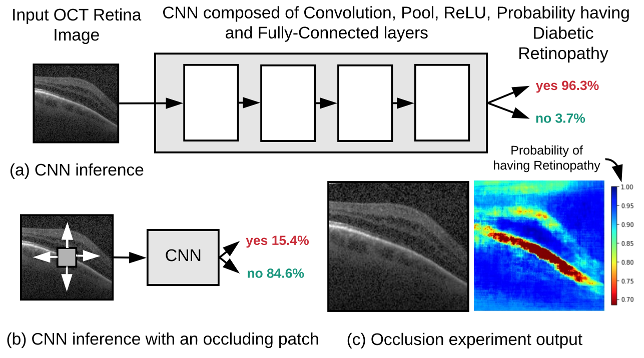
Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations

ABSTRACT

Deep Convolutional Neural Networks (CNNs) now match human accuracy in many image prediction tasks, resulting in a growing adoption in e-commerce, radiology, and other domains. Naturally, "explaining" CNN predictions is a key concern for many users. Since the internal workings of CNNs are unintuitive for most users, *occlusion-based explanations* (OBE) are popular for understanding which parts of a image matter most for a prediction. One occludes a region of the image using a patch and moves it around to produce 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 wastes time and raises resource costs. We tackle this issue by casting the OBE task as a new instance of the classical incremental view maintenance problem. We create a novel and comprehensive algebraic framework for incremental CNN inference combining materialized views with multi-query optimization to reduce computational costs. We then present two novel approximate inference optimizations that exploit the semantics of CNNs and the OBE task to further reduce runtimes. We prototype our ideas in Python to create a tool we call KRYPTON that supports both CPUs and GPUs. Experiments with real data and CNNs show that KRYPTON reduces runtimes by up to 5X (resp. 35X) to produce exact (resp. high-quality approximate) results 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, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



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

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

1 INTRODUCTION

Deep Convolution Neural Networks (CNNs) are now the state of the art method for many image prediction tasks [1]. Thus, there is growing interest in adopting deep CNNs in various application domains, including healthcare [2, 3], agriculture [4], security [5], and sociology [6]. Remarkably, even the US Food and Drug Administration recently approved the use of deep CNNs in radiology to assist radiologists in processing X-rays and other scans, cross-checking their decisions, and even mitigating the shortage of radiologists [7, 8].

Despite their successes, a key criticism of CNNs is that their internal workings are unintuitive to non-technical users. Thus, users often seek an "explanation" for why a CNN predicted a certain label. Explanations can help users trust CNNs [9], especially in high stakes applications such as radiology [10], and are a legal requirement for machine learning applications in some countries [11]. How to explain a CNN prediction is still an active research question, but in the practical literature, an already popular mechanism for CNN explanations is a simple procedure called *occlusion-based explanations* [12], or OBE for short.

OBE works as follows. Place a small square patch (usually gray or black) on the image to occlude those pixels. Rerun CNN inference, illustrated in Figure 1 (a), on the occluded image. The probability of the predicted label will change, as Figure 1 (b) shows. Repeat this process by moving the patch across the image to obtain a sensitivity *heat map* of the probability changes, as Figure 1 (c) shows. This heat map will highlight regions of the image that were highly sensitive or "responsible" for the prediction (red/orange color regions). Such *localization* of the regions of interest allows users to gain intuition on what "mattered" for the CNN prediction. For instance, the heat map can highlight the diseased areas of a tissue image, which a radiologist can then inspect more deeply for further tests. Overall, OBE is popular because it is easy for non-technical users to understand.

Alas, OBE is highly computationally expensive. Deep CNN inference is already expensive; OBE just amplifies it by issuing a large number of CNN re-inference requests (often 1000s) [13]. For example, [14] report over 500,000 re-inference requests for OBE on one image, which took 1hr even on a GPU! Such long wait times can hinder users' ability to consume explanations and reduce their productivity. One could use more compute hardware, if available, since OBE is embarrassingly parallel across re-inference requests. But throwing more machines at it may not always be affordable, especially for domain scientists, or feasible in all settings, e.g., in mobile clinical diagnosis. Using extra resources can also raise monetary costs, especially in the cloud.

In this paper, we use a database-inspired lens to formalize, optimize, and accelerate OBE. We start with a simple but crucial observation: *the occluded images are not disjoint but share most of their pixels; so, most of CNN re-inference computations are redundant.* This observation leads us to connect OBE with two classical data management concerns: *incremental view maintenance* (IVM) and *multi-query optimization* (MQO). Instead of treating a CNN as a "blackbox," we open it up and formalize *CNN layers* as "queries." Just like how a relational query coverts relations to other relations, a CNN layer converts *tensors* (multidimensional arrays) to other tensors. So, we reimagine OBE as *a set of tensor transformation queries* with incrementally updated inputs. With this fresh database-inspired view, we introduce several *novel CNN-specific query optimization techniques* to accelerate OBE.

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

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

We prototype our ideas in the popular deep learning framework PyTorch to create a tool we call Krypton. It works on both CPU and GPU and currently supports a few popular deep CNNs (VGG16, ResNet18, and InceptionV3). We perform a comprehensive empirical evaluation of Krypton with three real-world image datasets from recent radiology and computer vision papers. Krypton yields up to 35X speedups over the current dominant practice of running re-inference with just batching for producing high-quality approximate heat maps and up to 5X speedups for producing exact heat maps. We then analyze the utility of each of our optimizations. Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to formalize and optimize the execution of occlusion-based explanations (OBE) of CNN predictions from a data management standpoint.
- We cast OBE as an IVM problem to create a novel and comprehensive algebraic framework for incremental CNN inference. We also combine our IVM technique with an MQO-style technique to further reduce computational redundancy in CNN inference.
- We present two novel approximate inference optimizations for OBE that exploit the semantics of CNNs and properties of human perception.
- We prototype our ideas in a tool, Krypton, and perform an extensive empirical evaluation with real data and deep CNNs. Krypton speeds up OBE by even over an order of magnitude in some cases.

Outline. Section 2 explains our problem setup, assumptions, formalization of the dataflow of CNNs. Section 3 presents our incremental inference and multi-query optimizations. Section 4 presents our approximate inference optimizations. Section 5 presents the experiments. We discuss other related work in Section 6 and conclude in Section 7.

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)
$O_{:l}$	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

Table 1: Notation used in this paper.

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

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

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

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

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

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

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

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 is used for classification (or regression), since only such applications typically use OBE. One could create CNNs that predict an image "segmentation" instead, but labeling image segments for training such CNNs is tedious and expensive. Thus, most recent applications of CNNs in healthcare, sociology, and other domains rely on classification CNNs and use OBE [2–6]. Other approaches to explain CNN predictions have been studied, but since they are orthogonal to our focus, we summarize them in the appendix due to space constraints. We assume f is from a roster of well-known deep CNNs; we currently support VGG16, ResNet18, and InceptionV3. We think this is a reasonable start, since most recent OBE applications use only such well-known CNNs from model zoos [15, 16]. But we note that our techniques are generic enough to apply to any CNN; we leave support for arbitrary CNNs to future work.

2.2 Dataflow of CNN Layers

CNNs are organized as *layers* of various types, each of which transforms a tensor (multidimensional array, typically 3-D) into another tensor: *Convolution* uses image filters from graphics to extract features, but with parametric filter weights (learned during training); *Pooling* subsamples features in a spatial-aware manner; *Batch-Normalization* normalizes the output tensor; *Non-Linearity* applies an element-wise non-linear function (e.g., ReLU); *Fully-Connected* is an ordered collection of perceptrons [17]. The output tensor of a layer can have a different width, height, and/or depth than the input. An image can be viewed as a tensor, e.g., a 224×224 RGB image is a 3-D tensor with width and height

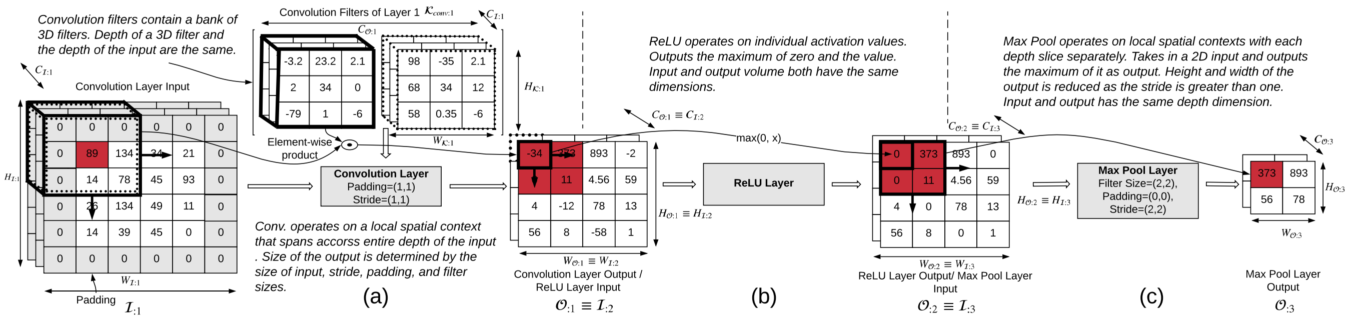

Convolution filters contain a bank of 3D filters. Depth of a 3D filter and the depth of the input are the same.
Convolution Filters of Layer 1 $\mathcal{K}_{conv:1}$
Convolution Layer Input
Element-wise product
Convolution Layer
Padding=(1,1)
Stride=(1,1)
Conv. operates on a local spatial context that spans accross entire depth of the input . Size of the output is determined by the size of input, stride, padding, and filter sizes.
Padding
$\mathcal{I}_{:1}$
(a)
Convolution Layer Output / ReLU Layer Input
$\mathcal{O}_{:1} \equiv \mathcal{I}_{:2}$
ReLU operates on individual activation values. Outputs the maximum of zero and the value. Input and output volume both have the same dimensions.
max(0, x)
ReLU Layer
(b)
ReLU Layer Output/ Max Pool Layer Input
$\mathcal{O}_{:2} \equiv \mathcal{I}_{:3}$
Max Pool operates on local spatial contexts with each depth slice separately. Takes in a 2D input and outputs the maximum of it as output. Height and width of the output is reduced as the stride is greater than one. Input and output has the same depth dimension.
Max Pool Layer
Filter Size=(2,2),
Padding=(0,0),
Stride=(2,2)
(c)
Max Pool Layer Output
$\mathcal{O}_{:3}$

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.

224 and depth 3. A Fully-Connected layer converts a 1-D tensor (or a "flattened" 3-D tensor) to another 1-D tensor. For simplicity of exposition, we group CNN layers into 3 main categories based on the *spatial locality* of how they transform a tensor: (1) Transformations with a *global context*, e.g., Fully-Connected; (2) Transformations at the granularity of *individual elements*, e.g., ReLU or Batch Normalization; and (3) Transformations at the granularity of a *local spatial context*, e.g. Convolution or Pooling.

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(s) 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. Non-Linearity 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 with 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 using our notation. Section 3 then uses this machinery to explain our optimizations.

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

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

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 on the depth slices of the input tensor. It takes as input a 3-D tensor O_l of depth $C_{\mathcal{I}:l}$, width $W_{\mathcal{I}:l}$, and height $H_{\mathcal{I}:l}$. It produces as output a 3-D tensor $O_{:l}$ with the same depth $C_{O:l} = C_{\mathcal{I}:l}$ 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. Thus, $W_{O:l}$ and $H_{O:l}$ are usually smaller than $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$. 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 $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$, $W_{\mathcal{K}:l}$ and $H_{\mathcal{K}:l}$, and two other parameters that are specific to that layer: *stride* $S_{:l}$ and *padding* $P_{:l}$. Stride is the number of pixels by which the filter kernel is moved at a time; it can differ along the width and height dimensions: $S_{x:l}$ and $S_{y:l}$, respectively. 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:

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

Computational Cost of Inference. Deep CNN inference is computationally expensive. Convolution layers typically account for a bulk of the cost (90% or more) [18]. Thus, we can roughly estimate the computational cost of inference by counting the number of *fused multiply-add* (FMA) floating point operations (FLOPs) needed for the Convolution layers. For example, applying a Convolution filter with dimensions $(C_{\mathcal{I}:l}, H_{\mathcal{K}:l}, W_{\mathcal{K}:l})$ to compute one element of the output tensor requires $C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l}$ FLOPs, with each FLOP corresponding to one FMA. Thus, the total computational cost $Q_{:l}$ of a layer that produces output $O_{:l}$ of dimensions $(C_{O:l}, H_{O:l}, W_{O:l})$ and the total computational cost Q

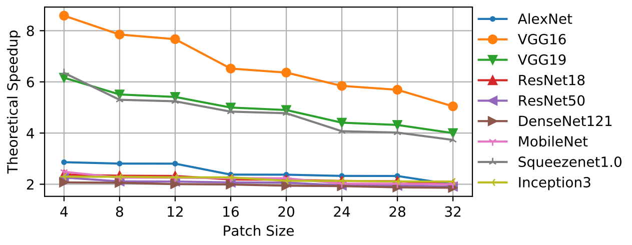


Figure 3: Theoretical speedups for popular deep CNN architectures with incremental inference.

of the entire set of Convolution layers of a given CNN f can be calculated as per Equations (9) and (10).

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

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

3 INCREMENTAL INFERENCE OPTIMIZATIONS

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

3.1 Expected Speedups

The basic reason why IVM 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; a CNN layer is our "query" and the materialized feature tensor is our "relation." OBE updates only a part of the image; so, only some parts of the tensors need to be recomputed. We create an algebraic framework to determine which parts these are for a CNN layer (Section 3.2) and how to propagate updates across layers (Section 3.3). Given a CNN f and the occlusion patch, our framework determines using "static analysis" over f how many FLOPs can be saved. This gives us 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 (11) and (12).

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

$$Q_{inc} = \sum_{l \text{ in } f} Q_{inc:l} \tag{12}$$

The above costs can be much smaller than $Q_{:l}$ and Q in Equations (9) and (10) 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, we plot the theoretical speedups of different deep CNNs. 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.

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 $(x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l})$ and has a height $H^{\mathcal{O}}_{\mathcal{P}:l}$ and width $W^{\mathcal{O}}_{\mathcal{P}:l}$. Due to overlaps among filter kernel positions during inference, computing the output update patch requires us to read a slightly larger spatial context than the input update patch–we call this the "read-in context," and it is illustrated by the blue/shaded region in Figure 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:

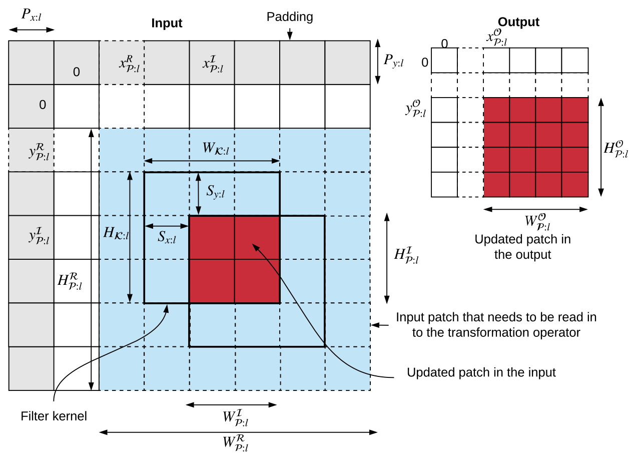

$P_{x:l}$
Input
Padding
Output
0
$x^{\mathcal{R}}_{\mathcal{P}:l}$
$x^{\mathcal{I}}_{\mathcal{P}:l}$
$P_{y:l}$
$x^{\mathcal{O}}_{\mathcal{P}:l}$
0
0
0
$y^{\mathcal{O}}_{\mathcal{P}:l}$
$H^{\mathcal{O}}_{\mathcal{P}:l}$
$y^{\mathcal{R}}_{\mathcal{P}:l}$
$W_{\mathcal{K}:l}$
$S_{y:l}$
$y^{\mathcal{I}}_{\mathcal{P}:l}$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

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

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

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

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

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

Equation (13) 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 (14) calculates the width of the output update patch. Given these, a start coordinate and width of the read-in context are given by Equations (15) and (16); similar equations hold for the height dimension (skipped for brevity).

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

Table 2: Additional notation for Sections 3 and 4.

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:

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

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

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

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

3.3 Propagating Updates across Layers

Sequential CNNs. Unlike relational IVM, CNNs have many layers, often in a sequence. This is analogous to having 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 lose spatial locality for subsequent layers.

Extension to DAG CNNs. Some recent deep CNNs have a more general directed acyclic graph (DAG) structure for layers. They have two new kinds of layers that "merge" two branches in the DAG: *element-wise addition* and *depth-wise concatenation*. Element-wise addition requires two input tensors with all dimensions being identical. Depth-wise concatenation takes two input tensors with the same height and width dimensions. We now face a new challenge–how to calculate the output update patch when the two input tensors differ on their input update patches locations and sizes? Figure 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 starting at coordinates $(x^{\mathcal{I}}_{\mathcal{P}_2:l}, y^{\mathcal{I}}_{\mathcal{P}_2:l})$ with dimensions $H^{\mathcal{I}}_{\mathcal{P}_2:l}$ and $W^{\mathcal{I}}_{\mathcal{P}_2:l}$. This issue can arise with both element-wise addition and depth-wise concatenation.

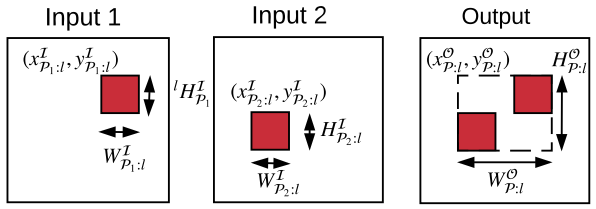

Input 1
Input 2
Output
$(x^{\mathcal{I}}_{\mathcal{P}_1:l}, y^{\mathcal{I}}_{\mathcal{P}_1:l})$
${}^{l}H^{\mathcal{I}}_{\mathcal{P}_1}$
$W^{\mathcal{I}}_{\mathcal{P}_1:l}$
$(x^{\mathcal{I}}_{\mathcal{P}_2:l}, y^{\mathcal{I}}_{\mathcal{P}_2:l})$
$H^{\mathcal{I}}_{\mathcal{P}_2:l}$
$W^{\mathcal{I}}_{\mathcal{P}_2:l}$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

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.

We propose a simple unified solution: compute the *bounding box* of the input update patches. So, the coordinates and dimensions of both read-in contexts and the output update patch will be identical. Figure 5 illustrates this. While this will potentially recompute parts of the output that do 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):

$$\begin{aligned} x^{\mathcal{O}}_{P:l} &= \mathtt{min}(x^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l}) \\ W^{\mathcal{O}}_{\mathcal{P}:l} &= \mathtt{max}(x^{\mathcal{I}}_{\mathcal{P}_1:l} + W^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l} + W^{\mathcal{I}}_{\mathcal{P}_2:l}) - \mathtt{min}(x^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l}) \end{aligned} \tag{20}$$

3.4 Multi-Query Incremental Inference

OBE issues $|G|$ re-inference requests *in one go*. Viewing each request as a "query" makes the connection with multi-query optimization (MQO) [19] clear. The $|G|$ queries are also *not disjoint*, since the occlusion patch is typically much smaller than the image, which means most the pixels are the same for each query. Thus, we now extend our IVM framework for one re-inference with an MQO-style optimization fusing multiple re-inference requests. An analogy with relational queries would be having many incremental update queries on the same relation in one go, with each query receiving a different incremental update.

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
1:  procedure BatchedIncrementalInference
2:      Calculate $[(x^{\mathcal{O}}_{\mathcal{P}_1:l}, y^{\mathcal{O}}_{\mathcal{P}_1:l}), ..., (x^{\mathcal{O}}_{\mathcal{P}_n:l}, y^{\mathcal{O}}_{\mathcal{P}_n:l})]$
3:      Calculate $(W^{\mathcal{O}}_{\mathcal{P}:1}, H^{\mathcal{O}}_{\mathcal{P}:l})$
4:      Calculate $[(x^{\mathcal{R}}_{\mathcal{P}_1:l}, y^{\mathcal{R}}_{\mathcal{P}_1:l}), ..., (x^{\mathcal{R}}_{\mathcal{P}_n:l}, y^{\mathcal{R}}_{\mathcal{P}_n} : l)]$
5:      Calculate $(W^{\mathcal{R}}_{\mathcal{P}:l}, H^{\mathcal{R}}_{\mathcal{P}:l})$
6:      Initialize $\mathcal{U} \in \mathbb{R}^{n \times \mathsf{depth}(\mathcal{I}_{:l}) \times H^{\mathcal{R}}_{\mathcal{P}:l} \times W^{\mathcal{R}}_{\mathcal{P}:l}}$
7:      for i in [1,...,n] do
8:          $T_1 \leftarrow \mathcal{I}_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}_i:l} : x^{\mathcal{R}}_{\mathcal{P}_i:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}_i:l} : y^{\mathcal{R}}_{\mathcal{P}_i:l} + H^{\mathcal{R}}_{\mathcal{P}:l}]$
9:          $T_2 \leftarrow T_1 \circ_{(x^{\mathcal{I}}_{\mathcal{P}_i:l} - x^{\mathcal{R}}_{\mathcal{P}_i:l}),(y^{\mathcal{I}}_{\mathcal{P}_i:l} - y^{\mathcal{R}}_{\mathcal{P}_i:l})} \mathcal{P}_{i:l}$
10:         $\mathcal{U}[i, :, :] \leftarrow T_2$
11:     $[\mathcal{P}^{\mathcal{O}}_{1:l}, ..., \mathcal{P}^{\mathcal{O}}_{n:l}] \leftarrow T(\mathcal{U})$          ▷ Batched version
12:     return $[\mathcal{P}^{\mathcal{O}}_{1:l}, ..., \mathcal{P}^{\mathcal{O}}_{n:l}]$,
13:         $[(x^{\mathcal{O}}_{\mathcal{P}_1:l}, y^{\mathcal{O}}_{\mathcal{P}_1:l}), ..., (x^{\mathcal{O}}_{\mathcal{P}_n:l}, y^{\mathcal{O}}_{\mathcal{P}_n:l})]$, $(W^{\mathcal{O}}_{\mathcal{P}:l}, H^{\mathcal{O}}_{\mathcal{P}:l})$
```

Batched Incremental Inference. Our optimization works as follows: materialize all tensors *once* and *reuse* them for incremental CNN inference across all $|G|$ queries. Since most of the occluded image pixels are identical, parts of the tensors will likely be identical too. Thus, 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 like GPUs, since it amortizes CNN set up costs, data movement costs, etc. Batch sizes are picked to optimize hardware utilization. We observed that batching is an *orthogonal* (albeit trivial) optimization compared to our optimization. Thus, we can combine both of these ideas to execute incremental inference in a batched manner. We call this approach "batched incremental inference." Empirically, we found that batching alone yields only limited speedups (under 2X), but combining batching and incremental inference as we do can amplify the speedups. Algorithm 1 formally presents the batched incremental inference operation for layer l.

BatchedIncrementalInference first calculates the geometric properties of the output update patches and read-in contexts. A temporary tensor $\mathcal{U}$ is initialized to hold the input update patches with their read-in contexts. The **for** loop iteratively populates $\mathcal{U}$ with corresponding patches. Finally, $T_{:l}$ is applied to $\mathcal{U}$ to compute the output patches. We note that 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 found a dichotomy between CPUs and GPUs: BatchedIncrementalInference yielded expected speedups on CPUs, but it performed dramatically poorly on GPUs. In fact, a naive implementation of BatchedIncrementalInference on GPUs was *slower* than full re-inference! We now shed light on why this is the case and how we tackled this issue. The **for** loop in line 7 of Algorithm 1 is essentially preparing the input for $T_{:l}$ by copying values (slices of the materialized tensor) from one part of GPU memory to another sequentially. A detailed profiling of the GPU showed that these *sequential memory copies are a bottleneck* for GPU throughput, since they throttle it from exploiting its massive parallelism effectively. To overcome this issue, we created a custom CUDA kernel to perform input preparation more efficiently by *copying memory regions in parallel* for all items in the batched inference request. This is akin to a parallel **for** loop tailored for slicing the tensor. We then invoke $T_{:l}$, which is already hardware-optimized by modern deep learning tools [20]. 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 GPU execution might be smaller than $|G|$.

3.5 Putting it All Together

We summarize the end-to-end workflow of our incremental inference optimizations for OBE. We are given the CNN f, image $\mathcal{I}_{:img}$, 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 $\mathcal{I}_{:img}$ as the input. Prepare occluded images ($\mathcal{I}'_{(x,y):img}$) for all positions in G. For batches of $\mathcal{I}'_{(x,y):img}$ as the input, invoke the transformations functions of the layers of f in topological order and calculate the corresponding entries of heat map M. For transformations with local spatial context, invoke BatchedIncrementalInference. For layer that precede 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.

4 APPROXIMATE INFERENCE OPTIMIZATIONS

Since incremental inference is *exact*, i.e., it yields the same heat map as full inference, it does not exploit a capability of human perception: tolerance of some degradation in visual quality. Thus, we now build upon our IVM framework to create two novel heuristic approximate inference optimizations that trade off the heat map's quality in a user-tunable manner

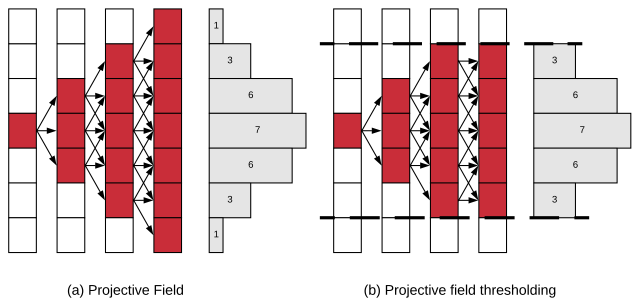


Figure 6: (a) Projective field growth for 1-D Convolution (filter size 2, stride 1). (b) Projective field *thresholding*; $\tau = 5/7$.

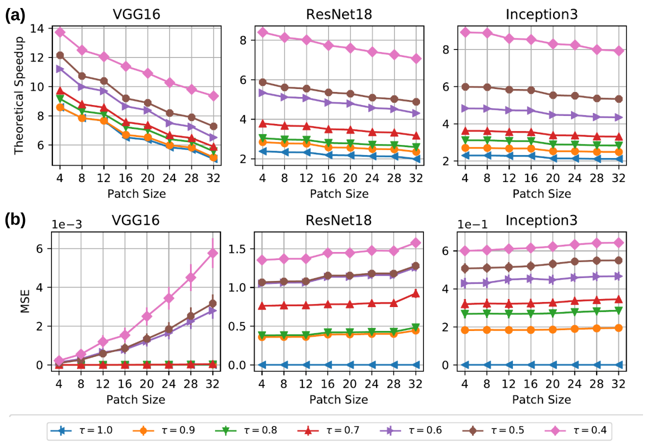


Figure 7: (a) Theoretical speedups with projective field thresholding. (b) Mean Square Error between exact and approximate output of final Convolution/Pooling layers.

to accelerate OBE further. We first explain the techniques and then explain how to tune them.

4.1 Projective Field Thresholding

The *projective field* of a CNN neuron is the slice of the output tensor that is connected to it [22]. It is a term from neuroscience to describe the effects of a retinal cell on the output of the eye's neuronal circuitry [23]. This notion sheds light on the *growth of the size* of the update patches through the layers of a CNN. The 3 kinds of layers (Section 2.2) affect the projective field size growth differently. Transformations at the granularity of individual elements do not alter the projective field size. Global context transformations increase it to the whole output. But local spatial context transformations, which are the most crucial, increase it *gradually* at a rate determined by the filter kernel's size and stride: additively in the size and multiplicatively in the stride. The growth of the projective field size implies the amount of FLOPs saved by IVM decreases as we go to the higher layers of a CNN. Eventually, the output update patch becomes as large as the output tensor. This growth is illustrated by Figure 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 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 do not consider the weights on these "paths," which is dependent on the given trained CNN. Since the weights can be arbitrary, a tight formal analysis is unwieldy. But under some assumptions on the weights values (similar to the assumptions in [24] for understanding the "receptive field" in CNNs), we show in the appendix that this distribution does indeed converge to a Gaussian. Thus, while this idea is a heuristic, it is grounded in a common behavior of real CNNs. 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):

$$W^{\mathcal{O}}_{\mathcal{P}:l} = \mathtt{min}\Big(\lceil (W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W^{\mathcal{O}}_{\mathcal{P}:l}\Big) \tag{21}$$

$$\text{If } W^{\mathcal{O}}_{\mathcal{P}:l} > \mathtt{round}(\tau \times W^{\mathcal{O}}_{:l}) : \tag{22}$$

$$W^{\mathcal{O}}_{\mathcal{P}:l} = \mathtt{round}(\tau \times W^{\mathcal{O}}_{:l}) \tag{23}$$

$$W^{\mathcal{I}}_{\mathcal{P}_{new}:l} = W^{\mathcal{O}}_{\mathcal{P}:l} \times S_{x:l} - W_{\mathcal{K}:l} + 1 \tag{24}$$

$$x^{\mathcal{I}}_{\mathcal{P}:l} \mathrel{+}= (W^{\mathcal{I}}_{\mathcal{P}:l} - W^{\mathcal{I}}_{\mathcal{P}_{new}:l})/2 \tag{25}$$

$$W^{\mathcal{I}}_{\mathcal{P}:l} = W^{\mathcal{I}}_{\mathcal{P}_{new}:l} \tag{26}$$

$$x^{\mathcal{O}}_{\mathcal{P}:l} = \mathtt{max}\Big(\lceil (P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\Big) \tag{27}$$

Equation (21) calculates the width assuming no thresholding. But if the output width exceeds the threshold, it is reduced as per Equation (23). Equation (24) calculates the input width that would produce an output of width $W^{O}_{\mathcal{P}:l}$; we can think of this as making $W^{\mathcal{I}}_{\mathcal{P}:l}$ the subject of Equation (21). If the new input width is smaller than the original input width, the starting x coordinate should be updated as per Equation (25) s.t. the new coordinates correspond to a "center crop" compared to the original. Equation (26) sets the input width to the newly calculated input width. Equation (27) calculates the x coordinate of the output update patch.

Theoretical Speedups. We modify our "static analysis" framework to determine the theoretical speedup of incremental inference (Section 3) to also include this optimization using the above formulas. Consider a square occlusion patch placed on the center of the input image. Figure 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 sample (n=30) of real-world images. 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 $r_{drill-down}$,

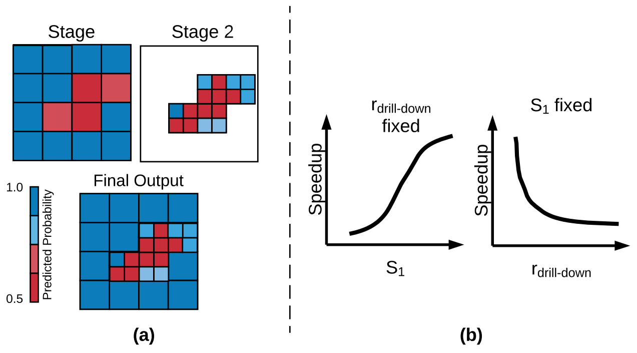


Figure 8: (a) Schematic illustration of the adaptive drill-down idea. (b) Conceptual depiction of the effects of S_1 and $r_{drill-down}$ on the theoretical speedup..

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 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 $\frac{H_{\mathcal{I}_{img}}}{S} \cdot \frac{W_{\mathcal{I}_{img}}}{S}$. Thus, the theoretical speedup is given by the following equation. Figure 8 (b) illustrates how this ratio varies with S_1 and $r_{drill-down}$.

$$\text{speedup} = \frac{S_1^2}{S_2^2 + r_{drill-down} \cdot S_1^2} \tag{28}$$

4.3 Automated Parameter Tuning

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

Tuning projective field thresholding. As Section 4.1 explained, τ controls the visual quality of the heat map. There is a spectrum of visual quality degradation: imperceptible changes to major structural changes. But mapping τ to visual quality directly is likely to be unintuitive for users. Thus, to measure visual quality more intuitively, we adopt a cognitive science-inspired metric called Structural Similarity (SSIM) Index, which is widely used to quantify human-perceptible differences between two images [25]. In our case, the two "images" are the original and approximate heat maps. SSIM is a number in $[-1, 1]$, with 1 meaning a perfect match. SSIM values in the $[0.90, 0.95]$ range are considered almost imperceptible distortions in many practical multimedia applications such as image compression and video encoding [25].

Our tuning process for τ has an offline "training" phase and an online usage phase. The offline phase relies on a set of sample images (default 30) from the same application domain. We compute SSIM for the approximate and exact heat maps for all sample images for a few τ values (default $1.0, 0.9, 0.8, \ldots, 0.4$). We then learn a second-degree polynomial curve for SSIM as a function of τ with these data points. Figure 9 (a) illustrates this phase and the fit SSIM-τ curves for 3 different CNNs using sample images from an OCT dataset (Section 5). In the online phase, when OBE is needed on a given image, we expect the user to provide a *target SSIM* for the quality–runtime trade-off they want (1 yields the exact heat map). We can then use our learned curve to map this target SSIM to the lowest τ. Figure 9 (b) shows the CDFs of differences between the target SSIM (0.9) and the actual SSIM yielded when using our auto-tuned τ on both the training set and a holdout test set (also 30 images). In 80% of the cases, the actual SSIM was *better* than the user-given target; never once did the actual SSIM go 0.1 below the target SSIM. This suggests that our auto-tuning method for τ works, is robust, and applicable to different CNNs.

Tuning adaptive drill-down. As Section 4.2 explained, the speedup offered by adaptive drill-down is controlled by two parameters: stage one stride S_1 and drill-down fraction $r_{drill-down}$. We expect the user to provide $r_{drill-down}$ (default 0.25), since it captures the user's intuition about how large or small the region of interest is likely to be in the images in their specific application domain and dataset. We also expect the user to provide a "target speedup" ratio (default 3) for using this optimization to capture their desired quality-runtime trade-off. Higher the user's target speedup, the more we sacrifice the quality of the "non-interesting regions" ($1 - r_{drill-down}$ fraction of the heat map). Our

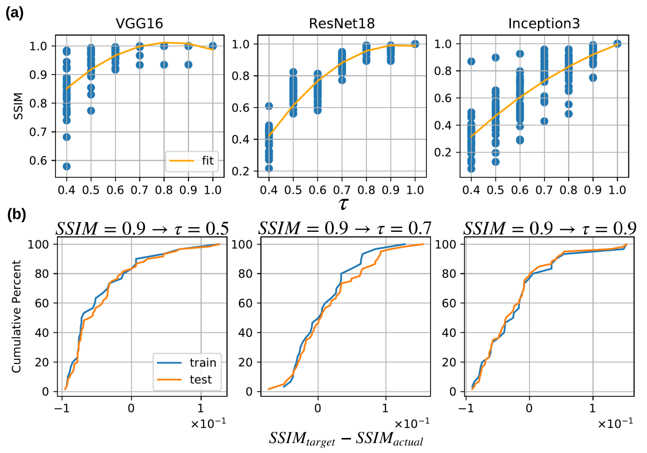


Figure 9: (a) Fitting a second-order curve for SSM against τ on a sample of the OCT dataset. (b) CDFs of deviation of actual SSIM from the target SSIM (0.9) with our auto-tuned τ, which turned out to be 0.5, 0.7, and 0.9 for VGG-16, ResNet-18, and Inception-V3, respectively.

automated tuning process sets S_1 using these two user-given settings. Unlike the tuning of τ, setting S_1 is more direct, since this optimization relies on the number of re-inference queries, not SSIM. Let *target* denote the target speedup; the original occlusion patch stride is S_2. Equation 29 shows how we calculate S_1. Since S_1 cannot be larger than the image width W_{img} (similarly H_{img}) and due to the constraint of $(1 - r_{drilldown} \cdot \texttt{speedup})$ being positive, we also have an upper bound on the possible speedups as per Equation 30.

$$S_1 = \sqrt{\frac{target}{1 - r_{drill-down} \cdot target}} \cdot S_2 \tag{29}$$

$$speedup < \texttt{min}\left(\frac{W_{img}^2}{S_2^2 + r_{drill-down} \cdot W_{img}^2}, \frac{1}{r_{drill-down}}\right) \tag{30}$$

5 EXPERIMENTAL EVALUATION

We integrated all of our optimization techniques with the popular deep learning environment PyTorch to create a tool we call Krypton. We now evaluate the speedups yielded by Krypton for OBE for several deep CNNs on real-world image datasets. We then drill into the contributions of each of our optimization techniques.

Datasets. We use three diverse real-world image datasets: *OCT*, *Chest X-Ray*, and a sample from *ImageNet*. *OCT* has about 84,000 optical coherence tomography retinal images with four classes: CNV, DME, DRUSEN, and NORMAL. CNV (choroidal neovascularization), DME (diabetic macular edema), and DRUSEN are three varieties of diabetic retinopathy. NORMAL corresponds to healthy retinal images. *Chest X-Ray* has about 6,000 X-ray images with three classes: VIRAL, BACTERIAL, and NORMAL. VIRAL and BACTERIAL are two varieties of pneumonia. NORMAL corresponds to healthy people. Both *OCT* and *Chest X-Ray* are from a recent scientific study that applied deep CNNs to radiology images to detect the respective diseases [2]. *ImageNet* is a benchmark dataset in computer vision [26]; our sample has 1,000 images with 200 classes.

Workloads. We use 3 diverse ImageNet-trained deep CNNs: VGG16 [27], ResNet18 [28], and Inception3 [29], obtained from [30]. They complement each other in terms of model size, architectural complexity, computational cost, and our predicted theoretical speedups (Figure 3 in Section 3). For *OCT* and *Chest X-Ray*, the 3 CNNs were fine-tuned by re-training their final Fully-Connected layers as per standard practice. The details of fine-tuning are not relevant for the rest of our discussion; so, we present further details in the appendix. The OBE heat maps are plotted using Python Matplotlib's `imshow` method using the `jet_r` color scheme; we set the maximum threshold to $\texttt{min}(1, 1.25p)$ and minimum to $0.75p$, where p is predicted class probability on a given image. All images are resized to the input size required by the CNNs (224×224 for VGG16 and ResNet18; 299×299 for Inception3); no additional pre-processing was done. The GPU-based experiments used a batch size of 128; for CPUs, the batch size was 16. All CPU-based experiments were executed with a thread parallelism of 8. All of our datasets, experimental scripts, and the Krypton codebase will be made publicly available on our project webpage.

Experimental Setup. We use a machine with 32 GB RAM, Intel i7-6700 3.40GHz CPU, and NVIDIA Titan X (Pascal) GPU with 12 GB memory. The machine runs Ubuntu 16.04 with PyTorch version 0.4.0, CUDA version 9.0, and cuDNN version 7.1.2. All reported runtimes are the average of 3 runs, with 95% confidence intervals shown.

5.1 End-to- End Runtimes

We focus on the most common OBE scenario of producing the whole heat map, i.e., G is automatically created ("non-interactive" mode). We use an occlusion patch of size 16 and stride 4. We compare two variants of Krypton: Krypton-Exact uses only incremental inference (Section 3), while Krypton-Approximate uses our approximate inference optimizations too (Section 4). The main baseline is *Naive*, the current dominant practice of performing full inference for OBE with just naive batching of images. We have another baseline for the GPU environment: *Naive Inc. Inference-Exact*, which is a direct implementation of Algorithm 1 in PyTorch/Python without using our GPU-optimized CUDA kernel, which Krypton uses (Section 3.4). Note that *Naive Inc. Inference-Exact* is not applicable to the CPU environment.

We set the user-given tuning parameters for adaptive drill-down based on the semantics of each dataset's prediction task (Section 4.3). For *OCT*, since the region of interest is likely to be small, we set $r_{drill-down} = 0.1$ and $target = 5$. For *Chest X-Ray*, the region of interest can be large; so, we set $r_{drill-down} = 0.4$ and $target = 2$. For *ImageNet*, which falls in between, we use the Krypton default values of $r_{drill-down} = 0.25$ and $target = 3$. For all experiments τ is auto-tuned with a target SSIM of 0.9 (Section 4.3). Figure 10 presents the results. Visual examples of images and the heat maps produced are presented in the appendix.

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

The speedups are lower with ResNet18 and Inception3 than VGG16 due to their architectural properties (kernel filter dimensions, stride, etc.) that make the projective field grow faster. Moreover, Inception3 has a complex DAG architecture with more branches and depth-wise concatenation, which limits GPU throughput for incremental inference. In fact, Krypton-Exact on GPU shows a minor slow-down (0.7X) with Inception3. But Krypton-Approximate still offers speedups on GPU with Inception3 (up to 4.5X). We also see that ResNet18 and VGG16 almost near their theoretical speedups (Figure 3) but Inception3 does not. Note that the theoretical speedup definition only counts FLOPs and does not account for memory stalls.

Finally, the speedups of Krypton are higher on CPU than GPU. This is because CPU does not suffer much due to memory stalls during incremental inference. But the *absolute* runtimes are almost an order of magnitude higher on CPUs than GPUs, which is to be expected. Overall, Krypton improves the efficiency of OBE significantly for multiple datasets and deep CNNs. We ran an additional experiment on the "interactive" mode by reducing $|G|$. The speedups go down as $|G|$ goes down, which is expected because the benefits of amortization are reduced. Due to space constraints, these additional results are presented in the appendix.

5.2 Ablation Study

We now analyze the contributions of our 3 optimizations individually. We compare the speedups of Krypton over *Naive* (batched inference) on both CPU and GPU, termed

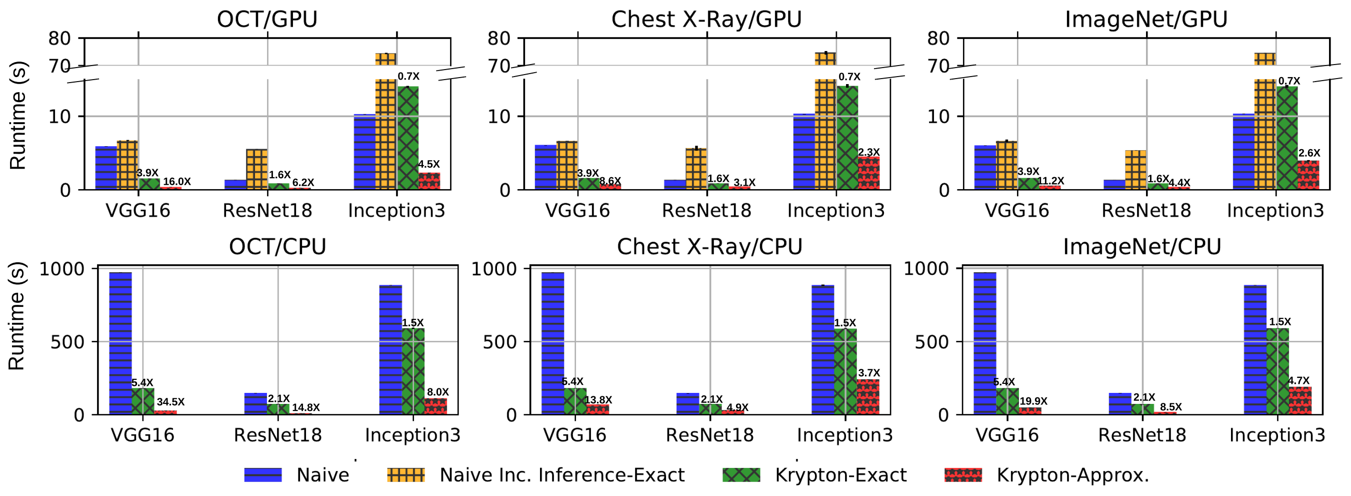


Figure 10: End-to-end runtimes of KRYPTON and baselines on all 3 datasets, 3 CNNs, and both GPU and CPU.

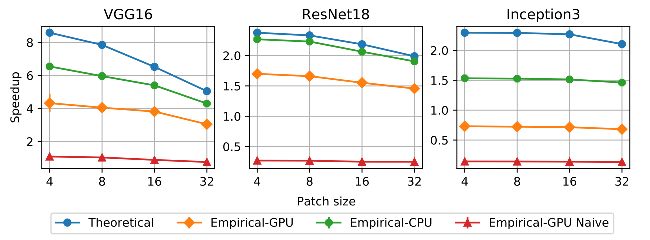


Figure 11: Speedups with only the incremental inference optimization (occlusion patch stride $S = 4$).

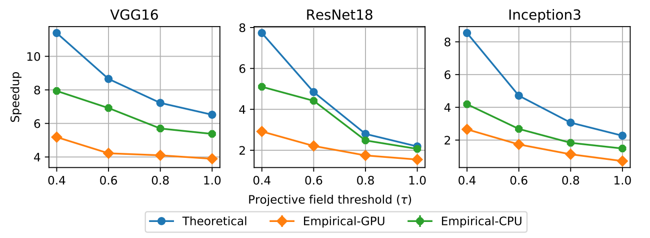


Figure 12: Speedups with incremental inference combined with only projective field thresholding.

Empirical-CPU and Empirical-GPU respectively, against the theoretical speedups (explained in Sections 3 and 4).

Only Incremental Inference. We vary the patch size and set the stride to 4. Figure 11 shows the results. As expected, the speedups go down as the patch size increases. Empirical-GPU Naive yields no speedups because it does not use our GPU-optimized kernel, while Empirical-GPU does. But Empirical-CPU is closer to theoretical speedup and almost matches it on ResNet18. Thus, there is still some room for improvement to improve the efficiency of incremental inference in both environments.

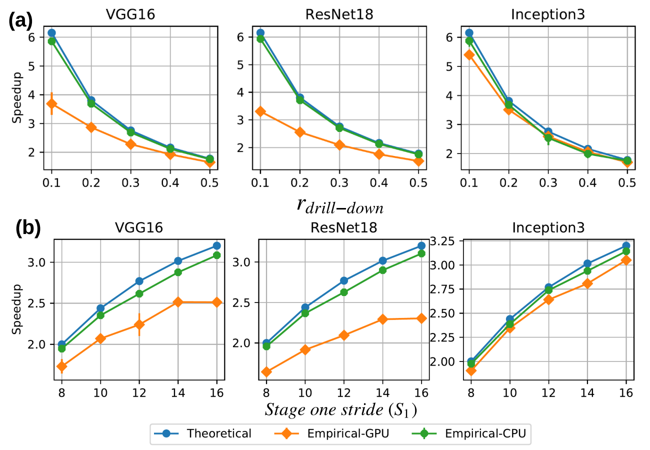


Figure 13: Speedups with incremental inference combined with adaptive drill-down. For (a), we set $S_1 = 16$. For (b), we set $r_{drill\_down} = 0.25$).

Projective Field Thresholding. We vary τ from 1.0 (no approximation) to 0.4. Adaptive drill-down is disabled but note that this optimization builds on top of our incremental inference. The occlusion patch size is 16 and stride is 4. Figure 12 shows the results. The speedups go up steadily as τ drops for all 3 CNNs. Once again, Empirical-CPU nears the theoretical speedups on ResNet18, but the gap between Empirical-GPU and Empirical-CPU remains due to the disproportionate impact of memory stalls on GPU. Overall, this approximation offers some speedups in both environments, but has a higher impact on CPU than GPU.

Adaptive Drill-Down. Finally we study the effects of adaptive drill-down (again, on top of incremental inference) and disable projective field thresholding. The occlusion patch size is 16. Stage two stride is $S_2 = 4$. First, we vary $r_{drill-down}$, while fixing stage one stride ($S_1 = 16$). Figure 13 (a) shows the results. Next, we vary S_1, while fixing $r_{drill-down} = 0.25$. Figure 13 (b) shows the results. As expected, the speedups go up as $r_{drill-down}$ goes down or S_1 goes up, since fewer re-inference queries arise in both cases. Empirical-CPU almost matches the theoretical speedups across the board; in fact, even Empirical-GPU almost matches theoretical speedups on Inception3. Empirical-GPU flattens out at high S_1, since the number of re-inference queries drops, thus resulting in diminishing returns for the benefits of batched execution on GPU. Overall, this optimization has a major impact on speeding up OBE for all CNNs in both environments.

Summary of Experiments. Overall, our empirical evaluation shows that Krypton is able to substantially accelerate the OBE workload for explaining CNN predictions, up to 16X speedups on GPU and 34.5X speedups on CPU. The speedups of all 3 of our optimizations depend on the CNN's architectural properties. The speedups of our approximate inference optimizations also depend on the dataset due to their tunable parameters, which Krypton can tune automatically. Finally, the speedups of Krypton are higher on CPU than GPU but the absolute runtimes are much lower on GPU. Overall, all of our optimizations in Krypton help reduce waiting times for users and can save resource costs, since they only use existing compute resources without forcing users to pay for more resources (say, renting more GPUs in the cloud).

6 OTHER RELATED WORK

Query Optimization. Our work is inspired by the long line of work on incremental view maintenance (IVM) in databases [31–33], but this is the first work to use the IVM lens for the occlusion-based CNN explanation workload. Our novel algebraic IVM framework for CNN inference is closely tied to the dataflow of CNN layers, which transform tensors in non-trivial ways. Closely related to our work is the IVM framework for linear algebra in [34]. They focus on bulk matrix operators and incremental addition of rows to the data matrix. We do not deal with bulk matrix operators or additions of rows, but more fine-grained CNN inference computations and in-place updates to image pixels due to occlusions. Also closely related is the IVM framework for distributed multi-dimensional array database queries in [35]. An interesting connection is that CNN layers with local spatial context (Section 2.2) can be viewed as a variant of spatial array join-aggregate queries. But our framework enables end-to-end incremental inference for entire CNNs, not just one-off spatial queries involving data materialization and loading. Our focus is also on popular deep learning environments, not array databases. Finally, we also go beyond algebraic IVM ideas to further exploit CNN-specific semantics and human perception properties in our problem setting.

Our work is also inspired by the multi-query optimization (MQO) literature [19, 36]. But we focus on CNN inference, not relational queries. To the best of our knowledge, ours is the first work to present an MQO-style optimization combined with IVM for optimizing CNN inference for occlusion-based explanations. Our approximate inference optimizations are inspired by approximate query processing (AQP) techniques [37, 38]. But unlike statistical approximations of aggregations over relations, our techniques are novel CNN-specific and human perception-oriented heuristic approximations tailored to reducing the computational cost of CNN inference for the OBE workload.

Multimedia DBMSs. There is much work in the database and multimedia literatures on multimedia DBMSs [39, 40]. The main focus of such work is on retrieval, including content-based image retrieval (CBIR) and video retrieval using similarity search or indexes. Our work is orthogonal to this body of work since we focus on accelerating CNN explanations, not multimedia retrieval queries. CBinfer is a video analytics tool for change-based approximate CNN inference that can accelerate real-time object recognition in video [18]. Our work also deals with incremental and approximate CNN inference, but our ideas exploit the specific properties of the OBE workload, not general object recognition in video. NoScope is a system to accelerate object detection in video streams using model cascades [41]. Our focus is on accelerating the OBE workload, not object recognition or video. Overall, both these tools are orthogonal to our focus.

7 CONCLUSIONS AND FUTURE WORK

Deep CNNs are gaining widespread adoption for image prediction tasks but their internal workings are unintuitive for most users. Thus, occlusion-based explanations (OBE) have become a popular mechanism for non-technical users to understand CNN predictions. But OBE is highly compute-intensive due to the large number of CNN inference requests produced. In this work, we formalize OBE from a data management standpoint and introduce several novel database-inspired optimization techniques to speed up OBE. Our techniques span exact incremental inference and multi-query optimization for CNN inference, as well as CNN-specific and human perception-aware approximate inference. Overall, our ideas yield even over an order of magnitude speedups for OBE in both CPU and GPU environments. As for future work, we plan to apply our ideas to other complex visual recognition tasks and video analytics. It is also interesting future work to generalize our framework to other CNN explanation mechanisms and data types.

REFERENCES

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

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

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

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

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

[6] Yilun Wang and Michal Kosinski. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. *Journal of personality and social psychology*, 114(2):246, 2018.

[7] 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.

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

[9] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

[10] 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.

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

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

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

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

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

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

[17] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

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

[41] 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.

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

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

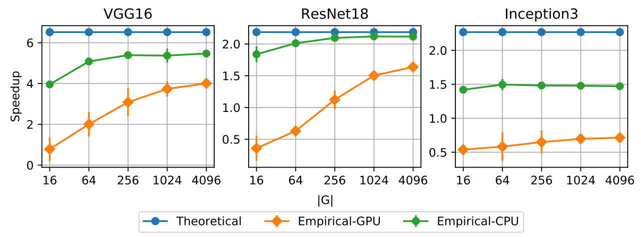
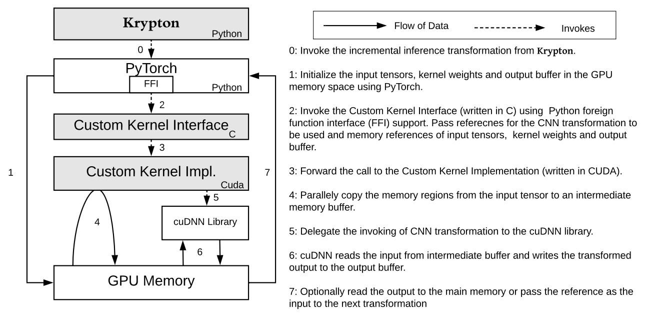


Figure 14: Interactive mode execution of incremental inference with Gs of different sizes



0: Invoke the incremental inference transformation from **Krypton**.

1: Initialize the input tensors, kernel weights and output buffer in the GPU memory space using PyTorch.

2: Invoke the Custom Kernel Interface (written in C) using Python foreign function interface (FFI) support. Pass referecnes for the CNN transformation to be used and memory references of input tensors, kernel weights and output buffer.

3: Forward the call to the Custom Kernel Implementation (written in CUDA).

4: Parallely copy the memory regions from the input tensor to an intermediate memory buffer.

5: Delegate the invoking of CNN transformation to the cuDNN library.

6: cuDNN reads the input from intermediate buffer and writes the transformed output to the output buffer.

7: Optionally read the output to the main memory or pass the reference as the input to the next transformation

Figure 15: Custom GPU Kernel integration architecture

[44] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *arXiv preprint arXiv:1706.07269*, 2017.

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

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 15

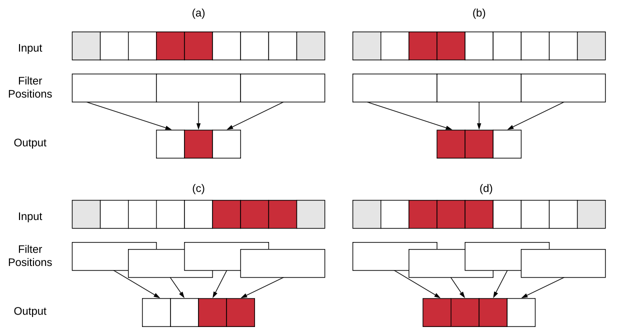


Figure 16: Illustration of special cases for which actual output size will be smaller than the value given by Equation (13). (a) and (b) show cases where the filter stride is equal to the filter size. (c) and (d) show situations where the position of the modified patch affecting the size of the output patch.

C SPECIAL CASES FOR INCREMENTAL INFERENCE

There are special cases under which the output patch size can be smaller than the values calculated in Section 3.2. Consider the simplified 1-D case shown in Figure 16 (a), where the filter stride[1] (3) is the same as the filter size (3). In this case, the size of the output update patch is one less than the value calculated by Equation (14). But this is not the case for the situation shown Figure 16 (b), which has the same input patch size but placed at a different location. Another case arises when the modified patch is placed at the edge of the input, as shown in Figure 16 (c). In this case, it is impossible for the filter to move freely through all positions, since it hits the input boundary. However, it is not the case for the modified patch shown in Figure 16 (d). In KRYPTON, we do not treat these cases separately but rather use the values calculated by Equation (14) for the width dimension (similarly for the height dimension), since they act as an upper bound. In the case of a smaller output patch, KRYPTON reads and updates a slightly bigger patch to preserve uniformity. This also requires updating the starting coordinates of the patch, as shown in Equation (31). This sort of uniform treatment is required for performing batched inference operations, which gives significant speedups compared to per-image inference.

$$\begin{aligned} &\text{If } x_{\mathcal{P}}^{O} + W_{\mathcal{P}}^{O} > W_O : \\ &x_{\mathcal{P}}^{O} = W_O - W_{\mathcal{P}}^{O}; x_{\mathcal{P}}^{\mathcal{I}} = W_{\mathcal{I}} - W_{\mathcal{P}}^{\mathcal{I}}; x_{\mathcal{P}}^{\mathcal{R}} = W_{\mathcal{I}} - W_{\mathcal{P}}^{\mathcal{R}} \end{aligned} \tag{31}$$

D EFFECTIVE PROJECTIVE FIELD SIZE

We formalize the effective projective field growth for the one dimensional scenario with n convolution layers (assuming certain conditions). This proof is motivated by a similar proof in [24] which characterizes the effective growth rate of the receptive field in a CNN.

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

The input is $u(t)$ where

$$u(t) = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \tag{32}$$

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

$$v(t) = \sum_{m=0}^{k-1} w(m)\delta(t - m) \tag{33}$$

where $w(m)$ is the weight for the m^{th} 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 n^{th} layer $o(t)$ is simply $o = u * v * \ldots * v$, convolving u with n such vs. To compute the convolution, we can use the Discrete Time Fourier Transform to convert the signals into the Fourier domain, and obtain

$$\begin{aligned} U(\omega) &= \sum_{t=-\infty}^{\infty} u(t)e^{-j\omega t} = 1,\ V(\omega) \\ &= \sum_{t=-\infty}^{\infty} v(t)e^{-j\omega t} = \sum_{m=0}^{k-1} w(m)e^{-j\omega t} \end{aligned} \tag{34}$$

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

$$\begin{aligned} \mathcal{F}(o) &= \mathcal{F}(u * v * \ldots * v)(\omega) = U(\omega).V(\omega)^n \\ &= \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t}\right)^n \end{aligned} \tag{35}$$

With inverse Fourier transform

$$o(t) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t}\right)^n e^{j\omega t} d\omega \tag{36}$$

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 [42]. It can be shown that if $\sum_m w(m) = 1$ and $w(m) \geq 0\ \forall\ m$, then

$$\begin{aligned} & o(t) = p(S_n = t) \\ & \text{where } S_n = \sum_{i=1}^{n} X_i \text{ and } p(X_i = m) = w(m) \end{aligned} \tag{37}$$

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

$$\mathbb{E}[S_n] = n\sum_{m=0}^{k-1} mw(m) \tag{38}$$

$$Var[S_n] = n\left(\sum_{m=0}^{k-1} m^2 w(m) - \left(\sum_{m=0}^{k-1} mw(m)\right)^2\right) \tag{39}$$

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

$$\sqrt{Var[S_n]} = \sqrt{nVar[X_i]} = O(\sqrt{n}) \tag{40}$$

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*, the 3 ImageNet-trained CNNs are fine-tuned by retraining the final Fully-Connected 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 [43] 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.

	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.

F DEEP CNN EXPLAINABILITY

Various approaches used to explain CNN predictions can be broadly divided into two categories, gradient-based and perturbation based approaches. Gradient-based approaches generate a sensitivity heat map by computing the partial derivatives of model output with respect to every input pixel via backpropagation. In perturbation based approaches the output of the model is observed by modifying regions on the input image and thereby identify the sensitive regions. Despite being time-consuming, in most real world use cases such as in medical imaging, practitioners tend to use occlusion experiments, a perturbation based approach, as the

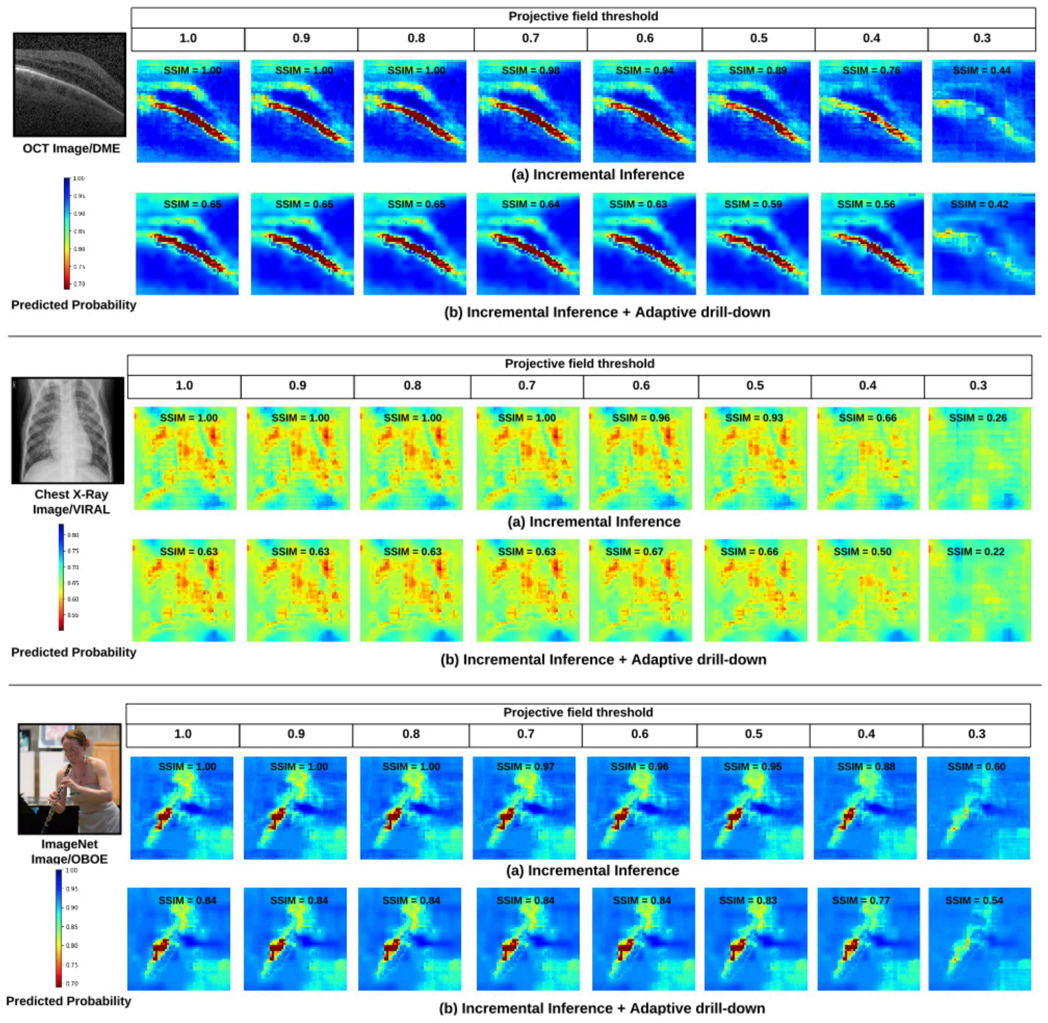


Figure 17: Occlusion heat maps for sample images (CNN model = VGG16, occlusion patch size = 16, patch color = black, occlusion patch stride (S or S_2) = 4. For *OCT* $r_{drill\_down} = 0.1$ and $target = 5$. For *Chest X-Ray* $r_{drill\_down} = 0.4$ and $target = 2$. For *ImageNet* $r_{drill\_down} =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

preferred approach for explanations as they produce high quality fine grained sensitivity heat maps using a process which is very intuitive to the human observer [10, 12, 44].

G VISUAL EXAMPLES

Figure 17 presents occlusion heat maps for a sample image from each dataset with (a) *incremental inference* for different *projective field threshold* values and (b) *incremental inference* with *adaptive drill-down* for different *projective field threshold* values. The predicted class label for *OCT*, *Chest X-Ray*, and *ImageNet* are DME, VIRAL, and OBOE respectively.