Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations

ABSTRACT

Deep Convolutional Neural Networks (CNNs) now match human accuracy in many image recognition tasks. This has lead to increasing adoption of deep CNNs in e-commerce, radiology, and other domains. Naturally, "explaining" CNN predictions is a key concern for many users. Since the internal working of CNNs are unintuitive for non-technical users, occlusion-based explanations are popular for determining which parts of an input image contribute the most to a given prediction. One occludes a region of the image using a patch and moves this patch across the image to yield a heat map of changes to the prediction probability. Alas, this approach is computationally expensive due to the large number of re-inference requests produced, which could waste human time and raise resource costs. In this paper, we resolve this issue by casting occlusion-based CNN explanations as a new instance of the incremental view maintenance problem. We create a novel and comprehensive algebraic framework for incremental CNN inference that combines materialized views with multi-query optimization to avoid computational redundancy across re-inference requests. We then introduce two approximate inference optimizations that exploit the semantics of CNNs and the occlusion task to further reduce runtimes. We prototype our ideas in Python to create a tool we call Krypton that can support both CPUs and GPUs. Experiments with real data and CNNs show that Krypton reduces runtimes by up to 5x (14x) to produce exact (approximate) heat maps without raising resource requirements.

ACM Reference Format:

. 2018. Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

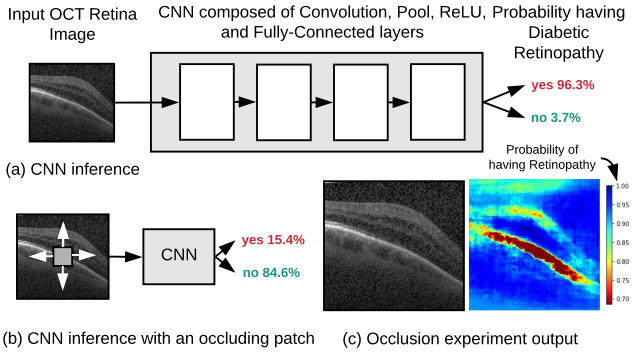

Figure 1: (a) Using CNNs for predicting Diabetic Retinopathy from OCT images. (b) Occluding parts of the OCT image changes the predicted probability for the disease. (c) By changing the position of the occlusion patch a sensitivity heat map is produced.

1 INTRODUCTION

Deep Convolution Neural Networks (CNNs) [1–4] have revolutionized the computer vision field with even surpassing human level accuracy in some of the image recognition challenges such as ImageNet [5]. As a result, there is wide adoption of Deep CNN technology in a variety of real-world image recognition tasks in several domains including healthcare [6, 7], agriculture [8], security [9], and sociology [10]. Remarkably, United States Food and Drug Administration Agency (US FDA) has already approved the use of Deep CNN based technologies for identifying diabetic retinopathy, an eye disease found in adults with diabetes [11]. It is expected that this kind of decision support systems will help the human radiologists in fulfilling their workloads efficiently, such as functioning as a cross-checker for the manual decisions and also to prioritize potential sever cases for manual inspection, and provide a remedy to the shortage of qualified radiologists globally [12].

Despite their many success stories, one of the major criticisms for deep CNNs and deep neural networks, in general, is the black-box nature of how they make predictions. In order to apply deep CNN based techniques in critical applications such as healthcare, the decisions should be explainable so that the practitioners can use their human judgment to decide whether to rely on those predictions or not [13]. One of the widely used approach for explaining CNN predictions

is the *occlusion based explanations* (OBE) approach [14]. In OBE experiments a square patch, usually of black or gray color, is used to occlude parts of the image and record the change in the predicted label probability as shown in Figure 1 (b). By changing the position of the occlusion patch a sensitivity heat map for the predicted label, similar to the one shown in Figure 1 (c), can be generated. Using this heat map, the regions in the image which are highly sensitive (or highly contributing) to the predicted class can be identified (corresponds to red color regions in the sensitivity heat map shown in Figure 1 (c)). This localization of highly sensitive regions then enables the practitioners to get an intuition of the the prediction process of the deep CNN.

Alas, this approach creates a new bottleneck: OBE experiments are highly compute intensive and time consuming due to the large number of re-inference requests that needs to be performed. The current approach to perform OBE experiments is to generate a large number of modified versions of the original image with each image corresponding to a specific occlusion patch position and perform CNN inference for those images using a tool like PyTorch [15]. In extreme cases this approach can generate more than 500,000 different images and can take up to more than one hour to complete, even by using a GPU [16]. Such long execution times hinders the data scientist's ability to analyze CNN predictions and also reduces their productivity.

In this work we apply database inspired optimizations to the OBE workload to reduce both the computational cost and the runtime. Our work is motivated by the simple yet crucial observation: different occluded instances of the original image are *not* independent – when performing CNN inference corresponding to each individual occluded image significant portion of redundant computation can be avoided. This observation leads us to a classical database systems-style concern: *view maintenance*. In database parlance, the current approach of performing CNN inference for each occluded instance of the image independently can be considered as a form of *complete view maintenance*, where a "view" is a layer of CNN features. This approach wastes runtimes due redundancy in CNN inference computations across occluded instances of the original image.

We formalize the dataflow of the layers of a CNN and create a novel and comprehensive algebraic framework for *incremental inference* of CNN which combines *incremental view maintenance* (IVM) [17–19] with *multi-query optimization* (MQO) to avoid computational redundancy across re-inference requests. Using our algebraic framework we theoretically characterize how much speedups one can expect from incremental inference for different CNN models. To the best of our knowledge, this is the first known instance of fusing an IVM-style techniques with an MQO-style technique for optimizing CNN inference.

We then introduce *approximate inference* optimizations that exploit a capability of human perception: tolerance of some degradation in visual quality of the heat maps. We build upon our incremental inference optimizations to create two novel approximate inference optimizations that trade off the quality of the generated heat map in a user tunable manner to accelerate OBE. Our approximate inference optimizations are inspired by the *approximate query processing* (AQP) techniques used for answering statistical analytical queries in RDBMs. However, our focus here is on minimizing the perceivable differences of the generated heat maps where as in traditional AQP the focus is on minimizing the statistical error. We also combine AQP and MQO style techniques to reduce OBE runtime.

We prototype our ideas as a system we call Krypton on top of PyTorch deep learning toolkit by adding custom implementations for incremental and approximate inference operations. It currently supports VGG16, ResNet18, and InceptionV3 both on CPU and GPU environments, which are three widely used Deep CNN architectures. We evaluate our system on three real-world datasets, 1) retinal optical coherence tomography dataset (OCT), 2) chest X-Ray, and 3) more generic ImageNet dataset. While we have implemented Krypton on top of PyTorch toolkit, our work is largely orthogonal to the choice of the deep learning toolkit; one could replace PyTorch with TensorFlow, Caffe2, CNTK, MXNet, or implement from scratch using C/CUDA and still benefit from our optimizations. Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to study the incremental inference and approximate inference optimizations for the OBE workload from a systems standpoint.
- We develop a comprehensive algebraic framework for incremental inference of CNN which combines IVM with MQO to avoid computational redundancy across re-inference requests.
- We also introduce two approximate inference optimizations that exploit the characteristics of human perception to further reduce runtimes of the OBE workload.
- Using a prototype system call Krypton, we present an extensive empirical evaluation of the benefits of our optimizations. Overall, Krypton can result in speedups up to 5x (14x) to produce exact (approximate) heat maps.

Outline. The rest of this paper is organized as follows. Section 2 formally define the problem, explains our assumptions, and formalizes the dataflow of layers of a CNN. Section 3 provides a theoretical characterization of potential speedups for different CNN models and presents our novel

Table 1: Notation used in this paper.

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	Heat map 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
$C_{O:l}, H_{O:l}, W_{O:l}$	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

algebraic framework for performing incremental inference for OBE. Section 4 presents our approximate inference optimizations. Section 5 presents the experimental evaluation. We discuss other related work in Section 6 and conclude in Section 7.

2 SETUP AND PRELIMINARIES

We now state our 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.

2.1 Problem Statement and Assumptions

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

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

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

$$M \in \mathbb{R}^{H_M \times W_M} \quad (3)$$

$$\forall\ (x, y) \in G: \quad (4)$$

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

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

Steps (1) and (2) calculate the dimensions of the heat map 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 heat map. In the *interactive* mode, the user may manually place the occlusion patch only at a few locations at a time, yielding partial heat maps.

We assume the CNN predicts a discrete set of labels, since only such applications typically use OBE. One could create CNNs that directly predict an image segmentation instead, but labeling image segments for training such CNNs is highly tedious and expensive. Thus, most recent applications of CNNs in healthcare, sociology, security, and other domains rely on discrete labels and use OBE [6–10]. There are other approaches to explain CNN predictions in the literature; since they are orthogonal to our focus, we summarize them in the appendix due to space constraints. We assume that f is from a roster of well-known deep CNNs; we currently support VGG-16, ResNet-18, and Inception V3. We think this is a reasonable start, since most recent applications in our target domains use only such well-known CNNs from model zoos [20, 21]. Nevertheless, our techniques are generic enough to apply to any CNN; we leave support for arbitrary CNNs to future work.

2.2 Dataflow of Deep CNN Layers

CNNs are a form of artificial neural networks specialized for image data. They are organized as *layers* of various

types, each of which transforms a tensor (a multidimensional array, typically 3-D) into another tensor: *Convolution* uses image filters from graphics to extract features, except with variable filter weights (learned during training); *Pooling* subsamples features in a spatially-aware manner; *Batch-Normalization* normalizes the output of that layer; *Non-Linearity* applies an element-wise non-linear transformation (e.g., ReLU); *Fully-Connected* is an ordered collection of perceptrons [**?**]. The output tensor of a layer can have a different width, height, and/or depth than the input tensor. The input 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 now group CNN layers into three main categories based on the *spatial locality* of how they transform a tensor:

- Transformations at the granularity of a *global context*, e.g., Fully-Connected
- Transformations at the granularity of *individual elements*, e.g., ReLU, Batch Normalization
- Transformations at the granularity of a *local spatial context*, e.g. Convolution, Pooling

Transformations at the granularity of a global context. 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 in deep CNNs (and never in some recent deep CNNs), and they typically account for a negligible fraction of the total computational cost. Thus, we do not focus on such layers for our optimizations.

Transformations at the granularity of individual elements. Such layers essentially apply a "map()" function on the elements of the input tensor, as illustrated in Figure 2(b). Thus, the output has the same dimensions as the input. NonLinearity falls under this category, e.g., with ReLU as the function. 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.

Transformations at the granularity of a local spatial context. Such layers essentially perform weighted aggregations of slices of the input tensor, called *local spatial contexts*, by multiplying them many a *filter kernel* (a tensor of weight parameters). Thus, the 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. Since such layers typically account for the bulk of the computational cost of deep CNN inference, enabling incremental inference for such layers in the OBE context is a key focus of this paper (Section 3). The rest of this section explains the machinery of the dataflow in such layers our notation. Section 3 then uses this machinery to explain our optimizations.

Dataflow of Convolution Layers. A layer l has $C_{O:l}$ 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 $W_{I:l}$ and height $H_{I:l}$ of the input tensor $\mathcal{I}_{:l}$ but the same depth $C_{I:l}$. During inference, c^{th} filter kernel is "strided" along the width and height dimensions of the input to produce a 2-D "activation map" $A_{:c} = (a_{y,x:c}) \in \mathbb{R}^{H_{O:l} \times W_{O:l}}$ 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 several 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 $O_{:l} \in \mathbb{R}^{C_{O:l} \times H_{O:l} \times W_{O:l}}$ as per Equation (8). Figure 2(a) presents a simplified illustration of a Convolution layer.

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

$$
O_{:l} = [A_{:0}, A_{:1}, ..., A_{(C_{O:l}-1)}] \tag{8}
$$

Dataflow of Pooling Layers. Such layers behave essentially like Convolution layers but with a fixed (i.e., not learned) 2-D filter kernel $\mathcal{K}_{pool:l}$. Such kernels aggregate a local spatial context to compute its maximum element ("max pooling") or average ("average pooling"). But unlike Convolution, Pooling operates independently the depth slices of the input tensor. It takes as input a 3-D tensor O_l of depth $C_{I:l}$, width $W_{I:l}$, and height $H_{I:l}$. It produces as output a 3-D tensor $O_{:l}$ with the same depth $C_{O:l} = C_{I:l}$ but a different width of $W_{O:l}$ and height $H_{O:l}$. The filter kernel is typically strided over more than one pixel at a time.

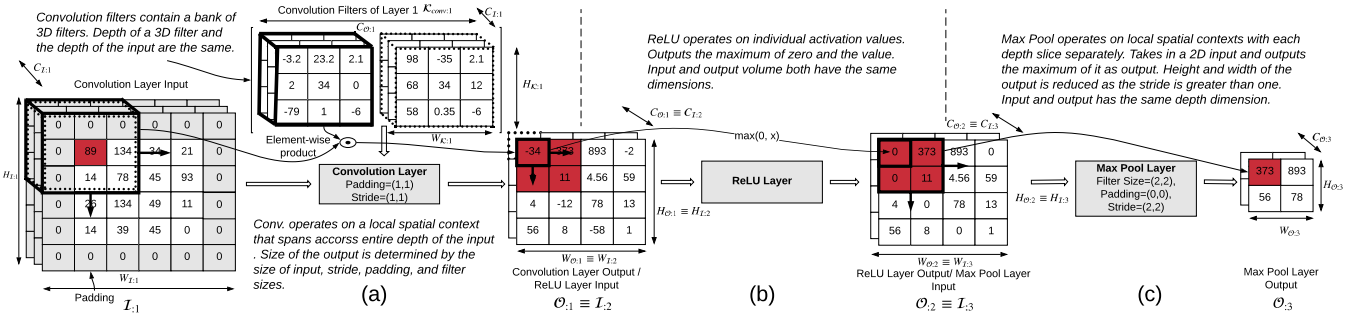

Figure 2: Simplified illustration of the key layers of a typical CNN. The highlighted cells (dark/red background) show how a small local spatial context in the first input propagates through subsequent layers. (a) Convolution layer (for simplicity sake, bias addition is not shown). (b) ReLU Non-linearity layer. (c) Pooling layer (max pooling). Notation is explained in Table 1.

Thus, $W_{O:l}$ and $H_{O:l}$ are usually smaller than $W_{I:l}$ and $H_{I:l}$. Figure 2(c) presents a simplified illustration of a Pooling layer. Overall, both Convolution and Pooling layers have a similar dataflow: they apply a filter kernel along the width and height dimensions of the input tensor but differ on how they handle the depth dimension. But since OBE only concerns the width and height dimensions of the image and subsequent tensors, we can treat both Convolution and Pooling layers in a unified manner for our optimizations.

Relationship between Input and Output Dimensions. For Convolution and Pooling layers, $W_{O:l}$ and $H_{O:l}$ are determined solely by $W_{I:l}$ and $H_{I:l}$, $W_{\mathcal{K}:l}$ 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. But 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 2, 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, the padding values can differ along the width and height dimensions: $P_{x:l}$ and $P_{y:l}$. In Figure 2(a), the Convolution layer has $P_{x:l} = P_{y:l} = 1$. Given all these parameters, the width (similarly height) of the output tensor is given by the following formula:

$$W_{O:l} = (W_{I:l} - W_{\mathcal{K}:l} + 1 + 2 \times P_{x:l})/S_{x:l} \tag{9}$$

Computational Cost of Inference. Deep CNN inference is computationally expensive. Convolution layers typically account for a bulk of the cost (90% or more) [**?**]. 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 $(C_{I:l}, H_{\mathcal{K}:l}, W_{\mathcal{K}:l})$ to compute one element of the output tensor requires $C_{I:l}H_{\mathcal{K}:l}W_{\mathcal{K}:l}$ FLOPs, with each FLOP corresponding to one FMA. Thus, the total computational cost $Q_{:l}$ of a layer that produces output $O_{:l}$ of dimensions $(C_{O:l}, H_{O:l}, W_{O:l})$ and the total computational cost Q of the entire set of Convolution layers of a given CNN f can be calculated as per Equations (10) and (11).

$$Q_{:l} = (C_{I:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{O:l} \cdot H_{O:l} \cdot W_{O:l}) \tag{10}$$

$$Q = \sum_{l \ in \ f} Q_{:l} \tag{11}$$

3 INCREMENTAL INFERENCE OPTIMIZATIONS

We start with a theoretical characterization of how much speedups one can expect from incremental inference for OBE. We then dive into our novel algebraic framework that enables incremental inference for CNN layers and integrates it with multi-query optimization for OBE.

3.1 Expected Speedups

The basic reason why incremental view maintenance (IVM) for relational queries offers speedups is that when a part of the input relation is updated, IVM only computes the part of output that gets changed. We bring the IVM notion to CNNs with CNN layers being our "queries" and materialized feature tensors being our "relations." OBE updates only a part of the input; so, only some parts of the output 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 can determine using "static analysis" over f how many FLOPs

can be saved. This let us derive an upper bound on the possible speedup–we call this the "theoretical speedup."

More precisely, let the output tensor dimensions of layer l be $(C_{O:l}, H_{O:l}, W_{O:l})$. An incremental update recomputes a smaller local spatial context with width $W_{\mathcal{P}:l} \leq W_{O:l}$ and height $H_{\mathcal{P}:l} \leq H_{O:l}$. Thus, the computational cost of incremental inference for layer l, $Q_{inc:l}$, and the total computational cost for incremental inference for f, Q_{inc}, are given by Equations (12) and (13).

$$Q_{inc:l} = (C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{O:l} \cdot H_{\mathcal{P}:l} \cdot W_{\mathcal{P}:l}) \quad (12)$$

$$Q_{inc} = \sum_{l \text{ in } f} Q_{inc:l} \quad (13)$$

The above costs can be much smaller than $Q_{:l}$ and Q in Equations (10) and (11) earlier. The *theoretical speedup* is defined 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 performed a sanity check experiment to ascertain the theoretical speedups for a few popular deep CNNs. For varying occlusion patch sizes (with the same stride of 1), we plot the theoretical speedups of different deep CNNs in Figure 3 shows the results. VGG-16 has the highest theoretical speedups, while DenseNet-121 has the lowest. Most CNNs fall in the 2X–3X 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). In turn, incremental inference is most beneficial for VGG-16. Note that we assumed an image size of 224 × 224 for this plot; if the image is larger, the theoretical speedups will be higher.

While one might be tempted to think that speedups of 2X-3X may not be "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 6min to just 2min, or on a GPU from 1min to just 20s. 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 2X for incremental inference to up to 8X with all of our optimizations enabled. Thus, incremental inference is critical to optimizing OBE.

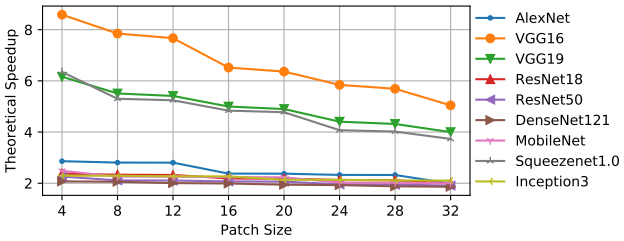

Figure 3: Theoretical speedups for popular deep CNN architectures with incremental inference.

3.2 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 4 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 $(x^{O}_{\mathcal{P}:l}, y^{O}_{\mathcal{P}:l})$ and has a height $H^{O}_{\mathcal{P}:l}$ and width $W^{O}_{\mathcal{P}:l}$. 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 4. 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:

$$x^{O}_{\mathcal{P}:l} = max\Big(\lceil (P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\Big) \quad (14)$$

$$W^{O}_{\mathcal{P}:l} = min\Big(\lceil (W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W_{O:l}\Big) \quad (15)$$

$$x^{\mathcal{R}}_{\mathcal{P}:l} = x^{O}_{\mathcal{P}:l} \times S_{x:l} - P_{x:l} \quad (16)$$

$$W^{\mathcal{R}}_{\mathcal{P}:l} = W_{\mathcal{K}:l} + (W^{O}_{\mathcal{P}:l} - 1) \times S_{x:l} \quad (17)$$

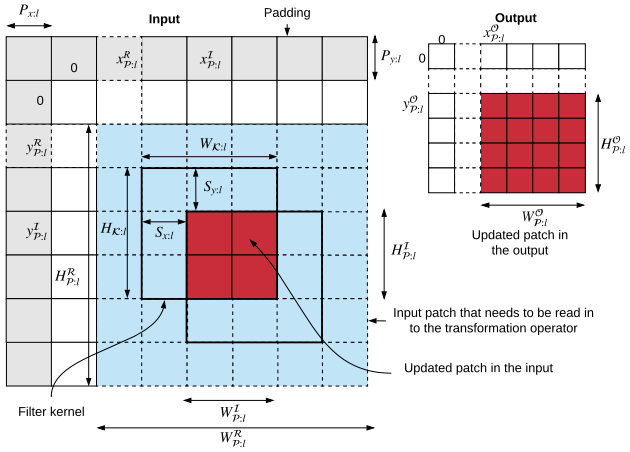

Figure 4: Simplified illustration of input and output update patches for Convolution/Pooling layers.

Symbol	Meaning
$x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l}$	Starting coordinates of the input patch for the l^{th} layer
$x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l}$	Starting coordinates of the patch that needs to be read in for the l^{th} layer transformation
$x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l}$	Starting coordinates of the output patch for the l^{th} layer
$H^{\mathcal{I}}_{\mathcal{P}:l}, W^{\mathcal{I}}_{\mathcal{P}:l}$	Height and width of the input patch for the l^{th} layer
$H^{\mathcal{R}}_{\mathcal{P}:l}, W^{\mathcal{R}}_{\mathcal{P}:l}$	Height and width of the patch that needs to be read in for the l^{th} layer transformation
$H^{\mathcal{O}}_{\mathcal{P}:l}, W^{\mathcal{O}}_{\mathcal{P}:l}$	Height and width of the output patch for the l^{th} layer
τ	Projective field threshold
$r_{drill-down}$	Stage two drill-down fraction used in *adaptive drill-down*

Table 2: Additional notation for Sections 3 and 4.

Equation (14) calculates the coordinates of the output update patch. 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 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 (15) calculates the width of the output update patch. Given these, a start coordinate and width of the read-in context are given by Equations (16) and (17); 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:

$$\mathcal{U} = \mathcal{I}_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}:l} : x^{\mathcal{R}}_{\mathcal{P}:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l} : y^{\mathcal{R}}_{\mathcal{P}:l} + H^{\mathcal{R}}_{\mathcal{P}:l}] \quad (18)$$

$$\mathcal{U} = \mathcal{U} \circ_{(x^{\mathcal{I}}_{\mathcal{P}:l} - x^{\mathcal{R}}_{\mathcal{P}:l}), (y^{\mathcal{I}}_{\mathcal{P}:l} - y^{\mathcal{R}}_{\mathcal{P}:l})} \mathcal{P}^{\mathcal{I}}_{:l} \quad (19)$$

$$\mathcal{P}^{\mathcal{O}}_{:l} = T_{:l}(\mathcal{U}) \quad (20)$$

Equation (18) slices the read-in context $\mathcal{U}$ from the pre-materialized input tensor $\mathcal{I}_{:l}$. Equation (19) 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 (20) 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.

3.3 Propagating Updates across Layers

Sequential CNNs. Unlike relational IVM, CNNs consist of many layers, often in a sequence. This is analogous to having not just one query, but a sequence of queries that require IVM on their predecessor's updated 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., going from a Convolution layer (or Pooling layer) to a Fully-Connected layer. In particular, we need to materialize the *full updated output* instead of propagating just the output update patches, since the global context transformation might lose spatial locality for subsequent layers.

Extension to DAG CNNs. Some recent deep CNNs have their layers organized as a more general directed acyclic graph (DAG) instead of a sequence. Such CNNs contain two additional layers apart from the layers described earlier: *element-wise addition* and *depth-wise concatenation*, which are needed to "merge" two branches in the DAG. 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 5 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

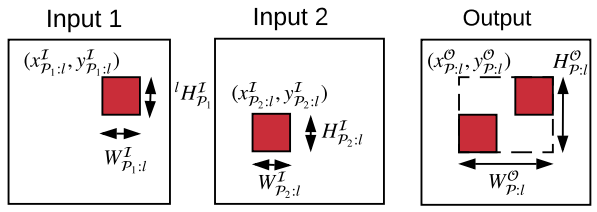

Figure 5: Illustration of bounding box calculation for differing input update patch locations for element-wise addition and depth-wise concatenation layers in DAG CNNs.

starting at coordinates $(x^I_{\mathcal{P}_2:l}, y^I_{\mathcal{P}_2:l})$ with dimensions $H^I_{\mathcal{P}_2:l}$ and $W^I_{\mathcal{P}_2:l}$. This issue can arise with both element-wise addition and depth-wise concatenation.

We propose a simple unified solution: compute the *bounding box* of both the input update patches. This means the coordinates and dimensions of both the output update patch and both the read-in contexts will be identical. Figure 5 illustrates this too. While our solution will potentially recompute portions of the output that did not get modified, we think this trade-off 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):

$$x^O_{\mathcal{P}:l} = \texttt{min}(x^I_{\mathcal{P}_1:l}, x^I_{\mathcal{P}_2:l})$$
$$W^O_{\mathcal{P}:l} = \texttt{max}(x^I_{\mathcal{P}_1:l} + W^I_{\mathcal{P}_1:l}, x^I_{\mathcal{P}_2:l} + W^I_{\mathcal{P}_2:l}) - \texttt{min}(x^I_{\mathcal{P}_1:l}, x^I_{\mathcal{P}_2:l}) \quad (21)$$

3.4 Multi-Query Incremental Inference

As explained in Section 2.1, the OBE workload issues multiple ($|G|$ to be precise) re-inference requests *in one go* for different occlusion patch locations. Viewig each re-inference request as a "query" itself, we now make a connection with multi-query optimization (MQO) [**?**]. We observe that the $|G|$ queries are *not disjoint*–they share large parts of the input, since the occlusion patch is typically much smaller than the image. Thus, we now extend our incremental inference optimization and integrate it with an MQO-style optimization for OBE. To the best of our knowledge, this is the first known instance of fusing an IVM-style techniques with an MQO-style technique for optimizing CNN inference. To draw an analogy to relational optimization, this situation is like a batch of incremental update queries arriving at once on the same relation, with each query modifying a different but small part of the input relation.

Batched Incremental Inference. Our optimization works as follows: materialize all the tensors produced by the CNN's layers *once*. Then, *reuse* these tensors for incremental inference for each of the $|G|$ queries. This is feasible due to our above observation–most of the occluded image's pixels are still identical across the queries, which means large parts of the tensors produced by a given CNN layer will likely be identical too. Essentially, we amortize the cost of materializing the tensors across all $|G|$ queries. One might wonder, why not just perform "batched" inference for the $|G|$ queries? Batched inference is standard practice today for high-throughput compute hardware such as GPUs, since it amortizes the CNN set up costs, data movement costs, etc. Batch sizes are picked to optimize hardware utilization. Our answer is that batched inference is an *orthogonal* (albeit trivial) optimization compared to our above optimization. In other words, we can combine these two ideas and execute incremental inference in a batched manner. We call this approach "batched incremental inference." As we show later (Section 5), batching alone yields only modest 2X speedups, while combining batching and incremental inference amplifies the speedups. Algorithm 1 formally presents the batched incremental inference operation for layer l.

Algorithm 1 BatchedIncrementalInference

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

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

BatchedIncrementalInference first calculates the geometric properties of the output update patches and read-in contexts. A temporary tensor U is initialized to hold the input update patches with their read-in contexts. The **for** loop iteratively populates U with corresponding patches. Finally, $T_{:l}$ is applied to U to compute the output patches. We note that only for the first layer, 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.

GPU Optimized Implementation. Empirically, we observed an interesting dichotomy between CPUs and GPUs: BatchedIncrementalInference yields expected speedups on CPUs, but it performs dramatically poorly on GPUs. Indeed, a naive implementation of BatchedIncrementalInference on GPUs was often *slower* than full inference! We now explain 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 input tensor) from one part of the GPU memory to another *sequentially*. A detailed profiling of the GPU showed that such sequential memory copies are a bottleneck for the naive GPU implementation, since the GPU is throttled from exploiting its massive parallelism effectively. To overcome this GPU-specific issue, we created a custom CUDA kernel to perform the input preparation more efficiently by copying the memory regions in parallel for all items in the batched inference request. Essentially, this is a parallel **for** loop customized for slicing the input tensor. We then invoke the CNN transformation function $T_{:l}$, which is already highly hardware-optimized by modern deep learning tools [22]. We defer more details on our custom CUDA kernel to the appendix due to space constraints. Also, since GPU memory might not be enough to fit all $|G|$ queries, the batch size for the GPU-based execution might be smaller than $|G|$. Overall, our incremental inference optimizations yield benefits directly in the CPU environment but requires our custom kernel for the GPU environment.

3.5 Putting it All Together

We now summarize the end-to-end workflow of our incremental inference optimizations for the OBE workload. We are given the CNN f, image $I_{:img}$, predicted class label L, occlusion patch $\mathcal{P}$ and its stride $S_{\mathcal{P}}$, and the set of occlusion patch positions G. Pre-materialize the output tensors of all layers of f with $I_{:img}$ as the input. Prepare the occluded images ($I'_{(x,y):img}$) for all positions in G. For batches of $I'_{(x,y):img}$ as the input, invoke the transformations in f in topological order and calculate the corresponding entries of the heat map M. For local context transformations, invoke BatchedIncrementalInference. For a local context transformation that precedes a global context transformation, materialize the full updated output. For all other layers, invoke the original transformation function. M is now the output heat map.

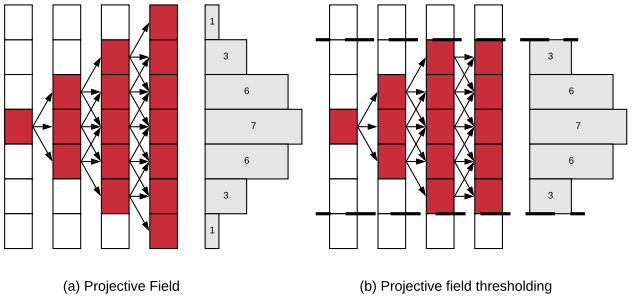

Figure 6: (a) 1-D Convolution showing growth of projective field (filter size = 2, stride = 1). (b) Projective field *thresholding* with $\tau = 5/7$.

4 APPROXIMATE INFERENCE OPTIMIZATIONS

Incremental inference is *exact*, i.e., it yields the same heat map as full inference. But we observe that it does not exploit a capability of human perception: tolerance of some degradation in visual quality. Thus, we build upon our incremental inference optimizations to create two novel heuristic *approximate* inference optimizations that trade off the heat map's quality in a user-tunable manner to accelerate OBE further. We first explain the techniques and then explain how we can help users tune them.

4.1 Projective Field Thresholding

The *projective field* of a neuron in a CNN is the local slice (including depth) of the output 3-D tensor that is connected to it [23, 24]. It is a term from neuroscience to describe the spatiotemporal effects of a retinal cell on the output of the eye's neuronal circuitry [25]. This notion helps us shed light on the *growth of the size* of the update patches truncate through the layers of a CNN. The three kinds of layers (Section 2.2) affect the projective field size 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* in a manner determined by the filter kernel's dimensions and stride: additively in the filter size and multiplicatively in the stride. The growth of the projective field size implies the amount of FLOPs saved by incremental inference decreases as we go to the higher layers of a CNN. Eventually, the output update patch of a layer becomes as large as the whole output. This is illustrated by Figure 6(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 6(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 3 elements in the next layer and then 5, but it then gets truncated because we set $\tau = 5/7$. This approximation can alter the accuracy of the output values and this, the heat map's visual quality. Empirically, we find that modest truncation is tolerable and does not affect the heat map's visual quality too significantly.

To provide intuition on why the above happens, consider histograms on the side of Figures 6(a,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 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 [26] for understanding the "receptive field" in CNNs), we show in the appendix that this distribution does indeed converge to a Gaussian. Thus, while this optimization is a heuristic, we think is grounded in a common behavior seen in real-world CNNs. Overall, since most of the contributions to the output elements are concentrated around the center, such 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):

$$W^O_{\mathcal{P}:l} = \texttt{min}\Big(\lceil (W^I_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W^O_{\mathcal{P}:l}\Big) \tag{22}$$

$$\text{If } W^O_{\mathcal{P}:l} > \texttt{round}(\tau \times W^O_{:l}) : \tag{23}$$

$$W^O_{\mathcal{P}:l} = \texttt{round}(\tau \times W^O_{:l}) \tag{24}$$

$$W^I_{\mathcal{P}_{new}:l} = W^O_{\mathcal{P}:l} \times S_{x:l} - W_{\mathcal{K}:l} + 1 \tag{25}$$

$$x^I_{\mathcal{P}:l} \mathrel{+}= (W^I_{\mathcal{P}:l} - W^I_{\mathcal{P}_{new}:l})/2 \tag{26}$$

$$W^I_{\mathcal{P}:l} = W^I_{\mathcal{P}_{new}:l} \tag{27}$$

$$x^O_{\mathcal{P}:l} = \texttt{max}\Big(\lceil (P_{x:l} + x^I_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\Big) \tag{28}$$

Equation (22) calculates the width assuming no thresholding. But if the output width exceeds the threshold, it is reduced as per Equation (24). Equation (25) calculates the input width that would produce an output of width $W^O_{\mathcal{P}:l}$; we can think of this as making $W^I_{\mathcal{P}:l}$ the subject of Equation (22). If the new input width is smaller than the original input width, the starting x coordinate should be updated as per Equation (26) s.t. the new coordinates correspond to a "center crop" compared to the original. Equation (27) sets the input width to the newly calculated input width. Equation (28) calculates the x coordinate of the output update patch.

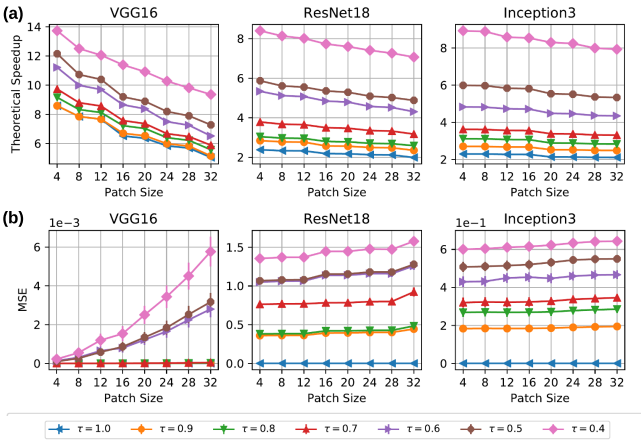

Figure 7: (a) Theoretical speedups with projective field thresholding. (b) Mean Square Error between exact and approximate output of final Convolution/Pooling layers.

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 7(a) plots the new theoretical speedups for varying patch sizes for 3 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 heat map 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 random real-world sample image. Figure 7(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.

4.2 Adaptive Drill-Down

This optimization is also a heuristic, but it is based on our observation about a peculiar semantics of the OBE task. It modifies the way G (the set of occlusion path 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" on uninteresting regions. In such cases, we modify the OBE workflow to produce an approximate heat map using a two-stage process, illustrated by Figure 8(a).

In stage one, we produce a *lower resolution* heat map by using a larger stride–we call it *stage one stride* S_1. Using this heat map, 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 heat map at the original higher resolution. Since this process "drills down" adaptively based on the lower resolution heat map, 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 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.

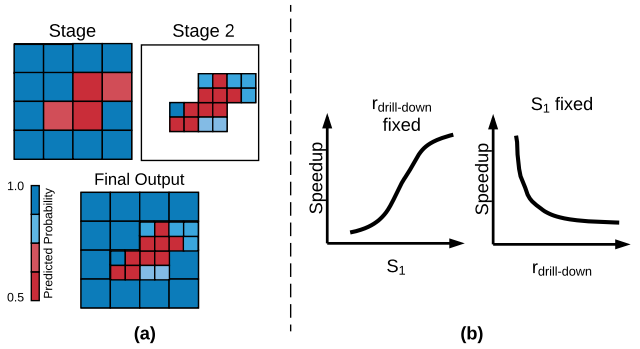

Figure 8: (a) Schematic illustration of the adaptive drill-down idea. (b) Conceptual depiction of the effects of S_1 and $r_{drill-down}$ on the theoretical speedup..

Figure 8(b) illustrates how this ratio varies with S_1 and $r_{drill-down}$.

$$\text{speedup} = \frac{S_1^2}{S_2^2 + r_{drill-down} \cdot S_1^2} \tag{29}$$

4.3 Automated Parameter Tuning

In this section we explain how Krypton sets its internal configuration parameters for *approximate inference* optimizations.

Tuning projective field threshold. The inaccuracies incurred when applying *projective field thresholding* can cause quality degradation in the generate approximate heat map all the way from indistinguishable changes major structural changes. To measure this quality degradation we use Structural Similarity (SSIM) Index [27], which is one of the widely used approaches to measure the *human perceived difference* between two similar images. When applying SSIM index, we treat the original heat map as the reference image with no distortions and the perceived image similarity of the approximate heat map is calculated with reference to it. 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 at the human perception level they produce indistinguishable distortions.

Tuning *projective field threshold* τ is done during a special initial tuning phase. During this tuning phase Krypton takes in a sample of images (default 30) from the operational workload and evaluates SSIM value of the approximate heat map (compared to the exact heat map) for different τ values (default values are 1.0, 0.9, 0.8, ..., 0.4). These τ versus SSIM data points are then used to fit a second-degree curve. At the operational time, Krypton requires the user

to provide the expected level of quality for the heat maps in terms of a SSIM value. τ is then selected from the curve fit to match this target SSIM value. Figure 9 (a) shows the SSIM variation and degree two curve fit for different τ values and three different CNN models for a tuning set (n=30) from OCT dataset. From the plots, it can be seen that the distribution of SSIM versus τ lies in a lower dimensional manifold and with increasing τ SSIM also increases. Figure 9 (b) shows the cumulative percentage plots for SSIM deviation for the tune and test sets (n=30) when the system is tuned for a target SSIM of 0.9. For a target SSIM of 0.9 system picks τ values of 0.5, 0.7, and 0.9 for VGG16, ResNet18, and Inception3 models respectively. It can be seen that approximately more than 50% of test cases will result in an SSIM value of 0.9 or greater. Even in cases where it performs worse than 0.9 SSIM, significant (95% – 100%) portion of them are within +0.1 deviation.

Tuning adaptive drill-down. As explained in section 4.2 the speedup obtained by *adaptive drill-down* approach is determined by two factors; stage one stride value S_1 and drill-down fraction $r_{drill-down}$. For configuring *adaptive drill-down* Krypton requires the user to provide $r_{drill-down}$ and a target speedup value. $r_{drill-down}$ should be selected based on the user's experience and understanding on the relative size of interesting regions compared to the full image. This is a fair assumption and in most cases such as in medical imaging, users will have a fairly good understanding on the relative size of the interesting regions. However, if the user is unable to provide this value Krypton will use a default value of 0.25 as $r_{drill-down}$. The speedup value basically captures user's input on how much faster the occlusion experiment should run. Higher speedup values will sacrifice the quality of non-interesting (1-$r_{drill-down}$) regions for faster execution. The default value for speedup value is three. The way how Krypton configures *adaptive drill-down* is different to how it configures *projective field thresholding*. The reason for this is, unlike in *projective field thresholding* in *adaptive drill-down* users have more intuition on the outcomes of $r_{drill-down}$ and target speedup parameters compared to the SSIM quality value of the final output. Given $r_{drill-down}$, target speedup value, and original occlusion patch stride value S_2 (also called stage two stride) Krypton then calculates the stage one stride value S_1 as per equation 30. As S_1 cannot be greater than the width W_{img} (similarly height H_{img}) of the image and with the mathematical constraint of $(1 - r_{drilldown} \times \text{speedup})$ being positive, it can be seen that possible values for the speedup value is upper-bounded as per equation 31.

$$S_1 = \sqrt{\frac{\text{speedup}}{1 - r_{drill-down} \times \text{speedup}}} \times S_2 \tag{30}$$

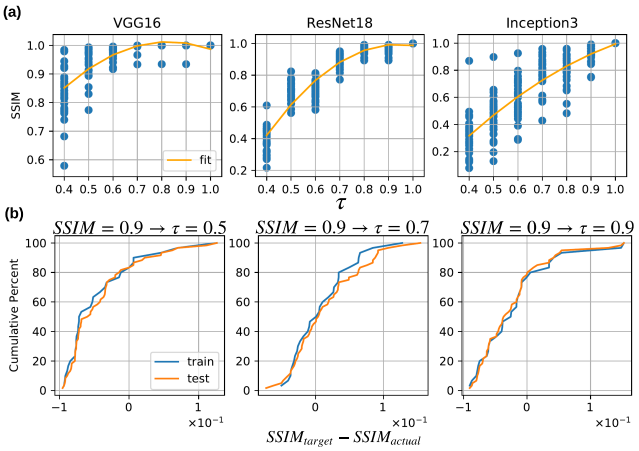

Figure 9: (a) SSIM variation and degree two curve fit for a sample of OCT dataset. (b) CDF plot for the SSIM deviation for the τ values picked from the curve fit for a target SSIM of 0.9.

$$S_1 = \sqrt{\frac{\text{speedup}}{1 - r_{drill-down} \times \text{speedup}}} \times S_2 < W_{img}$$
$$\text{speedup} < \min\left(\frac{W_{img}^2}{S_2^2 + r_{drill-down} \times W_{img}^2}, \frac{1}{r_{drill-down}}\right) \tag{31}$$

5 EXPERIMENTAL EVALUATION

We empirically validate if Krypton is able reduce the runtime taken for occlusion based deep CNN explainability workloads. We then conduct controlled experiments to show the individual contribution of each optimization in Krypton for the overall system efficiency.

Datasets. We use three real-world datasets: *OCT*, *Chest X-Ray*, and a sample from *ImageNet*. *OCT* has about 84,000 optical coherence tomography retinal images categorized into four categories: CNV, DME, DRUSEN, and NORMAL. CNV (choroidal neovascularization), DME (diabetic macular edema), and DRUSEN are three different varieties of Diabetic Retinopathy. NORMAL corresponds to healthy retinal images. *Chest X-Ray* has about 6,000 X-ray images categorized into three categories: VIRAL, BACTERIAL, and NORMAL. VIRAL and BACTERIAL categories correspond to two varieties of Pneumonia. NORMAL corresponds to chest X-Rays of healthy people. Both *OCT* and *Chest X-Ray* datasets are obtained from an original scientific study [6] which uses CNNs for predicting Diabetic Retinopathy and Pneumonia from radiological images. *ImageNet* sample dataset contains 1,000 images corresponding to two hundred categories selected from the original thousand categorical dataset [28].

Workloads. We use three popular ImageNet-trained Deep CNNs: VGG16 [2], ResNet18 [3], and Inception3 [4], obtained from [29]. They complement each other in terms of model size, computational cost, amount of theoretical redundancy that exist for occlusion experiments, and the level of architectural complexity of the CNN model. For *OCT* and *Chest X-Ray* datasets the three CNN models are fine-tuned by retraining the final fully-connected layer with hyper-parameter tuning as per standard practice. More details on the fine-tuning process are included in the Appendix. Heat map for the predicted probabilities is generated using Python Matplotlib library's `imshow` method using the `jet_r` color scheme. For the heat map, the maximum threshold value is set to $\min(1, 1.25 \times p)$ and minimum threshold value is set to $0.75 \times p$ where p is predicted class probability for the unmodified image. Original images were resized to the size required by the CNNs (224×224 for VGG16 and ResNet18 and 299×299 for Inception3) and no additional pre-processing is done. For GPU experiments a batch size of 128 and for CPU experiments a batch size 16 is used. CPU experiments are executed with a thread parallelism of 8. All of our datasets, fine-tuning, experiment, and system code will be made available on our project web page.

Experimental Setup. We use a workstation which has 32 GB RAM, Intel i7-6700 @ 3.40GHz CPU, 1 TB Seagate ST1000DM010-2EP1, and Nvidia Titan X (Pascal) 12 GB memory GPU. The system runs Ubuntu 16.04 operating system with PyTorch version of 0.4.0, CUDA version of 9.0, and cuDNN version of 7.1.2. Each runtime reported is the average of three runs with 95% confidence intervals shown.

5.1 End-to- End Evaluation

For the non-interactive GPU based environment we compare two variations of Krypton, Krypton-Exact which only applies the *incremental inference* optimization and Krypton-Approximate which applies both *incremental inference* and *approximate inference* optimizations, against two baselines. *Naive* is the current dominant practice of performing full inference for a batch of occluded instances of an image. *Naive Incremental Inference-Exact* is a pure PyTorch based implementation of Algorithm 1 which does not use any GPU optimized kernels for memory copying where as Krypton does. For non-interactive CPU based environment we only compare Krypton-Exact and Krypton-Approximate against *Naive* as no customization is needed for the pure PyTorch based implementation. For different datasets we set *adaptive drill-down* system tuning parameters differently. For *OCT* images the region of interest is relatively small and hence a $r_{drill-down}$ value of 0.1 and a target `speedup` of 5 is used. For *Chest X-Ray* images the region of interest can be large and hence a $r_{drill-down}$ value of 0.4 and a target `speedup` of 2 is used. For *ImageNet* experiments we use a $r_{drill-down}$ value of 0.25 and a target `speedup` value of 3 which are also the Krypton default values. For all experiments τ is configured using a separate tuning image dataset ($n = 30$) for a target SSIM of 0.9. Visual examples for each dataset is shown in Appendix. Figure 10 presents the final results.

We see that Krypton improves the efficiency of the occlusion based explainability workload across the board. Krypton-Approximate for *OCT* results in the highest speedup with VGG16 on both CPU and GPU environments (16X for CPU and 34.5X for GPU). Speedups obtained by Krypton-Exact for all the datasets are same for all three CNN models. However, with Krypton-Approximate they result in different speedup values. This is because with *approximate inference* each dataset uses different system configuration parameters. *OCT* which is configured with a low $r_{drill-down}$ of 0.1, high target `speedup` of 5, and a *projective field threshold* value of 0.5 results in the highest speedup. Speedup obtained by Krypton-Exact on GPU with Inception3 model (0.7X) is slightly lower than one. However ResNet18 which has roughly the same theoretical speedup (see Figure 3) results in a higher speedup value (1.6X). The reason for this is Inception3's internal architecture is more complex compared ResNet18 with more branches and depth-wise stacking operations. Thus Inception3 requires more memory copying operations whose overheads are not captured by our theoretical speedup calculation.

Compared to GPU environment, Krypton results in higher speedups on the CPU environment though the actual runtimes are much slower. GPUs enable higher parallelism with thousands of processing cores compared to CPUs with several cores. Hence computations are much cheaper on GPU. Memory operations required by Krypton throttles the overall performance on GPU and hinders it from achieving higher speedups. On CPU environment as computational cost dominates the overall runtime, the additional overhead introduced by the memory operations does not matter much. Therefore on CPU, Krypton achieves higher speedups which are closer to the theoretical speedup value. For the interactive mode with batched inference, Krypton yields similar results for the CPU environment. However, on the GPU environment with a smaller number of occlusion positions selected (i.e., small $|G|$) the speedup will be low. This is due to the high initialization overhead associated with GPU based CNN inference. As $|G|$ increases Krypton starts yielding higher speedups as it amortizes for the initialization overheads. Due to space constraints we include the results of interactive mode experiments in the Appendix.

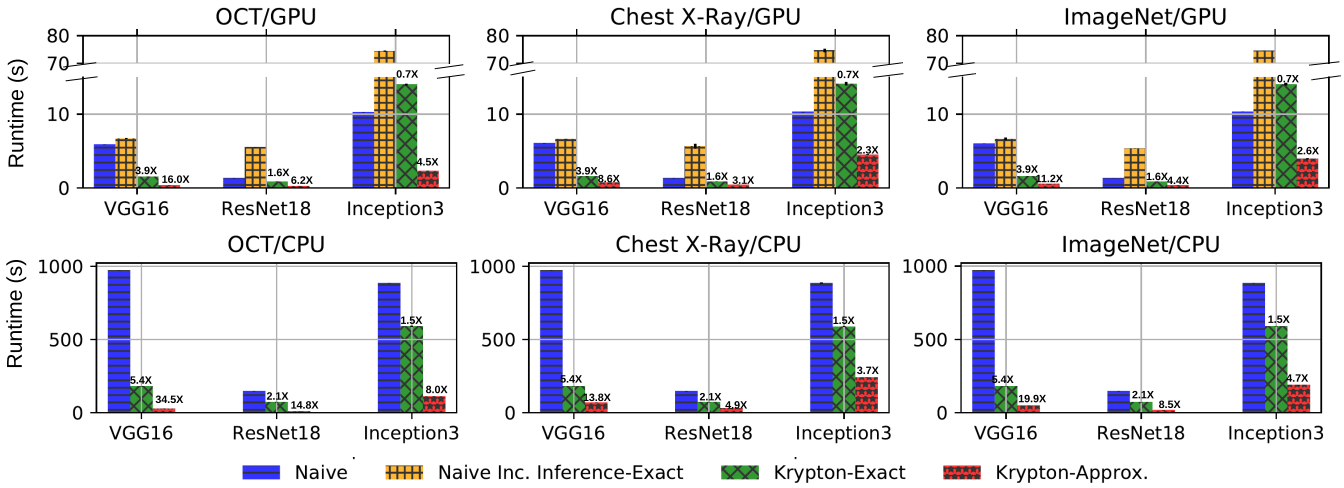

Figure 10: End-to-end efficiency achieved by KRYPTON over naive approaches.

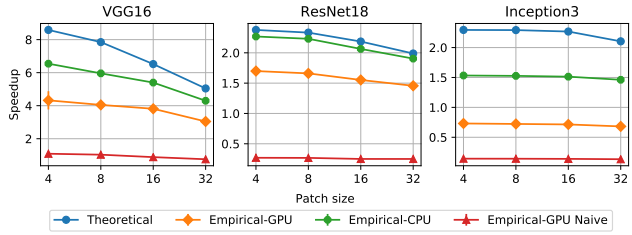

Figure 11: Theoretical versus empirical speedup for *incremental inference* (Occlusion patch stride $S = 4$).

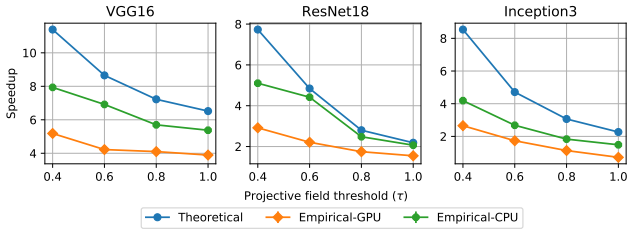

Figure 12: Theoretical versus empirical speedup for *incremental inference* with *projective field thresholding* (Occlusion patch size = 16×16, stride $S = 4$).

Overall KRYPTON improves the efficiency of the occlusion based deep CNN explainability workload. This confirms the benefits of different optimizations performed by KRYPTON for improving the efficiency of the workload and thereby to reduce the computational and runtime costs. Bringing down the runtimes also make occlusion experiments more amenable for interactive diagnosis of CNN predictions.

5.2 Lesion Study

We now present the results of controlled experiments that are conducted to identify the contribution of various optimizations discussed in Section 3 and Section 4. The speedup values are calculated compared to the runtime taken by the current dominant practice of performing full inference for batches of modified images. All the experiments are run in non-interactive mode.

Speedups from Incremental Inference. We compare theoretical speedup and empirical speedups obtained by *incremental inference* implementations for both CPU and GPU environments. The patch sizes that we have selected cover the range of sizes used in most practical applications. Occlusion patch stride is set to 4. Figure 11 shows the results. Empirical-GPU Naive results in the worst performance for all three CNN models. Empirical-GPU and Empirical-CPU implementations result in higher speedups with Empirical-CPU being closer to the theoretical speedup value. As the occlusion patch size increases the speedups decrease.

Speedups from Projective Field Thresholding. We vary *projective field threshold* τ from 1.0 (no thresholding) to 0.4 and evaluate the speedups. The occlusion patch size used is 16 and the stride is 4. The results are shown in Figure 12. Empirical-CPU and Empirical-GPU both results in higher speedups with Empirical-CPU being closer to the theoretical speedup value. When τ decreases the speedups increase as the amount of computational savings increase.

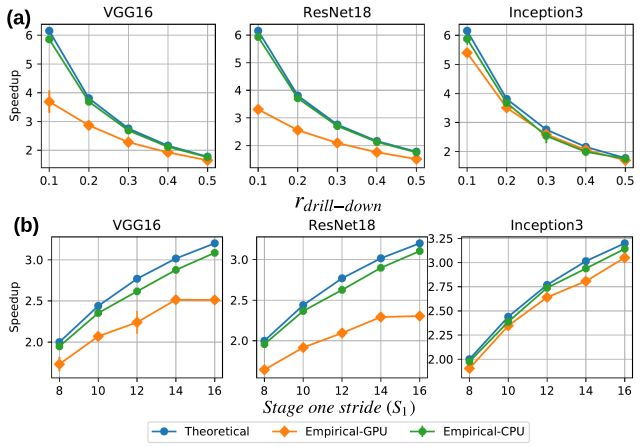

Figure 13: Theoretical versus empirical speedup for *adaptive drill-down* (Occlusion patch size = 16×16, stage two stride $S_2 = 4$, projective field threshold $\tau = 1.0$. For (a) S_1=16 and for (b) r_{drill_down}=0.25).

Speedups from Adaptive Drill-Down. Finally we evaluate the effect of *adaptive drill-down* on overall Krypton efficiency. The experiments are run on top of the *incremental inference* approach with no *projective field thresholding* (τ=1.0). $r_{drill-down}$ is varied between 0.1 to 0.5 fixing the stage one stride value S_1 to 16. Occlusion patch size is set to 16 and the stage two stride S_2 is set to 4. Figure 13 (a) shows the results. We also vary S_1 fixing $r_{drill-down}$ to 0.25. Occlusion patch size and the S_2 are set same as in the previous case. Figure 13 (b) presents the results. In both cases we see Empirical-GPU and Empirical-CPU achieve higher speedups with Empirical-CPU being very close to the theoretical speedup. On the CPU environment, the relative cost of other overheads is much smaller than the CNN computational cost. Hence on the CPU environment Krypton achieves near theoretical speedups for *adaptive drill-down*. Speedups decrease as we increase $r_{drill-down}$ and decrease S_1.

Summary of Experimental Results. Overall Krypton increases the efficiency of the occlusion based CNN explainability workload by up to 16X on GPU and 34.5X on CPU. Speedup obtained by *approximate inference* optimization (Krypton-Approximate) depends on the characteristics of the CNN model such as the effective growth of the projective field and the characteristics of the occlusion use case such as the relative size of the interesting regions on the image. Furthermore Krypton results in higher speedups on CPU environment compared to GPU environment. Increasing the occlusion patch size and τ decrease the speedup. Increasing $r_{drill-down}$ and decreasing S_1 also decrease the speedup.

6 OTHER RELATED WORK

Query Optimization. Our work is inspired by the long line of work on incremental view maintenance (IVM) based query processing [17–19]. The IVM approach presented in this paper falls into the broader category of *"lazy maintenance"* strategy. While there are many work in this context, most relevant to our work are [30] and [31]. [30] extended the IVM concept to support IVM of linear algebra operators as opposed to classical database queries. [31] extended the IVM concept to support multi-dimensional array data model with support for both relational style functions (e.g. filter and join) and also array style functions (e.g. smoothing and cross-matching). The spatially localized transformations explained in Section 2.2 can be thought of as a form of *"spatial array join"* introduced in [31]. However, the focus of [31] is on supporting efficient IVM for distributed sparse arrays by reducing the communication and chunk reassignment where as the focus of this work is on creating a comprehensive algebraic framework for the OBE workload by applying incremental inference optimization. We also takes into account the characteristics of human perception and semantics of CNN models to introduce approximate inference optimizations for OBE workload.

Our work also takes inspirations from the multi-query optimization (MQO) [32, 33] and approximate query processing (AQP) [34, 35] literature. We cast the multiple re-inference requests required for the OBE workload as multiple queries and apply MQO strategies. To the best of our knowledge, ours is the first to integrate IVM-style techniques with an MQO-style technique for optimizing CNN inference. The focus of existing AQP approaches is on answering analytical queries over relational data efficiently by trading-off the statistical error. In this work, we extend the AQP concepts to support CNN explanation queries over image data which trade-offs perceived quality of the heat maps for efficiency.

Multimedia DBMSs. There is prior work in the database and multimedia literatures on multimedia DBMSs [36, 37]. However, the focus of these work is on retrieval which includes content-based image retrieval (CBIR) and video retrieval using similarity searches or indices. Such systems are orthogonal to our work, since we focus on OBE for deep CNNs. CBinfer is a system for change based approximate evaluation of CNNs for real-time object recognition in video [38]. NoScope is a system to detect objects in video streams using cascades of CNNs [39]. Krypton is orthogonal to them, since it focuses on OBE, not object recognition. One could integrate Krypton with CBinfer and NoScope to explain the CNN predictions.

7 CONCLUSIONS AND FUTURE WORK

REFERENCES

[1] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[3] Kaiming He et al. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[5] Olga Russakovsky et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

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

[7] Mohammad Tariqul Islam et al. Abnormality detection and localization in chest x-rays using deep convolutional neural networks. *arXiv preprint arXiv:1705.09850*, 2017.

[8] Sharada P Mohanty et al. Using deep learning for image-based plant disease detection. *Frontiers in plant science*, 7:1419, 2016.

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

[10] Yilun Wang and Michal Kosinski. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. 2017.

[11] Ai device for detecting diabetic retinopathy earns swift fda approval. https://www.aao.org/headline/first-ai-screen-diabetic-retinopathy-approved-by-f. Accessed September 31, 2018.

[12] Radiologists are often in short supply and overworked deep learning to the rescue. https://government.diginomica.com/2017/12/20/radiologists-often-short-supply-overworked-deep-learning-rescue. Accessed September 31, 2018.

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

[14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[15] Nikhil Ketkar. Introduction to pytorch. In *Deep Learning with Python*, pages 195–208. Springer, 2017.

[16] Luisa M Zintgraf et al. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.

[17] Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

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

[19] Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. Answering queries using views. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM, 1995.

[20] Cafee model zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo. Accessed September 31, 2018.

[21] Models and examples built with tensorflow. https://github.com/tensorflow/models. Accessed September 31, 2018.

[22] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[23] Hung Le and Ali Borji. What are the receptive, effective receptive, and projective fields of neurons in convolutional neural networks? *arXiv preprint arXiv:1705.07049*, 2017.

[24] Basic operations in a convolutional neural network - cse@iit delhi. http://www.cse.iitd.ernet.in/~rijurekha/lectures/lecture-2.pptx. Accessed September 31, 2018.

[25] Saskia EJ de Vries et al. The projective field of a retinal amacrine cell. *Journal of Neuroscience*, 31(23):8595–8604, 2011.

[26] Wenjie Luo et al. Understanding the effective receptive field in deep convolutional neural networks. In *Advances in neural information processing systems*, pages 4898–4906, 2016.

[27] Zhou Wang et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[28] Jia Deng, Wei Dong, et al. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[29] torch vison models. https://github.com/pytorch/vision/tree/master/torchvision/models. Accessed September 31, 2018.

[30] Milos Nikolic, Mohammed ElSeidy, and Christoph Koch. Linview: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 253–264. ACM, 2014.

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

[32] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[33] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.

[34] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476. ACM, 2018.

[35] Minos N Garofalakis and Phillip B Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, pages 343–352, 2001.

[36] Donald A Adjeroh and Kingsley C Nwosu. Multimedia database managementrequirements and issues. *IEEE multimedia*, (3):24–33, 1997.

[37] Oya Kalipsiz. Multimedia databases. In *Information Visualization, 2000. Proceedings. IEEE International Conference on*, pages 111–115. IEEE, 2000.

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

[39] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

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

[41] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

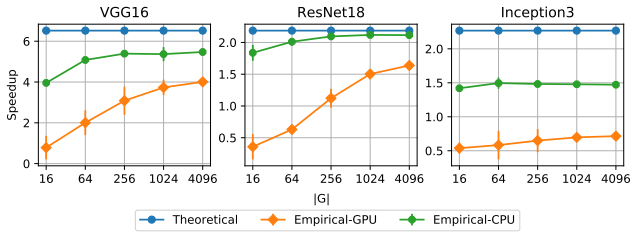

Figure 14: Interactive mode execution of incremental inference with Gs of different sizes

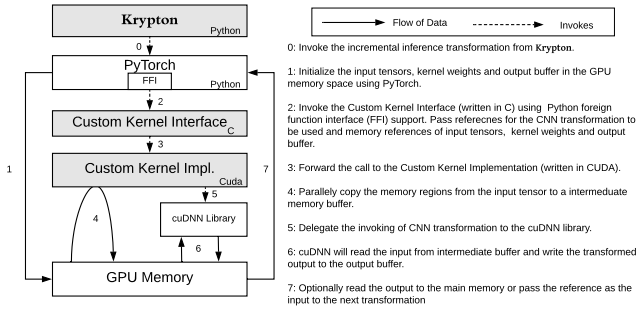

Figure 15: Custom GPU Kernel integration architecture

A 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 14 presents the final results.

B GPU OPTIMIZED KERNEL IMPLEMENTATION

We extend PyTorch by adding a custom GPU kernel which optimizes the input preparation for *incremental inference* by invoking parallel memory copy operations. This custom kernel is integrated to PyTorch using Python foreign function interface (FFI). Python FFI integrates with the Custom Kernel Interface layer which intern invokes the Custom Kernel Implementation. The high-level architecture of the Custom Kernel integration is shown in Figure **??**

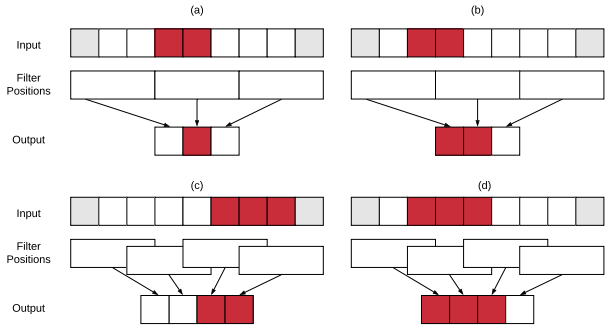

Figure 16: One dimensional representation showing special situations under which actual output size will be smaller than the value calculated by Equation 14. (a) and (b) shows a situation with filter stride being equal to the filter size. (c) and (d) shows a situation with input patch being placed at the edge of the input.

C SPECIAL SITUATIONS WITH INCREMENTAL INFERENCE

It is important to note that there are special situations under which the actual output patch size can be smaller than the values calculated in Section 3.2. Consider the simplified one dimensional situation shown in Figure 16 (a), where the stride value[1] (3) is same as the filter size (3). In this situation, the size of the output patch is one less than the value calculated by Equation 15. However, it is not the case in Figure 16 (b) which has the same input patch size but is placed at a different location. Another situation arises when the input patch is placed at the edge of the input as shown in Figure 16 (c). In this situation, it is not possible for the filter to move freely through all filter positions as it hits the input boundary compared to having the input patch on the middle of the input as shown in Figure 16 (c). In Krypton we do not treat theses differences separately and use the values calculated by Equation (15) for the horizontal dimension (similarly for the vertical dimension) as they act as an upper bound. In case of a smaller output patch, Krypton simply reads off and updates slightly bigger patches to preserve uniformity. This also requires updating the starting coordinates of the patches as shown in Equation (32). Such uniform treatment is required for performing batched inference operations which out of the box gives significant speedups compared to per image inference.

[1]Note that the stride value is generally less than or equal to the filter size.

$$\begin{aligned} &\text{If } x_{\mathcal{P}}^{O} + W_{\mathcal{P}}^{O} > W_O: \\ &\quad x_{\mathcal{P}}^{O} = W_O - W_{\mathcal{P}}^{O} \\ &\quad x_{\mathcal{P}}^{\mathcal{I}} = W_{\mathcal{I}} - W_{\mathcal{P}}^{\mathcal{I}} \\ &\quad x_{\mathcal{P}}^{\mathcal{R}} = W_{\mathcal{I}} - W_{\mathcal{P}}^{\mathcal{R}} \end{aligned} \tag{32}$$

D EFFECTIVE PROJECTIVE FIELD SIZE (ONE DIMENSIONAL SCENARIO)

We formalize the effective projective field growth for the one dimensional scenario with n convolution layers (assuming certain conditions).

The input is $u(t)$ where

$$u(t) = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \tag{33}$$

and $t = 0, 1, -1, 2, -2, \ldots$ indexes the input pixels.

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

$$v(t) = \sum_{m=0}^{k-1} w(m)\delta(t-m) \tag{34}$$

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

$$\begin{aligned} U(\omega) &= \sum_{t=-\infty}^{\infty} u(t)e^{-j\omega t} = 1,\ V(\omega) \\ &= \sum_{t=-\infty}^{\infty} v(t)e^{-j\omega t} = \sum_{m=0}^{k-1} w(m)e^{-j\omega t} \end{aligned} \tag{35}$$

Applying the convolution theorem, we have the Fourier transform of o is

$$\begin{aligned} \mathcal{F}(o) &= \mathcal{F}(u * v * \ldots * v)(\omega) = U(\omega).V(\omega)^n \\ &= \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t} \right)^n \end{aligned} \tag{36}$$

With inverse Fourier transform

$$o(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t} \right)^n e^{j\omega t} d\omega \tag{37}$$

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 [40]. It can be shown that if $\sum_{\mathbf{m}} \mathbf{w}(\mathbf{m}) = \mathbf{1}$ and $\mathbf{w}(\mathbf{m}) \geq \mathbf{0}\ \forall\ \mathbf{m}$, then

$$\begin{aligned} o(t) &= p(S_n = t) \\ \text{where } S_n &= \sum_{i=1}^{n} X_i \text{ and } p(X_i = m) = w(m) \end{aligned} \tag{38}$$

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

$$\mathbb{E}[S_n] = n \sum_{m=0}^{k-1} mw(m) \tag{39}$$

$$Var[S_n] = n\left(\sum_{m=0}^{k-1} m^2 w(m) - \left(\sum_{m=0}^{k-1} mw(m) \right)^2 \right) \tag{40}$$

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

$$\sqrt{Var[S_n]} = \sqrt{nVar[X_i]} = O(\sqrt{n}) \tag{41}$$

On the other hand stacking more convolution layers would grow the theoretical projective field linearly. But the effective projective field size is shrinking at a rate of $O(1/\sqrt{n})$.

E FINE-TUNING CNNS

For *OCT* and *Chest X-Ray* datasets the three ImageNet pretrained CNN models are fine-tuned by retraining the final layer. We use a train-validation-test split of 60-20-20 and the exact numbers for each dataset are shown in Table 3. Cross-entropy loss with L2 regularization is used as the loss function and Adam [41] is used as the optimizer. We tune learning rate $\eta \in [10^{-2}, 10^{-4}, 10^{-6}]$ and regularization parameter $\lambda \in [10^{-2}, 10^{-4}, 10^{-6}]$ using the validation set and train for 25 epochs. Table 4 shows the final train and test accuracies.

	Train	Validation	Test
OCT	50,382	16,853	16, 857
Chest X-Ray	3,463	1,237	1,156

Table 3: Train-validation-test split size for each dataset.

F VISUAL EXAMPLES

Figure 17 presents occlusion heat maps for a sample image from each dataset with (a) *incremental inference* and (b) *incremental inference* with *adaptive drill-down* for different

	Model	Accuracy(%)		Hyperparams.	
		Train	Test	η	λ
OCT	VGG16	79	82	10^{-4}	10^{-4}
	ResNet18	79	82	10^{-2}	10^{-4}
	Inception3	71	81	10^{-2}	10^{-6}
Chest X-Ray	VGG16	75	76	10^{-4}	10^{-4}
	ResNet18	78	76	10^{-4}	10^{-6}
	Inception3	74	76	10^{-4}	10^{-2}

Table 4: Train and test accuracies after fine-tuning.

projective field threshold values. The predicted class label for *OCT*, *Chest X-Ray*, and *ImageNet* are DME, VIRAL, and OBOE respectively.

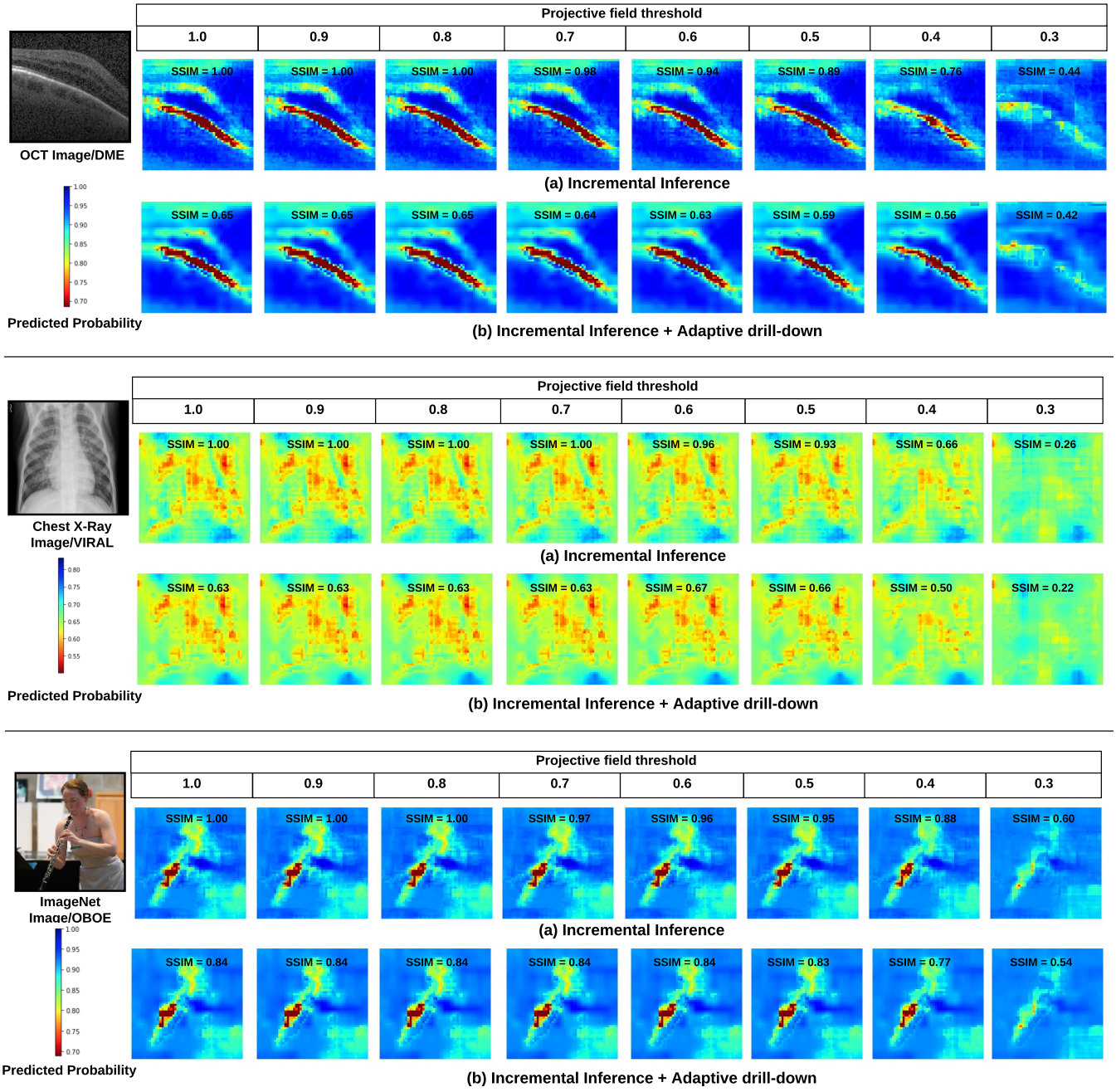

Figure 17: Occlusion heat maps for sample images (CNN model = VGG16, occlusion patch size = 16, patch color = black, occlusion patch stride (S or S_2) = 4. For *OCT* $r_{drill\_down} = 0.1$ and target speedup=5. For *Chest X-Ray* $r_{drill\_down} = 0.4$ and target speedup=2. For *ImageNet* $r_{drill\_down} = 0.25$ and target speedup=3).