Incremental and Approximate Computations for Accelerating Deep CNN Inference

SUPUN NAKANDALA, KABIR NAGRECHA, ARUN KUMAR, and
YANNIS PAPAKONSTANTINOU, University of California, San Diego

Deep learning now offers state-of-the-art accuracy for many prediction tasks. A form of deep learning called deep convolutional neural networks (CNNs) are especially popular on image, video, and time series data. Due to its high computational cost, CNN inference is often a bottleneck in analytics tasks on such data. Thus, a lot of work in the computer architecture, systems, and compilers communities study how to make CNN inference faster. In this work, we show that by elevating the abstraction level and re-imagining CNN inference as *queries*, we can bring to bear database-style query optimization techniques to improve CNN inference efficiency. We focus on tasks that perform CNN inference *repeatedly* on inputs that are only *slightly different*. We identify two popular CNN tasks with this behavior: *occlusion-based explanations* (OBE) and *object recognition in videos* (ORV). OBE is a popular method for "explaining" CNN predictions. It outputs a heatmap over the input to show which regions (e.g., image pixels) mattered most for a given prediction. It leads to many re-inference requests on locally modified inputs. ORV uses CNNs to identify and track objects across video frames. It also leads to many re-inference requests. We cast such tasks in a unified manner as a novel instance of the *incremental view maintenance* problem and create a comprehensive algebraic framework for incremental CNN inference that reduces computational costs. We produce *materialized views* of features produced inside a CNN and connect them with a novel *multi-query optimization* scheme for CNN re-inference. Finally, we also devise novel OBE-specific and ORV-specific approximate inference optimizations exploiting their semantics. We prototype our ideas in Python to create a tool called Krypton that supports both CPUs and GPUs. Experiments with real data and CNNs show that Krypton reduces runtimes by up to 5× (respectively, 35×) to produce exact (respectively, high-quality approximate) results without raising resource requirements.

CCS Concepts: • **Information systems → Query optimization**; **Database views**; **Data analytics**; • **Computing methodologies → Neural networks**;

Additional Key Words and Phrases: Incremental view maintenance, multi-query optimization, convolutional neural network explainability, systems for machine learning

ACM Reference format:
Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. 2020. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM Trans. Database Syst.* 45, 4, Article 16 (December 2020), 42 pages.
https://doi.org/10.1145/3397461

This work was supported in part by a Hellman Fellowship and by the NIDDK of the NIH under award number R01DK114945. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

Authors' addresses: S. Nakandala, K. Nagrecha, A. Kumar, and Y. Papakonstantinou, Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, Mail Code 0404, La Jolla, CA 92093-0404, USA; emails: snakanda@eng.ucsd.edu, knagrech@ucsd.edu, {arunkk, yannis}@eng.ucsd.edu.

0362-5915/2020/12-ART16 $15.00
https://doi.org/10.1145/3397461

1 INTRODUCTION

Deep Convolutional Neural Networks (CNNs) are now the state-of-the-art method for many prediction tasks on images, video, and time series data [1]. Thus, there is growing adoption of deep CNNs in numerous application domains such as healthcare [2, 3], agriculture [4], image search and recommendation systems [5], species monitoring [6], security [7], and sociology [8]. However, CNN inference is compute-intensive and time consuming. For example, inference using the popular VGG16 [9] CNN model requires 15 GFLOPs of computations. Furthermore, many analytics tasks involving CNNs perform repeated CNN inference, amplifying the computational cost and raising latency. This makes the adoption of CNNs unwieldy for interactive and/or resource-constrained settings as mobile, browser, and edge devices, while potentially raising resource costs in regular server and cloud settings.

In this work, we show that by re-imagining CNN inference as "queries," we can devise classical database-inspired query optimization techniques to reduce the computational cost of CNN inference in some popular analytics tasks. Specifically, we dive deeper into two tasks that perform *repeated* CNN re-inference on *slightly modified* inputs: (1) *occlusion-based explanations* (OBE) [10] and (2) *object recognition in videos* (ORV).

1.1 Occlusion-Based Explanations (OBE)

A key criticism of CNNs is that their internal workings are unintuitive to non-technical users. Thus, users often seek an "explanation" for why a CNN predicted a certain label. Explanations can help users trust CNNs [11], especially in high-stakes applications such as radiology [12], and are a legal requirement for machine learning applications in some countries [13, 14].

How to explain a CNN prediction is still an active research question, but in the practical literature OBE is a widely used method. OBE works as follows: Place a small square patch on the image to occlude those pixels. Rerun CNN inference, as Figure 1(b) illustrates, on the occluded image. The predicted class probability will change. Repeat this process by moving the patch across the image to obtain a sensitivity *heatmap* of probability changes, as Figure 1(c) shows. This heatmap highlights regions of the input that were highly "responsible" for the output (red/orange color regions). Such localization of regions of interest allows users to gain intuition on what "mattered" for a prediction. For instance, the heatmap can highlight diseased areas of a tissue image, which a radiologist can then inspect more deeply for further tests. Overall, OBE is popular because it is easy for non-technical users to understand.

1.2 Object Recognition in Videos (ORV)

Our second task applies CNN inference to video monitoring and analytics. CNNs are the state-of-the-art method to perform object recognition in videos. ORV is gaining popularity due to the mass deployment of video cameras in applications such as security surveillance [15], traffic monitoring [16], and tracking animal species in the wild [6]. An example use case from a trail camera video is shown in Figure 2. In ORV, each video frame is treated as an individual image. A trained CNN performed inference to identify the object. In this work, we focus specifically on *single-object* recognition over *fixed-angle camera* video feeds, a common setting in video monitoring applications such as species monitoring and surveillance security.

1.3 Problem: Compute-intensive, Slow, and Redundant Re-inference

Deep CNN inference is already computationally expensive; OBE and ORV just amplify that issue by creating a large number of CNN re-inference requests, even tens of thousands. However, we make a crucial observation about such tasks: *much of the re-inference computations are largely redundant,*

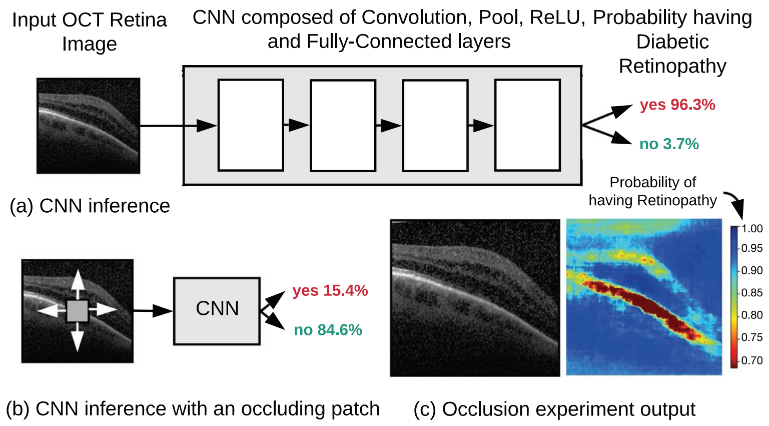

Input OCT Retina Image
CNN composed of Convolution, Pool, ReLU, and Fully-Connected layers
Probability having Diabetic Retinopathy
yes 96.3%
no 3.7%
(a) CNN inference
Probability of having Retinopathy
CNN
yes 15.4%
no 84.6%
1.00
0.95
0.90
0.85
0.80
0.75
0.70
(b) CNN inference with an occluding patch
(c) Occlusion experiment output

Fig. 1. (a) Using a CNN to predict diabetic retinopathy in an OCT image/scan. (b) Occluding a part of the image changes the prediction probability. (c) By moving the occluding patch, a sensitivity heatmap can be produced.

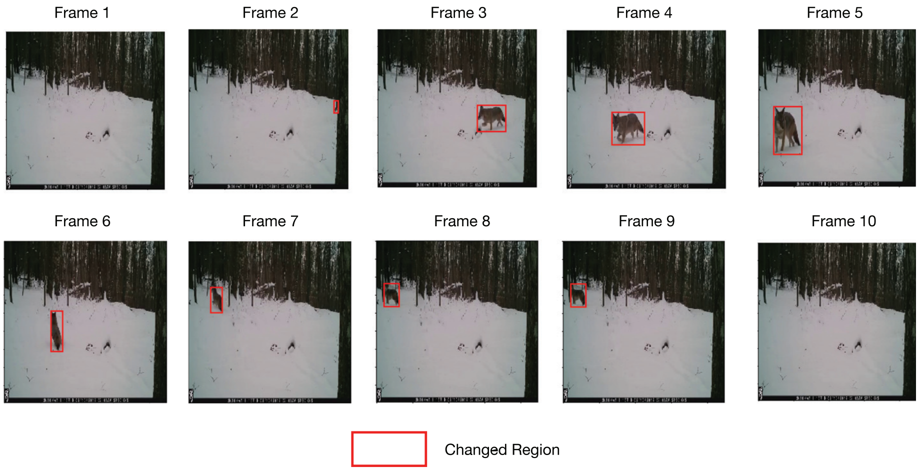


Fig. 2. Sample video frames obtained from a trail camera video (sampled at a rate of 1 frame per second). First frame shows the stationary background. Subsequent frames show the arrival of an animal into the scene and the corresponding changed region. A CNN can be trained to correctly recognize animals in video frames.

because the inputs differ only slightly. For example, Reference [17] reports 500,000 re-inference requests for performing OBE on a single image. Naively performing re-inference takes 1hr even on a GPU! Such long wait times can hinder users' ability to consume the explanations and reduce user productivity. But note that most of the pixels across occluded images are identical. Likewise, in ORV the number of re-inference requests issued is proportional to the length of the video. But note that most of the pixels across adjacent frames will be almost identical due to temporal locality. One simple way to reduce runtimes in such tasks is to throw more compute hardware at it, if possible, since OBE and offline ORV are embarrassingly parallel across re-inference requests. But this is not feasible for online ORV. Moreover, extra compute hardware may not be affordable, especially for domain scientists, or even feasible in all settings, e.g., in mobile clinical diagnosis or edge deployments. Throwing more resources at massively redundant computations also wastes money, especially in pay-as-you-go cloud environments.

1.4 Our Contributions

In this work, we approach the computational redundancy problem in CNN tasks such as OBE and ORV from the database standpoint by connecting their computational model to three classical database-inspired techniques: *incremental view maintenance* (IVM), *multi-query optimization* (MQO), and *approximate query processing* (AQP). Instead of treating a CNN as a "blackbox," we open it up and formalize *CNN layers* as "queries." Just like how a relational query converts relations to other relations, a CNN layer converts *tensors* (multidimensional arrays) to other tensors. So, we re-imagine OBE and ORV as *a set of tensor transformation queries* with incrementally updated inputs. With this fresh database-inspired view, we introduce several *novel CNN-specific query optimization techniques* to accelerate OBE and ORV.

Incremental inference for OBE. Our first optimization is *incremental inference*. We first *materialize* all tensors produced by the CNN. For every re-inference request, instead of rerunning inference from scratch, we treat it as an IVM query, with the "views" being the tensors. We rewrite such queries to *reuse* the materialized views as much as possible and recompute only what is needed, thus *avoiding computational redundancy*. Such rewrites are non-trivial, because they are tied to the complex geometric dataflows of CNN layers. We formalize such dataflows to create a novel *algebraic rewrite framework*. We also create a "static analysis" routine to tell us up front how much computations can be saved. Going further, we batch all re-inference requests to reuse the *same* materialized views. This is a form of MQO, which we call *batched incremental inference*. We also create a GPU-optimized kernel for such execution. To the best of our knowledge, this is the first instance of IVM being combined with MQO in query optimization, at least for CNN inference.

Approximate inference for OBE. For OBE, we also introduce two novel *approximate inference* optimizations that allow users to tolerate some degradation in visual quality of the heatmaps produced to reduce runtimes further. These optimizations build upon our incremental inference optimization to trade off heatmap quality in a user-tunable manner. Our first approximate optimization, *projective field thresholding*, draws upon an idea from neuroscience and exploits the internal semantics of how CNNs work. Our second approximate optimization, *adaptive drill-down*, exploits the semantics of the OBE task and the way users typically consume the heatmaps produced. We also present intuitive automated parameter tuning methods to help users adopt these optimizations.

Approximate inference for ORV. Similar to OBE, ORV can also be treated as a sequence of occluded images. In ORV, the image is the background of the video and occlusions are generated by a moving object. As the camera angle is fixed, based on some pixel-wise threshold, it is reasonable to assume the background to be fixed for a given episode of time. Thus, we approximate ORV as an extension of OBE. By taking the bounding box of the occluding object as the occluding patch, we leverage the same incremental inference infrastructure developed for OBE to perform ORV.

All of our techniques exist primarily at the logical level and help reduce computational costs. Thus, they are complementary to lower-level techniques to reduce runtimes from prior art such as reduced precision [18] or network pruning [19]. We prototype our techniques in the popular deep learning framework PyTorch to create a tool we call Krypton. It works on both CPU and GPU and supports any arbitrary CNN model architectures. We perform a comprehensive empirical evaluation of Krypton with real-world datasets and deep CNNs. For OBE, we use three image datasets and three CNNs from recent radiology and computer vision papers. Krypton yields up to 35× speedups over the current dominant practice of running re-inference with just batching

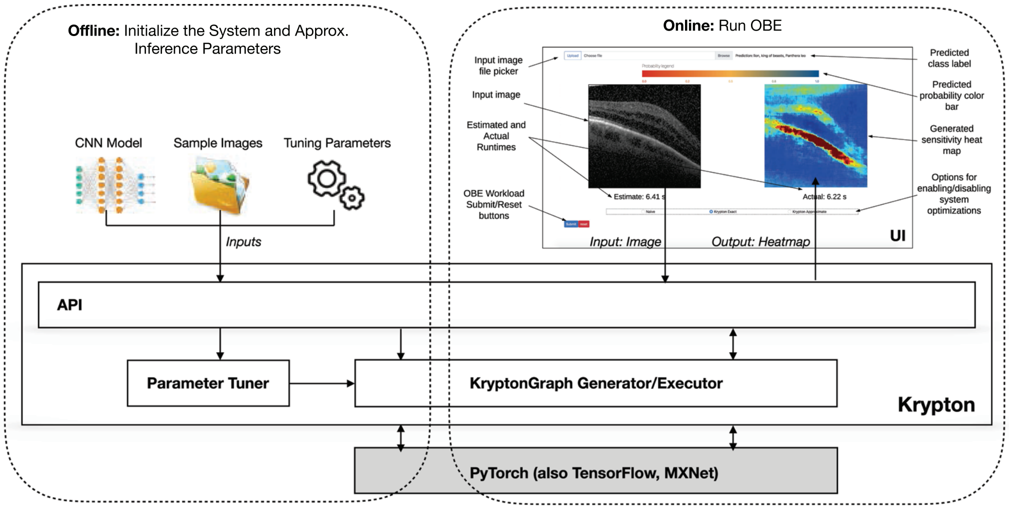

Offline: Initialize the System and Approx. Inference Parameters
CNN Model
Sample Images
Tuning Parameters
Inputs
Online: Run OBE
Input image file picker
Input image
Estimated and Actual Runtimes
OBE Workload Submit/Reset buttons
Upload
Choose file
Browse
Prediction: lion, king of beasts, Panthera leo
Probability legend
0.0
0.2
0.5
0.8
1.0
Estimate: 6.41 s
Actual: 6.22 s
Naive
Krypton Exact
Krypton Approximate
Submit
reset
Predicted class label
Predicted probability color bar
Generated sensitivity heat map
Options for enabling/disabling system optimizations
Input: Image
Output: Heatmap
UI
API
Parameter Tuner
KryptonGraph Generator/Executor
Krypton
PyTorch (also TensorFlow, MXNet)

Fig. 3. Krypton system architecture. It has four main components: API, Parameter Tuner, KryptonGraph Generator/Executor, and User Interface.

for producing high-quality approximate heatmaps and up to 5× speedups for producing exact heatmaps. We also show how Krypton can accelerate both OBE on real-world time series data analyzed with one-dimensional CNNs and ORV for real-world fixed-angle camera videos. We then tease apart the utility of each of our individual optimizations with drill-down experiments. Overall, this article makes the following contributions:

- To the best of our knowledge, this is the first article to formalize and optimize the execution of CNN inference from a data management standpoint.
- We create a novel and comprehensive algebraic framework for incremental CNN inference to reduce the computational cost of OBE. We integrate our IVM framework with an MQO technique to further reduce computational redundancy in CNN inference.
- We present two novel approximate inference optimizations for OBE that exploit the semantics of CNNs and properties of human perception.
- We approximate ORV as an extension of OBE and show how the same IVM machinery developed for OBE helps accelerate ORV, too.
- We prototype our ideas in a tool, Krypton, and perform an extensive empirical evaluation with real data and deep CNNs. Krypton offers substantial speedups for both OBE and ORV, even over an order of magnitude in some cases.

Outline. Section 2 presents the high-level system architecture of Krypton. Section 3 explains our problem setup, assumptions, and CNN dataflow model. Section 4 (respectively, Section 5) presents our incremental (respectively, approximate) inference optimizations for OBE. In Section 6, we explain the approximate inference optimizations for ORV. Section 6 presents the experimental evaluation. We discuss other related work in Section 7 and conclude in Section 8.

2 SYSTEM ARCHITECTURE

We explain the architecture of Krypton system in the context of the OBE workload. It has an offline setup phase and an online execution phase. For the ORV workload, we use only the online phase. The high-level architecture of the system is shown in Figure 3.

Krypton is implemented on top of the PyTorch deep learning library.[1] It has four main components:

- API
- Parameter Tuner
- `KryptonGraph` Generator/Executor
- User Interface (UI)

During the offline setup phase, Krypton takes in three inputs:

- Arbitrary PyTorch CNN model
- Sample of images from the OBE application (e.g., batch of OCT images)
- Several tuning parameters such as occlusion patch size, stride for the occlusion patch, and quality metrics for approximate inference

By analyzing the provided PyTorch CNN model, Krypton generates a DAG of incremental inference operators that we call a `KryptonGraph`. More details on `KryptonGraph` generation are explained in Section 4.5. Parameter Tuner then uses the user-provided sample of images and parameters to tune approximate inference subjected to user-defined quality metrics. More details on approximate inference are described in Section 5 and more details on parameter tuning are described in Section 5.3. During the online phase, user will provide an input image for which she wants to run OBE using the UI. After successful execution of OBE (respectively, ORV) using the `KryptonGraph` executor, Krypton will return the generated heatmap (respectively, predicted class labels). For the interested reader, more details about the UI and how it integrates with the Krypton engine can be found in our demo paper [20].

3 SETUP AND PRELIMINARIES

We now state the OBE problem formally and explain our assumptions. We then formalize the dataflow of the layers of a CNN, since these are required for understanding our techniques in Sections 3 and 4. Table 1 lists our notation. Finally, we briefly explain the Structural Similarity Index (SSIM), which is used to quantify the quality of the OBE-generated sensitivity heatmaps.

3.1 OBE Problem Statement and Assumptions

We are given a CNN f that has a sequence (or DAG) of *layers* l, each of which has a *tensor transformation function* $T_{:l}$. We are also given the image $\mathcal{I}_{:img}$ for which the OBE is desired, the class label L predicted by f on $\mathcal{I}_{:img}$, an occlusion patch $\mathcal{P}$ in RGB format, and occlusion patch *stride* $S_{\mathcal{P}}$. We are also given a set of patch positions G constructed either automatically or manually with a visual interface interactively. The OBE workload is as follows: produce a 2-D heatmap M, wherein each value corresponds to a position in G and has the predicted probability for label L by the CNN f on the occluded image $\mathcal{I}'_{x,y:img}$ (i.e., superimpose occlusion patch on image) or zero otherwise. More precisely, we can describe the OBE workload with the following logical statements:

$$W_M = \lfloor (\mathtt{width}(\mathcal{I}_{:img}) - \mathtt{width}(\mathcal{P}) + 1)/S_{\mathcal{P}} \rfloor \tag{1}$$

$$H_M = \lfloor (\mathtt{height}(\mathcal{I}_{:img}) - \mathtt{height}(\mathcal{P}) + 1)/S_{\mathcal{P}} \rfloor \tag{2}$$

$$M \in \mathbb{R}^{H_M \times W_M} \tag{3}$$

$$\forall\, (x, y) \in G : \tag{4}$$

$$\mathcal{I}'_{x,y:img} \leftarrow \mathcal{I}_{:img} \circ_{(x,y)} \mathcal{P} \tag{5}$$

$$M[x, y] \leftarrow f(\mathcal{I}'_{x,y:img})[L]. \tag{6}$$

[1] It is simple to extend it to support other libraries like TensorFlow as well.

Table 1. Notation Used in this Article

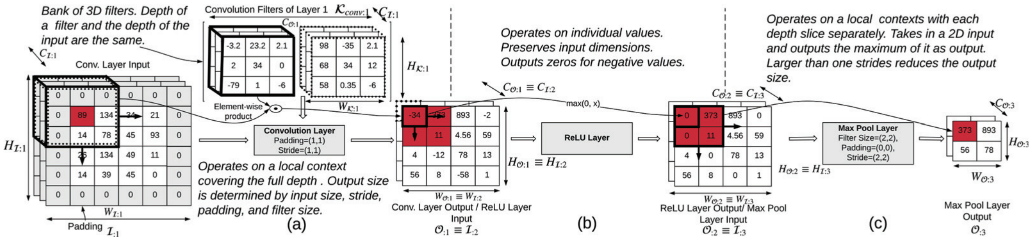
Symbol	Meaning
f	Given deep CNN; input is an image tensor; output is a probability distribution over class labels
L	Class label predicted by f for the original image $\mathcal{I}_{:img}$
$T_{:l}$	Tensor transformation function of layer l of the given CNN f
$\mathcal{P}$	Occlusion patch in RGB format
$S_{\mathcal{P}}$	Occlusion patch striding amount
G	Set of occlusion patch superimposition positions on $\mathcal{I}_{:img}$ in (x,y) format
M	Heatmap produced by the OBE workload
H_M, W_M	Height and width of M
$\circ_{(x,y)}$	Superimposition operator. $A \circ_{(x,y)} B$, superimposes B on top of A starting at (x, y) position
$\mathcal{I}_{:l}$ ($\mathcal{I}_{:img}$)	Input tensor of layer l (Input Image)
$\mathcal{O}_{:l}$	Output tensor of layer l
$C_{\mathcal{I}:l}, H_{\mathcal{I}:l}, W_{\mathcal{I}:l}$	Depth, height, and width of input of layer l
$C_{\mathcal{O}:l}, H_{\mathcal{O}:l}, W_{\mathcal{O}:l}$	Depth, height, and width of output of layer l
$\mathcal{K}_{conv:l}$	Convolution filter kernels of layer l
$\mathcal{B}_{conv:l}$	Convolution bias value vector of layer l
$\mathcal{K}_{pool:l}$	Pooling filter kernel of layer l
$H_{\mathcal{K}:l}, W_{\mathcal{K}:l}$	Height and width of filter kernel of layer l
$S_{:l}; S_{x:l}; S_{y:l}$	Filter kernel striding amounts of layer l; $S_{:l} \equiv (S_{x:l}, S_{y:l})$, strides along width and height dimensions
$P_{:l}; P_{x:l}; P_{y:l}$	Padding amounts of layer l; $P_{:l} \equiv (P_{x:l}, P_{y:l})$, padding along width and height dimensions

Steps (1) and (2) calculate the dimensions of the heatmap M. Step (5) superimposes $\mathcal{P}$ on $\mathcal{I}_{:img}$ with its top left corner placed on the (x, y) location of $\mathcal{I}_{:img}$. Step (6) calculates the output value at the (x, y) location by performing CNN inference for $\mathcal{I}'_{x,y:img}$ using f and picks the prediction probability of L. Steps (5) and (6) are performed *independently* for every occlusion patch position in G. In the *non-interactive* mode, G is initialized to $G = [0, H_M) \times [0, W_M)$. Intuitively, this represents the set of all possible occlusion patch positions on $\mathcal{I}_{:img}$, which yields a full heatmap. In the *interactive* mode, the user may manually place the occlusion patch only at a few locations at a time, yielding partial heatmaps.

We assume the CNN is used for classification (or regression), since OBE is typically only used in these applications. One could create CNNs that predict an image "segmentation" instead, but labeling image segments for training such CNNs is tedious and expensive. Thus, most recent applications of CNNs in healthcare, sociology, and other domains rely on classification CNNs and use OBE [2–4, 7, 8]. Other approaches to explain CNN predictions have been studied, but since they are orthogonal to our focus, we summarize them in the Related Work section (Section 8).

3.2 Dataflow of CNN Layers

CNNs are organized as *layers* of various types, each of which transforms a tensor (multidimensional array, typically 3-D) into another tensor: *Convolution* uses image filters from graphics to extract features, but with parametric filter weights (learned during training); *Pooling* subsamples features in a spatial-aware manner; *Batch-Normalization* normalizes the output tensor; *Non-linearity*


Bank of 3D filters. Depth of a filter and the depth of the input are the same.

Operates on a local context covering the full depth . Output size is determined by input size, stride, padding, and filter size.

Operates on individual values. Preserves input dimensions. Outputs zeros for negative values.

Operates on a local contexts with each depth slice separately. Takes in a 2D input and outputs the maximum of it as output. Larger than one strides reduces the output size.

Fig. 4. Simplified illustration of the key layers of a typical CNN. The highlighted cells (dark/red background) show how a small local spatial context in the first input propagates through subsequent layers. (a) Convolution layer (for simplicity sake, bias addition is not shown). (b) ReLU Non-linearity layer. (c) Pooling layer (max pooling). Notation is explained in Table 1.

applies an element-wise non-linear function (e.g., ReLU); *Fully-Connected* is an ordered collection of perceptrons [21]. The output tensor of a layer can have a different width, height, and/or depth than the input. An image can be viewed as a tensor, e.g., a 224 × 224 RGB image is a 3-D tensor with width and height 224 and depth 3. A Fully-Connected layer converts a 1-D tensor (or a "flattened" 3-D tensor) to another 1-D tensor. For simplicity of exposition, we group CNN layers into three main categories based on the *spatial locality* of how they transform a tensor: (1) Transformations with a *global context*, e.g., Fully-Connected; (2) Transformations at the granularity of *individual elements*, e.g., ReLU or Batch Normalization; and (3) Transformations at the granularity of a *local spatial context*, e.g., Convolution or Pooling.

Global context granularity. Such layers convert the input tensor holistically into an output tensor without any spatial context, typically with a full matrix-vector multiplication. Fully-Connected is the only layer of this type. Since every element of the output will likely be affected by the entire input, such layers do not offer a major opportunity for faster incremental computations. Thankfully, Fully-Connected layers typically arise only as the last layer(s) in deep CNNs (and never in some recent deep CNNs), and as shown in Figure 5, they typically account for a negligible fraction of the total computational cost and runtime. Thus, we do not focus on such layers for our optimizations.

Individual element granularity. Such layers apply a "map()" function on the elements of the input tensor, as Figure 4(b) illustrates. Thus, the output has the same dimensions as the input. Non-Linearity (e.g., with ReLU) falls under this category. The computational cost is proportional to the "volume" of the input tensor (product of the dimensions). If the input is incrementally updated, only the corresponding region of the output will be affected. Thus, incremental inference for such layers is straightforward. The computational cost of the incremental computation is proportional to the volume of the updated region and is typically a small fraction of the overall computation cost, as shown in Figure 5.

Local spatial context granularity. Such layers perform weighted aggregations of slices of the input tensor, called *local spatial contexts*, by multiplying them with a *filter kernel* (a tensor of weights). Thus, input and output tensors can differ in width, height, and depth. If the input is incrementally updated, the region of the output that will be affected is not straightforward to ascertain—this requires non-trivial and careful calculations due to the overlapping nature of how filters get applied to local spatial contexts. Both Convolution and Pooling fall under this category. As shown in Figure 5, such layers typically account for the bulk of the computational cost of deep CNN inference (over 90%). Thus, enabling incremental inference for such layers in the OBE context is a key

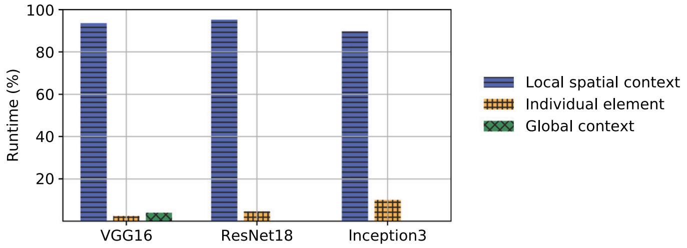


Fig. 5. Runtime distribution for different CNN layer families (based on the transformation granularity) for three popular deep CNNs. Experiments are performed on a CPU. For ResNet18 and Inception3, the runtime percentage spent on global context transformations is zero. Details on experimental setup are provided in Section 7.

focus of this article (Section 3). The rest of this section explains the machinery of the dataflow in such layers using our notation. Section 3 then uses this machinery to explain our optimizations.

Dataflow of Convolution Layers. A layer l has $C_{\mathcal{O}:l}$ 3-D filter kernels arranged as a 4-D array $\mathcal{K}_{conv:l}$, with each having a smaller spatial width $W_{\mathcal{K}:l}$ and height $H_{\mathcal{K}:l}$ than the width $W_{\mathcal{I}:l}$ and height $H_{\mathcal{I}:l}$ of the input tensor $\mathcal{I}_{:l}$ but the same depth $C_{\mathcal{I}:l}$. During inference, cth filter kernel is "strided" along the width and height dimensions of the input to produce a 2-D "activation map" $A_{:c} = (a_{y,x:c}) \in \mathbb{R}^{H_{\mathcal{O}:l} \times W_{\mathcal{O}:l}}$ by computing element-wise products between the kernel and the local spatial context and adding a bias value as per Equation (7). The computational cost of each of these small matrix products is proportional to the volume of the filter kernel. All the 2-D activation maps are then stacked along the depth dimension to produce the output tensor $\mathcal{O}_{:l} \in \mathbb{R}^{C_{\mathcal{O}:l} \times H_{\mathcal{O}:l} \times W_{\mathcal{O}:l}}$. Figure 4(a) presents a simplified illustration of this layer.

$$\begin{aligned} a_{y,x:c} = & \sum_{k=0}^{C_{\mathcal{I}:l}} \sum_{j=0}^{H_{\mathcal{K}:l}-1} \sum_{i=0}^{W_{\mathcal{K}:l}-1} \mathcal{K}_{conv:l}[c,k,j,i] \\ & \times \mathcal{I}_{:l}\left[k, y - \left\lfloor \frac{H_{\mathcal{K}:l}}{2} \right\rfloor + j, x - \left\lfloor \frac{W_{\mathcal{K}:l}}{2} \right\rfloor + i\right] \\ & + \mathcal{B}_{conv:l}[c] \end{aligned} \tag{7}$$

Dataflow of Pooling Layers. Such layers behave essentially like Convolution layers but with a fixed (not learned) 2-D filter kernel $\mathcal{K}_{pool:l}$. These kernels aggregate a local spatial context to compute its maximum or average element. But unlike Convolution, Pooling operates independently on the depth slices of the input tensor. It takes as input a 3-D tensor $\mathcal{O}_l$ of depth $C_{\mathcal{I}:l}$, width $W_{\mathcal{I}:l}$, and height $H_{\mathcal{I}:l}$. It produces as output a 3-D tensor $\mathcal{O}_{:l}$ with the same depth $C_{\mathcal{O}:l} = C_{\mathcal{I}:l}$ but a different width of $W_{\mathcal{O}:l}$ and height $H_{\mathcal{O}:l}$. The filter kernel is typically strided over more than one pixel at a time. Thus, $W_{\mathcal{O}:l}$ and $H_{\mathcal{O}:l}$ are usually smaller than $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$, respectively. Figure 4(c) presents a simplified illustration of this layer. Overall, both Convolution and Pooling layers have a similar dataflow along the width and height dimensions, while differing on the depth dimension. Since OBE only concerns the width and height dimensions of the image and subsequent tensors, we can treat both these types of layers in a unified manner for our optimizations.

Relationship between Input and Output Dimensions. For Convolution and Pooling layers, $W_{\mathcal{O}:l}$ and $H_{\mathcal{O}:l}$ are determined by $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$, $W_{\mathcal{K}:l}$ and $H_{\mathcal{K}:l}$, and two other parameters that are specific to that layer: *stride* $S_{:l}$ and *padding* $P_{:l}$. Stride is the number of pixels by which the filter kernel is moved at a time; it can differ along the width and height dimensions: $S_{x:l}$ and $S_{y:l}$, respectively. In practice, most CNNs have $S_{x:l} = S_{y:l}$. Typically, $S_{x:l} \leq W_{\mathcal{K}:l}$ and $S_{y:l} \leq H_{\mathcal{K}:l}$. In

Figure 4, the Convolution layer has $S_{x:l} = S_{y:l} = 1$, while the Pooling layer has $S_{x:l} = S_{y:l} = 2$. For some layers, to help control the dimensions of the output to be the same as the input, one "pads" the input with zeros along the width and height dimensions. *Padding* $P_{:l}$ captures how much such padding extends these dimensions; once again, padding values can differ along the width and height dimensions: $P_{x:l}$ and $P_{y:l}$. In Figure 4(a), the Convolution layer has $P_{x:l} = P_{y:l} = 1$. Given these parameters, width (similarly height) of the output tensor is given by the following formula:

$$W_{\mathcal{O}:l} = (W_{\mathcal{I}:l} - W_{\mathcal{K}:l} + 1 + 2 \times P_{x:l})/S_{x:l}. \tag{8}$$

Computational Cost of Inference. Deep CNN inference is computationally expensive. Convolution layers typically account for a bulk of the cost (90% or more) [22]. Thus, we can roughly estimate the computational cost of inference by counting the number of *fused multiply-add* (FMA) floating point operations (FLOPs) needed for the Convolution layers. For example, applying a Convolution filter with dimensions $(C_{\mathcal{I}:l}, H_{\mathcal{K}:l}, W_{\mathcal{K}:l})$ to compute one element of the output tensor requires $C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l}$ FLOPs, with each FLOP corresponding to one FMA. Thus, the total computational cost $Q_{:l}$ of a layer that produces output $\mathcal{O}_{:l}$ of dimensions $(C_{\mathcal{O}:l}, H_{\mathcal{O}:l}, W_{\mathcal{O}:l})$ and the total computational cost Q of the entire set of Convolution layers of a given CNN f can be calculated as per Equations (9) and (10):

$$Q_{:l} = (C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{\mathcal{O}:l} \cdot H_{\mathcal{O}:l} \cdot W_{\mathcal{O}:l}), \tag{9}$$

$$Q = \sum_{l \, in f} Q_{:l}. \tag{10}$$

3.3 Estimating the Quality of Generated Approximate Heatmaps

When applying approximate inference optimizations for OBE, Krypton trades off the accuracy/quality of the generated heatmap in favor of faster execution. To measure this drop of accuracy, we use SSIM Index [23], one of the widely used approaches to measure the *human-perceived difference* between two similar images. When applying SSIM index, we treat the original heatmap as the reference image with no distortions; the perceived image similarity of the Krypton-generated heatmap is calculated with reference to that. The generated SSIM index is a value between −1 and 1, where 1 corresponds to perfect similarity. Typically SSIM index values in the range of $0.90 - 0.95$ are used in practical applications such as image compression and video encoding, as they produce distortions indistinguishable for humans. For more details on SSIM Index metric, we refer the reader to the original SSIM Index paper [23].

4 INCREMENTAL INFERENCE OPTIMIZATIONS

We start with a theoretical characterization of the speedups incremental inference can yield for OBE. We then dive into our novel algebraic framework to enable incremental CNN inference and combine it with our multi-query optimization for OBE.

4.1 Expected Speedups

In relational IVM, when a part of the input relation is updated, we recompute only the part of output that gets changed. We bring this notion to CNNs; a CNN layer is our "query" and the materialized feature tensor is our "relation." OBE updates only a part of the image; so, only some parts of the tensors need to be recomputed. We create an algebraic framework to determine which parts these are for a CNN layer (Section 3.2) and how to propagate updates across layers (Section 3.3). Given a CNN f and the occlusion patch, our framework determines using "static analysis" over f how many FLOPs can be saved, yielding us an upper bound on speedups.

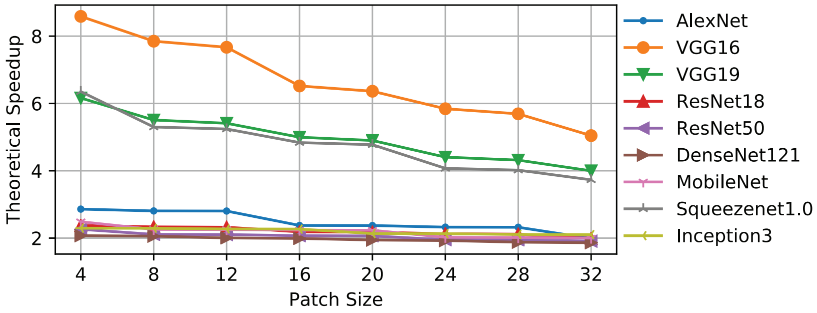


Fig. 6. Theoretical speedups for popular deep CNN architectures with incremental inference.

More precisely, let the output tensor dimensions of layer l be $(C_{\mathcal{O}:l}, H_{\mathcal{O}:l}, W_{\mathcal{O}:l})$. An incremental update recomputes a smaller local spatial context with width $W_{\mathcal{P}:l} \leq W_{\mathcal{O}:l}$ and height $H_{\mathcal{P}:l} \leq H_{\mathcal{O}:l}$. Thus, the computational cost of incremental inference for layer l, denoted by $Q_{inc:l}$, is equal to the volume of the individual filter kernel times the total volume of the updated output, as given by Equation (11). The total computational cost for incremental inference, denoted Q_{inc}, is given by Equation (12).

$$Q_{inc:l} = (C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{\mathcal{O}:l} \cdot H_{\mathcal{P}:l} \cdot W_{\mathcal{P}:l}), \tag{11}$$

$$Q_{inc} = \sum_{l\ inf} Q_{inc:l}. \tag{12}$$

The above costs can be much smaller than $Q_{:l}$ and Q in Equations (9) and (10) earlier. We define the *theoretical speedup* as the ratio $\frac{Q}{Q_{inc}}$. It tells us how beneficial incremental inference can be in the best case *without* performing the inference itself. It depends on several factors: the occlusion patch size, its location, the parameters of layers (kernel dimensions, stride, etc.), and so on. Calculating it is non-trivial and requires careful analysis, which we perform. The location of patch affects this ratio, because a patch placed in the corner leads to fewer updates overall than one placed in the center of the image. Thus, the "worst-case" theoretical speedup is determined by placing the patch at the center.

We perform a sanity check experiment to ascertain the theoretical speedups for a few popular deep CNNs. For varying occlusion patch sizes (with a stride of 1), we plot the theoretical speedups in Figure 6. VGG-16 has the highest theoretical speedups, while DenseNet-121 has the lowest. Most CNNs fall in the 2×–3× range. The differences arise due to the specifics of the CNNs' architectures: VGG-16 has small Convolution filter kernels and strides, which means full inference incurs a high computational cost (Q = 15 GFLOPs). Thus, VGG-16 benefits the most from incremental inference. Note the image size is assumed to be 224 × 224 for this plot; if the image is larger, the theoretical speedups will be higher.

While speedups of 2×–3× may sound "not that significant" in practice, we find that they indeed are significant for at least two reasons. First, *users often wait in the loop* for OBE workloads for performing interactive diagnoses and analyses. Thus, even such speedups can improve their productivity, e.g., reducing the time taken on a CPU from about 6 mins to just 2 mins, or on a GPU from 1 min to just 20 s. Second, and equally importantly, incremental inference is the *foundation for our approximate inference* optimizations (Section 4), which amplify the speedups we achieve for OBE. For instance, the speedup for Inception3 goes up from only 2× for incremental inference to up to 8× with all of our optimizations enabled. Thus, incremental inference is critical to optimizing OBE.

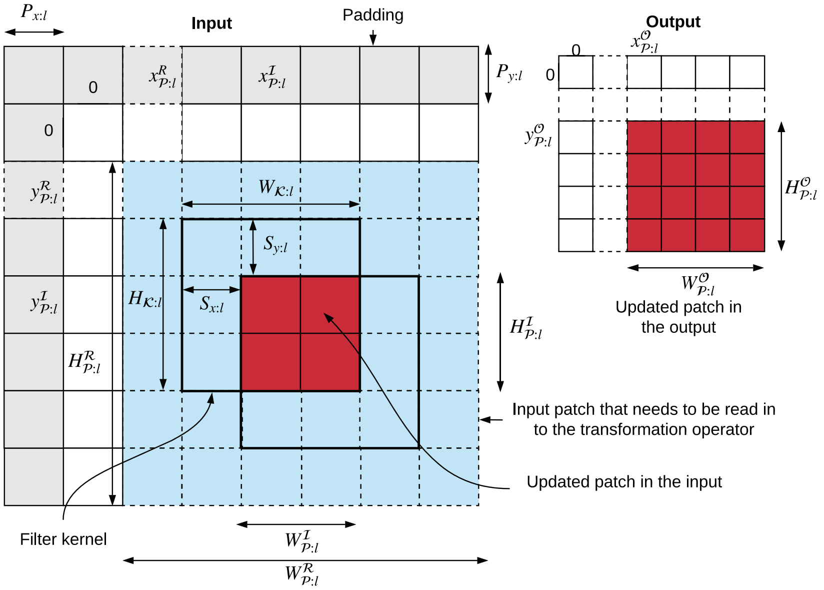

$P_{x:l}$
Input
Padding
Output
0
0
$x^{\mathcal{R}}_{\mathcal{P}:l}$
$x^{\mathcal{I}}_{\mathcal{P}:l}$
$P_{y:l}$
0
$x^{\mathcal{O}}_{\mathcal{P}:l}$
0
$y^{\mathcal{O}}_{\mathcal{P}:l}$
$H^{\mathcal{O}}_{\mathcal{P}:l}$
$y^{\mathcal{R}}_{\mathcal{P}:l}$
$W_{\mathcal{K}:l}$
$S_{y:l}$
$W^{\mathcal{O}}_{\mathcal{P}:l}$
Updated patch in the output
$y^{\mathcal{I}}_{\mathcal{P}:l}$
$H_{\mathcal{K}:l}$
$S_{x:l}$
$H^{\mathcal{I}}_{\mathcal{P}:l}$
$H^{\mathcal{R}}_{\mathcal{P}:l}$
Input patch that needs to be read in to the transformation operator
Updated patch in the input
Filter kernel
$W^{\mathcal{I}}_{\mathcal{P}:l}$
$W^{\mathcal{R}}_{\mathcal{P}:l}$

Fig. 7. Simplified illustration of input and output update patches for Convolution/Pooling layers.

4.2 Single Layer Incremental Inference

We now present our algebraic framework for incremental updates to the materialized output tensor of a CNN layer. As per the discussion in Section 2.2, we focus only on the non-trivial layers that operate at the granularity of a local spatial context (Convolution and Pooling). We call our modified version of such layers "incremental inference operations."

Determining Patch Update Locations. We first explain how to calculate the coordinates and dimensions of the *output update patch* of layer l given the *input update patch* and layer-specific parameters. Figure 7 presents a simplified illustration of these calculations. Our coordinate system's origin is at the top left corner. The input update patch is shown in red/dark color and starts at $(x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l})$, with height $H^{\mathcal{I}}_{\mathcal{P}:l}$ and width $W^{\mathcal{I}}_{\mathcal{P}:l}$. The output update patch starts at $(x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l})$ and has a height $H^{\mathcal{O}}_{\mathcal{P}:l}$ and width $W^{\mathcal{O}}_{\mathcal{P}:l}$. Due to overlaps among filter kernel positions during inference, computing the output update patch requires us to read a slightly larger spatial context than the input update patch—we call this the "read-in context," and it is illustrated by the blue/shaded region in Figure 7. The read-in context starts at $(x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l})$, with its dimensions denoted by $W^{\mathcal{R}}_{\mathcal{P}:l}$ and $H^{\mathcal{R}}_{\mathcal{P}:l}$. Table 2 summarizes all this additional notation for this section. The relationship between these quantities along the width dimension (similarly along the height dimension) can be expressed as follows:

$$x^{\mathcal{O}}_{\mathcal{P}:l} = max\Big(\lceil\big(P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1\big)/S_{x:l}\rceil, 0\Big), \tag{13}$$

$$W^{\mathcal{O}}_{\mathcal{P}:l} = min\Big(\Big\lceil(W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l}\Big\rceil, W_{\mathcal{O}:l}\Big), \tag{14}$$

$$x^{\mathcal{R}}_{\mathcal{P}:l} = x^{\mathcal{O}}_{\mathcal{P}:l} \times S_{x:l} - P_{x:l}, \tag{15}$$

$$W^{\mathcal{R}}_{\mathcal{P}:l} = W_{\mathcal{K}:l} + \Big(W^{\mathcal{O}}_{\mathcal{P}:l} - 1\Big) \times S_{x:l}. \tag{16}$$

Equation (13) calculates the coordinates of the output update patch. As shown in Figure 7, padding effectively shifts the coordinate system and thus, $P_{x:l}$ is added to correct it. Due to overlaps among the filter kernels, the affected region of the input update patch (blue/shaded region in

Table 2. Additional Notation for Sections 4 and 5

Symbol	Meaning
$x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l}$	Start coordinates of input update patch for layer l
$x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l}$	Start coordinates of read-in context for layer l
$x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l}$	Start coordinates of output update patch for layer l
$H^{\mathcal{I}}_{\mathcal{P}:l}, W^{\mathcal{I}}_{\mathcal{P}:l}$	Height and width of input update patch for layer l
$H^{\mathcal{R}}_{\mathcal{P}:l}, W^{\mathcal{R}}_{\mathcal{P}:l}$	Height and width of read-in context for layer l
$H^{\mathcal{O}}_{\mathcal{P}:l}, W^{\mathcal{O}}_{\mathcal{P}:l}$	Height and width of output update patch for layer l
τ	Projective field threshold
$r_{drill-down}$	Drill-down fraction for adaptive drill-down

Figure 7) will be increased by $W_{\mathcal{K}:l} - 1$, which needs to be subtracted from the input coordinate $x^{\mathcal{I}}_{\mathcal{P}:l}$. A filter of size $W_{\mathcal{K}:l}$ that is placed starting at $x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1$ will see an update starting from $x^{\mathcal{I}}_{\mathcal{P}:l}$. Equation (14) calculates the width of the output update patch, which is essentially the number of filter kernel stride positions on the read-in input context. However, this value cannot be larger than the output size. Given these, a start coordinate and width of the read-in context are given by Equations (15) and (16); similar equations hold for the height dimension (skipped for brevity).

Incremental Inference Operation. For layer l, given the transformation function $T_{:l}$, the pre-materialized input tensor $\mathcal{I}_{:l}$, input update patch $\mathcal{P}^{\mathcal{O}}_{:l}$, and the above calculated coordinates and dimensions of the input, output, and read-in context, the output update patch $\mathcal{P}^{\mathcal{O}}_{:l}$ is computed as follows:

$$\mathcal{U} = \mathcal{I}_{:l}\left[\ :, x^{\mathcal{R}}_{\mathcal{P}:l} : x^{\mathcal{R}}_{\mathcal{P}:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l} : y^{\mathcal{R}}_{\mathcal{P}:l} + H^{\mathcal{R}}_{\mathcal{P}:l}\right], \tag{17}$$

$$\mathcal{U} = \mathcal{U} \circ_{(x^{\mathcal{I}}_{\mathcal{P}:l} - x^{\mathcal{R}}_{\mathcal{P}:l}),(y^{\mathcal{I}}_{\mathcal{P}:l} - y^{\mathcal{R}}_{\mathcal{P}:l})} \mathcal{P}^{\mathcal{I}}_{:l}, \tag{18}$$

$$\mathcal{P}^{\mathcal{O}}_{:l} = T_{:l}(\mathcal{U}). \tag{19}$$

Equation (17) slices the read-in context $\mathcal{U}$ from the pre-materialized input tensor $\mathcal{I}_{:l}$. Equation (18) superimposes the input update patch $\mathcal{P}^{\mathcal{I}}_{:l}$ on it. This is an in-place update of the array holding the read-in context. Finally, Equation (19) computes the output update patch $\mathcal{P}^{\mathcal{O}}_{:l}$ by invoking $T_{:l}$ on $\mathcal{U}$. Thus, we avoid performing inference on all of $\mathcal{I}_{:l}$, thus achieving incremental inference and reducing FLOPs.

Special Cases for Incremental Inference. There are special cases under which the output patch size can be smaller than the values calculated above. Consider the simplified 1-D case shown in Figure 8 (a), where the filter stride[2] (3) is the same as the filter size (3). In this case, the size of the output update patch is one less than the value calculated by Equation (14). But this is not the case for the situation shown Figure 8(b), which has the same input patch size but placed at a different location. Another case arises when the modified patch is placed at the edge of the input, as shown in Figure 8(c). In this case, it is impossible for the filter to move freely through all positions, since it hits the input boundary. However, it is not the case for the modified patch shown in Figure 8(d). In Krypton, we do not treat these cases separately but rather use the values calculated by Equation (14) for the width dimension (similarly for the height dimension), since they act as an upper bound. In the case of a smaller output patch, Krypton reads and updates a

[2] Note that stride is typically less than or equal to filter size.

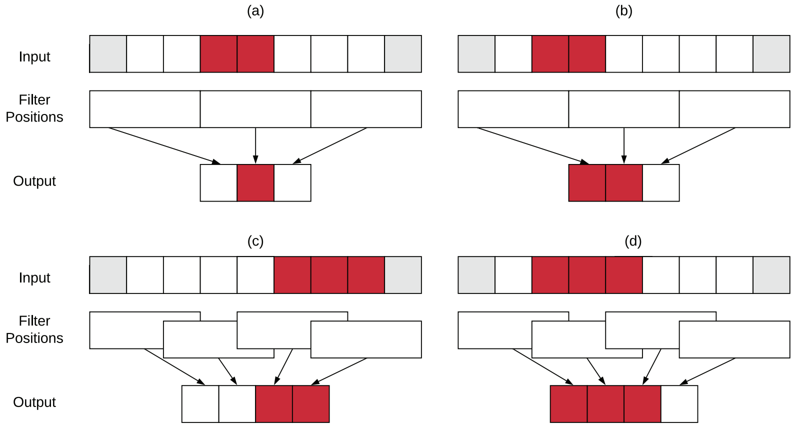


Fig. 8. Illustration of special cases for which actual output size will be smaller than the value given by Equation (14). (a) and (b) show cases where the filter stride is equal to the filter size. (c) and (d) show situations where the position of the modified patch affecting the size of the output patch.

slightly bigger patch to preserve uniformity. This also requires updating the starting coordinates of the patch, as shown in Equation (20). This sort of uniform treatment is required for performing batched inference operations, which gives significant speedups compared to per-image inference.

$$\begin{aligned}&\text{If}\, x^{\mathcal{O}}_{\mathcal{P}} + W^{\mathcal{O}}_{\mathcal{P}} > W_{\mathcal{O}} :\\ &x^{\mathcal{O}}_{\mathcal{P}} = W_{\mathcal{O}} - W^{\mathcal{O}}_{\mathcal{P}}; x^{\mathcal{I}}_{\mathcal{P}} = W_{\mathcal{I}} - W^{\mathcal{I}}_{\mathcal{P}}; x^{\mathcal{R}}_{\mathcal{P}} = W_{\mathcal{I}} - W^{\mathcal{R}}_{\mathcal{P}}\end{aligned} \tag{20}$$

Special Types of CNNs. So far in our formulation, we focused on use cases where CNNs are being used on image data. However, CNNs can be applied to sequence data, such as time-series, by splitting the sequence into equal-sized windows. In the case of time series data, this is done along the time axis. Windowed sequence can be considered as a special type of image where the height is always one and width is equal to the window size. The number of channels in the image will be equal to the number of attributes in a multi-variate sequence. All convolution operations will be applied only over the windowing axis and hence they are called one-dimensional CNNs. OBE is still useful in these scenarios to explain CNN predictions. However, in this case OBE will produce a sequence of probabilities instead of a heatmap. Due to the generic nature of our incremental inference formulation, Krypton can accelerate OBE for 1-D CNNs through incremental inference by simply setting the height of the occlusion patch to one.

4.3 Propagating Updates across Layers

Sequential CNNs. Unlike relational IVM, CNNs have many layers, often in a sequence. This is analogous to a sequence of queries, each requiring IVM on its predecessor's output. This leads to a new issue: correctly and automatically configuring the update patches across all layers of a CNN. Specifically, output update patch $\mathcal{P}^{\mathcal{O}}_{:l}$ of layer l becomes the input update patch of layer $l+1$. While this seems simple, it requires care at the boundary of a local context transformation and a global context transformation, e.g., between a Convolution (or Pooling) layer and a Fully-Connected layer. In particular, we need to materialize the full updated output, not just the output update patches, since global context transformations lose spatial locality for subsequent layers.

Extension to DAG CNNs. Some recent deep CNNs have a more general directed acyclic graph (DAG) structure for layers. They have two new kinds of layers that "merge" two branches in the

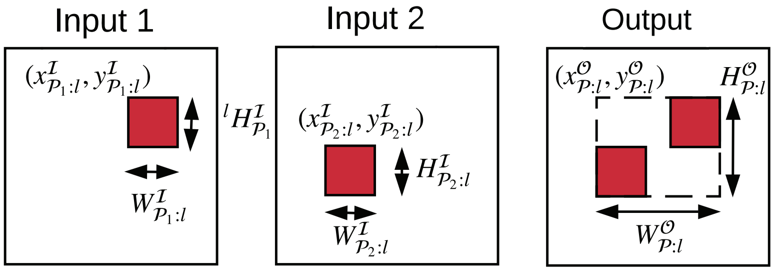

Input 1
Input 2
Output
$(x^{\mathcal{I}}_{\mathcal{P}_1:l}, y^{\mathcal{I}}_{\mathcal{P}_1:l})$
$W^{\mathcal{I}}_{\mathcal{P}_1:l}$
$^{l}H^{\mathcal{I}}_{\mathcal{P}_1}$
$(x^{\mathcal{I}}_{\mathcal{P}_2:l}, y^{\mathcal{I}}_{\mathcal{P}_2:l})$
$H^{\mathcal{I}}_{\mathcal{P}_2:l}$
$W^{\mathcal{I}}_{\mathcal{P}_2:l}$
$(x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l})$
$H^{\mathcal{O}}_{\mathcal{P}:l}$
$W^{\mathcal{O}}_{\mathcal{P}:l}$

Fig. 9. Illustration of bounding box calculation for differing input update patch locations for element-wise addition and depth-wise concatenation layers in DAG CNNs.

DAG: *element-wise addition* and *depth-wise concatenation*. Element-wise addition requires two input tensors with all dimensions being identical. Depth-wise concatenation takes two input tensors with the same height and width dimensions. We now face a new challenge—how to calculate the output update patch when the two input tensors differ on their input update patches locations and sizes? Figure 9 shows a simplified illustration of this issue. The first input has its update patch starting at coordinates $(x^{\mathcal{I}}_{\mathcal{P}_1:l}, y^{\mathcal{I}}_{\mathcal{P}_1:l})$ with dimensions $H^{\mathcal{I}}_{\mathcal{P}_1:l}$ and $W^{\mathcal{I}}_{\mathcal{P}_1:l}$, while the second input has its update patch starting at coordinates $(x^{\mathcal{I}}_{\mathcal{P}_2:l}, y^{\mathcal{I}}_{\mathcal{P}_2:l})$ with dimensions $H^{\mathcal{I}}_{\mathcal{P}_2:l}$ and $W^{\mathcal{I}}_{\mathcal{P}_2:l}$. This issue can arise with both element-wise addition and depth-wise concatenation.

We propose a simple unified solution: compute the *bounding box* of the input update patches. So, the coordinates and dimensions of both read-in contexts and the output update patch will be identical. Figure 9 illustrates this. While this will potentially recompute parts of the output that do not get modified, we think this tradeoff is acceptable, because the gains are likely to be marginal for the additional complexity introduced into our framework. Overall, the output update patch coordinate and width dimension are given by the following (similarly for the height dimension):

$$\begin{aligned} x^{O}_{P:l} &= \mathtt{min}\left(x^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l}\right), \\ W^{O}_{\mathcal{P}:l} &= \mathtt{max}\left(x^{\mathcal{I}}_{\mathcal{P}_1:l} + W^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l} + W^{\mathcal{I}}_{\mathcal{P}_2:l}\right) - \mathtt{min}\left(x^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l}\right). \end{aligned} \tag{21}$$

4.4 Multi-query Incremental Inference

OBE issues $|G|$ re-inference requests *in one go*. Viewing each request as a "query" makes the connection with multi-query optimization (MQO) [24] clear. The $|G|$ queries are also *not disjoint*, since the occlusion patch is typically small, which means most pixels are the same for each query. Thus, we now extend our IVM framework for re-inference with an MQO-style optimization fusing multiple re-inference requests. An analogy with relational queries would be having many incremental update queries on the same relation in one go, with each query receiving a different incremental update.

Batched Incremental Inference. Our optimization works as follows: materialize all CNN tensors *once* and *reuse* them for incremental inference across all $|G|$ queries. Since the occluded images share most of their pixels, parts of the tensors will likely be identical, too. Thus, we can amortize the materialization cost. One might ask: why not just perform "batched" inference for the $|G|$ queries? Batched execution is standard practice on high-throughput compute hardware like GPUs, since it amortizes CNN setup costs, data-movement costs, and so on. Batch sizes are tuned to optimize hardware utilization. We note that batching is an *orthogonal* (albeit trivial) optimization compared to our MQO. Thus, we combine both of these ideas to execute incremental inference in a batched manner. We call this approach "batched incremental inference." Empirically, we find that batching

ALGORITHM 1: BatchedIncrementalInference

```
Input:
  $T_{:l}$ : Original Transformation function
  $\mathcal{I}_{:l}$ : Pre-materialized input from original image
  $[\mathcal{P}^{\mathcal{I}}{}_{1:l}, \ldots, \mathcal{P}^{\mathcal{I}}{}_{n:l}]$ : Input patches
  $[(x^{\mathcal{I}}_{\mathcal{P}_1:l}, y^{\mathcal{I}}_{\mathcal{P}_1:l}), \ldots, (x^{\mathcal{I}}_{\mathcal{P}_n:l}, y^{\mathcal{I}}_{\mathcal{P}_n:l})]$ : Input patch coordinates
  $W^{\mathcal{I}}_{\mathcal{P}:l}, H^{\mathcal{I}}_{\mathcal{P}:l}$ : Input patch dimensions

procedure BatchedIncrementalInference
    Calculate $[(x^{\mathcal{O}}_{\mathcal{P}_1:l}, y^{\mathcal{O}}_{\mathcal{P}_1:l}), \ldots, (x^{\mathcal{O}}_{\mathcal{P}_n:l}, y^{\mathcal{O}}_{\mathcal{P}_n:l})]$
    Calculate $(W^{\mathcal{O}}_{\mathcal{P}:1}, H^{\mathcal{O}}_{\mathcal{P}:l})$
    Calculate $[(x^{\mathcal{R}}_{\mathcal{P}_1:l}, y^{\mathcal{R}}_{\mathcal{P}_1:l}), \ldots, (x^{\mathcal{R}}_{\mathcal{P}_n:l}, y^{\mathcal{R}}_{\mathcal{P}_n} : l)]$
    Calculate $(W^{\mathcal{R}}_{\mathcal{P}:l}, H^{\mathcal{R}}_{\mathcal{P}:l})$
    Initialize $\mathcal{U} \in \mathbb{R}^{n \times \mathtt{depth}(\mathcal{I}_{:l}) \times H^{\mathcal{R}}_{\mathcal{P}:l} \times W^{\mathcal{R}}_{\mathcal{P}:l}}$
    for i in [1,. . . ,n] do
        $T_1 \leftarrow \mathcal{I}_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}_i:l} : x^{\mathcal{R}}_{\mathcal{P}_i:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}_i:l} : y^{\mathcal{R}}_{\mathcal{P}_i:l} + H^{\mathcal{R}}_{\mathcal{P}:l}]$
        $T_2 \leftarrow T_1 \circ_{(x^{\mathcal{I}}_{\mathcal{P}_i:l} - x^{\mathcal{R}}_{\mathcal{P}_i:l}), (y^{\mathcal{I}}_{\mathcal{P}_i:l} - y^{\mathcal{R}}_{\mathcal{P}_i:l})} \mathcal{P}_{i:l}$
        $\mathcal{U}[i, :, :] \leftarrow T_2$
    $[\mathcal{P}^{\mathcal{O}}_{1:l}, \ldots, \mathcal{P}^{\mathcal{O}}_{n:l}] \leftarrow T(\mathcal{U})$        ▷ Batched version
    return $[\mathcal{P}^{\mathcal{O}}_{1:l}, \ldots, \mathcal{P}^{\mathcal{O}}_{n:l}]$,
        $[(x^{\mathcal{O}}_{\mathcal{P}_1:l}, y^{\mathcal{O}}_{\mathcal{P}_1:l}), \ldots, (x^{\mathcal{O}}_{\mathcal{P}_n:l}, y^{\mathcal{O}}_{\mathcal{P}_n:l})]$, $(W^{\mathcal{O}}_{\mathcal{P}:l}, H^{\mathcal{O}}_{\mathcal{P}:l})$
```

alone yields limited speedups (under 2×), but our batched incremental inference amplifies the speedups. Algorithm 1 formally presents the batched incremental inference operation for layer l.

BatchedIncrementalInference first calculates the geometric properties of the output update patches and read-in contexts. A temporary tensor $\mathcal{U}$ is initialized to hold the input update patches with their read-in contexts. The **for** loop iteratively populates $\mathcal{U}$ with corresponding patches. Finally, $T_{:l}$ is applied to $\mathcal{U}$ to compute the output patches. We note that for the first layer in OBE, all input update patches will be identical to the occlusion patch. But for the later layers, the update patches will start to deviate depending on their locations and read-in contexts.

4.5 Automating KryptonGraph Generation/Execution for Arbitrary CNNs Represented as PyTorch Neural Computational Graphs

Krypton can accelerate OBE for arbitrary PyTorch CNNs. To achieve this, we develop a high-level abstraction called KryptonGraph and automate the generation and execution of it. For a given CNN, KryptonGraph handles the incremental CNN inference of that CNN by using PyTorch. The high-level process for KryptonGraph generation and execution is shown in Figure 10 and works as follows:

(1) Given a CNN model f, we use the utilities available in PyTorch to trace the structure of the CNN by providing a sample image as input. Since all CNNs are static in nature (i.e., the order of operator execution is not dependent on data), the structure obtained by tracing is guaranteed to be correct. The trace output is then exported to ONNX format [25], which is a convenient representation format for subsequent analysis.

(2) Dropout [26] operators in the CNN model are simply ignored, as they do not have any effect on CNN inference.

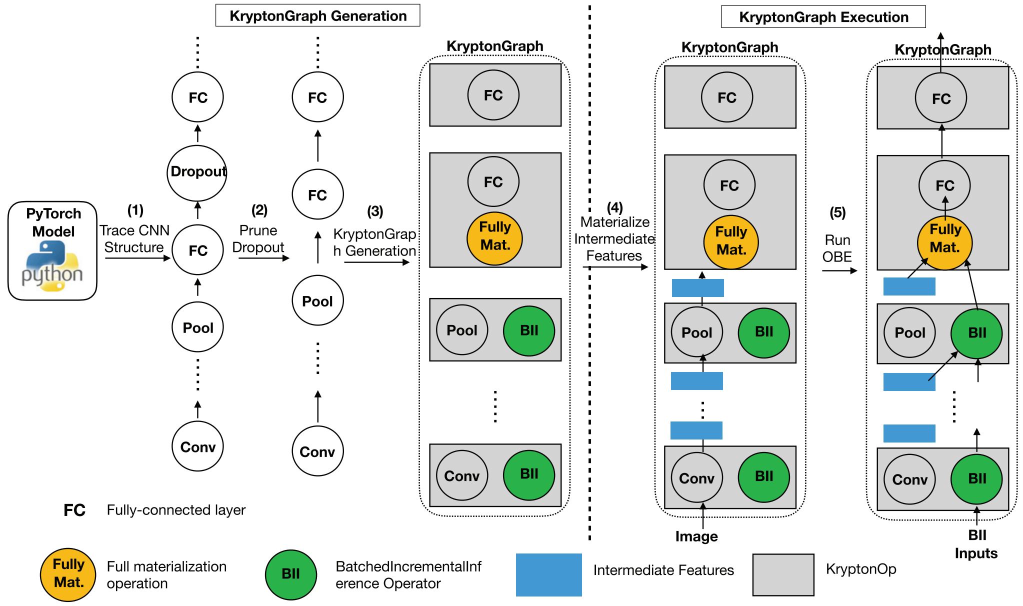


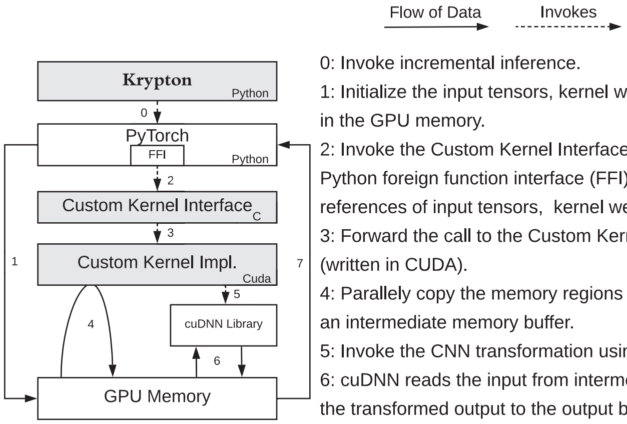
Fig. 10. KryptonGraph generation and execution process. For brevity, only a subgraph of a linear CNN is shown. The same method also applies to arbitrary DAG like CNNs.

(3) We then traverse the exported CNN model in topological order and create the corresponding KryptonGraph. For each operator T in the original CNN f, there will be a corresponding KryptonOP in the KryptonGraph that implements the BatchedIncrementalInference (Algorithm 1) for local context operators. Each KryptonOp also has a reference to the original CNN operator T, which will be used in the BatchedIncrementalInference method or directly invoked for global context operators that do not support incremental inference (e.g., Fully-Connected). Under the hood, KryptonOP is relying on the PyTorch framework for the actual execution of the corresponding CNN operator. The first global context operator that succeeds a local context operator will first fully materialize the updated input before invoking the full inference operator. Since all CNNs are created using a small number of low-level operators (e.g., convolution, pooling, and Fully-Connected), by implementing all corresponding types of KryptonOps, we are able to support arbitrary PyTorch CNNs as input.

(4) The generated KryptonGraph is then used for performing CNN inference for OBE. Given an input image $\mathcal{I}_{:img}$, we first materialize all intermediate outputs corresponding to incremental inference operators using one full inference.

(5) We then prepare occluded images ($\mathcal{I}'_{(x,y):img}$) for all positions in G. For batches of $\mathcal{I}'_{(x,y):img}$ as the input, we invoke the KryptonGraph in topological order and calculate the corresponding entries of heatmap M.

GPU Optimized Implementation. Empirically, we found a dichotomy between CPUs and GPUs: BatchedIncrementalInference yielded expected speedups on CPUs, but it performed dramatically poorly on GPUs. In fact, a naive implementation of BatchedIncrementalInference on GPUs was *slower* than full re-inference! We now shed light on why this is the case and how we tackled this issue. The **for** loop in line 7 of Algorithm 1 is essentially preparing the input for $T_{:l}$ by copying values (slices of the materialized tensor) from one part of GPU memory to another sequentially. A



0: Invoke incremental inference.
1: Initialize the input tensors, kernel weights and output buffer in the GPU memory.
2: Invoke the Custom Kernel Interface (written in C) using Python foreign function interface (FFI) support. Pass memory references of input tensors, kernel weights and output buffer.
3: Forward the call to the Custom Kernel Implementation (written in CUDA).
4: Parallely copy the memory regions from the input tensor to an intermediate memory buffer.
5: Invoke the CNN transformation using cuDNN.
6: cuDNN reads the input from intermediate buffer and writes the transformed output to the output buffer.
7: Read the output to the main memory or pass reference as the input to the next transformation.

Fig. 11. Custom GPU Kernel integration architecture.

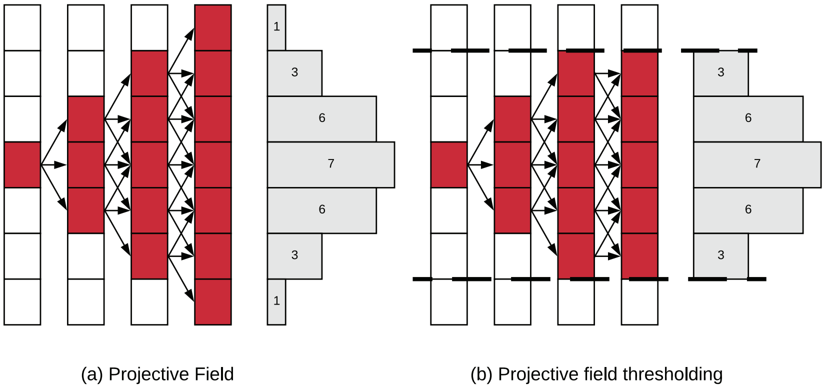
detailed profiling of the GPU showed that these *sequential memory copies are a bottleneck* for GPU throughput, since they throttle it from exploiting its massive parallelism effectively. To overcome this issue, we extended PyTorch by creating a custom CUDA kernel to perform input preparation more efficiently by *copying memory regions in parallel* for all items in the batched inference request. This is akin to a parallel **for** loop tailored for slicing the tensor. We then invoke $T_{:l}$, which is already hardware-optimized by modern deep learning tools [27]. This custom kernel is integrated to PyTorch using Python foreign function interface (FFI). The high-level architecture of the Custom Kernel integration is shown in Figure 11. Python FFI integrates with the Custom Kernel Interface layer, which then invokes the Custom Memory Copy Kernel Implementation. Also, since GPU memory might not be enough to fit all $|G|$ queries, the batch size for GPU execution might be smaller than $|G|$.

5 APPROXIMATE INFERENCE OPTIMIZATIONS FOR OBE

Since incremental inference is *exact*, i.e., it yields the same OBE heatmap as full inference, it does not exploit a capability of human perception: tolerance of some degradation in visual quality. Thus, we now build upon our IVM framework to create two novel heuristic approximate inference optimizations that trade off the heatmap's quality in a user-tunable manner to accelerate OBE further. We note that our optimizations operate at the logical level and are complementary to more physical-level optimizations such as low-precision computation [18] and model pruning [19]. We first present the techniques and then explain how to tune them.

5.1 Projective Field Thresholding

The *projective field* of a CNN neuron is the slice of the output tensor that is connected to it [28]. It is a term from neuroscience to describe the effects of a retinal cell on the output of the eye's neuronal circuitry [29]. This notion sheds light on the *growth of the size* of the update patches through the layers of a CNN. The three kinds of layers (Section 2.2) affect the projective field size growth differently. Transformations at the granularity of individual elements do not alter the projective field size. Global context transformations increase it to the whole output. But local spatial context transformations, which are the most crucial, increase it *gradually* at a rate determined by the filter kernel's size and stride: additively in the size and multiplicatively in the stride. The growth of the



(a) Projective Field

(b) Projective field thresholding

Fig. 12. (a) Projective field growth for 1-D Convolution (filter size 2, stride 1). (b) Projective field *thresholding*; $\tau = 5/7$.

projective field size implies the amount of FLOPs saved by IVM decreases as we go to the higher layers of a CNN. Eventually, the output update patch becomes as large as the output tensor. This growth is illustrated by Figure 12(a).

Our above observation motivates the main idea of this optimization, which we call projective field thresholding: *truncate* the projective field from growing beyond a given *threshold fraction* τ $(0 < \tau \leq 1)$ of the output size. This means inference in subsequent layers is approximate. Figure 12(b) illustrates the idea for a filter size 3 and stride 1. One input element is updated (shown in red/dark); the change propagates to three elements in the next layer and then five, but it then gets truncated, because we set $\tau = 5/7$. This approximation can alter the accuracy of the output values and the heatmap's visual quality. Empirically, we find that modest truncation is tolerable and does not affect the heatmap's visual quality too significantly.

To provide intuition on why the above happens, consider histograms on the side of Figures 12(a) and (b) that list the number of unique "paths" from the updated element to each element in the last layer. It resembles a Gaussian distribution, with the maximum paths concentrated on the middle element. Thus, for most of the output patch updates, truncation will only discard a few values at the "fringes" that contribute to an output element. Of course, we do not consider the weights on these "paths," which is dependent on the given trained CNN. Since the weights can be arbitrary, a tight formal analysis is unwieldy. But under some assumptions on the weights values (similar to the assumptions in Reference [30] for understanding the "receptive field" in CNNs), we can see that this distribution does indeed converge to a Gaussian. Thus, while this idea is a heuristic, it is grounded in a common behavior of real CNNs. In the following proposition, we formalize the effective projective field growth for a one-dimensional CNN with n convolutions layers. We also assume that all layers have the same weight normalized CNN filter kernel (i.e., sum of the weights add up to one).

Proposition 5.1. *For a one-dimensional CNN with n layers that uses the same weight normalized filter kernel, the theoretical projective field will grow $O(n)$ and the effective projective field will grow $O(\sqrt{n})$.*

Proof. The input is $u(t)$ and $t = 0, 1, -1, 2, -2, \ldots$ indexes the input pixels. Assume $u(t)$ is such that

$$u(t) = \begin{cases} 1, & t = 0, \\ 0, & t \neq 0. \end{cases} \tag{22}$$

Each layer has the same kernel $v(t)$ of size k. The kernel signal can be formally defined as

$$v(t) = \sum_{m=0}^{k-1} w(m)\delta(t-m), \tag{23}$$

where $w(m)$ is the weight for the mth pixel in the kernel. Without losing generality, we can assume the weights are normalized, i.e., $\sum_m w(m) = 1$. The output signal of the nth layer $o(t)$ is simply $o = u * v * \cdots * v$, convolving u with n such vs. To compute the convolution, we can use the Discrete Time Fourier Transform to convert the signals into the Fourier domain, and obtain

$$\begin{aligned} U(\omega) &= \sum_{t=-\infty}^{\infty} u(t)e^{-j\omega t} = 1,\ V(\omega) \\ &= \sum_{t=-\infty}^{\infty} v(t)e^{-j\omega t} = \sum_{m=0}^{k-1} w(m)e^{-j\omega t}. \end{aligned} \tag{24}$$

Applying the convolution theorem, we get the Fourier transform of o

$$\begin{aligned} \mathcal{F}(o) &= \mathcal{F}(u * v * \ldots * v)(\omega) = U(\omega).V(\omega)^n \\ &= \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t}\right)^n. \end{aligned} \tag{25}$$

With inverse Fourier transform

$$o(t) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t}\right)^n e^{j\omega t} d\omega. \tag{26}$$

The space domain signal $o(t)$ is given by the coefficients of $e^{-j\omega t}$. These coefficients turn out to be well studied in the combinatorics literature [31]. It can be shown that if $\sum_m w(m) = 1$ and $w(m) \geq 0\,\forall\, m$, then

$$\begin{aligned} o(t) &= p(S_n = t), \\ \text{where } S_n &= \sum_{i=1}^{n} X_i \text{ and } p(X_i = m) = w(m). \end{aligned} \tag{27}$$

From the central limit theorem, as $n \to \infty$, $\sqrt{n}(\frac{1}{n}S_n - \mathbb{E}[X]) \sim \mathcal{N}(0, Var[X])$ and $S_n \sim \mathcal{N}(n\,\mathbb{E}[X]), nVar[X])$. As $o(t) = p(S_n = t)$, $o(t)$ also has a Gaussian shape with

$$\mathbb{E}[S_n] = n\sum_{m=0}^{k-1} mw(m), \tag{28}$$

$$Var[S_n] = n\left(\sum_{m=0}^{k-1} m^2 w(m) - \left(\sum_{m=0}^{k-1} mw(m)\right)^2\right). \tag{29}$$

This indicates that $o(t)$ decays from the center of the projective field squared exponentially according to the Gaussian distribution. As the rate of decay is related to the variance of the Gaussian and assuming the size of the effective projective field is one standard deviation, the size can be expressed as

$$\sqrt{Var[S_n]} = \sqrt{nVar[X_i]} = O(\sqrt{n}). \tag{30}$$

□

However stacking more convolution layers would grow the theoretical projective field linearly. But the effective projective field size is shrinking at a rate of $O(1/\sqrt{n})$. Overall, since most of the contributions to the output elements are concentrated around the center, projective field truncation is often affordable. Note that this optimization is only feasible *in conjunction with* our incremental inference framework (Section 3) to reuse the remaining parts of the tensors and save FLOPs. We extend the formulas for the output-input coordinate calculations to account for τ. For the width dimension, the new formulas are as follows (similarly for the height dimension):

$$W^{\mathcal{O}}_{\mathcal{P}:l} = \mathtt{min}\left(\left\lceil \left(W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1\right)/S_{x:l} \right\rceil, W^{\mathcal{O}}_{\mathcal{P}:l}\right), \tag{31}$$

$$\text{If } W^{\mathcal{O}}_{\mathcal{P}:l} > \mathtt{round}\left(\tau \times W^{\mathcal{O}}_{:l}\right) :, \tag{32}$$

$$W^{\mathcal{O}}_{\mathcal{P}:l} = \mathtt{round}\left(\tau \times W^{\mathcal{O}}_{:l}\right), \tag{33}$$

$$W^{\mathcal{I}}_{\mathcal{P}_{new}:l} = W^{\mathcal{O}}_{\mathcal{P}:l} \times S_{x:l} - W_{\mathcal{K}:l} + 1, \tag{34}$$

$$x^{\mathcal{I}}_{\mathcal{P}:l} \mathrel{+}= \left(W^{\mathcal{I}}_{\mathcal{P}:l} - W^{\mathcal{I}}_{\mathcal{P}_{new}:l}\right)/2, \tag{35}$$

$$W^{\mathcal{I}}_{\mathcal{P}:l} = W^{\mathcal{I}}_{\mathcal{P}_{new}:l}, \tag{36}$$

$$x^{\mathcal{O}}_{\mathcal{P}:l} = \mathtt{max}\left(\left\lceil \left(P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1\right)/S_{x:l} \right\rceil, 0\right). \tag{37}$$

Equation (31) calculates the width assuming no thresholding. But if the output width exceeds the threshold, it is reduced as per Equation (33). Equation (34) calculates the input width that would produce an output of width $W^{\mathcal{O}}_{\mathcal{P}:l}$; we can think of this as making $W^{\mathcal{I}}_{\mathcal{P}:l}$ the subject of Equation (31). If the new input width is smaller than the original input width, the starting x coordinate should be updated as per Equation (35) s.t. the new coordinates correspond to a "center crop" compared to the original. Equation (36) sets the input width to the newly calculated input width. Equation (37) calculates the x coordinate of the output update patch.

Theoretical Speedups. We modify our "static analysis" framework to determine the theoretical speedup of incremental inference (Section 3) to also include this optimization using the above formulas. Consider a square occlusion patch placed on the center of the input image. Figure 13(a) plots the new theoretical speedups for varying patch sizes forthree popular CNNs for different τ values. As expected, as τ goes down from 1, the theoretical speedup goes up for all CNNs. Since lowering τ approximates the heatmap values, we also plot the mean square error (MSE) of the elements of the exact and approximate output tensors produced by the final Convolution or Pooling layers on a sample (n = 30) of real-world images. Figure 13(b) shows the results. As expected, as τ drops, MSE increases. But interestingly, the trends differ across the CNNs due to their different architectural properties. MSE is especially low for VGG-16, since its projective field growth is rather slow relative to the other CNNs. We acknowledge that using MSE as a visual quality metric and tuning τ are both unintuitive for humans. We mitigate these issues in Section 4.3 by using a more intuitive quality metric and by presenting an automated tuning method for τ.

5.2 Adaptive Drill-down

This heuristic optimization is based on our observation about a peculiar semantics of OBE that lets us modify how G (the set of occlusion patch locations) is specified and handled, especially in the non-interactive specification mode. We explain our intuition with an example. Consider a radiologist explaining a CNN prediction for diabetic retinopathy on a tissue image. The region of interest typically occupies only a tiny fraction of the image. Thus, it is an overkill to perform regular OBE for *every* patch location: most of the (incremental) inference computations are effectively "wasted"

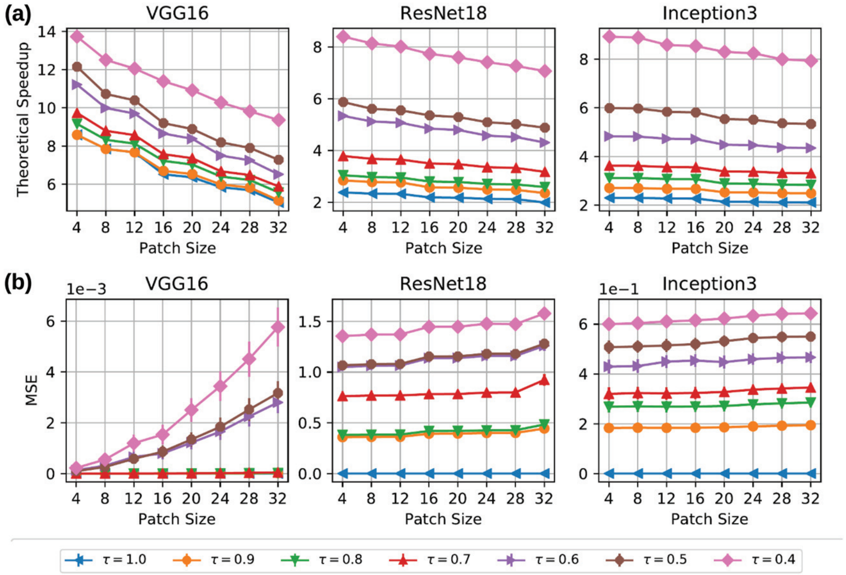


Fig. 13. (a) Theoretical speedups with projective field thresholding. (b) Mean Square Error between exact and approximate output of final Convolution/Pooling layers.

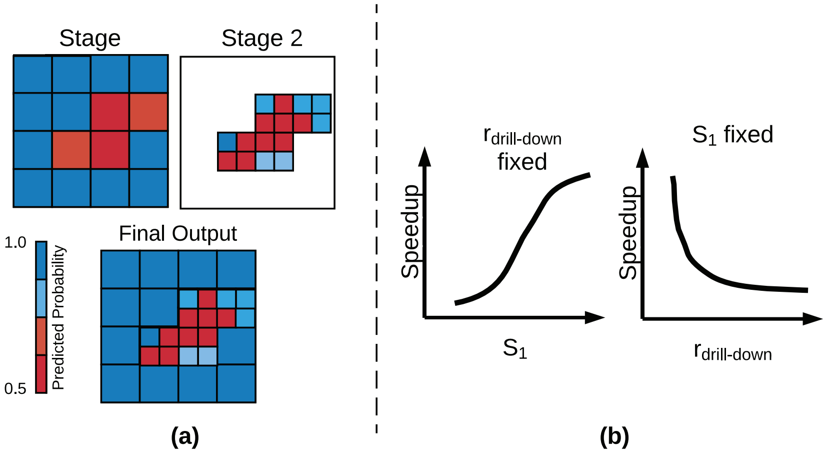


Fig. 14. (a) Schematic illustration of the adaptive drill-down idea. (b) Conceptual depiction of the effects of S_1 and $r_{drill-down}$ on the theoretical speedup.

on uninteresting regions. In such cases, we modify the OBE workflow to produce an approximate heatmap using a two-stage process, illustrated by Figure 14(a).

In stage one, we produce a lower resolution heatmap by using a larger stride—we call it *stage one stride* S_1. Using this heatmap, we identify the regions of the input that see the largest drops in predicted probability of the label L. Given a predefined parameter *drill-down fraction*, denoted $r_{drill-down}$, we select a proportional number of regions based on the probability drops. In stage two, we perform OBE only for these regions with original stride value (we also call this *stage two stride*, S_2) for the occlusion patch to yield a portion of the heatmap at the original higher resolution. Since this process "drills down" adaptively based on the lower resolution heatmap, we call it adaptive drill-down. Note that this optimization also builds upon the incremental inference optimizations of Section 3, but it is *orthogonal* to projective field thresholding and can be used in addition.

Theoretical Speedups. We now define a notion of theoretical speedup for this optimization; this is independent of the theoretical speedup of incremental inference. We first explain the effects of $r_{drill-down}$ and S_1. Setting these parameters is an application-specific balancing act. If $r_{drill-down}$ is low, only a small region will need re-inference at the original resolution, which will save a lot of FLOPs. But this may miss some regions of interest and thus, compromise important explanation details. Similarly, a large S_1 also saves a lot of FLOPs by reducing the number of re-inference queries in stage one. But it runs the risk of misidentifying interesting regions, especially when the size of those regions are smaller than the occlusion patch size. We now define the theoretical speedup of adaptive drill-down as the ratio of the number of re-inference queries for regular OBE without this optimization to that with this optimization. We only need the counts, since the occlusion patch dimensions are unaltered, i.e., the cost of a re-inference query is the same with or without this optimization. Given a stride S, the number of re-inference queries is $\frac{H_{I_{img}}}{S} \cdot \frac{W_{I_{img}}}{S}$. Thus, the theoretical speedup is given by the following equation. Figure 14(b) illustrates how this ratio varies with S_1 and $r_{drill-down}$.

$$\mathsf{speedup} = \frac{S_1^2}{S_2^2 + r_{drill-down} \cdot S_1^2} \tag{38}$$

5.3 Automated Parameter Tuning

We now present automated parameter tuning methods for easily configuring our approximate inference optimizations.

Tuning Projective Field Thresholding. As Section 4.1 explained, τ controls the visual quality of the heatmap. There is a spectrum of visual quality degradation: imperceptible changes to major structural changes. But mapping τ to visual quality directly is likely to be unintuitive for users. Thus, to measure visual quality more intuitively, we adopt a cognitive science-inspired metric called Structural Similarity (SSIM) Index, which is widely used to quantify human-perceptible differences between two images [23]. In our case, the two "images" are the original and approximate heatmaps. SSIM is a number in $[-1, 1]$, with 1 meaning a perfect match. SSIM values in the $[0.90, 0.95]$ range are considered almost imperceptible distortions in many practical multimedia applications such as image compression and video encoding [23].

Our tuning process for τ has an offline "training" phase and an online usage phase. The offline phase relies on a set of sample images (default 30) from the same application domain. We compute SSIM for the approximate and exact heatmaps for all sample images for a few τ values (default $1.0, 0.9, 0.8, \ldots, 0.4$). We then learn a second-degree polynomial curve for SSIM as a function of τ with these data points. Figure 15(a) illustrates this phase and the fit SSIM-τ curves for three different CNNs using sample images from an OCT dataset (Section 5). In the online phase, when OBE is needed on a given image, we expect the user to provide a *target SSIM* for the quality–runtime tradeoff they want (1 yields the exact heatmap). We can then use our learned curve to map this target SSIM to the lowest τ. Figure 15(b) shows the CDFs of differences between the target SSIM (0.9) and the actual SSIM yielded when using our auto-tuned τ on both the training set and a holdout test set (also 30 images). In 80% of the cases, the actual SSIM was *better* than the user-given target; never once did the actual SSIM go 0.1 below the target SSIM. This suggests that our auto-tuning method for τ works, is robust, and applicable to different CNNs.

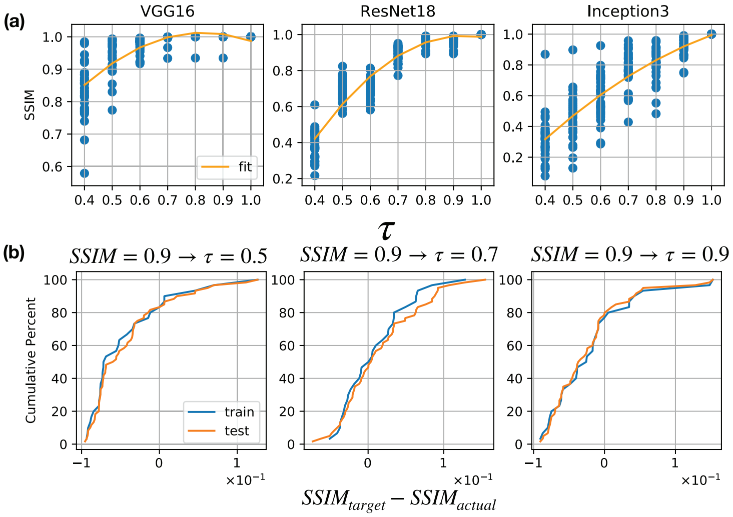


Fig. 15. (a) Fitting a second-order curve for SSM against τ on a sample of the OCT dataset. (b) CDFs of deviation of actual SSIM from the target SSIM (0.9) with our auto-tuned τ, which turned out to be 0.5, 0.7, and 0.9 for VGG-16, ResNet-18, and Inception-V3, respectively.

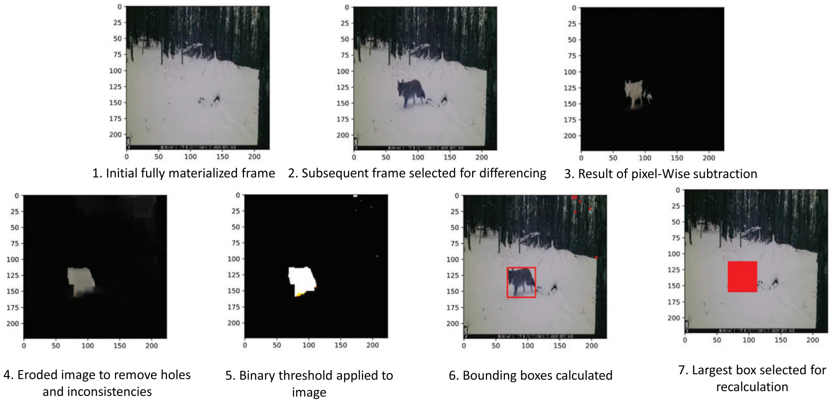
Tuning Adaptive Drill-down. As Section 5.2 explained, the speedup offered by adaptive drill-down is controlled by two parameters: stage one stride S_1 and drill-down fraction $r_{drill-down}$. We expect the user to provide $r_{drill-down}$ (default 0.25), since it captures the user's intuition about how large or small the region of interest is likely to be in the images in their specific application domain and dataset. We also expect the user to provide a "target speedup" ratio (default 3) for using this optimization to capture their desired quality-runtime tradeoff. The higher the user's target speedup, the more we sacrifice the quality of the "non-interesting regions" ($1 - r_{drill-down}$ fraction of the heatmap). Our automated tuning process sets S_1 using these two user-given settings. Unlike the tuning of τ, setting S_1 is more direct, since this optimization relies on the number of re-inference queries, not SSIM. Let *target* denote the target speedup; the original occlusion patch stride is S_2. Equation (39) shows how we calculate S_1; it is obtained by making $S1$ the subject of Equation (38). Since S_1 cannot be larger than the image width W_{img} (similarly H_{img}) and due to the constraint of $(1 - r_{drill-down} \cdot \texttt{speedup})$ being positive, we also have an upper bound on the possible speedups as per Equation (40):

$$S_1 = \sqrt{\frac{target}{1 - r_{drill-down} \cdot target}} \cdot S_2, \tag{39}$$

$$speedup < \texttt{min}\left(\frac{W_{img}^2}{S_2^2 + r_{drill-down} \cdot W_{img}^2}, \frac{1}{r_{drill-down}}\right). \tag{40}$$

6 APPROXIMATE INFERENCE OPTIMIZATIONS FOR ORV

We first explain the idea and the intuition behind our approach to support ORV using Krypton's incremental inference engine. We then dive into the two main stages in our approach: (1) *frame differencing* and (2) *scene separation*. We conclude by presenting the end-to-end workflow for executing ORV using Krypton.



1. Initial fully materialized frame 2. Subsequent frame selected for differencing 3. Result of pixel-Wise subtraction

4. Eroded image to remove holes and inconsistencies 5. Binary threshold applied to image 6. Bounding boxes calculated 7. Largest box selected for recalculation

Fig. 16. The process of frame differencing. To better illustrate the frame differencing scheme, we select a subsequent frame some significant distance away from the base frame.

6.1 High-level Idea

Krypton's ability to reduce the necessary recalculations across similar images provides a unique opportunity for accelerating CNN-based object recognition in fixed-angle camera videos. Frame-to-frame differences in video are unlikely to be significant, and Krypton can exploit the similarities between frames to limit re-inferencing to only the region of change. After performing an initial full inference using a base frame, Krypton only needs to perform incremental inference on the changed regions, whereas the naive approach will perform full inference for all video frames. Reducing strain on systems for video analytics is an area of high importance, as the various applications of ORV—traffic monitoring, surveillance, animal tracking—will largely run on edge devices without high compute power. In these settings, using Krypton can allow for faster and more efficient inference on video inputs, thus improving the operation of these devices.

However, accelerating ORV using Krypton poses several unique challenges. First, unlike in OBE where the changed region is exact and of rectangular shape, changed region in ORV can be of arbitrary shape. Furthermore, due to the inherent noise in video frames there can be more than one potentially small changed regions. Hence, finding the most important changed region from a given frame is non-trivial. Second, over time the materialized intermediate features can get invalid. This can happen in various ways such as due to a slight movement in the camera and change in the lighting levels. To overcome these issues, we develop techniques for *frame differencing* and *scene separation*, which essentially incorporate approximations to cast ORV as an OBE problem. Next, we dive into these techniques in more detail.

6.2 Frame Differencing

We use an approximate frame differencing approach to identify the single most important changed region for incremental inference in each frame. Algorithm 2 formally and Figure 16 pictorially present our approach. It gets three inputs: $base\_frame$, which is treated as the background; new_frame, from which we want to find the changed region; and $threshold$, which will be use to identify the changed pixels. By using pixel-subtraction, we identify all of the changes between current frame and the base frame on a per-pixel basis. Thresholding the resultant data eliminates noise and restricts the necessary re-inferencing to a more limited scope. We calculate bounding boxes for the remaining areas of difference to provide a more regular shape for feeding them into Krypton. These bounding boxes can often overlap, so we collapse them into larger bounding boxes to eliminate any overlaps. The largest of the resultant bounding boxes is selected as the most important changed region for incremental inference, and we return the coordinates and dimensions of this box. Smaller *threshold* values tend to select smaller changed regions and hence higher speedups. But they also reduce the accuracy of the generated predictions. The most optimal value for *threshold* (value between 0 and 255) is largely dependent on the chosen use case. In our experiments, we found that a *threshold* value of 40 provides a reasonable tradeoff between runtime and accuracy.

ALGORITHM 2: FrameDifferencing

```
Input:
  base_frame : Base frame (background of the video)
  new_frame : New frame with a potential object
  threshold : Pixel-wise change identification threshold
Output:
  $x_{\mathcal{P}}, y_{\mathcal{P}}$ : Starting coordinates of the largest changed region
  $w_{\mathcal{P}}, h_{\mathcal{P}}$ : Width and the height of the largest changed region

procedure FrameDifferencing(base_frame, new_frame, threshold)
    binary_img ← abs(new_frame - base_frame) // threshold
    bboxes ← Calculate bounding boxes for objects in binary_img
    collapsed_bboxes ← Collapse overlapping bounding boxes in bboxes
    $x_{\mathcal{P}}, y_{\mathcal{P}}, w_{\mathcal{P}}, h_{\mathcal{P}}$ ← Find the largest bounding box from collapsed_bboxes
    return $x_{\mathcal{P}}, y_{\mathcal{P}}, w_{\mathcal{P}}, h_{\mathcal{P}}$
```

6.3 Scene Separation

Our approach for ORV assumes there is a fixed background on which an object may appear. We call it a base frame. When starting, we use the first frame in the video as the base frame. However, as time passes it is possible that the actual background of the current frame is different to the selected base frame. This interferes with our frame differencing approach and ends up generating very large changed regions, which diminishes the gains of incremental inference. To address this problem, we introduce the notion of a *scene* to ORV. When the size of the selected changed region is larger than some significant fraction of the size of the base frame, we create a new scene and reset the base frame to the current frame. This fraction controls the tradeoff between how often we need to fully materialize a frame and how often the materialized features get reused subsequently. As a practical rule of thumb, we find that a fraction of 50% suffices to balance this tradeoff and still obtain good speedups without affecting accuracy much. At this point Krypton also re-materializes all the intermediate features for the current base frame that will be used in subsequent incremental inference of the next scene.

6.4 Putting It All Together

We summarize the end-to-end workflow for supporting ORV using Krypton. Algorithm 3 presents it formally. We are given a video V, *threshold* for frame differencing, *max_patch_size* for scene separation, `KryptonGraph` *kg*, which performs the incremental inference, and the *batch_size*, which will be used for batching multiple incremental inference requests. We first initialize the base frame to the first frame in the video. We then start iterating through frames in V, find the changed region by calling `FrameDifferencing`, and append them to a batch. Two possible events can occur to trigger incremental inference to be run on the compiled batch of changed regions. First, if the changed region size exceeds *max_patch_size* and encounters a new scene. Second, the current batch size reaches the *max_batch_size*. This *max_batch_size* is necessary to avoid the possibility

ALGORITHM 3: ObjectRecognitionInVideo

```
Input:
  $V$ : Input Video
  $threshold$ : Pixel-wise change identification threshold
  $max\_patch\_size$ : Maximum size of a patch for separating scenes
  $batch\_size$ : Batch size for incremental inference
  $kg$ : KryptonGraph
Output:
  $predictions$ : Predicted label for each frame

procedure ObjectRecognitionInVideo
    $X_{\mathcal{P}} \leftarrow []$; $Y_{\mathcal{P}} \leftarrow []$; $W_{\mathcal{P}} \leftarrow []$; $H_{\mathcal{P}} \leftarrow []$; $frames \leftarrow []$; $predictions \leftarrow []$
    $base\_frame \leftarrow V.next()$; $new\_frame \leftarrow V.next()$
    while $new\_frame \neq$ NULL do
        $x_{\mathcal{P}}, y_{\mathcal{P}}, w_{\mathcal{P}}, h_{\mathcal{P}} \leftarrow$ FrameDifferencing($base\_frame$, $new\_frame$, $threshold$)
        if $w_{\mathcal{P}} \times h_{\mathcal{P}} \geq max\_patch\_size$ then                    ▷ new scene
            $labels \leftarrow$ RunIncrementalInference($frames$, $X_{\mathcal{P}}$, $Y_{\mathcal{P}}$, $W_{\mathcal{P}}$, $H_{\mathcal{P}}$, $kg$)
            $predctions$.extend($labels$)
            $label \leftarrow kg$.materialize_intermediate_data($new\_frame$)
            $predictions$.append($label$)
            $base\_frame \leftarrow new\_frame$
            $X_{\mathcal{P}} \leftarrow []$; $Y_{\mathcal{P}} \leftarrow []$; $W_{\mathcal{P}} \leftarrow []$; $H_{\mathcal{P}} \leftarrow []$; $frames \leftarrow []$
        else                                                                 ▷ same scene
            $X_{\mathcal{P}}$.append($x_{\mathcal{P}}$), $Y_{\mathcal{P}}$.append($y_{\mathcal{P}}$), $W_{\mathcal{P}}$.append($w_{\mathcal{P}}$), $H_{\mathcal{P}}$.append($h_{\mathcal{P}}$)
            $frames$.append($new\_frame$)
            if len($frames$) == $batch\_size$ then                          ▷ batch size reached
                $labels \leftarrow$ RunIncrementalInference($frames$, $X_{\mathcal{P}}$, $Y_{\mathcal{P}}$, $W_{\mathcal{P}}$, $H_{\mathcal{P}}$, $kg$)
                $predctions$.extend($labels$)
                $X_{\mathcal{P}} \leftarrow []$; $Y_{\mathcal{P}} \leftarrow []$; $W_{\mathcal{P}} \leftarrow []$; $H_{\mathcal{P}} \leftarrow []$; $frames \leftarrow []$
        $new\_frame \leftarrow V$.next()
    return $predictions$

procedure RunIncrementalInference($frames$, $X_{\mathcal{P}}$, $Y_{\mathcal{P}}$,$W_{\mathcal{P}}$, $H_{\mathcal{P}}$, $kg$)
    $W_{\mathcal{P}_{max}} \leftarrow$ max($W_{\mathcal{P}}$); $H_{\mathcal{P}_{max}} \leftarrow$ max($H_{\mathcal{P}}$); $\mathcal{P} \leftarrow []$
    for $i \in [1, \ldots,$ len($X_{\mathcal{P}}$)$]$ do
        if $X_{\mathcal{P}}$ [i] + $W_{\mathcal{P}_{max}}$ > width($frames$[i]) then
            $X_{\mathcal{P}}$ [i] $\leftarrow$ width($frames$[i]) - $W_{\mathcal{P}_{max}}$
        if $Y_{\mathcal{P}}$ [i] + $H_{\mathcal{P}_{max}}$ > height($frames$[i]) then
            $Y_{\mathcal{P}}$ [i] $\leftarrow$ height($frames$[i]) - $H_{\mathcal{P}_{max}}$
        $\mathcal{P}$.append($frames$[i][$X_{\mathcal{P}}$[i]:$X_{\mathcal{P}}$[i]+$W_{\mathcal{P}_{max}}$, $Y_{\mathcal{P}}$[i]:$Y_{\mathcal{P}}$[i]+$H_{\mathcal{P}_{max}}$])
    $labels \leftarrow kg$.incremental_inference($\mathcal{P}$, $X_{\mathcal{P}}$, $Y_{\mathcal{P}}$, $W_{\mathcal{P}_{max}}$, $H_{\mathcal{P}_{max}}$)
    return $labels$
```

of exhausting hardware resource such as GPU memory. Unlike in OBE where all patches are of same size, changed regions in ORV are of arbitrary size. Thus, when invoking incremental inference on a batch of changed regions, we first find the maximum size as the final patch size. This is formally presented in the `RunIncrementalInference` procedure. Finally, we return predicted class labels for all the frames in V as the output.

7 EXPERIMENTAL EVALUATION

We integrated our optimization techniques with the popular deep learning framework PyTorch to create a tool we call Krypton. We now evaluate the speedups yielded by Krypton for OBE and ORV for different deep CNNs and datasets. We also deep dive into the contributions of each of our optimization techniques.

Datasets. For OBE, we use four diverse real-world datasets: *OCT*, *Chest X-Ray*, *ImageNet*, and *HAR*. *OCT* has about 84,000 optical coherence tomography retinal images with four classes: CNV, DME, DRUSEN, and NORMAL; CNV (choroidal neovascularization), DME (diabetic macular edema), and DRUSEN are varieties of diabetic retinopathy. *Chest X-Ray* has about 6,000 X-ray images with three classes: VIRAL, BACTERIAL, and NORMAL; VIRAL and BACTERIAL are varieties of pneumonia. *HAR* is a time series dataset of sensor data collected from body-worn accelerometers and gyroscopes at 50 Hz and has six modalities (acceleration and orientation along x,y,z axes). It has about 10,000 data points with each having a window size of 2.56 secs and has six classes: SITTING; STANDING; WALKING; WALKING UP; WALKING DOWN; and LAYING. Both *OCT* and *Chest X-Ray* are from a recent radiology study that applied deep CNNs to detect the respective diseases [2]. *ImageNet* is a benchmark dataset in computer vision [32]; we use a sample of 1,000 images with 200 classes. *HAR* is also a benchmark dataset used for human activity recognition [33]. For ORV, we use a sample (n = 5) of fixed-angle trail camera videos collected from [34]. Collectively they have 106 seconds of video data.

Workloads. For OBE on image data, we use three diverse ImageNet-trained deep CNNs: VGG16 [9], ResNet18 [35], and Inception3 [36], obtained from Reference [37]. They complement each other in terms of model size, architectural complexity, computational cost, and our predicted theoretical speedups (Figure 6 in Section 4). For *OCT* and *Chest X-Ray*, the three CNNs were fine-tuned by retraining their final Fully-Connected layers as per standard practice. The OBE heatmaps are plotted using Python Matplotlib's `imshow` method using the `jet_r` color scheme; we set the maximum threshold to $\mathtt{min}(1, 1.25p)$ and minimum to $0.75p$, where p is predicted class probability on a given image. All images are resized to the input size required by the CNNs (224×224 for VGG16 and ResNet18; 299×299 for Inception3); no additional pre-processing was done. For OBE on *HAR* data, we use a custom one-dimensional CNN. The architecture of the CNN is similar to other one-dimensional CNNs used in similar tasks in practice and is shown in Figure 19(a). We use a window size of 2.56 seconds and after training it yielded a test accuracy of 85%. For ORV, we use ImageNet-trained VGG16 model to recognize animals from video frames. All CPU-based experiments were executed with a thread parallelism of 8.

Experimental Setup. We use a machine with 32 GB RAM, Intel i7-6700 3.40 GHz CPU, and NVIDIA Titan X (Pascal) GPU with 12 GB memory. The machine runs Ubuntu 16.04 with PyTorch version 0.4.0, CUDA version 9.0, and cuDNN version 7.1.2. All reported runtimes are the average of three runs, with 95% confidence intervals shown.

7.1 End-to-end Runtimes for OBE

OBE for two-dimensional CNNs. We focus on the most common OBE scenario of producing the whole heatmap; G is automatically created ("non-interactive" mode). We use an occlusion patch of size 16 and stride of 4. We compare two variants of Krypton: Krypton-Exact uses only incremental inference (Section 3), while Krypton-Approximate uses our approximate inference optimizations, too (Section 4). The main baseline is *Naive*, the current dominant practice of performing full inference for OBE with just only batching. We have another baseline on GPU: *Naive Inc. Inference-Exact*, which is a direct implementation of Algorithm 1 in PyTorch/Python without

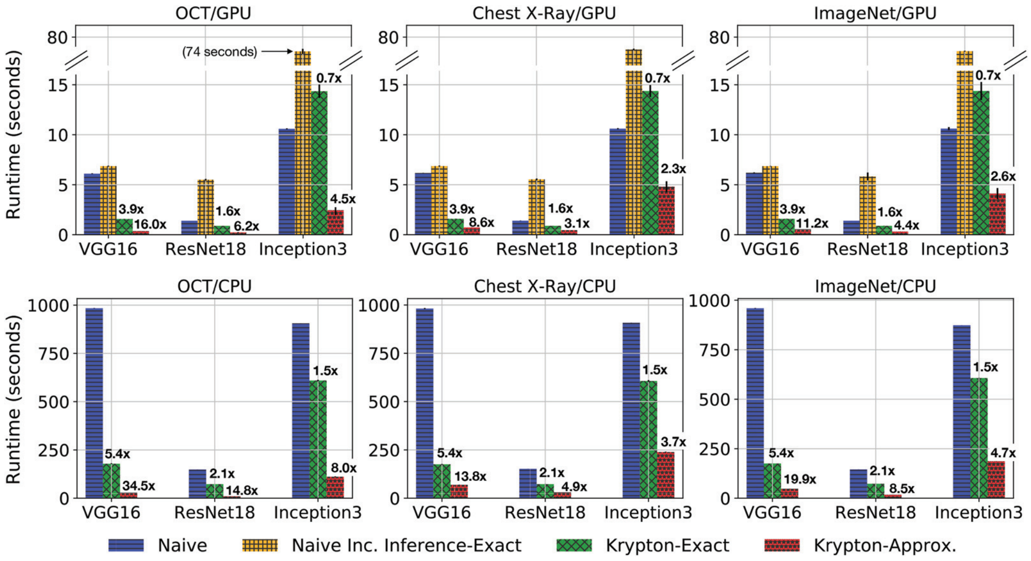


Fig. 17. End-to-end runtimes of Krypton and baselines for OBE on three image datasets, three CNNs, and both GPU and CPU.

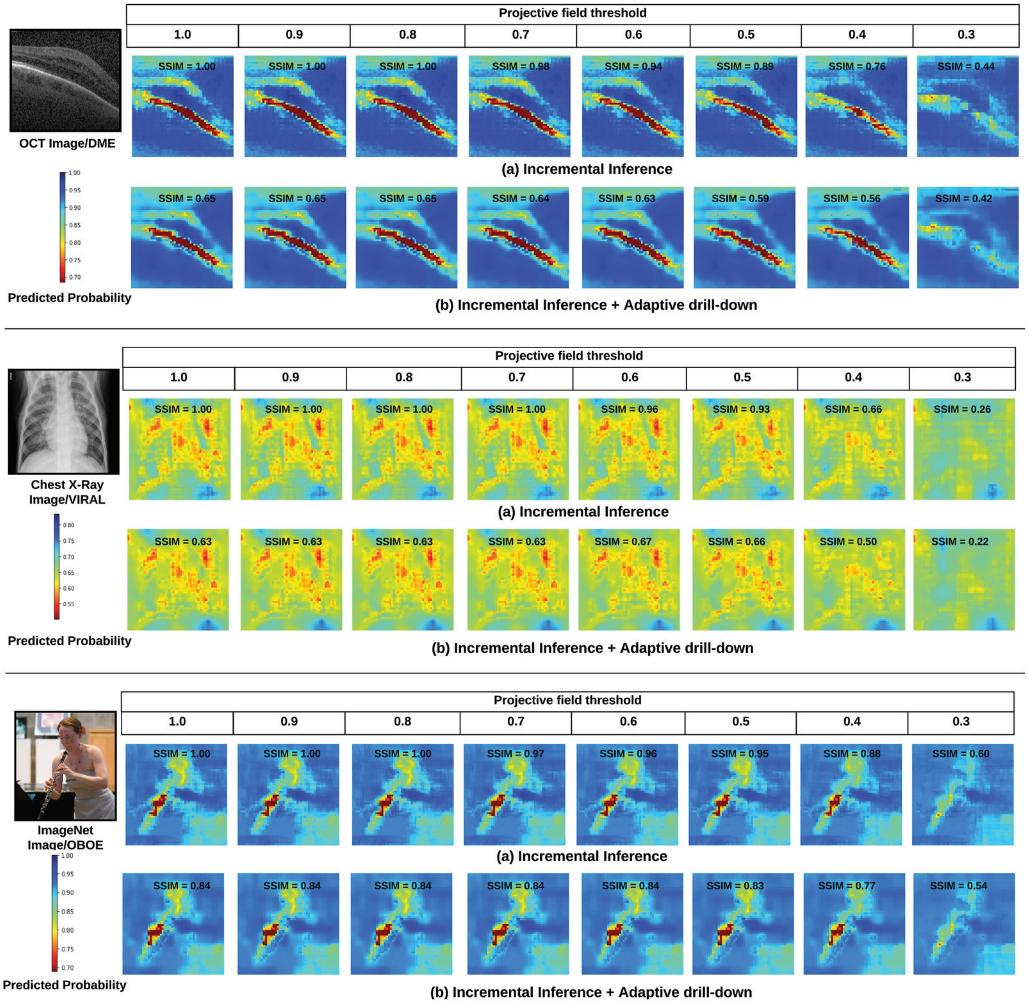
using our GPU-optimized CUDA kernel (Section 3.4). Note that *Naive Inc. Inference-Exact* is not relevant on CPU.

We set the adaptive drill-down parameters based on the semantics of each dataset's prediction task (Section 4.3). For *OCT*, since the region of interest is likely to be small, we set $r_{drill-down} = 0.1$ and $target = 5$. For *Chest X-Ray*, the region of interest can be large; so, we set $r_{drill-down} = 0.4$ and $target = 2$. For *ImageNet*, which is in between, we use the Krypton default of $r_{drill-down} = 0.25$ and $target = 3$. Throughout, τ is auto-tuned with a target SSIM of 0.9 (Section 4.3). All GPU-based experiments use a batch size of 128; for CPUs, the batch size is 16. Figure 17 presents the results.

Overall, we see Krypton offers significant speedups across the board on both GPU and CPU, with the highest speedups seen by Krypton-Approximate on *OCT* with VGG16: 16× on GPU and 34.5× on CPU. The highest speedups of Krypton-Exact are also on VGG16: 3.9× on GPU and 5.4× on CPU. The speedups of Krypton-Exact are identical across datasets for a given CNN, since it does not depend on the image semantics, unlike Krypton-Approximate due to its parameters. Krypton-Approximate sees the highest speedups on *OCT,* because our auto-tuning yielded the lowest $r_{drill-down}$, highest target speedup, and lowest τ on that dataset.

The speedups are lower with ResNet18 and Inception3 than VGG16 due to their architectural properties (kernel filter dimensions, stride, etc.) that make the projective field grow faster. Moreover, Inception3 has a complex DAG architecture with more branches and depth-wise concatenation, which limits GPU throughput for incremental inference. In fact, Krypton-Exact on GPU shows a minor slowdown (0.7×) with Inception3. But Krypton-Approximate still offers speedups on GPU with Inception3 (up to 4.5×). We also see that ResNet18 and VGG16 almost near their theoretical speedups (Figure 6) but Inception3 does not. Note that the theoretical speedup definition only counts FLOPs and does not account for memory stalls.

Finally, the speedups are higher on CPU than GPU; this is because CPU suffers less from memory stalls during incremental inferences. But the *absolute* runtimes are much lower on GPU as expected. Overall, Krypton reduces OBE runtimes substantially for multiple datasets and deep CNNs. We also ran an experiment in the "interactive" mode by reducing $|G|$. As expected, speedups



(a) Incremental Inference

(b) Incremental Inference + Adaptive drill-down

(a) Incremental Inference

(b) Incremental Inference + Adaptive drill-down

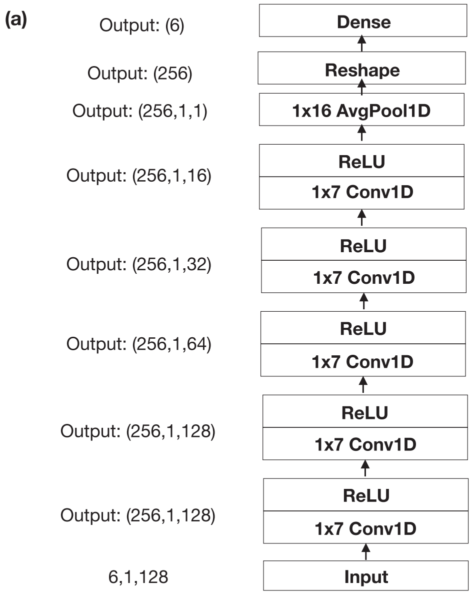
(a) Incremental Inference

(b) Incremental Inference + Adaptive drill-down

Fig. 18. Occlusion heatmaps for sample images (CNN model = VGG16, occlusion patch size = 16, patch color = black, occlusion patch stride ($S\ or\ S_2$) = 4. For *OCT* $r_{drill\_down} = 0.1$ and *target* = 5. For *Chest X-Ray* $r_{drill\_down} = 0.4$ and *target* = 2. For *ImageNet* $r_{drill\_down} = 0.25$ and *target* = 3). For a projective field threshold value of 0.3, we see significant degradation of heatmap quality due to the significant information loss from truncation.

go down with $|G|$ due to the reduction in amortization benefits. These additional results are presented in Section 7.2. Figure 18 presents occlusion heatmaps for a sample image from each dataset with (a) *incremental inference* for different *projective field threshold* values and (b) *incremental inference* with *adaptive drill-down* for different *projective field threshold* values. The predicted class label for *OCT*, *Chest X-Ray*, and *ImageNet* are DME, VIRAL, and OBOE, respectively.

OBE for one-dimensional CNNs. We compare full CNN inference time versus incremental inference time for running OBE using a CNN trained to identify different human postures from body-worn sensor data [33]. We use a zero valued occlusion patch of size 4, stride of 1, batch size of 125, and compare runtimes on both CPU and GPU environments. Figure 19(b) presents the results.



(b)

	Full Inference	Inc. Inference (speedup)
Theoretical MFLOPs	106	19 (5.6X)
CPU Time	0.41 s	0.13 s (3.2X)
GPU Time	0.01 s	0.01 s (1.0X)

Fig. 19. (a) Architecture of the one-dimensional CNN used for the human activity classification task. (b) Runtimes and theoretical FLOP counts for full inference and incremental inference ($\tau = 1$) when running OBE using the one-dimensional CNN (occlusion patch size = 4, occlusion patch value = 0, occlusion patch stride = 1, batch size = 125).

One full inference through the one-dimensional CNN requires performing 106 Mega floating point operations (MFLOPs) and performing incremental inference drops it to 19 MFLOPs resulting in a theoretical speedup of 5×. It should be noted that the amount of computations performed by a typical one-dimensional CNN is relatively smaller than the amount of computations performed by a two-dimensional image CNN. For example, VGG16 performs 16 GFLOPs versus 19 MFLOPs by our one-dimensional CNN. On the CPU environment incremental inference reduces the OBE runtime for a single time series window by 3×. However, on the GPU environment it did not yield any speedups. This is because on the GPU the relatively low amount of computations needed for OBE makes the overheads of invoking GPU kernels dominate the overall runtime. Figure 20 presents visuals on how the probability for the predicted class label change for a sample of time series windows as we slide the occlusion patch.

7.2 Deep Dive into OBE Optimizations

We now analyze the contributions of our optimizations individually in the context of OBE. We compare the speedups of Krypton over *Naive* (batched inference) on both CPU and GPU, termed Empirical-CPU and Empirical-GPU, respectively, against the theoretical speedups (explained in Sections 3 and 4).

Only Incremental Inference. We vary the patch size and set the stride to 4. Figure 21 shows the results. As expected, the speedups go down as the patch size increases. Empirical-GPU Naive yields no speedups, because it does not use our GPU-optimized kernel, while Empirical-GPU does. But Empirical-CPU is closer to theoretical speedup and almost matches it on ResNet18. Thus, there is still some room for improvement to improve the efficiency of incremental inference in both environments.

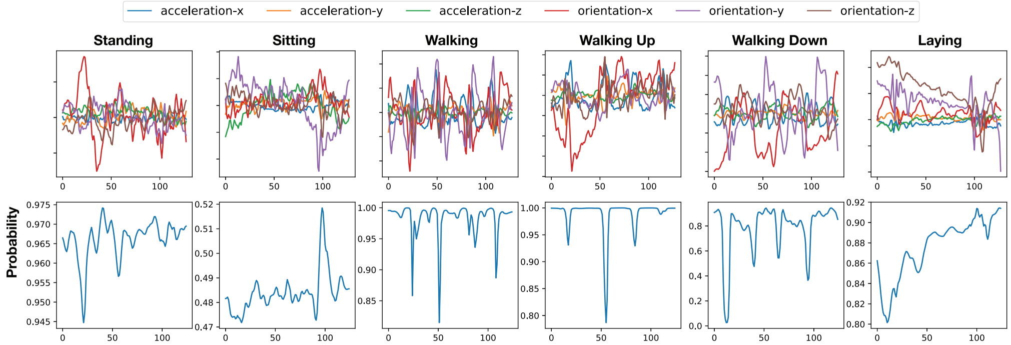


Fig. 20. OBE outputs for sample time series windows corresponding to difference activities (occlusion patch size = 4, occlusion patch value = 0, occlusion patch stride = 1, batch size = 125). Top row shows the input data corresponding to tri-axial acceleration and orientation. Bottom row shows the change in probability for the predicted class for each occlusion patch position.

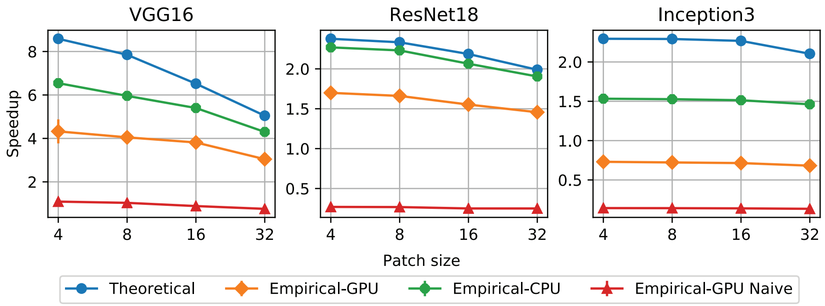


Fig. 21. Speedups with only the incremental inference optimization (occlusion patch stride $S = 4$).

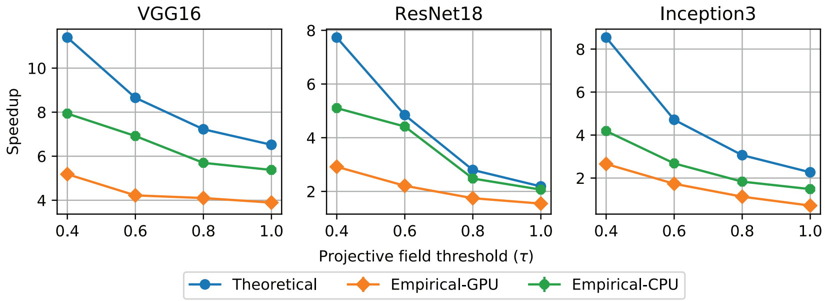


Fig. 22. Speedups with incremental inference combined with only projective field thresholding.

Projective Field Thresholding. We vary τ from 1.0 (no approximation) to 0.4. Adaptive drill-down is disabled but note that this optimization builds on top of our incremental inference. The occlusion patch size is 16 and stride is 4. Figure 22 shows the results. The speedups go up steadily as τ drops for all three CNNs. Once again, Empirical-CPU nears the theoretical speedups on ResNet18, but the gap between Empirical-GPU and Empirical-CPU remains due to the disproportionate impact of memory stalls on GPU. Overall, this approximation offers some speedups in both environments but has a higher impact on CPU than GPU.

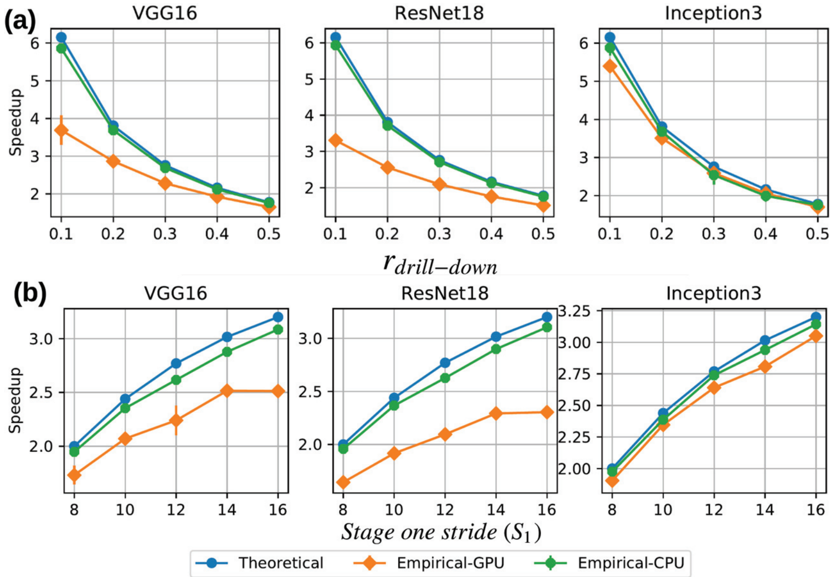


Fig. 23. Speedups with incremental inference combined with adaptive drill-down. For (a), we set $S_1 = 16$. For (b), we set $r_{drill\_down} = 0.25$).

Adaptive Drill-down. Finally, we study the effects of adaptive drill-down (again, on top of incremental inference) and disable projective field thresholding. The occlusion patch size is 16. Stage two stride is $S_2 = 4$. First, we vary $r_{drill-down}$, while fixing stage one stride ($S_1 = 16$). Figure 23(a) shows the results. Next, we vary S_1, while fixing $r_{drill-down} = 0.25$. Figure 23(b) shows the results. As expected, the speedups go up as $r_{drill-down}$ goes down or S_1 goes up, since fewer re-inference queries arise in both cases. Empirical-CPU almost matches the theoretical speedups across the board; in fact, even Empirical-GPU almost matches theoretical speedups on Inception3. Empirical-GPU flattens out at high S_1, since the number of re-inference queries drops, thus resulting in diminishing returns for the benefits of batched execution on GPU. Overall, this optimization has a major impact on speeding up OBE for all CNNs in both environments.

Interactive Mode Execution. We evaluate interactive-mode incremental inference execution (no approximate inference optimizations) with Gs of different sizes. Similar to non-interactive mode experiments presented in Section 5, all experiments are run in batched mode with a batch size of 16 for CPU based experiments and a batch size 128 for GPU-based experiments. If the size of G (formally $|G|$) or the remainder of G is smaller than the batch size, that value is used as the batch size (e.g., $|G| = 16$ results in a batch size of 16). Figure 24 presents the final results.

Memory Overhead. We evaluate the memory overhead of IVM approach, with no projective field thresholding ($\tau = 1.0$) and a projective field thresholding value of $\tau = 0.6$, compared to the full CNN inference. For this, we record the peak GPU memory utilization while the CNN models perform inference on image batches of size 128. We found that incremental inference approach can enable up to 58% lower memory overhead (see Figure 25). Krypton materializes a single copy of all CNN layers corresponding to the unmodified image and reuses it across a batch of occluded images with IVM. For IVM the size of required memory buffers are much smaller than the full inference, as only the updated patches need to be propagated.

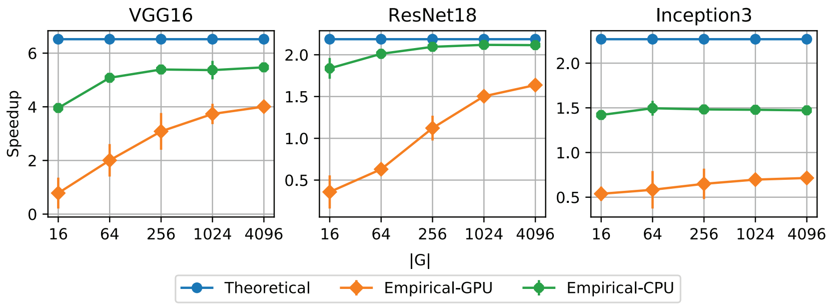


Fig. 24. Interactive mode execution of incremental inference with Gs of different sizes.

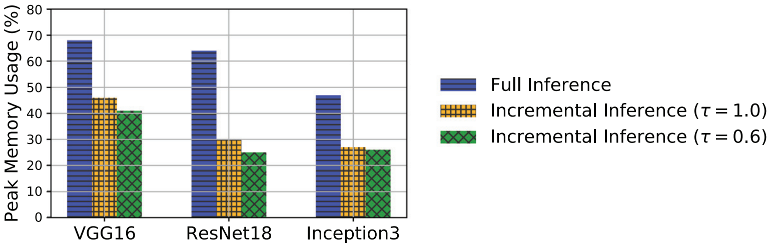


Fig. 25. Peak GPU memory usage when performing CNN inference on a batch of 128 images.

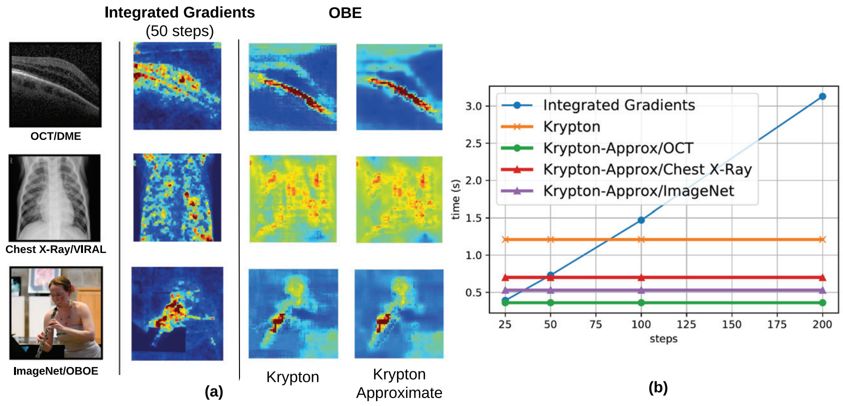


Fig. 26. Comparison of integrated gradients method against OBE. (a) Heatmaps generated by integrated gradients method with a step size of 50. The three color channel gradients of pixels at the same point are aggregated using L2 norm.

Integrated Gradients Method. Explaining deep CNN predictions is still an active area of research. While OBE is popular among many domain users, there are several other methods that can be used for the same task. Integrated Gradients (IG) is one such recently proposed method that claims to overcome many shortcomings of previous methods. We evaluate the runtime and visual quality of the generated heatmaps for IG [38] and OBE methods on three representative images from our datasets (see Figure 26). In general, OBE can better localize relevant regions from the input images. IG method requires tuning a hyper-parameter called *steps*, which determines the number of steps to be used in the gradient integration approximation. Increasing steps improves

	Full Inference	Inc. Inference (speedup)
Theoretical TFLOPs	53.4	9.3 (5.8X)
CPU Time	42.1 min	9.4 min (4.4X)
GPU Time	42 s	41 s (1.0X)

Fig. 27. Runtimes and theoretical FLOP counts for full inference and incremental inference for running ORV (frame differencing threshold = 40, frame sampling rate = 30fps, CPU max batch size = 1, GPU max batch size = 64).

both the runtime and heatmap quality of the IG method. For performing OBE, we used the same hyper-parameters that were used in Section 7.1.

7.3 End-to-end Runtimes for ORV

We now compare the full CNN inference time versus incremental inference time for running ORV on a sample of wildlife trail-camera videos. We set the frame differencing threshold to 40 and use a frame sampling rate of 30 fps. For CPU, we use a maximum batch size of 1; for GPU, the maximum batch size is 64. Figure 27 presents the results.

At the selected frame differencing threshold value and frame sampling rate, our approximate incremental inference approach for ORV achieves 89% accuracy. Performing full inference for ORV requires 53.4 Tera floating point operations (TFLOPs) and performing incremental inference drops it to 9.3 TFLOPs, resulting in a 5.8× theoretical speedup. On CPU Krypton is able to yield a 4.4× speedup and drop inference time from 41.6 mins to 9.4 mins. However, on GPU ORV with incremental inference is unable to provide significant speedups at any threshold, in fact being slower at some thresholds. As most use-cases for video-inferencing are on edge devices, CPU-based systems are likely to be the more common choice for applications of ORV. Hence, we believe Krypton's incremental inference optimizations are more likely to be applicable in those settings.

7.4 Deep Dive into ORV Optimizations

We now dive into how the frame differencing threshold and the frame sampling rate affects the speedup and accuracy for ORV on Krypton.

Frame Differencing Threshold. We vary the frame differencing threshold for ORV with incremental inference and compare how it affects both the runtime and accuracy. The frame rate of the videos are fixed at 30 fps. For CPU, we use a maximum batch size of 1; for GPU, the maximum batch size is 64. Figure 28 presents the results.

On CPU, ORV with incremental inference provides significant speedups at all thresholds, though higher thresholds do perform better than lower ones. This is because a lower threshold causes larger areas to be selected for recalculation, increasing the runtime and inference costs. The effect is mitigated by the choice to select only the largest region for recalculation, but higher thresholds continue to have better runtime performance. The frame differencing threshold also presents a tradeoff between runtime and accuracy, as these lower thresholds with larger areas of recalculation make it more likely that the incremental inference system will produce an accurate result compared to the full inferencing system. For this particular dataset, we find a threshold value of 40 gives an accuracy of 89% with 5× speedup on CPU.

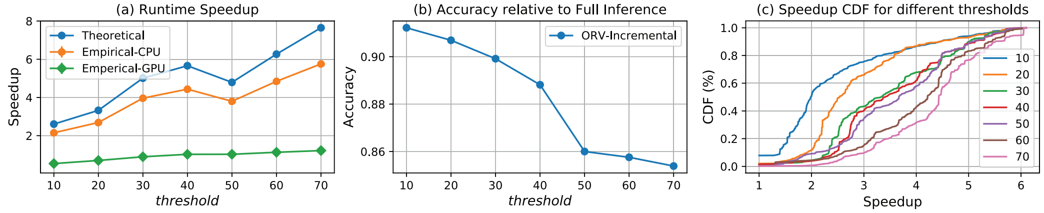


Fig. 28. (a) Speedups obtained by incremental inference for ORV on both CPU and GPU. (b) Accuracy of ORV with incremental inference compared to full inference with varying frame differencing threshold. (c) CDF plot of the speedup improvements at various frame differencing thresholds.

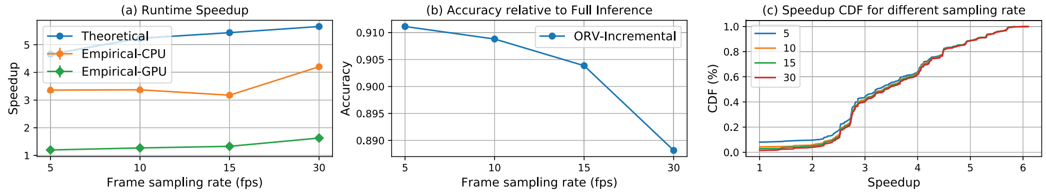


Fig. 29. (a) Runtimes of ORV with incremental and full inference on both CPU and GPU with varying frame sampling rate. (b) Accuracy of ORV with incremental inference compared to full inference with varying sampling rate. (c) CDF plot of the speedup improvements at various sampling rates.

GPU systems do not benefit much from Krypton's application to ORV. The speedups there are minimal at best. Due to the nature of ORV, the dimensions of different input batches are different among multiple batches. Hence, it requires allocating new memory buffers on the fly, which causes significant overheads. Another likely issue is the reduced effective batch sizes due to scene separation triggering before reaching the max batch size, which reduces the hardware utilization.

Interestingly, the speedup does not monotonically increase with the threshold value. We see a sudden drop in speedup from threshold value 40 to value 50 and it starts increasing thereafter. We found that thresholds past 40 actually have fewer full materializations than those prior to 40. This is because a too large threshold value eliminates too large of a region in each frame differencing, so as it minimizes the patch size, the scene separation is unlikely to ever trigger. While this reduces computations performed by full inference, it adversely affects the subsequent incremental inference operations by selecting relatively larger regions due not picking an appropriate base frame. In other words, while the sizes of the changed regions are not too large—they are also not too small! As a result, the gains achieved by reduced full inferences are not sufficient to offset the added computations of larger incremental inference regions. The speedup CDF plots for thresholds 40 and 50 also confirm our observation; CDF plot for 50 is towards the left of the CDF plot for 40 at lower speedup values. This issue is largely data-dependent—the intensity of changes across frames would affect the prevalence of this phenomenon. It also explains the sudden drop in accuracy at threshold 50. Since this problem only begins past threshold 40, all subsequent thresholds show significantly lower accuracy.

Frame Sampling Rate. We now vary the frame sampling rate for ORV with incremental inference and compare how it affects both the runtime and accuracy. For these experiments, we fix the frame differencing threshold to 40. On CPU, we use a maximum batch size of 1; for GPU, the maximum batch size is 64. Figure 29 presents the results.

For both ORV with incremental inference and ORV with full inference, the relationship between the sampling rate and runtime is generally linear and the results are not surprising. The speedups remain quite consistent on CPU systems, ranging between 3–4.5× improvements. On GPU, there appears to be a more consistent improvement, but the rate of change is slight, and the difference is between 1–1.6× improvement. Accuracy also appears to be affected, being reduced slightly as the frame sampling rate increases. We could attribute this to the fact that between two directly adjacent frames, there may not be a large (if one exists at all!) change region that gets past the threshold, as not enough time has passed for any change to occur. Therefore, the incremental inference will be applied to a very small portion of the image—limiting the model's ability to update to the new inputs.

7.5 Summary of Experimental Results

Overall, our experiments show that Krypton can substantially accelerate CNN inference for both OBE and ORV workloads. For OBE it yields up to 16× speedups on GPU and 34.5× speedups on CPU, and for ORV it yields up to 4.4× speedups on CPU. The benefits of our optimizations depend on the CNN's architectural properties. Our approximate inference optimizations also depend on the dataset's properties due to their tunable parameters, which Krypton can tune automatically or easily set by the user based on her judgment. Finally, Krypton sees higher speedups on CPU than GPU, but the runtimes are much lower on GPU. Overall, our optimizations in Krypton help reduce runtimes for OBE and ORV by improving utilization of existing resources rather than forcing users to buy more resources.

7.6 Extensions and Limitations

In this work, we use IVM-based incremental inference as a post hoc optimization to accelerate CNN inference. Going further, "IVM-friendliness" can be baked into the very model selection process that crafts the CNN architecture so the model is both accurate and amenable to fast explanations [39] or fast video analytics. These extensions are complementary to our work and we leave such extensions to future work.

Our approximate CNN inference optimizations rely on the domain expertise of the user to pick configuration parameters such as target SSIM for projective field thresholding and drill-down ratio for adaptive drill-down. Improperly choosing these parameters can lead to misleading explanations.

8 OTHER RELATED WORK

Methods for Explaining CNN Predictions. Perturbation-based and gradient-based are the two main kinds of methods. Perturbation-based methods observe the output of the CNN by modifying regions of the input image [10, 11, 17]. OBE belongs to this category. Gradient-based methods generate a sensitivity heatmap by computing the partial derivatives of model outputs with respect to every input pixel [38, 40, 41]. The recently proposed "Integrated Gradients" (IGD) method belongs in this category [38]. Empirically, we found that OBE produces higher quality heatmaps with better localized regions of interest compared to IGD, while being competitive on runtime. In practice, however, OBE is usually the method of choice for scientific domain users, especially in radiology [12, 42], since it is easy to understand for non-technical users and typically produces high-quality and well-localized heatmaps.

Faster CNN Inference. Several techniques have been proposed to accelerate deep neural network inference that are also applicable to CNNs. One such technique is model quantization technique [43–45], which reduces the precision of model parameters and performs low-precision arithmetic during inference. Another technique to reduce the inference times of CNNs is to prune the number of channels in convolution layers [19, 46]. Furthermore, one can also train a computationally less expensive model using a technique called model distillation [47]. All these techniques trade off the accuracy of a model for faster inference and are orthogonal to our work. Krypton's incremental and approximate inference optimizations are higher-level optimizations that can still operate on quantized, pruned, or distilled CNN models.

Video Inference Optimization. DeepCache [48] is a system for accelerating CNN-based video inference for mobile vision. Similar to Krypton, it exploits the temporal locality in input video streams, but by using a cache of CNN features generated by previous frames. Given a new frame, it tries to find reusable image regions and use their CNN features to reduce the amount of required computations. Krypton's focus is on fixed-angle camera videos, and hence it is sufficient to materialize/cache the features only from the base frame in a scene. EVA^2 [49] is a custom software-hardware integrated stack for exploiting temporal redundancy in video frames. It does so by performing motion estimation computations on video frames and incrementally updating the activation values. Similarly, Diffy [50] is another custom hardware accelerator that exploits the spatial correlation of activation values to reduce inference times. It performs differential convolutions that operate on delta values instead of original values to reduce inference times. These systems are complementary to our work, as our optimizations are at the logical level; they are also applicable to any compute hardware. CBinfer performs change-based approximate CNN inference to accelerate real-time object recognition on video [22]. Similar to Krypton, it also uses thresholding to identify the changed region in video frames. However, it does this thresholding to the outputs of all convolution layers to gain even higher speedups. NoScope accelerates object classification on video streams using model cascades [16]. Panorama also accelerates unbounded vocabulary object recognition on video streams using a deeply supervised cascade built into a new CNN architecture that can be weakly supervised by a domain-specific CNN [51]. Krypton uses frame differencing and incremental inference to accelerate ORV, not model cascades, and it does not logically modify the CNN given for inference. Nexus [52] is a GPU cluster engine for accelerating CNN-based video inference. It uses novel batching techniques, which are orthogonal to the optimizations introduced in Krypton.

Query Optimization. Our work is inspired by the long line of work on relational IVM [53–55], but ours is the first work to use the IVM lens for OBE with CNNs. Our novel algebraic IVM framework is closely tied to the dataflow of CNN layers, which transform tensors in non-trivial ways. Our work is related to the IVM framework for linear algebra in Reference [56]. They focus on bulk matrix operators and incremental addition of rows. We do not deal with bulk matrix operators or addition of rows but more fine-grained CNN inference computations and in-place updates to image pixels due to occlusions. Also related is the IVM framework for distributed multi-dimensional array database queries in Reference [57]. An interesting connection is that CNN layers with local spatial context (Section 2.2) can be viewed as a variant of spatial array join-aggregate queries. But our work enables end-to-end IVM for entire CNNs, not just one-off spatial queries involving data materialization and loading. Our focus is on popular deep learning tools, not array databases. Finally, we also introduce novel CNN-specific and human perception-aware optimizations to accelerate OBE.

Our work is also inspired by relational MQO [24, 58], but our focus is on CNNs, not relational queries. A recent line of work in the database community studies MQO-style techniques for ML systems [59–61], both classical statistical ML systems [62–69] and deep learning systems [70, 71]. Our work adds to this growing direction but to the best of our knowledge, ours is the first work to combine MQO with IVM, at least in ML systems. Our approximate inference optimizations are inspired by approximate query processing (AQP) techniques [72]. But unlike statistical approximations of SQL aggregates, our techniques are CNN-specific. For OBE, we also take into account human perception-aware heuristics; for ORV, we take into account properties of video data.

A shorter version of this article was published at ACM SIGMOD 2019 [73]. In that paper, we introduced Krypton aimed only at OBE for images and outlined incremental and approximate inference optimizations for OBE. We showcased the utility of our optimizations on three popular CNNs and real-world image datasets. Compared to that paper, in this article, we generalize Krypton to support IVM for arbitrary CNNs by introducing the notion of `KryptonGraph` and automating its generation and execution. To demonstrate the generality of our system, we also present a new OBE experiment on time series data using a one-dimensional CNN. We also provide a theoretical proof for our insights underpinning projective field thresholding. Finally, this article also goes beyond OBE to show how the same IVM framework also helps accelerate ORV as an extension of OBE.

9 CONCLUSIONS AND FUTURE WORK

Deep CNNs are gaining widespread adoption for analytics over images, video, and time series data. Occlusion-based explanation (OBE) and object recognition in video (ORV) are two popular CNN-based workloads on such data that are highly compute-intensive due to the large number of CNN re-inference requests produced. In this work, we formalize OBE and ORV from a data management standpoint by re-imagining CNN inference as queries and by devising a suite of novel database-inspired query optimization techniques to accelerate these workloads. Our techniques span exact incremental inference and multi-query optimization for CNN inference, as well as CNN-specific and human-perception-aware approximate inference. Overall, our ideas yield over an order of magnitude speedups for OBE and over 4× speedups for ORV.

An interesting avenue for future work is to extend our IVM and MQO techniques to reduce computational costs and runtimes of more deep learning and other ML models and workloads, as well as on other data types such as relational (tabular), text, and graph data. More broadly, we believe database systems techniques can help reduce resource costs and improve usability of deep learning and ML systems significantly, thus enabling a wider base of application users to benefit from modern ML. We hope our work helps inspire more followup work in the database community on this interesting and important direction.

ACKNOWLEDGMENTS

We thank NVIDIA Corporation for the donation of the Titan Xp GPU used for this work. We also thank the anonymous reviewers of our SIGMOD 2019 paper on this work for their insightful feedback and suggestions on extensions. Some of the extensions shown in this article were motivated by those discussions.

REFERENCES

[1] Olga Russakovsky et al. 2015. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252.

[2] Daniel S. Kermany et al. 2018. Identifying medical diagnoses and treatable diseases by image-based deep learning. *Cell* 172, 5 (2018), 1122–1131.

[3] Mohammad Tariqul Islam et al. 2017. Abnormality detection and localization in chest x-rays using deep convolutional neural networks. *Arxiv Preprint Arxiv:1705.09850* (2017).

[4] Sharada P. Mohanty et al. 2016. Using deep learning for image-based plant disease detection. *Front. Plant Sci.* 7 (2016), 1419.

[5] An Yan, Shuo Cheng, Wang-Cheng Kang, Mengting Wan, and Julian McAuley. 2019. CosRec: 2D convolutional neural networks for sequential recommendation. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2173–2176.

[6] Mohammad Sadegh Norouzzadeh, Anh Nguyen, Margaret Kosmala, Alexandra Swanson, Meredith S. Palmer, Craig Packer, and Jeff Clune. 2018. Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning. *Proc. Nat. Acad. Sci.* 115, 25 (2018), E5716–E5725.

[7] Farhad Arbabzadah et al. 2016. Identifying individual facial expressions by deconstructing a neural network. In *Proceedings of the German Conference on Pattern Recognition*. Springer, 344–354.

[8] Yilun Wang and Michal Kosinski. 2018. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images.*J. Person. Social Psychol.* 114, 2 (2018), 246.

[9] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *Arxiv Preprint Arxiv:1409.1556* (2014).

[10] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Proceedings of the European Conference on Computer Vision*. Springer, 818–833.

[11] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should I trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1135–1144.

[12] Kyu-Hwan Jung et al. 2017. Deep learning for medical image analysis: Applications to computed tomography and magnetic resonance imaging. *Han. Med. Rev.* 37, 2 (2017), 61–70.

[13] Paul Voigt and Axel Von dem Bussche. 2017. *The EU General Data Protection Regulation (GDPR)*. Vol. 18. Springer.

[14] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and benchmarking the impact of GDPR on database systems. *Proc. VLDB Endow.* 13 (Mar. 2020), 1064–1077. DOI: http://dx.doi.org/10.14778/3384345.3384354

[15] G. Sreenu and M. A. Saleem Durai. 2019. Intelligent video surveillance: A review through deep learning techniques for crowd analysis. *J. Big Data* 6, 1 (2019), 48.

[16] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.* 10, 11 (2017), 1586–1597.

[17] Luisa M. Zintgraf et al. 2017. Visualizing deep neural network decisions: Prediction difference analysis. *Arxiv Preprint Arxiv:1702.04595* (2017).

[18] Bert Moons and Marian Verhelst. 2016. A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale convnets. In *Proceedings of the IEEE Symposium on VLSI Circuits (VLSI-Circuits'16)*. IEEE, 1–2.

[19] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 1389–1397.

[20] Allen Ordookhanians, Xin Li, Supun Nakandala, and Arun Kumar. 2019. Demonstration of Krypton: Optimized CNN inference for occlusion-based deep CNN explanations. *Proc. VLDB Endow.* 12, 12 (2019), 1894–1897.

[21] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*. Vol. 1. The MIT Press, Cambridge, MA.

[22] Lukas Cavigelli, Philippe Degen, and Luca Benini. 2017. Cbinfer: Change-based inference for convolutional neural networks on video data. In *Proceedings of the 11th International Conference on Distributed Smart Cameras*. ACM, 1–8.

[23] Zhou Wang et al. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE Trans. Image Proc.* 13, 4 (2004), 600–612.

[24] Timos K. Sellis. 1988. Multiple-query optimization. *ACM Trans. Datab. Syst.* 13, 1 (1988), 23–52.

[25] ONNX Model Format. Retrieved on March 31, 2020 from https://onnx.ai.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Lear. Res.* 15, 1 (2014), 1929–1958.

[27] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. CUDNN: Efficient primitives for deep learning. *Arxiv Preprint Arxiv:1410.0759* (2014).

[28] Basic Operations in a Convolutional Neural Network—CSE@IIT Delhi. Retrieved on March 31, 2020 from http://www.cse.iitd.ernet.in/ rijurekha/lectures/lecture-2.pptx.

[29] Saskia E. J. de Vries et al. 2011. The projective field of a retinal amacrine cell. *J. Neurosci.* 31, 23 (2011), 8595–8604.

[30] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. 2016. Understanding the effective receptive field in deep convolutional neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 4898–4906.

[31] Steffen Eger. 2013. Restricted weighted integer compositions and extended binomial coefficients. *J. Integ. Seq.* 16, 13.1 (2013), 3.

[32] Jia Deng, Wei Dong, et al. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE, 248–255.

[33] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. 2012. Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine. In *Proceedings of the International Workshop on Ambient Assisted Living*. Springer, 216–223.

[34] Winter PhotoVideo Contest Predator Game Camera Pictures & Videos. Retrieved on March 31, 2020 from https://www.trailcampro.com/pages/2017-winter-trail-camera-contest-predator-trailcam-photos.

[35] Kaiming He et al. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[36] Christian Szegedy et al. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2818–2826.

[37] Torch Vison Models. Retrieved on March 31, 2020 from https://github.com/pytorch/vision/tree/master/torchvision/models.

[38] Mukund Sundararajan et al. 2017. Axiomatic attribution for deep networks. *Arxiv Preprint Arxiv:1703.01365* (2017).

[39] Mohammad Motamedi, Felix Portillo, Mahya Saffarpour, Daniel Fong, and Soheil Ghiasi. 2018. Resource-scalable CNN synthesis for IoT applications. *Arxiv Preprint Arxiv:1901.00738* (2018).

[40] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *Arxiv Preprint Arxiv:1312.6034* (2013).

[41] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV'17)*. IEEE, 618–626.

[42] Tim Miller. 2017. Explanation in artificial intelligence: Insights from the social sciences. *Arxiv Preprint Arxiv:1706.07269* (2017).

[43] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning*. 1737–1746.

[44] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *Arxiv Preprint Arxiv:1510.00149* (2015).

[45] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless CNNs with low-precision weights. *Arxiv Preprint Arxiv:1702.03044* (2017).

[46] Jianbo Ye, Xin Lu, Zhe Lin, and James Z. Wang. 2018. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. *Arxiv Preprint Arxiv:1802.00124* (2018).

[47] Seyed-Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, and Hassan Ghasemzadeh. 2019. Improved knowledge distillation via teacher assistant: Bridging the gap between student and teacher. *Arxiv Preprint Arxiv:1902.03393* (2019).

[48] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 129–144.

[49] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting temporal redundancy in live computer vision. *Arxiv Preprint Arxiv:1803.06312* (2018).

[50] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. 2018. Diffy: A Déjà vu-free differential deep neural network accelerator. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 134–147.

[51] Yuhao Zhang and Arun Kumar. 2019. Panorama: A data system for unbounded vocabulary querying over video. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 477–491. DOI: http://dx.doi.org/10.14778/3372716.3372721.

[52] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[53] Rada Chirkova, Jun Yang, et al. 2012. Materialized views. *Found. Trends® Datab.* 4, 4 (2012), 295–405.

[54] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[55] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. 1995. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 95–104.

[56] Milos Nikolic, Mohammed ElSeidy, and Christoph Koch. 2014. LINVIEW: Incremental view maintenance for complex analytical queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 253–264.

[57] Weijie Zhao, Florin Rusu, Bin Dong, Kesheng Wu, and Peter Nugent. 2017. Incremental view maintenance over array data. In *Proceedings of the ACM International Conference on Management of Data*. ACM, 139–154.

[58] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable multi-query optimization for SPARQL. In *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE'12)*. IEEE, 666–677.

[59] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems*. Morgan & Claypool Publishers.

[60] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. Association for Computing Machinery, New York, NY, 1717–1722. DOI: http://dx.doi.org/10.1145/3035918.3054775

[61] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. 2016. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec.* 44, 4 (May 2016), 17–22. DOI: http://dx.doi.org/10.1145/2935694.2935698

[62] Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization optimizations for feature selection workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. Association for Computing Machinery, New York, NY, 265–276. DOI: http://dx.doi.org/10.1145/2588555.2593678

[63] Pradap Konda, Arun Kumar, Christopher Ré, and Vaishnavi Sashikanth. 2013. Feature selection in enterprise analytics: A demonstration using an r-based data analytics system. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1306–1309. DOI: http://dx.doi.org/10.14778/2536274.2536302

[64] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning generalized linear models over normalized data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Association for Computing Machinery, New York, NY, 1969–1984. DOI: http://dx.doi.org/10.1145/2723372.2723713

[65] Arun Kumar, Mona Jalal, Boqun Yan, Jeffrey Naughton, and Jignesh M. Patel. 2015. Demonstration of Santoku: Optimizing machine learning over normalized data. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1864–1867. DOI: http://dx.doi.org/10.14778/2824032.2824087

[66] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2017. Towards linear algebra over normalized data. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1214–1225. DOI: http://dx.doi.org/10.14778/3137628.3137633

[67] Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *Proceedings of the International Conference on Management of Data (SIGMOD'19)*. Association for Computing Machinery, New York, NY, 1571–1588. DOI: http://dx.doi.org/10.1145/3299869.3319878

[68] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A layered aggregate engine for analytics workloads. In *Proceedings of the International Conference on Management of Data*. ACM, 1642–1659. DOI: http://dx.doi.org/10.1145/3299869.3324961

[69] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An intermediate representation for optimizing machine learning pipelines. *Proc. VLDB Endow.* 12, 11 (July 2019), 1553–1567. DOI: http://dx.doi.org/10.14778/3342263.3342633

[70] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2019. Cerebro: Efficient and reproducible model selection on deep learning systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning (DEEM'19)*. Association for Computing Machinery, New York, NY. DOI: http://dx.doi.org/10.1145/3329486.3329496

[71] Supun Nakandala and Arun Kumar. 2020. Vista: Optimized system for declarative feature transfer from deep CNNs at scale. In *Proceedings of the International Conference on Management of Data (SIGMOD'20)*. Association for Computing Machinery.

[72] Minos N. Garofalakis and Phillip B. Gibbon. 2001. Approximate query processing: Taming the terabytes. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 725.

[73] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2019. Incremental and approximate inference for faster occlusion-based deep CNN explanations. In *Proceedings of the International Conference on Management of Data*. 1589–1606.

Received January 2020; accepted April 2020