Accelerating Convolutional Neural Networks for Continuous Mobile Vision via Cache Reuse

Mengwei Xu, Xuanzhe Liu, Yunxin Liu, and Felix Xiaozhu Lin

***Abstract*—Convolutional Neural Network (CNN) is the state-of-the-art algorithm of many mobile vision fields. It is also applied in many vision tasks such as face detection and augmented reality on mobile devices. Though benefited from the high accuracy achieved via deep CNN models, nowadays commercial mobile devices are often short in processing capacity and battery to continuously carry out such CNN-driven vision applications. In this paper, we propose a transparent caching mechanism, named CNNCache, that can substantially accelerate CNN-driven mobile continuous vision tasks without any efforts from app developers. To cache and reuse the computations of the similar image regions which are consecutively captured by mobile devices, CNNCache leverages two novel techniques: an image matching algorithm that quickly identifies similar image regions between images, and a cache-aware CNN inference engine that propagates the reusable regions through varied layers and reuses the computation results at layer granularity. We implement a prototype of CNNCache to run on commodity Android devices, and evaluate it via typical CNN models. The results show that CNNCache can accelerate the execution of CNN models by 20.2% on average and up to 47.1% under certain scenarios, with no more than 3.51% accuracy loss.**

I. Introduction

Benefiting from powerful and ubiquitous cameras equipped on mobile devices, *continuous mobile vision* has surprisingly enriched and facilitated human lives via "showing computers what you see". The roles of such continuous vision tasks can be varied and generalized, including but not limited to 1) **image classification**, where a 3-years old child uses a smart tablet to scan the room and learn to differentiate objects such as bookcase, air condition, etc. 2) **face recognition**, where a user is wearing smart glasses during social activities, then a sequence of images will be captured and processed to identify the faces of the people in view. As people are identified, information about them can be displayed on glass screen. 3) **image-to-text translation**, where an English speaker walks around Lyon and uses her smartphone scanning the road signs written in French, then the text is translated into English and displayed on phone screen at the meantime. To make such vision intelligence practical and efficient under mobile scenarios, both industry and academia have put tremendous efforts via developing inspiring applications [8], [4], [45], [64], [22], [32], optimizing the runtime [39], [42], and building new hardware prototypes [38], [43], [44].

The state-of-the-art algorithm that drives continuous mobile vision is deep learning (DL), which feeds the captured images into a well-trained convolutional neural network (CNN) frame-by-frame and outputs task-specific results. Though proven to be very accurate compared to traditional vision algorithms, running these CNN models requires substantial hardware resources, e.g., CPU and memory, thus making it quite challenging on commodity mobile devices such as smartphones and head-mount devices. A straightforward way to overcome this limitation is offloading deep learning workloads to a much more powerful remote cloud [25], [2], [1]. However, this offloading strategy requires high network bandwidth and can sometimes have substantial energy and monetary cost. More importantly, sending image data via network continuously introduces big privacy concerns as images captured on mobile devices often contain personal information.

In this paper, we propose a cache mechanism for CNN, named ***CNNCache*** (CNN Optimization on Mobile by cache reuse), that leverages layer-level cache reuse to accelerate CNN execution in continuous mobile vision scenarios. It is based on a key observation that mobile devices are often in slow motion or even held still when running continuous vision tasks [42], [20]. It indicates an opportunity to exploit high temporal locality within adjacent frames in a video stream, e.g., consecutively captured images by mobile cameras might have substantial overlapped regions that are very similar. However, this opportunity is missed by traditional deep-learning engines, which always run as end-to-end systems by taking the whole image as input and directly outputting the results. Some prior work [43], [20] uses a coarse-grained caching strategy, which simply reuses the final result from the previous frame when the difference between frames is under a certain threshold. In comparison, CNNCache achieves a more fine-grained cache mechanism, as it reuses the intermediate computation results of identified similar regions (not the whole image) at layer granularity. The saved time from such caching mechanism can help deliver better user experience to end-users via a higher frame rate as well as a longer battery life.

Designing a practical and feasible caching mechanism for CNN inference at layer granularity, however, is challenging because of the large layer numbers and complicated inter-layer processing for deep models. Moreover, properly identifying *which part of images shall be reused* is difficult due to the presence of large environmental variations such as camera motion, object appearance, illumination conditions, etc. In addition, aggressive cache reuse might result in unacceptable accuracy loss as the reused image regions are unlikely to be identical. To this end, we carefully design CNNCache to maximumly reuse the computation results with minimal accuracy loss via two novel techniques. First, we design an image matching algorithm to identify the similar regions between two consecutive images. We retrofit the knowledge from video compression community [63] for our CNN-specific caching mechanism. Second, we dig deeply into the CNN execution

procedure and build cache-awareness into the execution of a CNN inference engine, so that the engine propagates similar regions through the internal layers and therefore reuses the computational results.

We implement CNNCache to be transparent to application developers and able to run off-the-shelf CNN models that are trained in the traditional way. We then evaluate CNNCache on commodity Android device with a variety of popular CNN models and real-world video benchmarks. The results show that CNNCache can accelerate the execution of CNN models by 20.2% on average and up to 47.1% in some scenarios, but only has minimal accuracy loss. In addition, around 19.7% energy consumption can be saved after applying our approach.

To summarize, we make following contributions.

- We design CNNCache, a transparent cache mechanism for CNN of mobile continuous vision (Section IV). The two core modules of CNNCache are a novel image matching algorithm for identifying similar regions among adjacent images in a video stream (Section V), and a cache-aware CNN inference engine that can reuse the computation results at layer granularity for each image frame (Section VI).
- We implement CNNCache to be transparent to app developers and run unmodified CNN models on commodity devices (Section VII).
- We comprehensively evaluate CNNCache via a variety of popular CNN models and real-world benchmarks (Section VIII). The results show that CNNCache can substantially benefit end-users in consideration of end-to-end latency and energy consumption with minimal accuracy loss.

The rest of paper is organized as follows. We survey the related work in Section II. We present the background about CNN and challenges of achieving layer-level caching in Section III. We conclude the paper and discuss possible future work in Section IX.

II. Related Work

Continuous Mobile Vision. Many mobile vision applications have been developed to assist end-users, spanning from industrial products [8], [4] to research prototypes [45], [64], [22], [28], [56], [32]. There are also projects aiming at optimizing mobile vision tasks. Starfish [39] allows concurrent vision applications to share computation and memory objects. RedEye [38] saves image sensor energy consumption via performing layers of a CNN in the analog domain before quantization. DeepMon [42] designs a suite of optimization techniques to efficiently offload CNN to mobile GPUs. DeepEye [43] makes a solid step to enable rich analysis of images in near real-time via a match-box sized collar or lapel-worn camera wearable. All these existing work are motivational to us.

DL Cloud Offloading. Running deep neural networks (DNN), especially deep CNN models for vision tasks, often requires large amount of computation resource. A common wisdom has been that commercial smartphones cannot support such high computational workloads with reasonable end-to-end latency and energy consumption. Thus, a traditional way of utilizing DL algorithms is offloading the execution procedure from mobile devices to high-end clouds. Many companies such as Google and Apple have already built powerful web services to support such offloading tasks from intelligent applications [2], [1].

In addition to the industrial efforts, some academic projects go further in some certain aspects. DjiNN [26] is a novel service infrastructure in warehouse scale computers (WSCs) specialized to handle large-scale DNN tasks offloaded from remote applications. Zhang et al. [58] propose a privacy-preserving deep model using the BGV encryption scheme to encrypt the private data and employing cloud servers to perform the high-order back-propagation algorithm on the encrypted data. Neurosurgeon [33] can automatically partition DNN computation between mobile devices and data-centers at the granularity of neural network layers. This partition strategy is based on the consideration of latency and energy consumption of mobile devices, and can adapt to different hardware specifications and other environment elements such as network connectivity. In addition, there are many work that targets at more generic offloading ideas [59], [51], [60].

Optimizing Mobile Execution. Optimization for mobile systems have gone for a long period in academia [41], [30], [40], [62], [54], [61]. Researchers have also been exploring several directions to make DL execution practical on mobile devices. Some efforts [37], [18], [50], [29] have proposed DL models that are much smaller than normal without sacrificing too much accuracy, so they can run directly on mobile devices. Some efforts [19], [57], [21], [24] aimed at building customized hardware architectural support for DL algorithms. Model compression [34], [36], [53], [23], [25] is another popular approach of accelerating DL and reducing the energy consumption. For example, DeepX [36] dramatically reduces the resource overhead by leveraging Runtime Layer Compression (RLC) and Deep Architecture Decomposition (DAD). Differently from the preceding work, CNNCache explores how to efficiently utilize the frame-by-frame similarity for mobile vision tasks. All the above techniques can be leveraged atop CNNCache to optimize the deep learning processing collaboratively.

Convolutional Layer Caching. DeepMon [42] and CBinfer [17] are the two most related efforts to CNNCache, as they explore a similar caching mechanism. However, they are both short in several aspects compared to CNNCache. First, they only match the image blocks (or pixels) to the same position from last frame. This limitation makes their approaches not feasible under some mobile scenarios when the mobile camera is not held stably or even moving. As a comparison, CNNCache adopts a novel image matching algorithm so that current image blocks can be matched to a more proper position of last frame. Second, DeepMon simply matches image blocks based on the histogram distribution and CBinfer uses pixel-level matching in each layer, which is either not accurate enough or causes too much overhead compared to our image matching algorithm presented in Section V. Third, neither DeepMon nor CBinfer makes a detailed design on how cached regions shall be propagated and altered during DL inference as discussed in Section VI. Overall, our work presents a deeper and more thorough study on this specific caching mechanism in CNN.

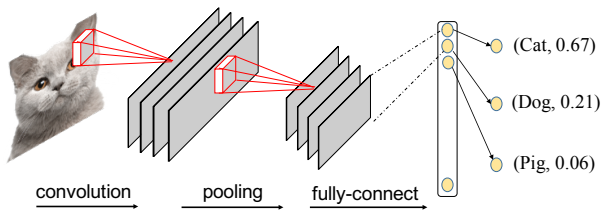


Fig. 1: A typical CNN model structure.

III. Background and Challenges

A. Convolutional Neural Network

Convolutional Neural Network (CNN) is the state-of-the-art algorithm in many computer vision tasks, and is recently adopted in many mobile scenarios [56], [15], [55], [45], [3], [43]. As shown in Figure 1, a typical CNN model repeatedly uses convolution and pooling layers to extract certain features from the whole image, and then applies fully-connected layers (fc) to finalize the classification task (or other vision tasks). The primary computation of CNN is in the convolutional layers (conv), where filters are applied on the input data (called feature map) to extract embedded visual characteristics and generate output data. The difference between fully-connected layers and convolutional (pooling) layers is that the former connects each units of input and output together, while the latter only extract simple features from input and capture local data properties.

B. Layer-level Cache in CNN and Challenges

CNNCache is based on a key observation that consecutively captured images often have substantial overlapped (similar) regions. The reason is that mobile devices, e.g., smartphones and head-mounted devices, are in slow motion or even held still especially when users are using these devices for vision tasks such as augmenting reality [42], [20]. Thus, CNNCache tries to cache the intermediate computation results of previous frames, and reuse the results of unchanged regions to accelerate the processing of current frame. Designing and implementing an efficient cache mechanism in CNN, however, is not easy because of the following challenges.

• **Dealing with inter-layer processing of CNN.** CNN models usually have many layers and complicated inter-layer processing which is regarded as a black-box to applications. To enable layer-level cache, deep understanding on how each layer is processed inside CNN execution engine and a careful design dealing with the complex intermediate results are required to make CNNCache effective.

• **Image variations.** Images consecutively captured in real world can have various aspects of differences for the presence of large variations in camera motion, object appearance, object scale, cluttered background, illumination conditions, etc. Those complicated conditions make it difficult to find out *"what should be reused and what should not"*.

• **Trade-off between performance and accuracy.** Cache reuse can inevitably result in accuracy loss for CNN models since image regions identified as similar (reusable) are not likely to be numerically identical. Proper trade-off between the performance (latency & energy) improvement and accuracy preservation is important to make CNNCache practical.

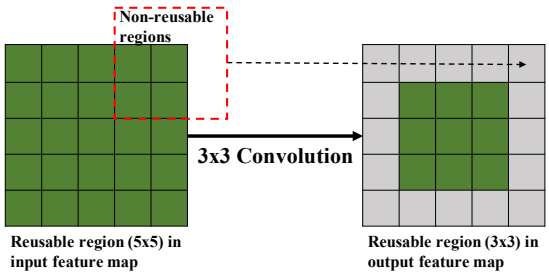


Fig. 2: An example showing how the reusable (cache-able) region is shrunk by convolution operation.

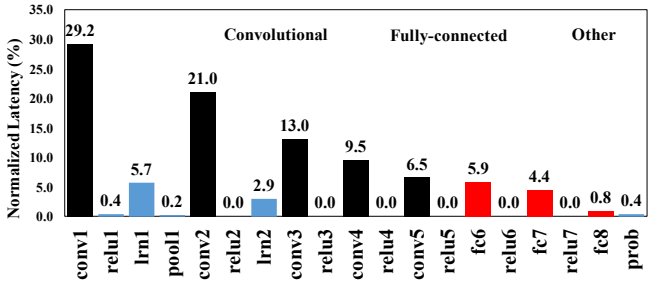


Fig. 3: Latency breakdown at layer granularity for AlexNet model. Layers are presented at the order of execution: left-side layer will be executed first and the output will be fed to the right-side layer as input.

• **Low processing overhead.** The extra overhead of CNNCache in identifying reusable image regions and applying the cached results in the CNN execution engine must be low enough so that the total processing latency can be improved by reusing the computational results inside CNN layers. Otherwise, CNNCache is not useful.

• **Cache erosion.** Due to the nature of CNN processing, an unchanged (reusable) image region may become "shrunk" as the execution progresses into further layers in a CNN model. Take convolutional layers as an example. As shown in Figure 2, the input feature map of a convolution layer (kernel=3x3, stride=1, padding=1) has a reusable region of 5x5 pixels. However, when the kernel reaches the edge of this reusable region as shown in the red dot-line box, we cannot reuse the previously computed result because the kernel refers to pixels outside the reusable region. Thus, the reusable region in the output feature map (i.e., the next layer) is shrunk to 3x3. Some other types of layers such as pooling and LRN can also result in a similar shrinking behavior as convolution. Even worse, fully-connect layer can totally destroy the regions to be reused since it breaks the data localization by producing output units based on each input unit. We call such a shrinking behavior as ***cache erosion***, referring to the phenomenon that an unchanged image region that we can cache and reuse between consecutive images becomes smaller during CNN inference procedure. As you might expect, a relatively large unchanged image region can vanish after repeated *cache erosion* of several layers, imposing a big challenge to CNNCache.

Fortunately, we have observed that for most popular CNN

models, layers at the beginning of model (mostly convolutional layers, pooling layers, and activation layers) contribute to the majority of computations, while fully-connect layers only reside at the end of models and contribute to a small part of computations. As a typical example, Figure 3 breaks down the processing time of each layer of AlexNet [35] model. As observed, fully-connected layers only reside at last few stages, contributing to around **11.5%** of the total latency. In other words, there are still plenty of room (**88.5%**) to be improved via reusing the cache of layers before fully-connected layers.

In the next sections, we describe how we address the above challenges through a fast image matching algorithm and a lightweight cache-aware CNN inference engine, including minimizing the *cache erosion* effect by identifying the largest possible reusable regions in image matching.

IV. System Design

The goal of our proposed CNNCache system is to maximize the reuse of computation results when running a specific CNN model. We guide the design of CNNCache with the following design principles.

• **No cloud offloading**. CNNCache targets at only using the local resources of a device without any remote offloading [2], [1] to process DL tasks, therefore no concern over the availability of network connectivity and privacy leak.

• **Minimal accuracy loss**. While cache reusing can help reduce processing time, it may also cause accuracy loss of DL tasks. CNNCache aims to carefully identify which part of results can be reused and make appropriate trade-off among the latency improvement and accuracy loss.

• **Applicable to any CNN model**. Off-the-shelf, pre-trained CNN models should be able to run directly on CNNCache without retraining or any tuning on the model structure or parameters.

• **Zero effort for app developers**. Developers do NOT need to write any code to support our cache mechanism in their applications. All caching details specifying what to reuse and how to reuse shall be totally hidden within CNNCache.

Figure 4 shows the architecture design and overall workflow of CNNCache. CNNCache runs in the user-space inside a continuous vision application by revising only the CNN part. All other components are the same to the original application, including loading the CNN model file from storage, reading continuous images (i.e., frames) from the camera, preprocessing the frames, running the CNN model on CPU and GPU, and returning the final model outputs to the application. This design makes CNNCache do its work entirely on the local device, support existing CNN models, and transparent to legacy applications.

To achieve a good trade-off between performance improvement and model accuracy loss, CNNCache employs two key components including a novel image matching algorithm and a cache-aware CNN inference engine.

• **Image matching algorithm.** When a new image frame captured and passed to CNNCache, it will first be matched to the previous frame. The goal of matching is to identify a similar block in the previous frame for each block in the current frame if any. The matching results can be represented as a set of rectangle-to-rectangle mappings, e.g., $(x_i, y_i, w, h) \rightarrow (x'_i, y'_i, w, h)$, where (x_i, y_i) $((x'_i, y'_i))$ is the left-top point in the current (previous) frame and w (h) is the width (height) of a certain rectangle. The algorithm details are shown in Section V.

• **Cache-aware CNN inference engine.** When the current frame has been matched to the last frame, the raw image data and the mappings of reusable blocks are sent to our cache-aware CNN inference engine together. The inference is performed as normal, except for two key differences. First, the mappings from reusable regions (current frame) to cached regions (last frame) are also propagated and changed among CNN layers just like input data. Second, the spatial convolution operation is customized so that it will skip calculating the regions that can be reused from cache but directly copy from corresponding cached region of the last frame. In addition, the output feature map of each convolutional layer will be cached until the inference of the next frame is done. Section VI explains the details of our cache mechanism.

Though CNNCache reuses only the computation of similar image blocks, there is still accuracy loss since the matched blocks may not be numerically identical. For two consecutive frames, the output disparity can be negligible. However, if the caching goes on for more frames, the accuracy loss might be non-ignorable. To mitigate the superposition of accuracy loss caused by reusing cache between two consecutive frames, CNNCache periodically clean its cache and calculates a whole new frame every $\mathcal{N}$ frames (default to 10).

When designing and developing the above two components, we have two important considerations in optimizing their performance.

Block-wise matching vs. pixel-wise matching. Theoretically, identifying each pixel's matching level (pixel-wise matching) and reusing its cached results can minimize the model accuracy loss. However, we have observed that even similar scenes in two overlapped images can have relatively low matching scores of corresponding pixels (pixel mutation). Those "unmatched" pixels can lead to significant reduction of cache reuse due to the *cache erosion* mentioned in Section III-B. Thus, we use block-wise matching rather than pixel-wise matching, taking a block (e.g., 8x8 pixels) as the basic unit to tell if it's successfully matched to a corresponding block in the previous image. In this way, a mutated pixel will not affect the block-wise matching decision if other surrounding pixels in the block are well matched. Section VIII-G evaluates the difference between these two strategies.

One-time matching vs. each-layer matching. Also due to the *cache erosion* mentioned in Section III-B, the unchanged regions detected on raw images can become smaller and even finally vanish when the CNN model goes deeper. In other words, the proportion of computations we can reuse in each layer tends to be smaller as inference goes on. One possible way to mitigate *cache erosion* is performing the block-matching not only at the input image, but before each convolution layer [17]. Though this approach can help increase the reuse of cache sometimes, our empirical development and experiments show matching every layer is not quite feasible or

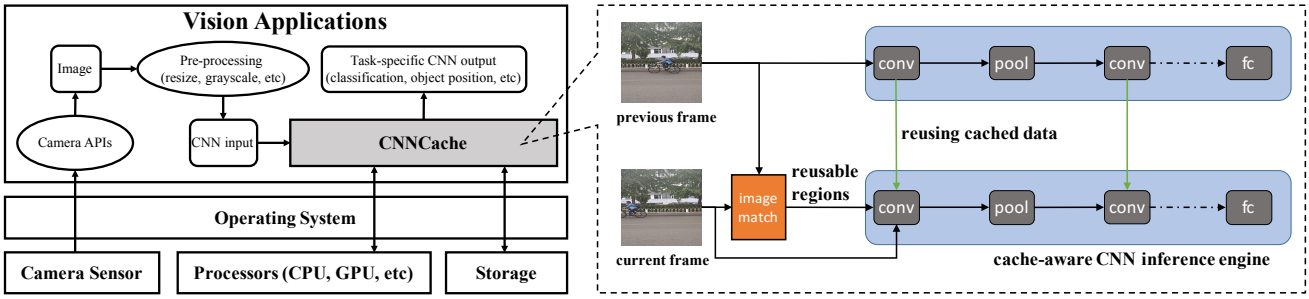


Fig. 4: Architecture design and overall workflow of CNNCache.

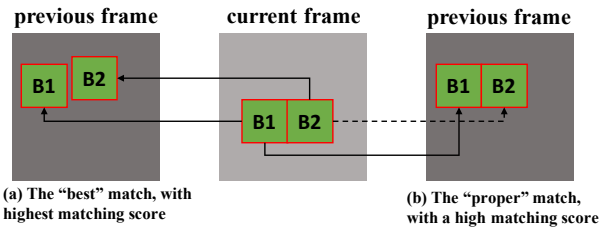


Fig. 5: Two matching examples, showing that the best matched block is not always what we want.

Fig. 6: Matched rectangles in two consecutive images via our proposed algorithm.

practical for two reasons: 1) the overhead of block matching in each layer can be higher than the gained benefit from cache reuse. 2) the threshold used to tell whether blocks are "similar enough to be reused" in the matching algorithm needs to be manually assigned for different layers. That is because, unlike input images that are all aligned in RGB format, the inputs (feature map) of different CNN layers are actually in different data dimensions with different semantics. This manual work is also model-specific and task-specific, thus substantial experimental efforts from application developers will be required. Given the above facts, CNNCache only matches the raw image for one time, and propagates the cached regions during the whole inference procedure.

V. Image Block Matching

The goal of our image matching algorithm is to find "similar" blocks (rectangles) between two images. Two principles should be considered into the design of our algorithm. First, the matching algorithm should run fast, keeping the processing overhead negligible compared to the improvement gained via cache reuse. Second, we want the resulted blocks to be likely merged into larger blocks. As shown in Figure 5, match(a) might have the highest matching scores for block B1 and B2, but it's not suitable in our cache mechanism since these small reusable blocks will quickly vanish after several layers due to *cache erosion* (Section III-B). Imagine that B1 and B2 have size 5x5, and the convolutional kernel is 3x3. After the *cache erosion*, the reusable regions become two 3x3 rectangles, 18 pixels in total. In comparison, match(b) finds two adjacent blocks in current frame that are similar to the blocks in previous frame, so that these two blocks can be merged into a larger one. In this case, the reusable region becomes one 3x10 rectangle after convolution, 30 pixels in total.

The overall flow of our matching algorithm is as follows.

• **Step 1**. The current frame (image) is divided into an NxN grid, where each grid block contains certain number of pixels.

• **Step 2**. For each divided grid block we find the most matched same-size block in previous frame. Here, we denote the left-top point of *i*-th block (i = 1 to N^2) in current frame as (x_i, y_i), and the corresponding matched block position in previous frame as (x_i', y_i'). We leverage the *diamond search* [63] algorithm which is widely used in video compression to quickly identify the most matched block. The matching level (similarity) between two image blocks is represented by the *PSNR* [10] metric: higher *PSNR* indicates that two blocks are more similar.

• **Step 3**. We calculate the average block movement (M_x, M_y) as the mean movement of the matched blocks whose *PSNR* is larger than the given threshold $\mathcal{T}$.

$$(M_x, M_y) = (\frac{\Sigma(x_i' - x_i)}{K}, \frac{\Sigma(y_i' - y_i)}{K}), \langle (x_i, y_i), (x_i', y_i') \rangle \in \mathcal{S}$$

where $\mathcal{S}$ is the collection of matched block pair whose PSNR is larger than $\mathcal{T}$, and K is the cardinality of $\mathcal{S}$.

• **Step 4**. For each block (x_i, y_i) in the current frame, we calculate its *PSNR* with block $(x_i + M_x, y_i + M_y)$ in the previous frame. If *PSNR* is larger than $\mathcal{T}$, these two blocks are considered to be properly matched.

• **Step 5**. We merge the small blocks that are properly matched in last step to larger ones. For example, if (x_i, y_i) and (x_j, y_j) in current frame are adjacent, then their matched blocks in Step 4 should also be adjacent since they share the same offset (M_x, M_y). Thus, we can directly merge them into a larger rectangle as well as their matched blocks.

Figure 6 shows an output example of applying our matching algorithm on two consecutively captured images. As observed, the second frame image is different from the first one in two

Layer Type	Layer Parameters	Output($\mathcal{D}_t$)
Convolution	kernel=k x k	$x' = \lceil (x+p)/s \rceil, y' = \lceil (y+p)/s \rceil$
Pooling	stride=s, padding=p	$w' = \lfloor (w-k)/s \rfloor, h' = \lfloor (h-k)/s \rfloor$
LRN [9]	radius=r	$x' = x + r, y' = y + r$ $w' = w - 2 * r, h' = h - 2 * r$
Concat [6]	input number=*N*	overlapped region of these *N* rectangles
Fully-connect	/	$(x', y', w', h') = (0, 0, 0, 0)$
Softmax [13]		
Others	/	$(x', y', w', h') = (x, y, w, h)$

TABLE I: How reusable regions are adjusted based on layer type ($\mathcal{D}_t$) with input rectangle (x, y, w, h).

aspects. First, the camera is moving, so the overall background also moves in certain direction. This movement is captured in Step 3 by looking into the movement of each small block and combining them together. Second, the objects in sight are also moving. Those moved objects (regions) should be detected and marked as non-reusable. This detection is achieved in Step 4.

Our experiments show that most of the processing time of the above matching algorithm is spent at Step 2 and Step 4. In Step 2, we need to explore the previous frame to identify the most matched block for every block in current image. We can accelerate this step by skipping some blocks in current frame, e.g., only matching blocks at $(i*k)$-th row and $(j*k)$-th column ($i*k, j*k \leqslant N$). Theoretically, a 2-skip (*k*=2) can save 75% of the computation time in this step, and a higher k can even achieve better improvements. However, a higher *k* might also result in inappropriately calculated (M_x, M_y), making the matched blocks in Step 4 fewer. We can further accelerate the computation of Step 4 by reusing the results in Step 2 since both of them need to calculate *PSNR* between two blocks. More specifically, if the *PSNR* between (x_i, y_i) (current frame) and $(x_i + M_x, y_i + M_y)$ (previous frame) is already calculated in Step 2, we simply reuse the result. We demonstrate the efficiency of our proposed algorithm as well as these acceleration approaches in Section VIII-G.

VI. Cache-aware CNN Inference Engine

To reuse the computation results inside CNN inference, CNNCache needs to identify which regions can be reused and where they are mapped to for each layer's output. As previously explained in Section III-B, the mappings obtained by matching raw images (Section V) need to be dynamically adjusted at inference runtime. This adjusting should also be performed on the corresponding cached blocks of previous frame. Obviously, the strategy about how reusable regions are adjusted is based on the forward implementation of different layer types. More specifically, a caching mapping $(x_i, y_i, w, h) \rightarrow (x'_i, y'_i, w, h)$ will be adjusted to $\mathcal{D}_t(x_i, y_i, w, h) \rightarrow \mathcal{D}_t(x'_i, y'_i, w, h)$ after operation *t*, where $\mathcal{D}_t$ indicates how a reusable region should be adjusted after going through a layer type *t*. We show the design details of $\mathcal{D}_t$ for every layer type in Table I.

Figure 7 shows an illustrating example about how a reusable region is propagated among different layers. The current raw image has been matched to previous image,

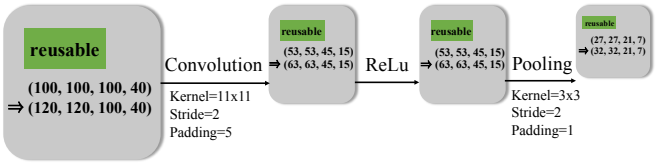


Fig. 7: An example showing how the mappings obtained by image matching are adjusted during the CNN inference.

Module	Language	LoC
Caching inside *ncnn*	C/C++	3,620
Image matching	Java, rs	410
Total		*4,030*

TABLE II: LoC (Lines of Code) of CNNCache.

and a block $(100, 100, 100, 40)$ (left-top point=$\langle 100, 100 \rangle$, width=100, height=40) is identified to be similar to the block $(120, 120, 100, 40)$ of last frame. This image is the input of a convolutional layer, with kernel=11x11, stride=2, and padding=5. The reusable region of computational output of this layer can be calculated as $(53, 53, 45, 15)$. This output is passed to an activation layer (ReLu) as input, but the reusable region is not changed since the activation layer performs just a certain activation function on every single input unit. Then, the output of ReLu is consumed by a pooling layer, with kernel=3x3, stride=2, padding=1. Similar to the convolutional layer, the reusable region becomes smaller due to the kernel padding.

After knowing which regions can be reused from cache, CNNCache customizes the convolutional forward so that the convolution computation of these reusable regions are skipped but directly copied from the corresponding cached region. When customizing convolution operations, it's important to achieve good a data locality since data movement is one of the computational bottlenecks [21] in the convolution processing. To this end, CNNCache splits the convolution operation into three steps. First, reusable regions are directly copied from cache of last frame. Second, a boolean map is created to specify whether a pixel (x, y) is already cached. Third, kernel travels on the input feature map and perform convolution only on the non-reusable pixels but skips reusable pixels.

CNNCache caches and reuses the computational results only in convolutional layers for two reasons. 1) As mentioned in Section III, convolution is often the dominant layer in inference time (e.g, 79.2% for AlexNet). 2) Caching the intermediate output for other layer types (e.g., pooling) requires additional memory overhead. In other words, CNNCache supports caching reuse only in convolution layers to make proper trade-off among latency improvement and memory overhead. But it's worth mentioning that we can easily extend this cache mechanism to other layer types.

VII. Implementation

We use RenderScript [11] to implement and accelerate our proposed image matching algorithm (Section V). RenderScript is coded in a C-like language and designed for accelerate computational-intensive, especially high data-parallel, tasks via mobile GPUs. In addition, RenderScript is a generic frame-

Model	Application	Output	Layers
AlexNet [35]	image classification	1x1000 vector, specifying the possibility of each class	25
GoogLeNet [49]			153
ResNet-50 [27]			246
YOLO [46]	object detection	7x7x30 tensor, specifying detected positions, categories, and confidences	58

TABLE III: Popular CNN models used to evaluate CNNCache.

work supported by all Android devices, without depending on any vendor-specific libraries.

We implement our cache-aware CNN inference engine based on a state-of-the-art DL framework *ncnn* [14], an open-source high-performance neural network inference framework optimized for the mobile platform, and already integrated in many popular apps such as WeChat and QQ [1]. Inside *ncnn* there is a corresponding implementation function for each supported layer type named `forward(top_blob, bottom_blob)`, where `top_blob` and `bottom_blob` stand for the output and input of this forward step respectively. We replace this `forward` function with our customized `c_forward(top_blob, bottom_blob, c_blob, c_regions)`, where `c_blob` stores the computation results of current layer from the last frame, and `c_regions` specifies which parts can be reused. `c_forward` calculates the output just as `forward` does, except that `c_forward` skips the calculation of cached regions but copies from `c_blob` directly. Before `c_forward` invoked, `c_regions` will be propagated from last layer. As mentioned in Section VI, cached regions will be compressed (conv, pooling) or vanished (full-connect) during the inference process, thus we use another function named `reg_forward` which calculates how cached regions are propagated among different layers. Currently, we only support `c_forward` for convolutional layer, since it dominants the processing time of overall CNN network as we have already shown. But we can easily extend the implementation of this caching mechanism to other layer types as well. Overall, our implemented system contains 4,030 lines of code as illustrated in Table II.

VIII. Evaluation

In this section we comprehensively evaluate CNNCache using 4 popular CNN models from multiple aspects.

A. Experimental Setup

Devices. We use Nexus 6 (Qualcomm 2.7 GHz quad-core CPU, Adreno 420 GPU) with Android 6.0 as the testing platform.

Workloads. We used a variety of popular CNN models to verify CNNCache as shown in Table III. Input images are all resized to 227x227 before feeding into these models. YOLO model is trained via Pascal VOC 2007 dataset [16], and other classification models are trained via ILSVRC 2012 dataset [47]. It is worth mentioning that these CNN models are quite generalized and can be used in many different tasks with few customization efforts.

[1]WeChat and QQ are two dominant social apps in China with billions of users all over the world.

Benchmarks. We use the UCF101 dataset [48], which contains 101 types of human activities and 13,421 short videos ($<$ one minute) created for activity recognition, as our test data. We randomly select 10 types from these activities and evaluate CNNCache across them: *Basketball* (**T1**), *ApplyEyeMakeup* (**T2**), *CleanAndJerk* (**T3**), *Billiards* (**T4**), *BandMarching* (**T5**), *ApplyLipstick* (**T6**), *CliffDiving* (**T7**), *BrushingTeeth* (**T8**), *BlowDryHair* (**T9**), and *BalanceBeam* (**T10**). In total, **55,680** images have been processed in our evaluation for each selected CNN model.

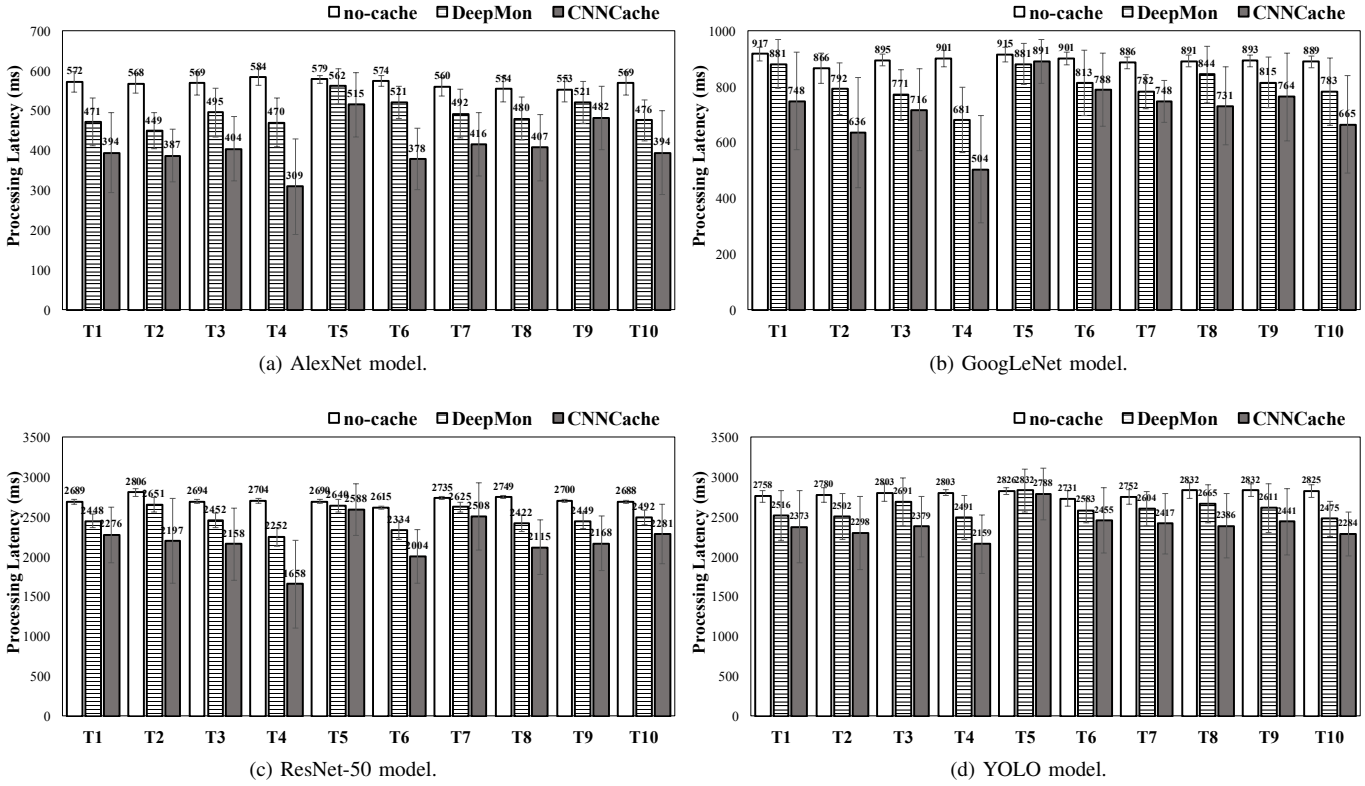
Metrics. We use accuracy, processing latency, and power consumption as our key evaluation metrics. For **accuracy**, we treat the output of exhaustively running complete model without cache mechanism as ground truth. Then we use two metrics to measure the accuracy loss of CNNCache: Euclidean distance (abbr. *ED*) [7] and top-k accuracy. *ED* is calculated as the distance between two points in n-space, taking every output number into consideration. But in some classification applications, we only care the top ranks from output. Thus, top-k accuracy is used in image classification tasks to capture how accurate the most confident class can be identified. Both metrics are measured between CNNCache and the ground truth (*no-cache*). For **latency**, we log the starting time when CNNCache receives the image and ending time when CNNCache outputs the inference result. The subtracted duration is reported as the processing latency, including the time spent on image matching and CNN inference. Finally, we measure the **energy consumption** via Qualcomm Snapdragon Profiler [12]. The baseline of phone's idle state is always subtracted.

Alternatives. We compare the performance of CNNCache to two alternative approaches: *no-cache* and *DeepMon* [42]. *no-cache* means running the complete model without cache reuse (ground truth used in measuring accuracy). *DeepMon* is the state-of-the-art framework that implements a relatively naive cache mechanism. We compare the approaches of *DeepMon* and CNNCache in Section II.

CNNCache settings. If not otherwise specified, we use a default block size of 10x10, and the matching threshold $\mathcal{T}$ of 20 in our image matching algorithm (Section V). In addition, the expiration time N of cache is set as 10 (Section IV).

B. Latency Improvement

Figure 8 shows the overall processing latency of CNNCache, *no-cache*, and *DeepMon*. Our primary observation is that applying CNNCache can have substantial latency improvement compared to *no-cache*, e.g., **20.2%** on average, while *DeepMon* has only **9.7%**. This improvement varies across different CNN models and benchmark scenarios. For a relatively small model AlexNet (5 convolutions, 25 layers in total), CNNCache results in **28.1%** saving up of total processing time on average, while *DeepMon* only has **13.1%** improvement. For a deeper model GoogLeNet (57 convolutions, 153 layers in total), the benefit from CNNCache reduces to **19.7%**, while *DeepMon* has only **10.2%**. For YOLO, CNNCache can have only **14.2%** latency improvement. The reason is that, differently from other classification models, YOLO is applied in object detection applications and outputs location-



(a) AlexNet model.

(b) GoogLeNet model.

(c) ResNet-50 model.

(d) YOLO model.

Fig. 8: Processing latency of CNNCache compared to *no-cache* and *DeepMon*.

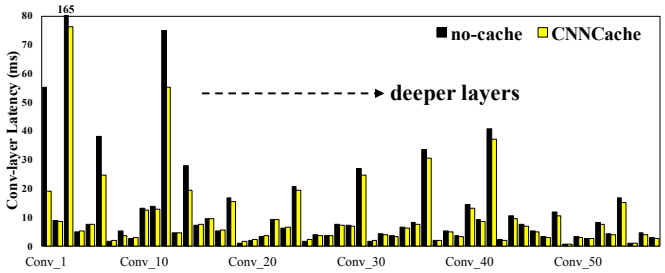


Fig. 9: Processing time of each individual convolutional layers in GoogLeNet.

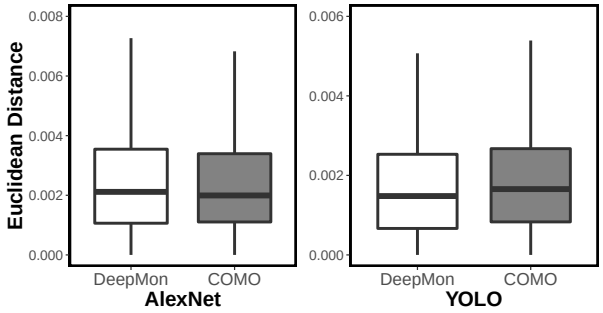


Fig. 10: Euclidean distance between the output of caching approaches (CNNCache, *DeepMon*) and ground truth (*no-cache*).

sensitive information. Thus, many computation-intensive fully-connected layers reside at the end of YOLO, making the benefit from convolution-layer cache smaller.

We also observe that the performance of CNNCache can differ a lot under different benchmarks. Taking AlexNet as an instance, CNNCache can save up to **47.1%** processing time under *Billiards* (**T4**) scenarios. We manually check the dataset videos and identify the reasons of such high speedup as following: 1) camera is held still or in very slow motion, 2) most objects are still except the player and balls, 3) indoor lighting is stable. In comparison, CNNCache has only **11.0%** latency improvement when processing *BandMarching* (**T5**) videos because the camera and most objects (people) in view are moving brokenly.

We further dig into the achieved improvement at each individual convolutional layer. As shown in Figure 9, the latency improvement mainly comes from the first few layers due to the *cache erosion* mentioned previously. Fortunately, these layers often contribute to the majority of overall latency, indicating that the benefit remains meaningful when models grow deeper.

C. Accuracy Loss

We then investigate how much accuracy CNNCache compromises in return for the latency benefits. Figure 10 shows the Euclidean distance (*ED*) between ground truth (*no-cache*) and other cache approaches (CNNCache and *DeepMon*) when running AlexNet and YOLO models. As observed, the median *ED* of CNNCache is **0.0021** and **0.0016** for AlexNet and YOLO respectively, quite similar to the results of *DeepMon* with **0.0023** and **0.0015**. To be compared, our above latency

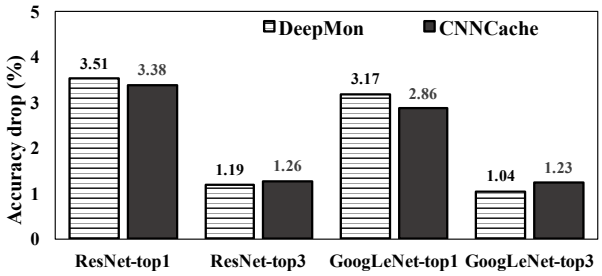


Fig. 11: Top-k accuracy drop of CNNCache.

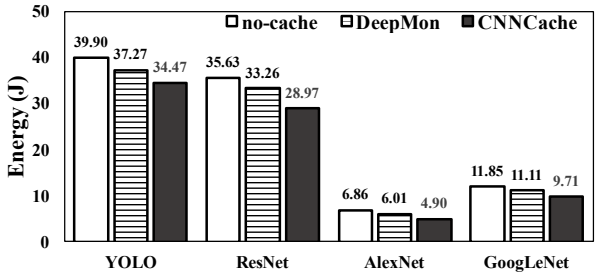


Fig. 12: Energy consumption of CNNCache.

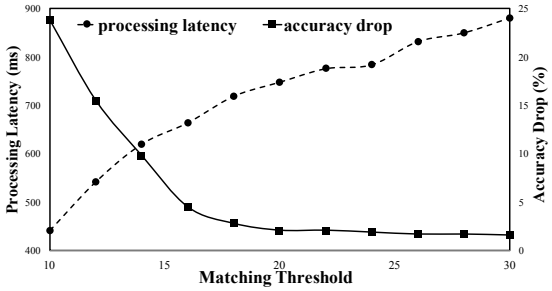


Fig. 13: Effect of varied matching threshold $\mathcal{T}$ on processing latency and top-1 accuracy drop of GoogLeNet model.

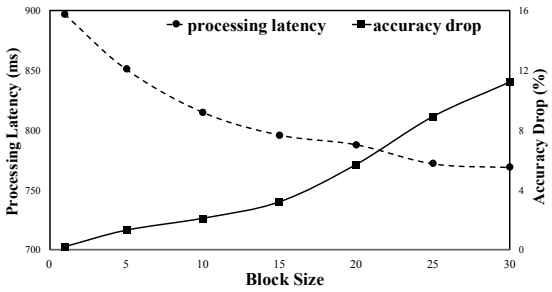


Fig. 14: Effect of varied block size on processing latency and top-1 accuracy drop of GoogLeNet model.

experiment shows that CNNCache can accelerate CNN models two times as *DeepMon*, e.g., **20.2%** vs. **9.7%** on average across all models and benchmarks.

Besides using *ED* to evaluate how CNNCache's output is similar to ground truth, we also test whether CNNCache can accurately classify the images (top-k accuracy). Since the dataset UCF101 used in our experiment is not built for image classification tasks, many images cannot be classified into any of the classification pre-defined in our training data (ILSVRC2012). To make the results more practically meaningful, we only keep the images with a relatively reasonable confidence ($\geqslant 0.3$) in ground truth and filter others.

The top-k accuracy drop is shown in Figure 11. CNNCache and *DeepMon* both have very small accuracy drop ($\leqslant$ **3.51%** for top-1 and $\leqslant$ **1.26%** for top-3). This is because that we have designed our image matching algorithm to carefully choose which part of computations to reuse, and these reusable information is properly propagated during inference, thus minimizing the impact on the recognition output.

D. Energy Saving

We now investigate the energy consumption of CNNCache across all selected benchmarks, and illustrate results in Figure 12. It is observed that CNNCache can save **19.7%** of energy consumption on average and up to **28.6%** (AlexNet), while *DeepMon* only has **8.0%** on average. This saving is mostly from the reduced processing time. Considering that vision tasks are very energy-intensive, this saving up is able to substantially lengthen battery life. For example, applying CNNCache on ResNet-50 to classify 10 images can help spare 66.8J energy, equaling to 40 seconds of video playing on Nexus 6 phone according to our measurement.

E. Parameters Choosing

In our matching algorithm mentioned in Section V, some variables can be used to make trade-off between latency improvement and accuracy drop. Matching threshold $\mathcal{T}$ is the key to decide whether two image blocks are similar enough to be reused. Figure 13 illustrates how $\mathcal{T}$ can affect the latency and accuracy (GoogLeNet + **T1**). As expected, higher $\mathcal{T}$ indicates fewer blocks can be matched, thus leading to less top-1 accuracy drop, but also higher processing latency. In our default setting ($\mathcal{T} = 20$), CNNCache can achieve considerable latency improvement, e.g., **18.3%** (from **917ms** to **748ms**), with acceptable accuracy loss (**2.1%**). This setting aligns with the fact that the acceptable values for wireless transmission quality loss are commonly considered to be about 20 to 25 [10]. However, the threshold can also be set by application developers to adapt to task-specific requirements. For applications that are not very sensitive to the output accuracy, developers can aggressively use a smaller $\mathcal{T}$ to achieve higher latency improvement.

Another configurable parameter in our image matching algorithm is the block size. As observed from Figure 14, a larger block size results in more latency improvement but also higher accuracy loss. This result is reasonable since splitting an image into large blocks indicates more coarse-grained matching. As an extreme case, when block size equals to 1, the accuracy loss is very small (**0.2**) but the latency improvement is also very low (**2.19%**). This is actually the *pixel-wise* approach discussed previously in Section IV, and the result is consistent with our discussion. Our empirical suggestion is setting block size around 10 for 227x227 images.

F. Memory Overhead

Figure 15 shows the memory overhead of CNNCache. Besides the 4 models used above, we also test on other three popular CNN models: MobileNet [29], SqueezeNet [31], and DeepFace [52]. Here we assume that all model parameters are

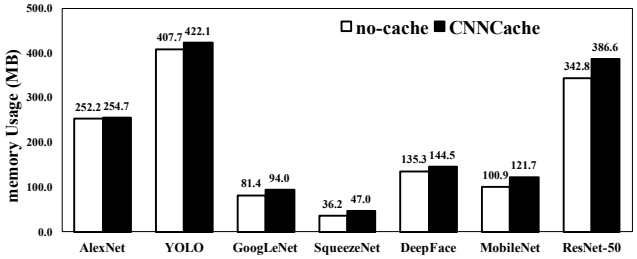


Fig. 15: Memory overhead of CNNCache.

	Basic	Basic+2-skip	Basic+2-skip+reusing
Latency (ms)	25.9 $\pm$ 6.15	12.4 $\pm$ 3.81	9.7 $\pm$ 3.35
Match Ratio	0.731	0.712	0.712

TABLE IV: Efficiency of proposed image matching algorithm.

read into memory once initialized without I/O transmission during the inference. As observed, the memory overhead occurred by CNNCache ranges from **2.5MB** to **43.8MB** depending on the internal structure of models. However, this overhead is quite trivial since nowadays mobile devices are usually equipped with large size of memory, e.g., 3GB in Nexus 6.

G. Image Matching Performance

Finally, we report the performance of our image matching algorithm individually. As shown in Table IV, the acceleration techniques mentioned in Section V can highly improve the processing latency from **25.9ms** to **9.7ms** on average, with only **0.019** loss in match ratio. This results indicate that our image matching algorithm is feasible for our CNN cache mechanism, as it occurs negligible overhead ($\leqslant$ **10ms**) compared to the benefit gained from cache reusing.

IX. Conclusion and future work

In this paper, we have proposed CNNCache to accelerate the execution of CNN models via caching and reusing similar image regions under continuous vision scenarios. We demonstrate two novel techniques to accomplish such caching mechanism for CNN: an image matching algorithm that identifies similar regions between two consecutively captured images, and a cache-aware CNN inference engine that propagates reusable regions among different layers and reuse the computation results. We implement CNNCache to run on commodity Android devices, and comprehensively evaluate its effectiveness via a set of experiments on typical CNN models.

Currently we have tested CNNCache on only Nexus 6 device and 4 popular CNN models. Though our proposed caching technique is quite generalized, we plan to apply it on more devices (e.g., head-mount glasses, iPhone, etc) and more CNN models (e.g., VGG, RCNN, etc). For now we focus on optimizing vision tasks, however, our caching mechanism is generalizable in other deep learning applications that receives similar consecutive inputs. We plan to explore the possibility of applying our technique on other fields such as *NLP* and audio processing. In addition, we propose to leverage Android video encoding/decoding hardware [5] to further accelerate the block matching algorithm mentioned in Section V.

References

[1] Apple moves to third-generation Siri back-end, built on open-source Mesos platform. https://9to5mac.com/2015/04/27/siri-backend-mesos/, 2016.

[2] Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, 2016.

[3] TensorZoom App. https://play.google.com/store/apps/details?id=uk.tensorzoom&hl=en, 2016.

[4] Amazon App. https://itunes.apple.com/us/app/amazon-app-shop-scan-compare/id297606951?mt=8, 2017.

[5] Android MediaCodec. https://developer.android.com/reference/android/media/MediaCodec.html, 2017.

[6] Concat Layer. http://caffe.berkeleyvision.org/tutorial/layers/concat.html, 2017.

[7] Euclidean distance. https://en.wikipedia.org/wiki/Euclidean_distance, 2017.

[8] Google Translate App. https://play.google.com/store/apps/details?id=com.google.android.apps.translate&hl=en, 2017.

[9] Local Response Normalization (LRN). http://caffe.berkeleyvision.org/tutorial/layers/lrn.html, 2017.

[10] Peak signal-to-noise ratio. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio, 2017.

[11] RenderScript. https://developer.android.com/guide/topics/renderscript/compute.html, 2017.

[12] Snapdragon Profiler. https://developer.qualcomm.com/software/snapdragon-profiler, 2017.

[13] Softmax Layer. http://caffe.berkeleyvision.org/tutorial/layers/softmax.html, 2017.

[14] Tencent ncnn deep learning framework. https://github.com/Tencent/ncnn, 2017.

[15] TensorFlow Android Camera Demo. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android, 2017.

[16] The PASCAL Visual Object Classes. http://host.robots.ox.ac.uk/pascal/VOC/, 2017.

[17] L. Cavigelli, P. Degen, and L. Benini. Cbinfer: Change-based inference for convolutional neural networks on video data. *arXiv preprint arXiv:1704.04313*, 2017.

[18] G. Chen, C. Parada, and G. Heigold. Small-footprint keyword spotting using deep neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*, pages 4087–4091, 2014.

[19] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, pages 269–284, 2014.

[20] T. Y. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys'15)*, pages 155–168, 2015.

[21] Y. Chen, J. S. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'16)*, pages 367–379, 2016.

[22] A. Das, M. Degeling, X. Wang, J. Wang, N. M. Sadeh, and M. Satyanarayanan. Assisting users in a world full of cameras: A privacy-aware infrastructure for computer vision applications. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR'17*, pages 1387–1396, 2017.

[23] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'14)*, pages 1269–1277, 2014.

[24] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'16)*, pages 243–254, 2016.

[25] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: an approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*, pages 123–136, 2016.

[26] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. N. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: DNN as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 27–40, 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR'16*, pages 770–778, 2016.

[28] S. Hodges, L. Williams, E. Berry, S. Izadi, J. Srinivasan, A. Butler, G. Smyth, N. Kapur, and K. R. Wood. Sensecam: A retrospective memory aid. In *UbiComp 2006: Ubiquitous Computing, 8th International Conference, UbiComp 2006, Orange County, CA, USA, September 17-21, 2006*, pages 177–193, 2006.

[29] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[30] G. Huang, M. Xu, F. X. Lin, Y. Liu, Y. Ma, S. Pushp, and X. Liu. Shuffledog: Characterizing and adapting user-perceived latency of android apps. *IEEE Transactions on Mobile Computing*, 2017.

[31] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[32] P. Jain, J. Manweiler, and R. R. Choudhury. Overlay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, pages 331–344, 2015.

[33] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. N. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, pages 615–629, 2017.

[34] K. Katevas, I. Leontiadis, M. Pielot, and J. Serrà. Practical processing of mobile sensor data for continual deep learning predictions. In *Proceedings of the 1st International Workshop on Embedded and Mobile Deep Learning (Deep Learning for Mobile Systems and Applications) (EMDL@MobiSys'17)*, pages 19–24, 2017.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS'12)*, pages 1106–1114, 2012.

[36] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2016)*, pages 23:1–23:12, 2016.

[37] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'15)*, pages 283–294, 2015.

[38] R. LiKamWa, Y. Hou, Y. Gao, M. Polansky, and L. Zhong. Redeye: Analog convnet image sensor architecture for continuous mobile vision. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture ISCA'16*, pages 255–266, 2016.

[39] R. LiKamWa and L. Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, pages 213–226, 2015.

[40] X. Liu, Y. Ma, Y. Liu, X. Wang, T. Xie, and G. Huang. Swarovsky: Optimizing resource loading for mobile web browsing. *IEEE Transactions on Mobile Computing*, 2016.

[41] X. Liu, Y. Ma, M. Yu, Y. Liu, G. Huang, and H. Mei. i-jacob: An internetware-oriented approach to optimizing computation-intensive mobile web browsing. *ACM Trans. Internet Techn.*, page Accepted to appear, 2017.

[42] H. N. Loc, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pages 82–95, 2017.

[43] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pages 68–81, 2017.

[44] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pages 292–305, 2017.

[45] Z. Ou, C. Lin, M. Song, and H. E. A cnn-based supermarket autocounting system. In *Proceedings of the 17th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'17)*, pages 359–371, 2017.

[46] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 779–788, 2016.

[47] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[48] K. Soomro, A. R. Zamir, and M. Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*, pages 1–9, 2015.

[50] E. Variani, X. Lei, E. McDermott, I. Lopez-Moreno, and J. Gonzalez-Dominguez. Deep deural detworks for small footprint text-dependent speaker verification. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*, pages 4052–4056, 2014.

[51] X. Wang, X. Liu, Y. Zhang, and G. Huang. Migration and execution of javascript applications between mobile devices and cloud. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 83–84, 2012.

[52] Y. Wen, K. Zhang, Z. Li, and Y. Qiao. A discriminative feature learning approach for deep face recognition. In *Proceedings of the 14th European Conference on Computer Vision (ECCV'16)*, pages 499–515, 2016.

[53] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, (CVPR'16)*, pages 4820–4828, 2016.

[54] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu. Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 143–152, 2017.

[55] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. F. Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web, (WWW'17)*, pages 351–360, 2017.

[56] X. Zeng, K. Cao, and M. Zhang. *MobileDeepPill*: A small-footprint mobile deep learning system for recognizing unconstrained pill images. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pages 56–67, 2017.

[57] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*, pages 161–170, 2015.

[58] Q. Zhang, L. T. Yang, and Z. Chen. Privacy preserving deep computation model on cloud for big data feature learning. *IEEE Trans. Computers*, pages 1351–1362, 2016.

[59] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 233–248, 2012.

[60] Y. Zhang, G. Huang, W. Zhang, X. Liu, and H. Mei. Towards module-based automatic partitioning of java applications. *Frontiers of Computer Science*, 6(6):725–740, 2012.

[61] Y. Zhang, Y. Liu, X. Liu, and Q. Li. Enabling accurate and efficient modeling-based CPU power estimation for smartphones. In *25th IEEE/ACM International Symposium on Quality of Service, IWQoS 2017, Vilanova i la Geltrú, Spain, June 14-16, 2017*, pages 1–10, 2017.

[62] Y. Zhang, X. Wang, X. Liu, Y. Liu, L. Zhuang, and F. Zhao. Towards better CPU power management on multicore smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems,*

HotPower 2013, Farmington, Pennsylvania, USA, November 3-6, 2013, pages 11:1–11:5, 2013.

[63] S. Zhu and K.-K. Ma. A new diamond search algorithm for fast block matching motion estimation. In *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, pages 292–296, 1997.

[64] Y. Zhu, Y. Yao, B. Y. Zhao, and H. Zheng. Object recognition and navigation using a single networking device. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pages 265–277, 2017.