eq. (8) all by tensors

$$\begin{aligned} &\boldsymbol{\mathcal{Y}}(j_1,j_2,\ldots,j_d) = \\ &\sum_{i_1=1}^{m_1}\sum_{i_2=1}^{m_2}\cdots\sum_{i_d=1}^{m_d}[\boldsymbol{\mathcal{G}}_1^t(j_1,i_1)\boldsymbol{\mathcal{G}}_2^t(j_2,i_2)\ldots\boldsymbol{\mathcal{G}}_d^t(j_d,i_d) \\ &\boldsymbol{\mathcal{X}}(i_1,i_2,\ldots,i_d)] + \boldsymbol{\mathcal{B}}(j_1,j_2,\ldots,j_d). \end{aligned} \quad (11)$$

This way, the computational complexity becomes $O(dr^2 \max_k(l_k))$ instead of $O((\max_k(l_k))^d)$ [19], where r is the maximum rank of cores $\boldsymbol{\mathcal{G}}_k$ (r is small and set to be 4 in our setup). This results in generally high compression ratios and much higher efficiency than doing the direct matrix-vector multiplication in traditional fully-connected layers.

The tensorization of the hidden-to-hidden weights is depicted in Fig. 5. The inputs to the T-LSTM are tensorized time-series features from feature extractor. In the input-to-hidden and hidden-to-hidden layers, the 2-dimensional weights are reshaped into d-dimensional tensors and decomposed into tensor cores $\boldsymbol{\mathcal{G}}_1, \boldsymbol{\mathcal{G}}_2, \cdots, \boldsymbol{\mathcal{G}}_d$. We remark that these cores are relatively small due to the small r, which brings about a high reduction in the number of network parameters.

C. Trained Quantization of DEEPEYE

The direct implementation of full-precision networks for video-scale data requires unnecessarily large software and hardware resources. Here we present an 8-bit[2] quantization strategy on the whole framework for further compression. According to the recent work [20], we present the trained quantization by low-bitwidth convolution with 8-bit-quantized weights that result in high hardware efficiency with only a slight drop in accuracy. Assuming $\boldsymbol{w_k} \in [-1,0) \cup (0,1]$ are the full-precision weights and $\boldsymbol{w_k^q} \in [-2^7+1,0) \cup (0,2^7-1]$ are the 8-bit quantized-valued weights in the k-th convolution layer, and they have the approximation $\boldsymbol{w_k} \approx \xi \cdot \boldsymbol{w_k^q}$ with a non-negative scaling factor ξ. The weight element $\boldsymbol{w_k}$ is quantized into 8-bit as

$$\boldsymbol{w_k^q} = \begin{cases} \frac{\boldsymbol{w_k}}{|\boldsymbol{w_k}|}, & 0 < |\boldsymbol{w_k}| \le \frac{1}{2^7}, \\ INT(2^7 \times \boldsymbol{w_k}), & \frac{1}{2^7} < |\boldsymbol{w_k}| < 1, \\ (2^7-1)\frac{\boldsymbol{w_k}}{|\boldsymbol{w_k}|}, & |\boldsymbol{w_k}| = 1, \end{cases} \quad (12)$$

where the function INT takes the smaller nearest integer. We further develop an activation with quantization which quantizes a real number feature $\boldsymbol{x_k} \in [0,1]$ into an 8-bit feature $\boldsymbol{x_k^q} \in [0,1]$, namely,

$$\boldsymbol{x_k^q} = \frac{1}{2^8} \times \begin{cases} INT(2^8 \times \boldsymbol{x_k}), & 0 \le \boldsymbol{x_k} < 1, \\ 2^8 - 1, & \boldsymbol{x_k} = 1. \end{cases} \quad (13)$$

Having both the 8-bit weights and features, the quantized convolution proceeds as

$$\boldsymbol{s_k^q}(h_1,h_2,h_3) = \sum_{i=1}^{D_k}\sum_{j=1}^{H_k}\sum_{l=1}^{V_k} \boldsymbol{w_k^q}(i,j,l,h_3) \times \boldsymbol{x_{k-1}^q}(i+h_1-1, j+h_2-1, l), \quad (14)$$

where $\boldsymbol{w_k^q} \in \mathbb{R}^{d_k \times u_k \times V_{k-1} \times V_k}$, $\boldsymbol{x_{k-1}^q} \in \mathbb{R}^{D_{k-1} \times U_{k-1} \times V_{k-1}}$ are the 8-bit weights and features, respectively. Here, D_k, U_k and V_k are mode sizes of the feature maps and d_k, u_k and v_k are those of the weight kernels. Since both the weights and features are computed and stored in an 8-bit format during training, the processor and memory resources required for the whole framework are substantially reduced.

IV. Deeply Compressed DEEPEYE at Terminal Device

A. Overall Workflow

DEEPEYE integrates the 8-bit-quantized feature extractor for real-time object detection and the T-LSTM which takes tensorized time-series features aggregated from feature extractor for action recognition. The workflow of DEEPEYE is shown in Fig. 6. First, the video clip is fed into the

[2]We determine the best quantization bitwidth by testing several realizations from 4-bit to 10-bit.

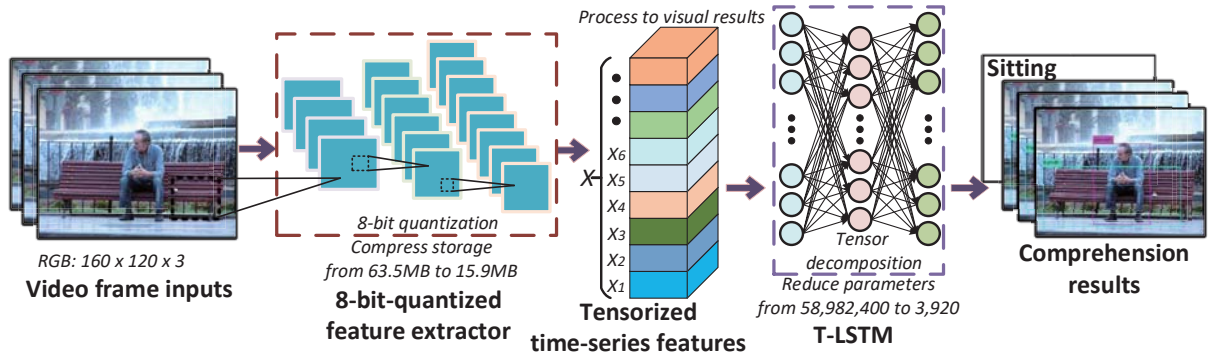

Fig. 6. DEEPEYE: an embedded video comprehension system integrating 8-bit-quantized feature extractor and T-LSTM.

8-bit-quantized feature extractor as inputs. Then, the time-series feature outputs of 8-bit-quantized feature extractor are stacked into a tensor and fed into the T-LSTM. We note in passing that the tensorized time-series features are coming from the last convolution layer (CONV-final) of the CNN-based feature extractor, which can also be processed to display real-time visual results. Finally, after the T-LSTM processing with tensorized-compression on both tensorial input-to-hidden and hidden-to-hidden mappings, one obtains the classification result towards video action recognition.

In short, DEEPEYE represents an imitation of the human physiology: recognizing the object and understanding the scene. DEEPEYE is highly compressed through tensorization and quantization, and enjoys remarkable speedup as well as saving in resources. Referring to the network in Fig. 6 with typical settings, the storage cost is compressed from 63.5MB to 15.9MB and the number of parameters is reduced from 58982400 to 3920, namely, a remarkable 15k$\times$ reduction.

B. Terminal Device Implementation

Remarkable speedup, energy-efficiency and saving in resources can be obtained through the tensorization and quantization, which make DEEPEYE become lightweight and hardware-friendly. To implement the proposed DEEPEYE framework at terminal devices, we use an ARM-core board of 6GB RAM memory and 64 GB ROM storage, which contains a cluster of four ARM Cortex-A73 cores with 2.36 GHz working frequency and four ARM Cortex-A53 cores with 1.80 GHz working frequency, whose architecture is shown in Fig. 7. These cores are organized in a big.LITTLE configuration [21]. This ARM-core board is able to support several operating systems such as Ubuntu, Debian and Android.

In practice, firstly, we burn the Ubuntu 16.04 64-bit firmware to the board, which acts as our operating system. Secondly, we cross-compile the necessary libraries such as Python, Tensorflow and OpenCV. Thirdly, we solve the incompatibility between these libraries on Ubuntu. After configuring the running environment, we realize DEEPEYE using Python codes. Finally, we run the real-time video comprehension on the ARM-core based IOT board.

V. Experiments

We now evaluate DEEPEYE for its video object detection and action recognition performance on both GPU and ARM-

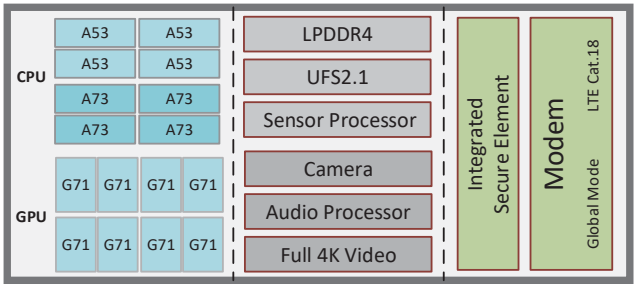

Fig. 7. Hardware architecture of the proposed ARM-core board.

core board. Specifically, we adopt tiny-YOLOv2 [22] for our feature extractor and the 8-bit-quantized tiny-YOLOv2 is called Q-YOLO. The reasons of using YOLO as the feature extractor are twofold: **1)** YOLO trains on full images and directly optimizes object detection performance. **2)** YOLO obtains excellent features with its powerful feature extraction ability, while achieving excellent accuracy and speed.

First, we report the accuracy and performance results of DEEPEYE on GPU. Our experimental setup employs Theano for coding and NVIDIA GTX-1080Ti for hardware realization. We validate DEEPEYE by presenting a comparison study on object dataset VOC [23] and two challenging video datasets, namely, MOMENTS [24] and UCF11 [6].

A. Comparison on Object Detection

To verify the performance on video object detection, firstly, we choose VOC dataset which contains 11530 images with 27450 annotated objects and 6929 segmentations, falling into 20 classes such as *person*, *car* or *tvmonitor*. Then we apply the large-scale video dataset MOMENTS that contains one million labeled 3-second video clips, involving people, animals, objects and natural phenomena, that capture the gist of a dynamic scene. Each clip is assigned with 339 action classes such as eating, bathing or attacking. Based on the majority of the clips we resize every frame to the standard size 340×256 at an FPS of 25. On account of the huge size of MOMENTS, we choose 10 representative classes for our experiments and the length of training sequences is set to be 80 while that of the test sequences is set to be 20.

1) Accuracy Comparison on Classes: Here we report the AP comparison between the proposed 8-bit-quantized model

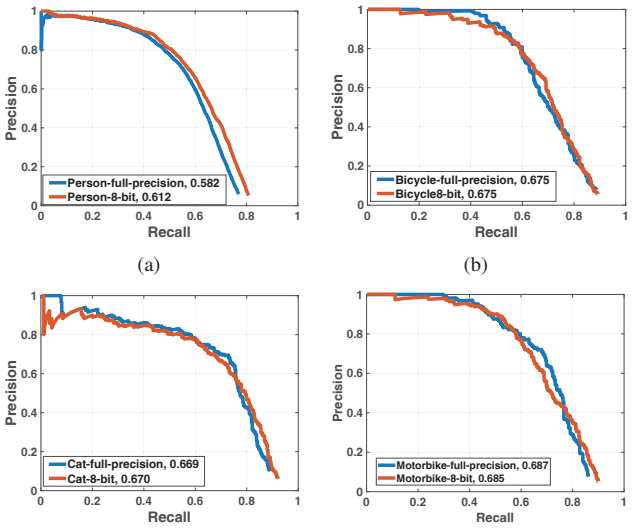

Fig. 8. Average precision comparison between 8-bit Q-YOLO and full-precision tiny-YOLOv2 on classes: (a) person, (b) bicycle, (c) cat and (d) motorbike.

and full-precision model on several representative classes in VOC, as shown in Fig. 8. We observe that the 8-bit Q-YOLO does not cause the Average Precision (AP) curves to deviate significantly different from their full-precision counterparts. Moreover, the mean Average Precision (mAP) among all 20 classes reaches 0.5350 in the 8-bit Q-YOLO, which causes only 0.47% decrease compared with the full-precision one (0.5397).

2) Visual Results: Sample results of the object detection tasks on MOMENTS are shown in Fig. 9. Experiments show that all existing objects in these video clips can be detected precisely in real time. In DEEPEYE, the tensorized features of each frame is in a size of $19 \times 19 \times 425$, which is fed into the T-LSTM for action recognition without delay.

B. Comparison on Action Recognition

Here we use UCF11 dataset for a performance comparison on action recognition. The dataset contains 1.6k video clips, falling into 11 action classes that summarize the human action visible in each clip such as *biking*, *diving* or *walking*. We resize the RGB frames into 160×120 at an FPS of 24. In T-LSTM, to determine the best tensor rank setting r_k, we set all r_k's to be the same and juggle it between 3 to 6 in experiments. The experimental results of variations in accuracy and total number of parameters shows that $r_k = 4$ strikes the optimal balance between accuracy and compression.

1) Accuracy with Different Models: We sample all frames of each video clip as the input data. The tensorization-based algorithm has been configured for both inputs and weights by the training process. Fig. 10 shows the training loss and top-1 accuracy comparison among: **1)** original LSTM, **2)** T-LSTM without Q-YOLO and **3)** DEEPEYE (Q-YOLO with T-LSTM). In practice, we set the parameters as follows: the tensor dimension is $d = 4$; the input shapes as: **1)** and **3)** $m_1 = 17, m_2 = 19, m_3 = 19, m_4 = 25$, **2)** $m_1 = 8, m_2 = 20, m_3 = 20, m_4 = 18$; the hidden shapes are: $n_1 = n_2 = n_3 = n_4 = 4$; and the ranks of T-LSTM are: $r_1 = r_5 = 1$, $r_2 = r_3 = r_4 = 4$.

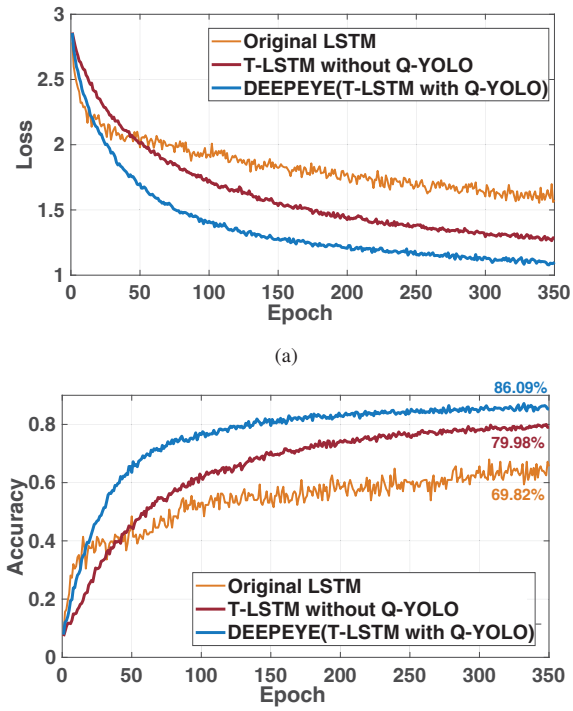

Fig. 10. Comparison among different LSTM models on (a) training loss curves and (b) top-1 accuracy curves.

TABLE I
THE TOP-1 ACCURACY COMPARISON OF ACTION RECOGNITION BETWEEN DEEPEYE AND STATE-OF-THE-ART APPROACHES ON THE UCF11 DATASET.

Initial approach	Bag-of-words method [6]	71.20%
Two-stream CNN	Simonyan et al. [8]	73.25%
RNN	Liu et al. [3]	76.06%
	Hasan et al. [4]	69.00%
	Sharma et al. [5]	84.96%
CNN+RNN	Ullah et al. [9]	79.67%
	Donahue et al. [10]	77.46%
3D-CNN	Tran et al. [11]	80.51%
	Chen et al. [12]	82.30%
Tensorized LSTM	Yang et al. [17]	81.30%
	He et al. [16]	79.29%
Our experiments	Original LSTM	69.82%
	Original LSTM with original YOLO	76.36%
	T-LSTM without Q-YOLO	79.98%
	DEEPEYE (Q-YOLO + T-LSTM)	**86.09%**

2) Peak Accuracy Comparison: Table I shows the state-of-the-art results of action recognition on the UCF11 dataset, and it can be seen that DEEPEYE outperforms all other approaches. The peak top-1 accuracy of DEEPEYE reaches 86.09%, 16.27% higher than the initial bag-of-words approach [6], 12.84% higher than the two-stream CNN based approach [8], 10.03% higher than the RNN-based approach [3], 6.42% higher than the traditional CNN+RNN based approach [9] with raw feature inputs, 3.79% higher than the 3D CNN-based approach [12] while 6.80% higher than the tensorized

Fig. 9. Sample visual results of DEEPEYE on the MOMENTS dataset.

LSTM-based approach [16] with direct video frame inputs. The comparison results demonstrate the *unique* benefit of the proposed DEEPEYE framework arises from the use of tensorized-compression.

C. Performance Analysis

Besides the outstanding function and accuracy of DEEPEYE, the compression ratio and speed are also remarkable. Table II shows the object detection performance comparison between DEEPEYE and state-of-the-arts. Note that all the related YOLO results are measured on a GTX-1080Ti. In particular, DEEPEYE exhibits major improvements in the speed of training and inference, and requires $3.99\times$ less storage compared with the full-precision tiny-YOLOv2 combined with the original LSTM.

The action recognition performance evaluation is shown in Table III based on different baselines. Among all schemes, DEEPEYE has the best performance with excellent accuracy even with several orders fewer parameters and $228.10\times$ less storage resources compared with the original LSTM. Furthermore, it achieves a remarkable FPS performance over state-of-the-art approaches and $2.87\times$ speedup compared with the original LSTM. With its low complexity, high throughput and high speed, DEEPEYE provides a turnkey solution for terminal devices where low-power and accurate video comprehension is required.

TABLE II
DETECTION PERFORMANCE ON MULTIPLE BASELINES VIA FLOPS (FLOATING-POINT OPERATIONS PER SECOND) AND FPS (FRAMES PER SECOND).

Framework	FLOPS	Storage size	Compression	FPS
Fast R-CNN [25]	58.90Billion	N/A	N/A	0.5
Faster R-CNN [26]	44.36Billion	N/A	N/A	5
SSD300 [27]	25.62Billion	N/A	N/A	61
SSD500 [27]	31.75Billion	N/A	N/A	46
YOLOv2	62.94Billion	194.3MB	N/A	28
Tiny-YOLOv2	5.41Billion	63.5MB	N/A	168
Q-YOLO	**4.88Billion**	**15.9MB**	**3.99**	**193**
DEEPEYE	**19.34Billion**	**18.8MB**	**3.38**	**108**

TABLE III
THE PERFORMANCE COMPARISON ON ACTION RECOGNITION BETWEEN DEEPEYE AND STATE-OF-THE-ARTS.

Approach	Storage size	Compression	Runtime	FPS
Sharma et al. [5]	616.3MB	N/A	26,011s	2
Kantorov et al. [28]	N/A	N/A	1,084s	48
Liu et al. [3]	586.8MB	N/A	865s	24
LSTM	661.5MB	N/A	1,369s	38
T-LSTM	**2.9MB**	**228.10**	**431s**	**118**
DEEPEYE	**18.8MB**	**35.19**	**477s**	**108**

D. Evaluation on IOT Board

After the evaluation on GPU, we further report the experiment results of DEEPEYE on the ARM-core based IOT board. The test verification platform is shown in Fig. 11, which

comprises of an ARM-core board, a DC regulated power supply and a computer monitor. From the functional point of view, the DEEPEYE shows virtually identical results on the GPU and ARM. The whole storage size of DEEPEYE on board is 18.8M, which occupies only 0.03% of all 64GB ROM storage. When the DEEPEYE is running, the memory consumption is up to 700M, consuming only 12% of all 6GB RAM memory. Furthermore, the average power consumption of the proposed framework is measured to be 2.4W.

Fig. 11. Verification platform for DEEPEYE on ARM-core board.

VI. Conclusion

This paper has proposed a compact yet accurate DNN video comprehension method at terminal devices. A LSTM spatio-temporal model is built with tensorized time-series features, called DEEPEYE, for object detection and action recognition. The DEEPEYE is deeply compressed through tensorizaiton and decomposition, and is further compressed with a trained quantization in 8-bit. Experiments using large-size MOMENTS and UCF11 benchmarks have shown that DEEPEYE can achieve a $228.1\times$ compression with only 0.47% mAP decrease, as well as $15k\times$ parameter reduction and up to 16.27% accuracy improvement with only 2.4W power. Experiments have shown that DEEPEYE outperforms state-of-the-art networks, and demonstrates the best storage and accuracy performance on both GPU and IOT devices.

References

[1] Y. Li, Z. Liu, W. Liu, Y. Jiang, Y. Wang, W. L. Goh, H. Yu, and F. Ren, "A 34-fps 698-gop/s/w binarized deep neural network-based natural scene text interpretation accelerator for mobile edge computing," *IEEE Transactions on Industrial Electronics*, 2018.

[2] G. Zhong, A. Dubey, C. Tan, and T. Mitra, "Synergy: An hw/sw framework for high throughput cnns on embedded heterogeneous soc," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 2, p. 13, 2019.

[3] D. Liu, M.-L. Shyu, and G. Zhao, "Spatial-temporal motion information integration for action detection and recognition in non-static background," in *Proceedings of the IEEE Conference on Information Reuse & Integration*, 2013, pp. 626–633.

[4] M. Hasan and A. K. Roy-Chowdhury, "Incremental activity modeling and recognition in streaming videos," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 796–803.

[5] S. Sharma, R. Kiros, and R. Salakhutdinov, "Action recognition using visual attention," *arXiv preprint arXiv:1511.04119*, 2015.

[6] J. Liu, J. Luo, and M. Shah, "Recognizing realistic actions from videos "in the wild"," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 1996–2003.

[7] L. Fan, W. Huang, C. Gan, S. Ermon, B. Gong, and J. Huang, "End-to-end learning of motion representation for video understanding," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6016–6025.

[8] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in Neural Information Processing Systems*, 2014, pp. 568–576.

[9] A. Ullah, J. Ahmad, K. Muhammad, M. Sajjad, and S. W. Baik, "Action recognition in video sequences using deep bi-directional lstm with cnn features," *IEEE Access*, vol. 6, pp. 1155–1166, 2018.

[10] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 2625–2634.

[11] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *IEEE International Conference on Computer Vision*, 2015, pp. 4489–4497.

[12] Y. Chen, Y. Kalantidis, J. Li, S. Yan, and J. Feng, "Multi-fiber networks for video recognition," in *European Conference on Computer Vision*, 2018, pp. 352–367.

[13] M. Denil, B. Shakibi, L. Dinh, N. De Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[14] S. Zhe, K. Zhang, P. Wang, K.-c. Lee, Z. Xu, Y. Qi, and Z. Ghahramani, "Distributed flexible nonlinear tensor factorization," in *Advances in Neural Information Processing Systems*, 2016, pp. 928–936.

[15] ——, "Distributed flexible nonlinear tensor factorization," in *Advances in Neural Information Processing Systems*, 2016, pp. 928–936.

[16] Z. He, S. Gao, L. Xiao, D. Liu, H. He, and D. Barber, "Wider and deeper, cheaper and faster: Tensorized lstms for sequence learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 1–11.

[17] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," *arXiv preprint arXiv:1707.01786*, 2017.

[18] R. Yu, S. Zheng, A. Anandkumar, and Y. Yue, "Long-term forecasting using tensor-train rnns," *arXiv preprint arXiv:1711.00073*, 2017.

[19] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 442–450.

[20] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[21] S. Kamdar and N. Kamdar, "big. little architecture: Heterogeneous multicore processing," *International Journal of Computer Applications*, vol. 119, no. 1, 2015.

[22] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2017.

[23] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010.

[24] M. Monfort, B. Zhou, S. A. Bargal, A. Andonian, T. Yan, K. Ramakrishnan, L. Brown, Q. Fan, D. Gutfruend, C. Vondrick *et al.*, "Moments in time dataset: one million videos for event understanding," *arXiv preprint arXiv:1801.03150*, 2018.

[25] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE Conference on Computer Vision*, 2015, pp. 1440–1448.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision*, 2016, pp. 770–778.

[27] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European Conference on Computer Vision*, 2016, pp. 21–37.

[28] V. Kantorov and I. Laptev, "Efficient feature extraction, encoding and classification for action recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 2593–2600.