Neural Computing and Applications (2022) 34:19937–19952
https://doi.org/10.1007/s00521-022-07538-1

ORIGINAL ARTICLE

An efficient hardware implementation of CNN-based object trackers for real-time applications

Al-Hussein A. El-Shafie[1] · Mohamed Zaki[2] · S. E. D. Habib[3]

Received: 20 September 2021 / Accepted: 13 June 2022 / Published online: 12 July 2022

Abstract
The object tracking field continues to evolve as an important application of computer vision. Real-time performance is typically required in most applications of object tracking. The recent introduction of Convolutional Neural network (CNN) techniques to the object tracking field enabled the attainment of significant performance gains. However, the heavy computational load required for CNNs conflicts with the real-time requirements required for object tracking. In this paper, we address these computational limitations on the algorithm-side and the circuit-side. On the algorithm side, we adopt interpolation schemes which can significantly reduce the processing time and the memory storage requirements. We also evaluate the approximation of the hardware-expensive computations to attain an efficient hardware design. Moreover, we modify the online-training scheme in order to achieve a constant processing time across all video frames. On the circuit side, we developed a hardware accelerator of the online training stage. We avoid transposed reading from the external memory to speed-up the data movement with no performance degradation. Our proposed hardware accelerator achieves 44 frames-per-second in training the fully connected layers.

Keywords Object tracking · CNN · Online training · Deep-feature interpolation · Hardware accelerator

1 Introduction

The object tracking field has wide application domains which are ever increasing like surveillance systems, intelligent robotics, unmanned vehicles and virtual reality. The dramatic increase in the computational power has opened the door for the creation of new tracking algorithms to overcome the tracking field challenges [1].

Traditionally, tracking algorithms employed hand-crafted features like pixel intensity, color and Histogram of Oriented Gradients (HOG) [2] to represent the target. However, hand-crafted features are not robust against severe appearance variations [3]. Deep features obtained from Convolutional Neural Networks (CNNs) achieved better performance than that obtained by hand-crafted features on recent tracking benchmarks. The adoption of CNNs in the tracking field can be classified into three categories: First, CNNs are exploited to train the regression models of the Discriminative Correlation Filters (DCF)-based trackers as in [4–6]. Second, a Siamese structure [7–9] is adopted where the two identical CNN branches are used to measure the similarity of two input patches. Third, fully connected layers are added after the convolutional layers to score the input patches and classify them into object or background as in [3, 10–12]. All the three categories have achieved state-of-the-art performance on the object tracking benchmarks. We focus on the third category in this paper.

Generally, real-time processing and small-form factor are desirable in most of the tracking applications. Hence,

✉ S. E. D. Habib
serag@eng.cu.edu.eg

Al-Hussein A. El-Shafie
elshafie_a@yahoo.com

Mohamed Zaki
mzaki.azhar@gmail.com

[1] Electronics and Communication Engineering Department, Faculty of Engineering, Cairo University, Giza, Egypt

[2] Computer Engineering Department, Faulty of Engineering, Al-Azhar University, Cairo, Egypt

[3] Electronics and Communication Engineering Department, Faculty of Engineering, Cairo University, Giza, Egypt

embedded platforms are the typical choice for implementing the tracking algorithms [1]. The implementation of the CNN-based object trackers on embedded platforms, however, confronts several challenges because of the slow speed, large memory requirements and the online training computations. In general, CNN training requires at least 5 × more multiply and accumulate (MAC) operations than the inference phase [13] with much larger amount of external memory accesses. For an efficient embedded implementation, the limitations of the CNN-based trackers need to be tackled while keeping acceptable performance levels.

Initial results of this work were published in [12, 14]. In [12], we outlined the overall architecture and presented a Matlab model of our proposed object tracker named Interpolation and Localization Network (ILNET). Reference [14] presented an initial HW design of ILNET where we showed the benefits of applying interpolation schemes and word length exploration of the different variables to cut-down the required memory size and speed-up the processing time. We presented also a novel technique for the error back-propagation that avoids the transposed reading from the external memory.

In this paper, we sum-up the full development of our proposed object tracker, ILNET. The main objective of ILNET is to enhance the speed of CNN-based object trackers. The following new contributions are introduced in this paper:Computationbreakdown in fully

- We modify the online training scheme where we spread the required computations across all the frames instead of at every specific frame intervals, and hence, a steady frames-per-second (fps) throughput can be achieved for the whole system.
- Additional algorithm-level enhancements are explored aiming at a hardware-efficient implementation, like dispensing the hard-negative mining step, approximating the softmax function and adopting a fixed dropout pattern.
- A detailed description of our online-training hardware accelerator is provided including a detailed computation flow analysis, reduction in the external memory latency and top-level implementation details.
- Detailed comparison with a recent published tracker [13] and a speed analysis of the whole tracker implementation on embedded systems.
- This paper is organized as follows: Sect. 2 gives an overview of our proposed object tracker and the interpolation schemes employed. In Sect. 3, we evaluate additional algorithm-level features to simplify the hardware implementation. Section 4 describes our proposed hardware accelerator for the online training. Experimental results and speed analysis for the complete tracker system are presented in Sect. 4 and finally, section VI concludes our work.

2 Overview of ILNET tracker

ILNET is based on the MDNET tracker [3], while it is modified to be suitable for embedded systems. The MDNET network consists of three convolutional layers, conv1:3, followed by three fully connected layers fc4:6 to classify the tested patches into object or background.

Generally, the required processing can be divided into three phases: tracking phase, training phase and network update. In the MDNET tracking phase, candidate patches are generated with Gaussian distribution around the object location in the previous frame as shown in Fig. 2a. All these candidate patches are forwarded through the whole network and classified into object or background. The new predicted location can then be obtained by averaging the location of the candidate patches of the highest object score. In the training phase, training patches are generated around the new predicted location and the feature maps are extracted and stored every frame. The network update is carried out either at fixed frame intervals, which is the long-term update, or when the object score drops severely during the tracking, which is the short-term update as defined in [3].

2.1 ILNET network structure

In ILNET, the network is supplemented by fully connected localization layers, fc7:9, in order to classify also the object location inside the patch into five positions: up, down, right, left and middle. The input of fc7 is the conv3 output, the same input for fc4. Figure 1 shows the network structure of ILNET.

The computation in the convolutional layers is the dominant part in the tracker operation. Therefore, the main purpose of our tracker is to reduce the number of the convolutional computations in the network and reuse the feature maps in the tracking phase and the training phase. In ILNET, the whole Region of Interest (ROI) is forwarded

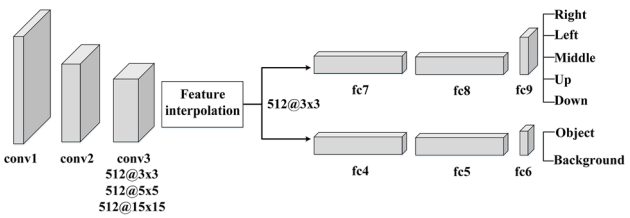

Fig. 1 ILNET network structure. Convolutional layers, conv1:3, Classification layers, fc4:6. Localization layers, fc7:9 [12]

to the network instead of small random patches and a $15 \times 15 \times 512$ feature map is obtained. The object is represented by $3 \times 3 \times 512$ inside the ROI feature map. Accordingly, a ROI feature map of size $15 \times 15 \times 512$ represents 169 fixed-spaced candidates, each with feature map of size $3 \times 3 \times 512$.

2.2 ILNET tracking phase

A coarse–fine localization scheme is adopted for the ILNET tracking phase. Figure 2b-f shows the main steps of the coarse localization. Figure 2b shows the ROI which is equivalent to fixed-spaced samples in Fig. 2c. For simplicity, nine fixed-spaced samples are shown. The steps of the coarse localization can be summarized as follows for this example:

- Classify all candidates into object or background.
- Discard the background candidates as in Fig. 2d.
- Move the object candidates using the localization layers as shown in Fig. 2e. The dominant move of candidate two, five and six are down, right and left, respectively, in this example.

Obtain a coarse predicted location which is the candidate with the maximum overlap with the other candidates as in Fig. 2f.

For the fine localization, fine samples are scored around the coarse locations. ILNET exploits interpolation schemes to approximate the feature maps of the fine samples. As the object is represented by a $3 \times 3 \times 512$ feature map, and if we extract feature maps for a region larger than the target size (e.g., $5 \times 5 \times 512$ feature maps), we would have nine 3×3 grid in total. Each 3×3 grid is displaced by dx and/or dy from its neighbors. The value of dx and dy depends on the network structure. Accordingly, we can obtain the feature maps of all image patches which have displacements ranging from 0 to dx or dy measured from the center by bilinear interpolation without forwarding the image patches through the convolutional layers. ILENT employs bilinear interpolation and linear interpolation to approximate the feature maps in the translation domain and the scale domain, respectively. We only need to extract two additional feature maps at scale-up and scale-down because we already generate a larger-sized feature map ($15 \times 15 \times 512$) at the normal scale in the coarse localization step. The fine localization steps are illustrated in Fig. 3 and summarized as follows:

Extract two additional feature maps of size $5 \times 5 \times 512$ around the coarse location at scale-up and scale-down.

Approximate the feature maps of the fine samples in the translation domain by adopting bilinear interpolation.

Approximate the final feature map in the scale domain by adopting linear interpolation.

Attain the fine location by averaging the location of the highest score samples.

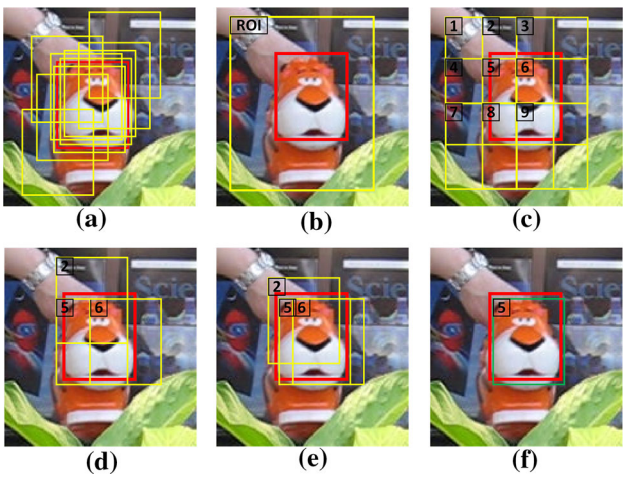

Fig. 2 Random samples and fixed-spaced samples: **a** MDNET random samples, **b-f** ILNET ROI and coarse localization steps

2.3 ILNET training phase

ILNET reuses the ROI feature maps obtained in the tracking phase to approximate the feature maps of the training patches by applying bilinear interpolation as well. Accordingly, a significant reduction in the processing time is achieved because the computation in the convolutional layers is dispensed in the training phase. A speed improvement of 8.8 × is obtained while achieving a comparable performance with the baseline tracker for all the tracking benchmark challenges [15].

2.4 ILNET algorithm and performance

Our overall tracking algorithm is presented in Algorithm 1. The convolutional layer weights, $[W_1{:}W_3]$, are initialized by the VGG-M [16] network model pre-trained on the ImageNet dataset, while the weights of fully connected layers, $[W_4{:}W_6]$ and $[W_7{:}W_9]$, are randomly initialized.

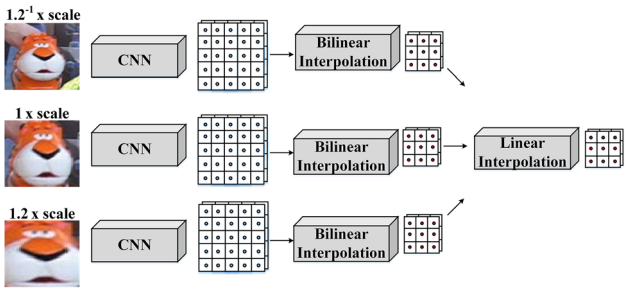

Fig. 3 ILNET fine localization step [12]

Hence, our tracker is not offline-trained on any video dataset. At frame i, $fmap_{1,i}$ is a conv3 feature map of size 15 × 15 × 512, while $fmap_{2,i}$ and $fmap_{3,i}$ are feature maps of size 5 × 5 × 512 at scale-up and scale-down, respectively.

Algorithm 1 ILNET tracking algorithm

Input : Network weights ($[W_1{:}W_3]$ initialized by VGG-M, $[W_4{:}W_6]$ & $[W_7{:}W_9]$ randomly initialized), initial object location z_1

Output: Estimated object location z_i

Obtain $fmap_{1,1}$ of size 15x15x512 around z_1
Obtain $fmap_{2,1}$ and $fmap_{3,1}$ of size 5x5x512 at scale up and scale down respectively around z_1
Train $[W_4{:}W_9]$ via interpolating $fmap_{1,1}, fmap_{2,1}, fmap_{3,1}$
Store $fmap_{1,1}, fmap_{2,1}, fmap_{3,1} \rightarrow U$
repeat
  Obtain $fmap_{1,i}$ of size 15x15x512 around z_i
  Estimate z_{coarse} using the localization layers
  Obtain $fmap_{2,i}$ and $fmap_{3,i}$ of size 5x5x512 at scale up and scale down round z_{coarse}
  Get z_{fine} by interpolating $fmap_{1,i}$, $fmap_{2,i}$, $fmap_{3,i}$
  if score(z_{fine}) > 0.5 **then**
    Obtain $fmap_{2,i}$ and $fmap_{3,i}$ of size 5x5x512 at scale up and scale down around z_{fine}
    Store $fmap_{1,i}$, $fmap_{2,i}$, $fmap_{3,i} \rightarrow U$
    if i mod 10 = 0 **then**
      Update $[W_4{:}W_9]$ via interpolating *fmaps* from U
  else then
    Update $[W_4{:}W_9]$ via interpolating *fmaps* from U
until end of sequence

Table 1 shows the breakdown of the processing time for ILNET and MDNET_N running on an Intel i7-3520 M CPU system. MDNET_N is the same as MDNET [3] but without offline training and bounding box regression for fair comparison with our tracker. Table 2 shows the success Area Under the Curve (AUC) and the precision of the Object Tracking Benchmark (OTB-100) [15]. Figure 4 shows illustrative ex from OTB-100 for MDNET_N and ILNET.

Table 1 Average computation time in seconds per frame*

	MDNET-N	ILNET	Speed-up factor
Tracking phase	3.4	0.36	9.4 ×
Training phase	3.3	0.21	15.7 ×
Network update	2.3	2.3	1 ×
First frame training	90	52	1.72 ×
Frame processing without first frame	7	0.8	8.8 ×

*Running on an Intel i7-3520 M CPU system

Table 2 performance Comparison of ilnet and mdnet_n

Tracker	Success AUC	Precision
MDNET_N	0.618	0.867
ILNET	0.622	0.863

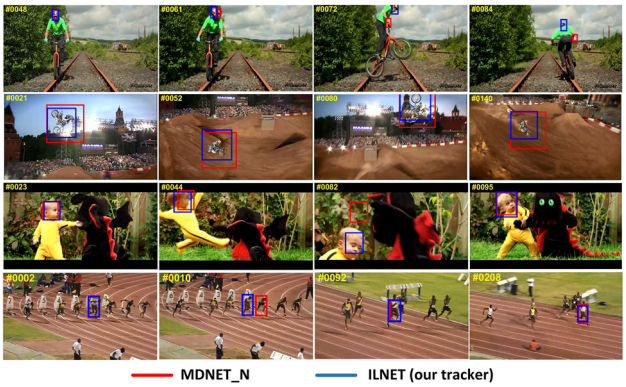

Fig. 4 Illustrative tracking examples from OTB-100 (Biker, Motor-Rolling, DragonBaby, Bolt2)

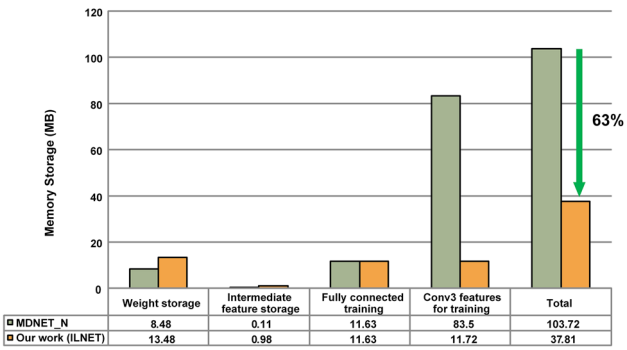

Fig. 5 Memory storage comparison between MDNET and ILNET

Moreover, our proposed tracker achieves a significant reduction in the memory storage requirement compared to the baseline tracker. Figure 5 shows the breakdown of the required memory storage. The memory requirement is divided into weight storage, intermediate feature storage, fully connected training and conv3 deep features which is used in the training. 16-bit fixed-point representation is assumed for all parameters. The localization layers used in ILNET adds an extra 5 MB for the weight storage. In addition, the storage of the intermediate feature is slightly increased as well because we have increased the size of the input patches. However, this increase in the storage of the weights and the intermediate feature maps for ILNET is insignificant to the achieved reduction in the storage of the conv3 feature maps.

The long-term and short-term network updates in both MDNET and ILNET are 100 and 20 frames, respectively. The required memory storage for conv3 feature maps in MDNET is given by Eqs. (1) and (2), where *fmap_sz* and *fmap_num* are the size and number of the feature maps, respectively. The number of the feature maps of the positive (object) class is 500 from the first frame and 50 each frame in the long-term. The number of the feature maps of the negative (background) class is 200 each frame in the short-term. For our tracker ILNET, conv3 feature maps are not stored for all positive and negative samples each frame. As we exploit interpolation schemes, we can obtain and store a single and larger-size feature map each frame which can be reused to obtain the required feature maps when the training process needs it. For positive training, three feature maps of size $5 \times 5 \times 512$ at three scales are stored each frame in the long-term. For negative training, a single $15 \times 15 \times 512$ feature map is stored for each frame in the short-term. The required memory storage for conv3 feature maps in ILNET is given by Eqs. (3) and (4), where *pfmap_sz*, *pfmap_num*, *nfmap_sz* and *nfmap_num* are the size and number of the feature maps required for the positive and negative training, respectively. As shown in Fig. 5, our proposed tracker achieves a total memory reduction of 63%.

$$\mathrm{MDNET}_{\mathrm{conv3}} = \mathrm{fmap\_sz} \times \mathrm{fmap\_num} \tag{1}$$

$$\begin{aligned}&\mathrm{MDNET}_{\mathrm{conv3}}\\ &= \underbrace{3 \times 3 \times 512 \times 2}_{\mathrm{fmap\_sz}} \times \left(\underbrace{500}_{\mathrm{initial}} + \underbrace{50}_{\mathrm{pos}} \times \underbrace{100}_{\mathrm{long}} + \underbrace{200}_{\mathrm{neg}} \times \underbrace{20}_{\mathrm{short}} \right)\\ &= 83.5\ \mathrm{MB}\end{aligned} \tag{2}$$

$$\begin{aligned}\mathrm{ILNET}_{\mathrm{conv3}} &= \mathrm{pfmap\_sz} \times \mathrm{pfmap\_num}\\ &\quad + \mathrm{nfmap\_sz} \times \mathrm{nfmap\_num}\end{aligned} \tag{3}$$

$$\begin{aligned}&\mathrm{ILNET}_{\mathrm{conv3}}\\ &= \underbrace{5 \times 5 \times 512 \times 2}_{\mathrm{pfmap\_sz}} \times \underbrace{100}_{\mathrm{long}}\\ &\quad \times \underbrace{3}_{\mathrm{scale}} + \underbrace{15 \times 15 \times 512 \times 2}_{\mathrm{nfmap\_sz}} \times \underbrace{20}_{\mathrm{short}}\\ &= 11.72\ \mathrm{MB}\end{aligned} \tag{4}$$

3 Evaluation of algorithm-level features

In this section, the algorithm-level features which have a severe impact on the hardware implementation are evaluated. We study the performance impact if we remove or approximate these features in addition to the gain that can be achieved in return. In [14], we explored the design space of the fixed-point representation of the main parameters of ILNET and we studied 11 ILNET variants using OTB-100 benchmark.

Although we showed in [14] that the fixed-point representation of the fully connected layers can be reduced down to 13-bit and 12-bit for the weights and the feature maps, respectively, with small performance degradation, we adopt 16-bit in this section because the external memories are typically byte-aligned and this 16-bit representation simplifies the memory data allocation and data access.

In this analysis, we incrementally study one system-feature at a time, create a tracker variant and check the performance delta by running OTB-100 on all tracker variants. Table 3 shows a summary of the results: the tracker variant number, OTB success AUC, OTB precision, memory estimate, speed-up factor and outline of the system modification.

3.1 Removal of the multi-scales in the training

As described in Algorithm 1, multi-scale feature maps are extracted at two steps: for the fine localization step and for collecting feature maps required in the training step. In ILNET12, we removed the extraction of multi-scale feature maps that were used in the training phase. We depend on one scale only for the online training. However, we still extract multi-scale feature maps for the fine localization step. Accordingly, the processing time and the required memory storage can be reduced, while trading-off a small performance degradation. Compared to ILNET5, the memory is reduced from 27.14 to 24.70 MB.

3.2 Removal of the hard-negative mining

The idea of hard-negative mining is to identify good negative samples for the training which is done via getting false positive classifications. In [3], 1024 negative samples are scored by the network, and then, the 96 negative samples with the highest positive score out of the 1024 tested samples are selected for the mini-batch training. The mini-batch consists of 32 and 96 positive and negative training samples, respectively. Although hard-negative mining would enhance the classification capability of the tracker, it would challenge the hardware implementation. We would need to run the whole network for 1024 samples, score them and select-out the samples of the highest object scores to be used in the training phase. Therefore, in ILNET13, we evaluate the performance degradation if we remove the hard-negative mining step. It can be noticed that there is a considerable amount of performance loss in ILNET13 accordingly.

Table 3 Exploration of system-level features*

Tracker	Success AUC	Precision	Memory (MB)	Speed-up factor	Modification outline
ILNET5	0.606	0.843	27.14	1x	Baseline [14]
ILNET12	0.599	0.832	24.70	1.02 ×	-Remove multi-scales used in the training
ILNET13	0.583	0.819	24.70	1.40 ×	Remove multi-scales used in the trainingRemove hard-negative mining
ILNET14	0.588	0.816	24.70	1.45 ×	Remove multi-scales used in the training
					Remove hard-negative mining -Proposed training scheme (update every 10 frames)
ILNET15	0.581	0.808	24.70	1.45 ×	Remove multi-scales used in the training
					Remove hard-negative mining -Proposed training scheme (update every 10 frames) -Softmax PWL
ILNET16	0.551[1]	0.757[1]	24.70	1.45 ×	Remove multi-scales used in the training
	0.575[2]	0.791[2]			Remove hard-negative mining -Proposed training scheme (update every 10 frames)
	0.580[3]	0.800[3]			
	0.586[4]	0.809[4]			Softmax PWL -Dropout fixed patterns
ILNET17	0.583	0.801	19.70	1.45 ×	Remove multi-scales used in the training -Remove hard-negative mining
					Proposed training scheme (update every frame)
					Softmax PWL
					Dropout fixed pattern (30 pattern)

*For all ILNET variants given in this table, the weights and the feature maps are represented by 8-bit for the convolutional layers, and by 16-bit for the fully connected layers

[1]No dropout, [2]dropout pattern: 8, [3]dropout pattern: 18, [4]dropout pattern: 30

3.3 Proposed online training scheme

The online training step is carried out every tenth frame and with 10 iterations per update in MDNET and ILNET. This update scheme results in a non-uniform processing time across the frames. In addition to localization and collecting training samples each frame, the tenth frames would require longer processing time because of the online update process. We believe this update scheme is undesirable for the hardware implementation and it is important to achieve a constant processing time across all the frames. Therefore, we propose to modify the online update process where we perform the online training each frame and with a single iteration only. Consequently, the overall processing time would not change compared to the baseline tracker, while the processing time of the online training is spread evenly on the frames. Figure 6 illustrates the advantage of using our proposed online training scheme where a steady processing time can be achieved for all video frames. The proposed update scheme is implemented in ILNET14 and the interesting point is that it has re-gained some of the performance loss caused by removing the hard-negative mining feature. Additionally, we made the Stochastic Gradient Descent (SGD) update much simpler by removing the momentum and decay update which would save 5 MB required for the momentum storage. In ILNET14, the online update that we perform each frame is not actually affecting the filter weights directly. We store the updated weights temporarily and apply them to the network at the tenth frames. The required filter storage of the fully connected layers would double consequently, while removing the momentum storage compensates that and there will be no memory increase in total. We attempt to keep our proposed update scheme close to that of the baseline tracker. The difference between both approaches would be in the training patches adopted in the update process. For a batch of 10 frames, the baseline update scheme is carried out utilizing all the training patches collected from the last 10 frames as well as

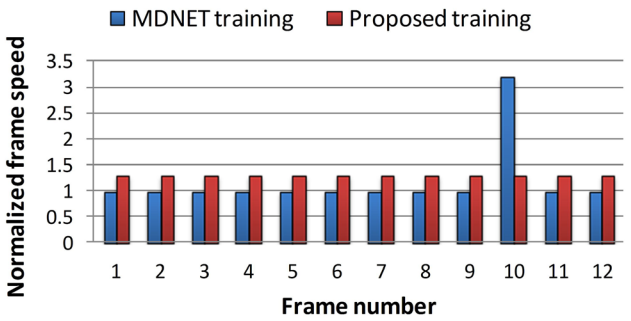

Fig. 6 Frame speed analysis using the proposed online training scheme

the previous frames. In our proposed update scheme, however, the update process is obviously carried out utilizing the training patches collected from the frames that are only received.

3.4 Approximation for Softmax layer

Softmax layer has been widely used in neural networks as an activation function and as a final classification layer. However, Softmax involves expensive division and exponentiation functions. Several works in literature, e.g., [17–21], proposed efficient hardware implementations for the softmax function where all of them adopted domain transformation technique. The domain transformation is described in Eqs. (5), (6) and (7).

$$f(a_i) = \frac{e^{a_i - a_{\max}}}{\sum_{k=1}^{z} e^{a_k - a_{\max}}} \tag{5}$$

$$\ln(f(a_i)) = (a_i - a_{\max}) - \ln\left(\sum_{k=1}^{z} \exp(a_k - a_{\max})\right) \tag{6}$$

$$f(a_i) = \exp\left((a_i - a_{\max}) - \ln\left(\sum_{k=1}^{z} \exp(a_k - a_{\max})\right)\right) \tag{7}$$

It can be noticed from Eq. (7) that the division is removed and a logarithmic function is added. In [17], the logarithmic and exponential functions are implemented by Look-Up Tables (LUT). Linear fitting is adopted in [18] to approximate the logarithmic and exponential functions to simplify the hardware implementation. In [19], Eq. (7) is further approximated to $f(a_i) = \exp(a_i - a_{\max})$ in case of $(a_i \neq a_{\max})$.

We can see that Eq. (7) has two exact exponential functions and one logarithmic function. Piece Wise Linear (PWL) approximation for the exponential function was presented in [20, 21]. In our work, we propose to adopt PWL approximation for both, the exponential and logarithmic functions. We started from Matlab to opt a suitable number of lines and the location of the breakpoints in addition to the proper fixed point representation of all the intermediate results. First for $\exp(a_k - a_{\max})$, we can notice that the operand $(a_k - a_{\max})$ is always less than or equal zero which would simplify the approximation. For the logarithmic function, the operand value $\left(\sum_{k=1}^{z} \exp(a_k - a_{\max})\right)$ would be range from 1 to z (z is two for the classification network and five for the localization network). Consequently, the output from the logarithmic function would range from 0 to $\sim$ 1.61. For the final exponential function, it can be noticed that the operand $\left((a_i - a_{max}) - \ln\left(\sum_{k=1}^{z} \exp(a_k - a_{max})\right)\right)$ would be also less than zero like the first exponential, and hence, we

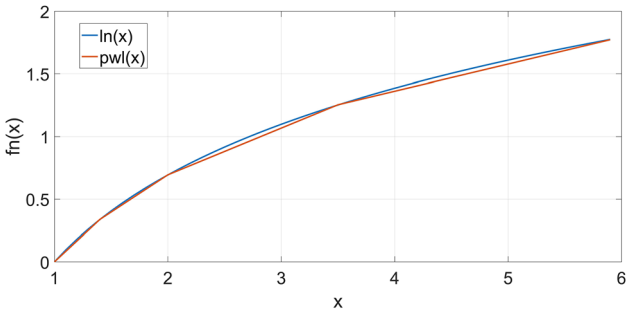

Fig. 7 PWL of the logarithmic function

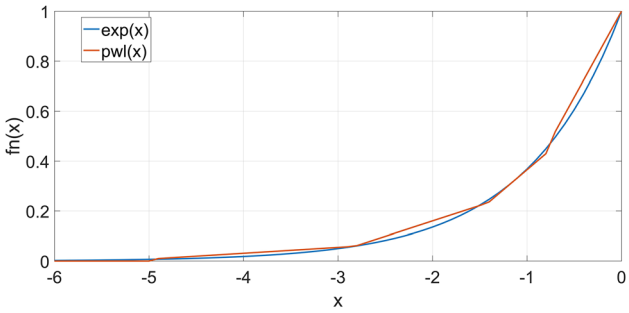

Fig. 8 PWL of the exponential function

can apply the same PWL for both exponential functions in Eq. (7). We adopted five linear pieces and four linear pieces for PWL_{exp} and PWL_{ln}, respectively. Figure 7 and Fig. 8 plot the PWL function along with the exact functions. A small performance loss is observed when we apply the PWL approximation to the ILNET15 tracker as shown in Table 3.

3.5 Dropout layer

We study the effect of the dropout layer on the training process in ILNET16. Having a fully random dropout in hardware would complicate the design. Hence, we apply pseudo-random dropout patterns and evaluate the performance. We tested few variants where the pattern for the same dropout layer is repeated every 8, 18 and 30 iterations. The performance of no dropout is also evaluated. It can be noticed that there is a significant impact of the dropout layer on the performance. Removing the dropout layer has a severe performance degradation (success AUC is lowered to 0.551), while for the pseudo-random patterns, the performance is getting better as the pattern size increases (success AUC is enhanced to 0.586 for a repeated pattern every 30 iterations).

3.6 Every-frame network update

In ILNET17, we extended our proposed update scheme by applying the new filter weights directly to the network each

frame. Hence, we would be able to get rid of the temporary filter storage and reduce the required memory storage. It can be noticed that performance is slightly affected compared to ILNET16 with a dropout pattern repeated every 30 iterations.

In addition to the advantages of simplifying the hardware implementation and the memory reduction that are gained by applying the schemes in ILNET12 to ILNET17, a speed-up improvement of 1.45 × is also obtained. The speed-up improvement arises mainly from removing the hard-negative mining in ILNET13. In addition, our proposed training scheme features a constant training time even in case of the frames that require a short-term update. We analyzed the speed of ILNET5 [14] and all the ILNET variants presented in this paper by running the Matlab system model on all OTB-100 videos using an Intel i7-3520 M CPU system. ILNET5 runs in 0.78 s per frame, while ILNET17 run in 0.54 s per frame.

4 Online training hardware accelerator

The implementation of online training capability in embedded systems is challenging because of the additional processing steps and memory storage. The main focus of most of the previous publications on CNN hardware implementations was on the inference phase. In such systems, offline training is mainly exploited to obtain the filter weights which would be fixed throughout the inference computations. However, online training is a vital for ILNET and other CNN-based trackers. In ILNET, we did not even train the network on any video dataset, while we solely depend on the online training in the initial frame and throughout the successive frames. Hence, we propose a hardware accelerator for the online training of the fully connected layers that we preset in this section.

We opted to develop and test our design on an FPGA as the target fabric, while our design is suitable for ASIC development as well. We target Xilinx VC709 evaluation board [22] which has Virtex-7 FPGA (XC7VX690T-2FFG1761C) [23].

4.1 Computation flow

The network weights and the feature maps are typically stored in the external memory due to the large data size. Chunks of data are fetched from the external memory and processed on chip. The result is written back to the external memory. There is a trade-off between the data chunk size and the on-chip memory versus the number of external memory accesses. As the number of external memory accesses increases, the processing time and the energy consumption intuitively increase. Figure 9 illustrates the computation flow in the ILNET fully connected layers. The input is X_4 which is selected from the stored feature maps. The mini-batch size is 128 and the feature map size is $3 \times 3 \times 512$. Hence, the size of X_4 is 128×4608. The fully connected layers fc4, fc5 and fc6 compute X_5, X_6 and X_7, respectively. The loss is calculated by the Softmax layer and back-propagated in the network to get the delta weights: dW_6, dW_5 and dW_4, and the new propagated errors: dX_6 and dX_5. Matrix transpose operation has to be applied to W_6, X_6, W_5, X_5 and X_4 in the backward path. Hence, the main computations are eight matrix multiplications with different sizes as shown in yellow rectangles in Fig. 9: three matrix multiplications in the forward path and five matrix multiplications in the backward path.

The transposed reading required in the backward path is time-consuming. It requires 4 × more memory accesses than that required in the forward path [13]. In order to overcome this issue, a transposed SRAM with two different access modes is proposed in [24] in order to allow transposed readings, while it requires 40% larger area. Binary Feedback Alignment (BFA) scheme is adopted in [13] where a constant matrix is used in the back-propagation path instead of the transposed weight matrix. BFA causes performance degradation as it replaces the standard back-propagation.

In our work, we address the transposed reading issue by re-forming the matrix multiplications in the backward path. We simply apply the transpose operator to both sides of the backward path equations. As shown in Fig. 10, we store $loss^T$ in the memory instead of *loss* and the back-propagated errors would be calculated in the transpose format as well, dX_6^T and dX_5^T. In this proposed scheme, we would need to perform the transpose operation only when writing the *loss* and the delta weights. The external memory accesses required for writing the result in a transposed form are significantly less than that required for the transposed-reading to perform matrix multiplications. In addition, we show that the weight update step can be done on the fly without requiring additional processing steps.

The arithmetic multiplier is a key element in this implementation. We can achieve better processing time by increasing the number of the parallel multiplications in the system. However, it is typically desired to reduce the number of multipliers to save the logic area. Moreover, the number of DSP elements in the FPGAs is limited. In order to get a sense of how many multipliers we need to achieve the target performance, we performed the quick analysis shown in Table 4. We listed the number of the required multiplications in the forward path and backward path for the three layers and assuming 200 MHz clock frequency and single update iteration. We can achieve 72.6 fps with 256 multipliers. This performance should be

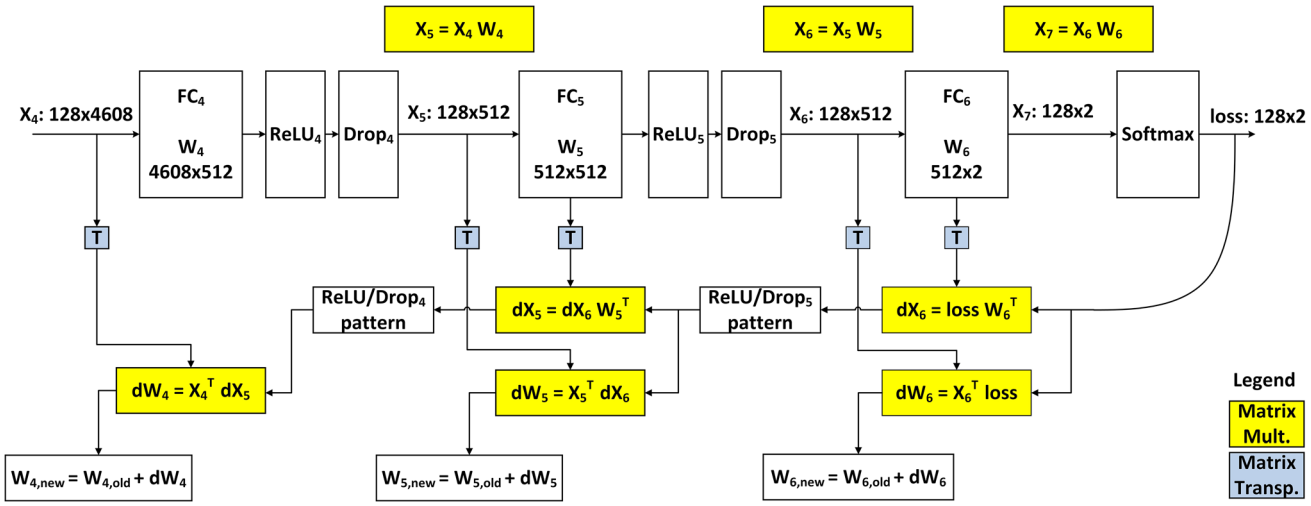

Fig. 9 The computation flow of a conventional online training [14]

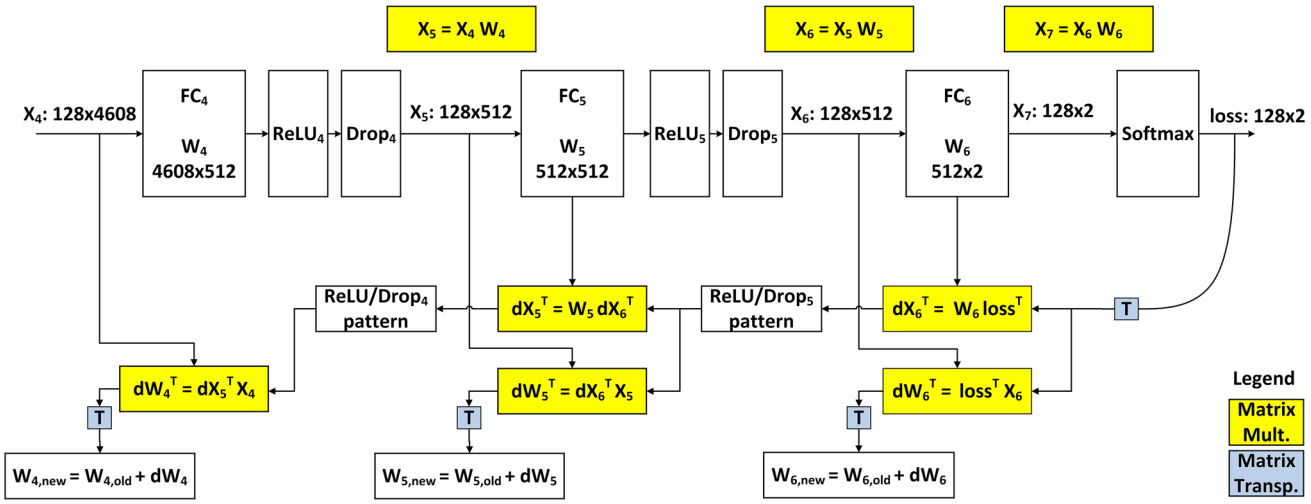

Fig. 10 The computation flow of the proposed online training

Table 4 Computation breakdown in fully connected layer

FCx	# of multiplications		FPS	
	Forward	Backward	Per layer	Total
FC4	128 × 4608 × 512	0	84.8	72.6
FC5	128 × 512 × 512	512 × 512 × 128 + 512 × 128 × 512	508.6	
FC6	128 × 512 × 2	512 × 2 × 128 + 2 × 128 × 512	130,208.3	

acceptable such that when the fully connected computation is added to the convolutional layers computation, the whole system can achieve a real-time operation (i.e., around 30 fps). It is worth mentioning that this performance of 72.6 fps is theoretical without taking into consideration the memory access latency or the Finite State Machine (FSM) delays.

4.2 Memory latency reduction

Previous works have focused on reducing the number of the external memory accesses via transferring parts of the data to the on-chip buffer and maximizing the reuse of this on-chip data. Compression techniques have been also proposed to reduce the data size that is transferred from the external memory. Although these works result is a reduced number of memory accesses, they are sub-optimal as they

do not consider the memory latency-per-access which also affects the energy consumption [25]. Therefore, in our work, we focus on reducing the external memory latency by adopting an efficient arrangement of storing and accessing the data.

First for the data storage, instead of storing the data matrix row-by-row or column-by-column, we store chunks of each row. The chunk is the maximum data that we can read from the DRAM in one clock cycle which is 512-bit (32 × 16-bit) on Xilinx VC709 evaluation board. Hence, starting from the first row, we store the first chunks (32 × 16-bit) of all the rows till we reach the last row, then we store the second (32 × 16-bit) chunks of all the rows and so on. Figure 11 illustrates the proposed order of data storage. If we manage to read and write the data in sequential addresses, this would yield the minimum memory latency.

Second for the memory access order and the matrix multiplication operation, we apply the same multiplication scheme for all the eight matrix multiplications where $[Z] = [Mat1] * [Mat2]$. Each matrix element is represented by 16-bit. We start by reading chunks (each chunk is 512-bit $\equiv$ 32 element) from eight rows of $[Mat1]$ and store them on-chip. This read data can be considered a sub-matrix of size 8 × 32 elements, we call it $[subMat1]$. We then read a single chunk (32 elements) from the first row of $[Mat2]$. We perform 256 multiplications between the first column of $[subMat1]$ and the 32 elements from $[Mat2]$. The partial sums are stored inside the registers of the MAC units and there is no need to transfer them to on-chip memory or the DRAM. Then, the next chunk is read from

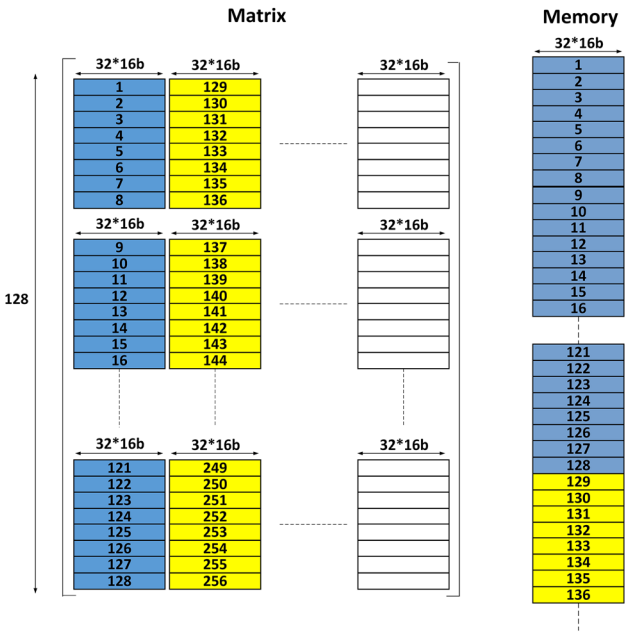

Fig. 11 Matrix storing order

$[Mat2]$ and multiplied with the second column of $[subMat1]$. The second column of $[subMat1]$ is already stored on chip and the partial sum is accumulated. A double-buffer is employed to store $[subMat1]$ on-chip so that we maintain available data from $[Mat1]$ when transitioning from the last column of $[subMat1]$. We continue the same sequence till reaching the chunk in the last row of $[Mat2]$. The output from the 256 multipliers will then give the first 8 × 32 elements of the final result $[Z]$ which are stored back to DRAM. We repeat the same sequence until we obtain all elements of $[Z]$.

It can be noticed that reading a single $[subMat1]$ yields the minimum DRAM latency as the read addresses are sequential. The latency of filling the double-buffer by different sub-matrices whose DRAM start addresses are not sequential should not affect the processing time because filling the double-buffer is carried out in parallel while performing the multiplications. The multiplications of all the 32 columns of a single $[subMat1]$ would take at least 32 clock cycles which are typically sufficient to fetch another $[subMat1]$ from the DRAM even if its start address is not contiguous with that of the previous sub-matrix.

For $[Mat2]$, we read the first 32 elements from all the rows sequentially. The MAC units perform the multiply and add operations on the fly while reading $[Mat2]$ chunks till the last row is read. Hence, this scheme should also yield the minimum DRAM latency as all the read addresses are contiguous.

For the result $[Z]$, it is worth mentioning that it will be read in the following computation steps, and hence, it will be considered $[Mat1]$ or $[Mat2]$ of the new computation step. Therefore, we write the result in the same order we described before. We always obtain chunks of the result of size 8 × 32 and we write the chunk data in sequential addresses.

4.3 Weight update

As we mentioned, we avoid the transposed reading of $[Mat1]$ and $[Mat2]$ by re-forming the matrix multiplications in the backward path. The transposed delta weights $[dw^T]$ would be obtained accordingly. If we opt to write the delta weights to DRAM, this would slow-down the processing speed due to the increased number of memory accesses required for the weight update step. Instead, we do not write the delta weights to DRAM in our update scheme. As the matrix multiplication result $[dw^T]$ is calculated into chunks of size 8 × 32, we read chunks of the old weights $[W_{old}]$ of size 32 × 8 and do the weight update and the transpose operation directly on this chunk. We then write chunks of size 32 × 8 of the new weights $[W_{new}]$ back to DRAM.

4.4 ReLU and dropout storage

fc4 and fc5 are followed by ReLU and Dropout layers. ReLU layer saturates all negative values in the feature maps to zero and Dropout layer sets some values in the feature maps to zero randomly. We need to store the values that are set to zero in the forward path by ReLU and Dropout layers because they are required in the backward path when back-propagating the loss. Instead of storing the feature maps at each layer output (i.e., fc, ReLU and Dropout outputs), we store the output at the Dropout layer only in DRAM. We build a small array and store it on-chip memory where one bit per output neuron is employed to indicate whether this output neuron has been set to zero by ReLU, Dropout or not. As shown in Fig. 12, we store the pattern of ReLU and Dropout together which requires 8 KB (128 × 512 × 1-bit) instead of 128 KB if we store the complete feature maps (128 × 512 × 16-bit). Figure 13 shows an example of applying the ReLU and Dropout functions where we use a value of '1' to indicate that the neuron output is set to zero by either ReLU or Dropout layer.

4.5 Memory allocation

We store the feature maps in one DRAM and the fitter weights in the other DRAM. Our objective is to maximize the memory throughput in the system. Hence, we avoid reading $[Mat1]$ and $[Mat2]$ from the same DRAM. Therefore, we start by having $X4$ in DRAM1, and all the weights:$W4$, $W5$ and $W6$ in DRAM2. Then, we obtain and store $X5$ and $X6$ in DRAM1. After obtaining the *loss*, we store it in both memories because the *loss* would be required as $[Mat1]$ when we calculate $\left[dW_6^T\right]$, while it would be required as $[Mat2]$ when we calculate $\left[dX_6^T\right]$. We follow the same scheme after obtaining the propagated error $\left[dX_6^T\right]$, we write it in both memories because it would be considered $[Mat1]$ and $[Mat2]$ in the following computation steps. This writing scheme allows us to avoid reading $[Mat1]$ and $[Mat2]$ from the same DRAM in any of the computation steps.

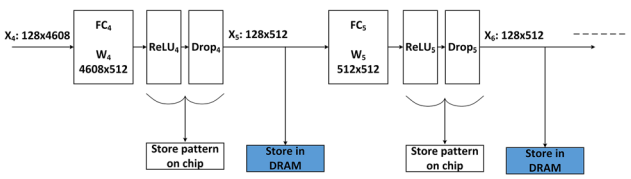

Fig. 12 ReLU and Dropout storage

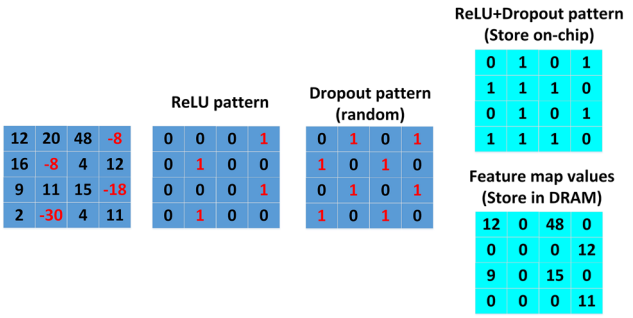

Fig. 13 Example of applying ReLU and Dropout patterns

4.6 Top-level design description

Figure 14 shows the top-level block diagram of our proposed hardware design of the fully connected layers. The blue arrows represent the data signals only and not the address or side-band signals. There are two FSMs to read $[Mat1]$ and $[Mat2]$ from the DRAMs. The maximum data size we can read from DRAM per clock cycle is 512-bit (32 elements each of 16-bit). As previously mentioned, the data for $[Mat1]$ is stored in a double buffer of size 2 × 8x32 × 16-bit. There are 256 Processing Elements (PE) employed in our design which perform the MAC operations on two inputs. A bias would be added if required by the computation step. The data read from the double buffer are of size 8 × 16-bit, and are then replicated 32 times to generate input A (256 × 16-bit) of the MAC units. Input B (256 × 16-bit) of the MAC units comes from $[Mat2]$ RD FSM after replicating the data eight times.

The output from the MAC units is stored in FlipFlops (FF) which will be written back to DRAM directly or processed first based on the computation step. In the softmax step, the MAC outputs are processed first by the softmax logic before writing to DRAM. In the weight update steps, the old weights are read by the weight RD FSM, got updated by the delta weights stored in the FFs and then written back to DRAM. We choose which DRAM to write in based on the computation step and the scheme we described previously. For the bias read and update, we store the bias values in chip RAM as they do not need large memory size. The bias is required for the matrix multiplication in the forward path. The bias update is carried out by the summation of the *loss* and the back-propagated errors.

The proposed design is described in Verilog with several parameterized parameters so that the same design can be ported easily to support other fully connected layers with different structures, input and weight sizes. The main parameters in our Verilog code are as follows:-

- *a_total*, *b_total*: first and second dimension of $[Mat1]$

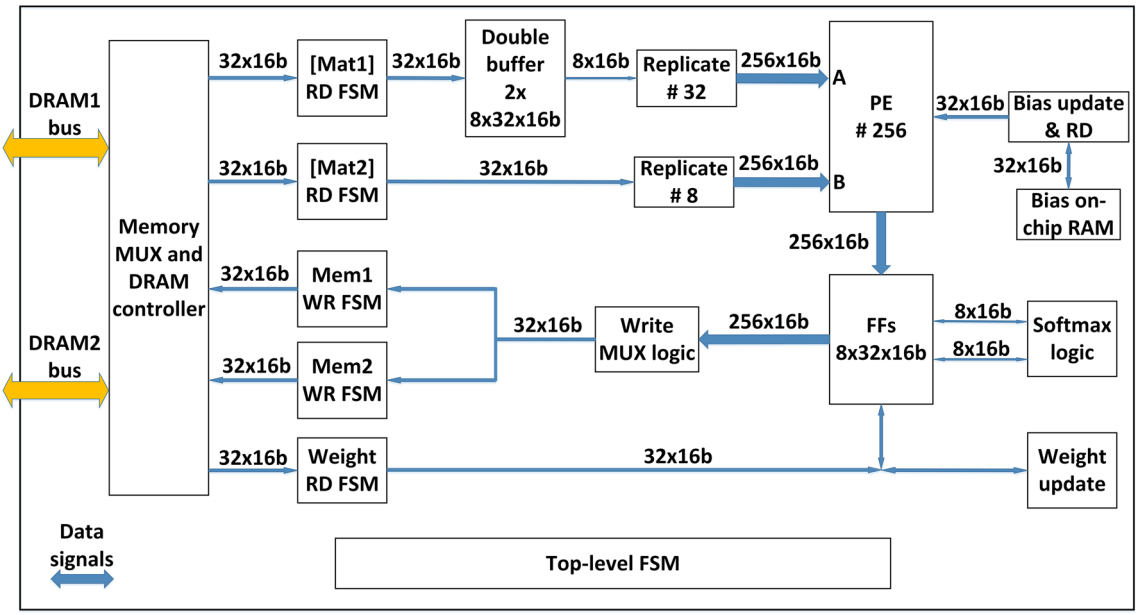

Fig. 14 Top-level block diagram of our proposed accelerator

- *c_total*, *d_total*: first and second dimension of $[Mat2]$ (*b_total* = *c_total*).
- *a_step* and *b_step*: step size for reading $[Mat1]$
- *c_step* and *d_step*: step size for reading $[Mat2]$
- *memwr_step*: step size for writing the result to DRAM.

In addition, the number of the PEs and the associated interface is parameterized so that we can study the area-performance tradeoff when increasing or decreasing the number of the PEs and choose the appropriate number based on the system needs.

There are also run-time configurations that would allow flexibility in the system when integrating the hardware accelerator with CPU. The start address of reading $[Mat1]$ and $[Mat2]$ from DRAM is configurable for all the computation steps and they should be provided by the CPU. In addition, the start address for writing the result to DRAM is configurable for all the computation steps. The selection of which DRAM to write the result in is also configurable. The CPU can also change the learning rate used by the accelerator at run-time based on the performance and system needs.

5 Experimental results and analysis

Test vectors are dumped from the Matlab model for all the design intermediate nodes. The same input vector and the initial weight values are used in simulating the design on the Register Transfer Level (RTL). The outputs at all the intermediate nodes from the hardware design are bit-matched with those of the Matlab model. We measured the design speed using an ideal memory model and a DRAM3 memory model. The ideal memory model has the same interface as that of DRAM3, while there are no wait states for read and write requests. Table 5 shows the performance of our design with the theoretical speed where there is no FSM or memory access overhead. It can be noticed that FSM overhead in our design is minimal where our design with the ideal memory model achieves pretty much the same performance as the theoretical case. For DRAM3 model, we achieved a memory access efficiency of 64.5% using our access scheme and the proposed order of the data storage.

Our proposed design is synthesized and implemented by the Xilinx Vivado 2018.1 tool. We adopted the Xilinx generated design for the DRAM3 memory controllers. The FPGA resource utilization is shown in Table 6. In general, the utilization is low for all types of resources on the Virtex-7 FPGA. It is worth mentioning that the DRAM3 memory controller consumes 24.6% (22,613 slices) of the whole LUT slices (91,954 slices) used by our design.

Although our design is relatively small and Virtex-7 FPGA is relatively big in logic size, there was a congestion during placement and during routing which initially rendered the Vivado tool unable to meet the required timing. For such congested designs, the Vivado tool would prioritize the routing completion of all signals over meeting the required timing. There are a couple of reasons for this congestion: the double buffer was implemented initially with BRAMs which may have limited the capability of the placer and router. High fan-out of several signals and big combinational logic for fixed-point conversion were

Table 5 Performance of the proposed accelerator

Design	Performance at 200 MHz and 256 multipliers
Theoretical (no FSM, no memory)	72.6 fps
Our design (ideal memory model)	71.2 fps
Our design (DRAM3 model)	44 fps

Table 6 Resource utilization using Xilinx Virtex-7

Resource	Utilization	Available	Utilization %
LUT	91,954	433,200	21.23
LUTRAM	8637	174,200	4.96
FF	40,338	866,400	4.66
BRAM	585	1470	16.25
DSP	380	3600	44.71
IO	6	850	18.75
MMCM	3	20	15
PLL	2	20	10

reasons for the congestion as well. Therefore, we implemented the double-buffer with registers (FFs) and added several pipelined FlipFlops to break long combinational paths. In addition, we added several Multi Cycle Path (MCP) constraints for many signals. If intended by the design, adding MCP constraints on the failing paths would allow to meet the required timing for these paths without breaking the paths and adding more FlipFlops and latency. Moreover, we had to enable a special strategy for the Vivado implementation (Congestion_SpreadLogic_medium) which is recommended for highly congested designs. In some cases, there might be a need to explore the special strategy (Congestion_SpreadLogic_high) as well depending on the design complexity. After applying all these techniques, we are able to meet the required timing at 200 MHz.

Our HW accelerator can be adopted to train other Multi-Layer Perceptron (MLP) neural networks that are employed in real-time applications on embedded devices. Some prior works [13, 26, 27] exist in literature to accelerate the training of the MLP or fully connected neural networks. Approximate computing is adopted in [26] via inexact multipliers and bit-precision reduction to reduce the power consumption. The synapses that have lesser impact on the final error are obtained from the training phase and approximated by the inexact multipliers. In [27], the authors proposed a pipeline implementation of the Quasi-Newton method for training instead of the conventional batch training. The Quasi-Newton training method speeds-up the training convergence at the cost of memory resources.

A HW accelerator close to our work was recently published in [13]. The authors implement a HW accelerator in 65 nm ASIC to train the last fully connected layers in the MDNET-based object tracker similar to our work. However, there are several differences. The whole network in [13] is trained offline on the VOT dataset. The authors added a new fully connected layer of weight size $1 \times 1\text{x}512 \times 512$ in order not to train fc4 which has a weight size of $3 \times 3\text{x}512 \times 512$. Hence, in the modified network, there will be four fully connected layers. fc5, fc6 and fc7 are only trained online, while the weights of fc4 are fixed. The idea behind this is to reduce the External Memory Access (EMA) where fc4 has the main contribution because of its large weight size. This modification in MDNET has negatively affected the performance by 5% on a Tracking Accuracy (TA) metric defined by the authors. In addition, in order to overcome the matrix transpose operation required in the error propagation and weight update steps, the authors employed BFA instead of reading transposed weight matrices. BFA has performance degradation as well, reported to by 2.93% on TA. It is not mentioned, however, the actual impact of BFA and the scheme of not training fc4 online on the OTB benchmark. We believe it is important to have a unique metric, like OTB success AUC, across the different designs in order to achieve a fair comparison. The HW design in [13] implements a Run-Length Coding (RLC) to store the feature maps completely on-chip memory, while it brings chunks of the weights from the external memory to an on-chip memory of size 1 KB.

On the other hand, in our proposed HW accelerator, we did not modify the fully connected layers in order not to affect the performance. We still train fc4, fc5 and fc6 online. For the matrix transpose issue, we re-form the matrix multiplications in the backward path so that we do not need to read transposed weight and feature map matrices from the external memory. Instead, the propagated errors are calculated in the transposed form. Hence, we employ the normal back-propagation for the weight updates and avoid degrading the performance as is reported by [13] due to BFA. We store the feature maps and the propagated errors to the external memory which makes our design more scalable to larger-size networks. We have provided detailed OTB results on all the system-level features we studied. Table 7 shows a comparison summary of our work with [13]. Both accelerators operate at the

Table 7 Comparison summary of online training accelerators used in object tracking

	[13]	Our work
Online-trained only	No	Yes
Error propagation	BFA	Back-propagation
EMA reduction scheme	RLC, BFA	No
Implementation fabric	ASIC 65 nm	FPGA Virtex-7
Online training FPS	16.2 FPS	44 FPS

same frequency of 200 MHz and achieve a theoretical throughput of 51.2 GOPS. However, our accelerator achieves 44 fps compared to 16.2 fps achieved by [13].

As we target embedded systems, it is important to get the performance of the whole tracker operation including the inference and the training phases. We assume the CNN inference is carried out by the inference processor [28] which is the same inference processor assumed in [13]. We estimate the speed of the our tracker ILNET and MDNET when employing our training accelerator and the training accelerator in [13], respectively, while using the same inference processor for both. Table 8 shows the maximum FPS that can be obtained when using the CNN processor [28] as the CNN inference for both MDNET and our proposed tracker ILNET along with the computation breakdown. This analysis is carried out through obtaining the required MAC operations of the convolutional computations in both trackers. For MDNET, the number of MAC operations required to obtain the feature map of one candidate is calculated. This number is then multiplied by the total number of candidates in the tracking phase and the training phase to get the total number of MAC operations. For ILNET, the number of MAC operations is calculated for each feature map size: one feature maps of size $15 \times 15 \times 512$ and two feature maps of size $5 \times 5 \times 512$. There are no convolutional computations required for the training phase. Accordingly, the CNN accelerator [28] can run ILNET convolution computations with 167.45 fps, while it can run MDNET convolution computations with 4.87 fps. We can then add the speed of the online training accelerator published in [13] and the speed of our online training accelerator to the MDNET and ILNET analysis, respectively, to estimate the performance of the whole tracker system. Our proposed tracker can run at 34.81 fps on the mentioned hardware accelerators compared to 3.74 fps for MDNET. It is evident that our modifications on the system-level of ILNET have achieved a significant improvement of the total tracker speed.

Table 8 Tracker performance on a CNN accelerator

Computation breakdown	MDNET	ILNET
# of conv MAC per one candidate	122×10^6 (3×3 fmap)	1323×10^6 (15×15 fmap) 231×10^6 (5×5 fmap)
# of runs in the tracking phase	256	1 (15×15 fmap)
2 (5×5 fmap)		
# of runs in the training phase	250	0
# of interpolation MAC per frame	NA	6.5×10^6
Total # of MAC per frame	61.6 GMAC	1.8 GMAC
Inference processor throughput [28]	300 GOPS	
Maximum inference fps of the tracker	4.87 fps	166.7 fps
Total tracker speed	3.74 fps	34.81 fps

It is worth mentioning that our contributions presented in ILNET can be utilized in many other CNN-based trackers, but the performance impact of each modification may differ from one tracker system to another. Therefore, we recommend running the object tracker benchmarks to obtain the actual system impact when applying these modifications to other CNN-based trackers. Some contributions we present that can be utilized in other CNN-based trackers include the interpolation schemes, the uniform processing time across all frames and the approximation of Softmax and dropout layers. Some other contributions are even general and can benefit CNN networks in other applications, like avoiding the transposed read in the training of the fully connected layers and the proposed scheme for memory latency reduction.

6 Conclusion

The full hardware design of our CNN-based object tracker, code named ILNET, is exposed. ILNET targets speed enhancement and relies solely on online training. A multitude of algorithmic and circuit design techniques are integrated to attain an efficient, and real-time object tracker.

On the algorithmic level, interpolation schemes and a novel technique for online training with uniform per frame latency are utilized. Also, the impact of removing or approximating the hardware-expensive computations is evaluated at the system-level.

On the circuit level, a fixed-point hardware accelerator for the online training of the fully connected layers is presented. We introduce a novel technique for error back-propagation in the fully connected layers that avoids the

costly reading of transposed data from external memories. In addition, we present a novel technique for memory latency reduction via adopting an efficient arrangement of storing and accessing the data.

Thanks to this multitude of algorithmic and circuit level enhancements, our proposed accelerator can train the fully connected layers at a rate of 44 fps compared to 16.2 fps achieved with the prior-art designs at the same clock frequency.

Funding Open access funding provided by The Science, Technology & Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB).

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. El-Shafie AHA, Habib SED (2019) Survey on hardware implementations of visual object trackers. IET Image Proc 13:863–876. https://doi.org/10.1049/iet-ipr.2018.5952
2. Dalal N, Triggs B (2005) Histograms of oriented gradients for human detection. In: Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005: pp 886–893
3. Nam H, Han B (2016) Learning Multi-domain Convolutional Neural Networks for Visual Tracking. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society: pp 4293–4302
4. Ma C, Huang JB, Yang XK, Yang MH (2015) Hierarchical Convolutional Features for Visual Tracking. IEEE International Conference on Computer Vision. Santiago, CHILE, pp 3074–3082
5. Danelljan M, Hager G, Khan FS, Felsberg M (2016) Convolutional Features for Correlation Filter Based Visual Tracking. In: Proceedings of the IEEE International Conference on Computer Vision. Institute of Electrical and Electronics Engineers Inc., pp 621–629
6. Danelljan M, Robinson A, Khan FS, Felsberg M (2016) Beyond correlation filters: Learning continuous convolution operators for visual tracking. In: Leibe B, Matas J, Sebe N, Welling M (eds) Computer Vision – ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part V. Springer International Publishing, Cham, pp 472–488. https://doi.org/10.1007/978-3-319-46454-1_29
7. Tao R, Gavves E, Smeulders AWM (2016) Siamese instance search for tracking. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society pp 1420–1429
8. Bertinetto L, Valmadre J, Henriques JF, Vedaldi A, Torr PHS (2016) Fully-Convolutional Siamese Networks for Object Tracking. In: Hua G, Jégou H (eds) Computer Vision – ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8-10 and 15-16, 2016, Proceedings, Part II. Springer International Publishing, Cham, pp 850–865. https://doi.org/10.1007/978-3-319-48881-3_56
9. Held D, Thrun S, Savarese S (2016) Learning to track at 100 FPS with deep regression networks. In: Computer Vision – ECCV 2016. Lecture Notes in Computer Science
10. Yun S, Choi J, Yoo Y et al (2017) Action-decision networks for visual tracking with deep reinforcement learning. IEEE Conf Comput Vis Pattern Recognit (CVPR). https://doi.org/10.1109/cvpr.2017.148
11. Yang L, Liu R, Zhang D, Zhang L (2017) Deep location-specific tracking. In: Proceedings of the 25th ACM International Conference on Multimedia. ACM, New York, NY, USA, pp 1309–1317
12. El-Shafie AHA, Zaki M, Habib SED (2019) Fast CNN-based object tracking using localization layers and deep features interpolation. In: 15th International Wireless Communications and Mobile Computing Conference, IWCMC 2019
13. Han D, Lee J, Lee J, Yoo HJ (2019) A low-power deep neural network online learning processor for real-time object tracking application. IEEE Trans Circuits Syst I Regul Pap 66:1794–1804. https://doi.org/10.1109/TCSI.2018.2880363
14. El-Shafie AHA, Zaki M, Habib SED (2021) Towards an efficient hardware implementation of CNN-based object trackers. In: IEEE International Symposium on Circuits and Systems (ISCAS)
15. Wu Y, Lim J, Yang MH (2015) Object tracking benchmark. IEEE Trans Pattern Anal Mach Intell 37:1834–1848. https://doi.org/10.1109/tpami.2014.2388226
16. Chatfield K, Simonyan K, Vedaldi A, Zisserman A (2014) Return of the devil in the details: delving deep into convolutional nets. In: British Machine Vision Conference
17. Yuan B (2016) Efficient hardware architecture of softmax layer in deep neural network. In: International System on Chip Conference
18. Wang M, Lu S, Zhu D, et al (2019) A high-speed and low-complexity architecture for softmax function in deep learning. In: IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2018
19. Kouretas I, Paliouras V (2019) Simplified hardware implementation of the softmax activation function. In: 8th International Conference on Modern Circuits and Systems Technologies, MOCAST 2019
20. Geng X, Lin J, Zhao B, et al (2019) Hardware-aware softmax approximation for deep neural networks. In: Proc. Asian Conference on Computer Vision. Springer Verlag, pp 107–122
21. Wei Z, Arora A, Patel P, John L (2020) Design space exploration for softmax implementations. In: Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors. Institute of Electrical and Electronics Engineers Inc., pp 45–52
22. VC709 Evaluation Board for the Virtex-7 FPGA. https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf
23. 7 Series FPGAs Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

24. Bong K, Choi S, Kim C, et al (2017) A 0.62mW ultra-low-power convolutional-neural-network face-recognition processor and a CIS integrated with always-on haar-like face detector. In: Digest of Technical Papers - IEEE International Solid-State Circuits Conference. Institute of Electrical and Electronics Engineers Inc., pp 248–249
25. Wicaksana Putra RV, Abdullah Hanif M, Shafique M (2020) DRMap: A generic DRAM data mapping policy for energy-efficient processing of convolutional neural networks. In: Proceedings - Design Automation Conference
26. Kim D, Kung J, Mukhopadhyay S (2017) A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing. IEEE Trans Emerg Top Comput. https://doi.org/10.1109/TETC.2017.2673548
27. Liu Q, Liu J, Sang R et al (2018) Fast neural network training on FPGA using quasi-Newton optimization method. IEEE Trans Very Large Scale Integr VLSI Syst 26:1575–1579. https://doi.org/10.1109/TVLSI.2018.2820016
28. Shin D, Lee J, Lee J, Yoo HJ (2017) DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In: Digest of Technical Papers - IEEE International Solid-State Circuits Conference