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

ABSTRACT

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

1 INTRODUCTION

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

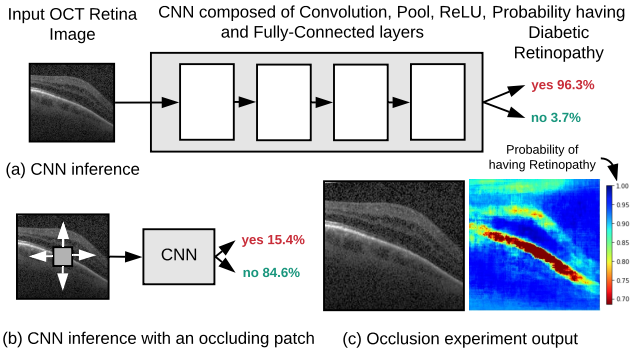


Figure 1: (a) Using CNNs for predicting Diabetic Retinopathy from OCT images. (b) Occluding parts of the OCT image changes the predicted probability for the disease. (c) By changing the position of the occlusion patch a sensitivity heat map is produced.

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

In order to improve the explainability of Deep CNN predictions several approaches have been proposed. One of the most widely used approach in image recognition tasks is occlusion experiments [14]. In occlusion experiments, as shown in Figure 1 (b), a square patch usually of black or gray color is used to occlude parts of the image and record the change in the predicted label probability. By changing the position of the occlusion patch, usually by a small fixed number of pixels called stride, a sensitivity heat map for the predicted label can be generated (similar to one shown in Figure 1 (c)). If the occlusion experiment is performed in interactive mode, the human operator has the option of picking the occlusion patch positions by marking a region on a visual interface. For example, if the scenario shown in

Figure 1 is performed in interactive mode, the human operator who understands OCT images will start evaluating the image from the central region where she expects the pathological region to most likely to be. In the non-interactive mode, which is also the most common mode of performing occlusion experiments due to the high runtimes which are not amenable for interactive performance, the heat map values are evaluated for all possible occlusion patch positions. Using this heat map, the regions in the image which are highly sensitive (or highly contributing) to the predicted class can be identified (corresponds to red color regions in the sensitivity heat map shown in Figure 1 (c)). This localization of highly sensitive regions then enables the practitioners to get an idea of the the prediction process of the Deep CNN.

However, occlusion experiments are highly compute intensive and time consuming as each occlusion position has to be treated as a new image and requires a separate CNN inference. In this work, our goal is to apply database inspired optimizations to the occlusion based explainability workload to reduce both the computational cost and the runtime. This will also make occlusion experiments more amenable for interactive diagnosis of CNN predictions. Our main motivation is based on the observation that when performing CNN inference corresponding to each individual occlusion patch position, there is a significant portion of redundant computations which can be avoided. To avoid redundant computations we introduce the notion of *incremental inference* of Deep CNN which is inspired by the incremental view maintenance technique studied in the context of relational databases.

Due to the overlapping nature of how the Convolution kernel operates (details to follow in Section 2), the size of the modified patch will start growing as it progresses through more layers in a CNN and the amount of redundant computations will reduce. However, at deeper layers, the effect over the patch coordinates which are radially further away from the center of the occlusion patch position will be diminishing. Our second optimization is based on this observation where we apply a form of *approximate inference* which applies a threshold to limit the growth of the updating patch. By applying propagation thresholds, a significant amount of computation redundancy can be retained. We refer to this optimization as *projective field thresholding*.

The third optimization is also a form of *approximate inference* which is applicable only in the context of non-interactive mode. In most occlusion experiment use cases, such as in medical imaging, the object or pathological region of interest is contained in a relatively small region of the image. In such situations, it is unnecessary to inspect the original image at the same high resolution of striding the occluding patch few pixels at a time, at all possible occlusion patch positions. In this approach first, a low-resolution heat map is generated using a larger stride value with a relatively low computational cost. Only the interesting regions will be then inspected further with a smaller stride to produce a higher resolution output. In the interactive mode, as the human operator will be actively picking a set of occlusion patch positions for the system to evaluate this optimization will not be applicable. We refer to this optimization as *adaptive drill-down*.

Unlike the *incremental inference* approach which is exact, *projective field thresholding* and *adaptive drill-down* are approximate approaches. They essentially trade-off accuracy of the generated sensitivity heat map compared to the original, in favor of faster execution. These changes in accuracy in the generated heat map can be visible all the way from quality differences which are almost indistinguishable to the human eye to drastic structural differences, depending on the level of approximation. This opens up an interesting trade-off space of quality/accuracy versus runtime. Krypton provides user configurable tuning parameters for easily picking an operational point on this quality-runtime trade-off space.

Finally, we have implemented Krypton on top of PyTorch Deep learning toolkit by adding custom implementations for incremental and approximate inference operations. It currently supports VGG16, ResNet18, and InceptionV3 both on CPU and GPU environments, which are three widely used Deep CNN architectures. We evaluate our system on three real-world datasets, 1) retinal optical coherence tomography dataset (OCT), 2) chest X-Ray, and 3) more generic ImageNet dataset, and show that Krypton can result in speedups over 10X. While we have implemented Krypton on top of PyTorch toolkit, our work is largely orthogonal to the choice of the Deep learning toolkit; one could replace PyTorch with TensorFlow, Caffe2, CNTK, MXNet, or implement from scratch using C/CUDA and still benefit from our optimizations. Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to study

Outline. The rest of this paper is organized as follows.

2 PRELIMINARIES AND OVERVIEW

In this section, we first formally state the problem and explain our assumptions. Then we formalize the internals of critical layers in a deep CNN for the purpose of proposing our *incremental inference* approach in Section 4.

Table 1: Symbols used in the Section 3

Symbol	Meaning
f	Fine-tuned CNN which takes in an input image and outputs a probability distribution over the class labels
$T_{:l}$	Tensor transformation function used in the l^{th} layer of the CNN f
L	Class label predicted by f for the original image $\mathcal{I}_{:img}$
$\mathcal{P}$	Occluding patch in RGB format
$S_{\mathcal{P}}$	Occluding patch striding amount
G	Set of occluding patch superimposition positions on $\mathcal{I}_{:img}$ in (x,y) format
M	Heat map produced by the occlusion experiment
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 off at (x, y) position
$\mathcal{I}_{:l}(\mathcal{I}_{:img})$	Input tensor of the l^{th} layer (Input Image)
$\mathcal{O}_{:l}$	Output tensor of the l^{th} layer
$C_{I:l}, H_{I:l}, W_{I:l}$	Depth, height, and width of l^th layer Input
$C_{O:l}, H_{O:l}, W_{O:l}$	Depth, height, and width of l^{th} layer Output
$\mathcal{K}_{conv:l}$	Convolution filter kernels for the l^{th} layer
$\mathcal{B}_{conv:l}$	Convolution bias value vector for the l^{th} layer
$\mathcal{K}_{pool:l}$	Pooling filter kernel for the l^{th} layer
$H_{\mathcal{K}:l}, W_{\mathcal{K}:l}$	Height and width of the filter kernel for the l^{th} layer
$S_{:l} \equiv (S_{x:l}, S_{y:l})$	Filter kernel patch striding amount for the l^{th} layer ($S_{x:l}$ and $S_{y:l}$ corresponds to width and height dimensions)
$P_{:l} \equiv (P_{x:l}, P_{y:l})$	Padding amount for the l^{th} layer ($P_{x:l}$ and $P_{y:l}$ corresponds to padding along width and height dimensions)

2.1 Problem Statement and Assumptions

We are given a CNN f which consists of a sequence or a DAG of tensor transformation functions $T_{:l}$s, an image $\mathcal{I}_{:img}$ on which the occlusion experiment needs to be run, the predicted class label L for $\mathcal{I}_{:img}$, an occluding patch $\mathcal{P}$ in RGB format, and occluding patch striding amount $S_{\mathcal{P}}$. We are also given a set of interested occluding patch positions G, constructed either automatically or by the human with a visual interface interactively. The occlusion experiment workload is to generate a 2-D heat map M, where each value correspond to the coordinates in G contains the predicted probability for L by f for the occluded image $\mathcal{I}'_{x,y:img}$ or zero otherwise. More precisely, we can state the workload using the following set of logical statements:

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

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

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

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

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

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

Step (1), and (2) calculates the dimensions of the generated heat map M which is dependent on the dimensions of $\mathcal{I}_{:img}$, $\mathcal{P}$, and $S_{\mathcal{P}}$. 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 picking the predicted probability for the label L. Step (5) and (6) are run for all occluding patch position values in G. In the non-interactive mode G is initialized to $G = [0, H_M) \times [0, W_M)$, which corresponds to the set of all possible occlusion patch positions on $\mathcal{I}_{:img}$. In the interactive mode it is possible that human operator would provide multiple Gs, one after the other, for which the system has to evaluate iteratively.

The focus of this work is on the occlusion based deep CNN explainability approach [14], which is widely used by practitioners in several domains including healthcare, sociology, security, and agriculture [6–10]. However, we acknowledge that there are other methods for CNN explainability and a summary of those approaches is included in the Appendix. We assume that f is a CNN from a roster of well-known CNNs (currently, we support VGG 16-layer version, ResNet 18-layer version, and Inception version 3). We think this is a reasonable start, since most recent CNN-based image recognition applications use only such well-known CNNs from model zoos [15, 16]. Nevertheless, our work is orthogonal to the specifics of a particular CNN architecture, and our proposed techniques can be easily extended to any CNN architecture. We leave support for arbitrary CNNs to future work.

2.2 Deep CNN Internals

Deep CNNs are a type of neural networks specialized for image data. They are organized into multiple layers of various types including: *Convolution*, which use image filters from graphics, except with variable filter weights, to extract features; *Pooling*, which subsamples features in a spatial locality-aware way; *Batch-Normalization*, which normalizes the output of the layer; *Non-Linearity*, which applies a non-linear transformation (e.g., ReLU); *Fully-Connected* which is an ordered collection of perceptrons. Input and output of individual layers in a deep CNN, except for Fully-Connected ones, are arranged into 3-D tensors that have a

width, height, and depth. For example an RGB input image of 224×224 dimensions can be considered as an input tensor having a width and height of 224 and a depth of 3. Every non-Fully-Connected layer will take in an input tensor and transform it into another tensor. A Fully-Connected layer takes in a 1-D tensor or a flattened 3-D tensor as input and transforms it into another 1-D tensor. For our purpose, these transformations can be broadly divided into three categories based on their spatial locality:

- Transformations that operate at the granularity of a global context.
  - E.g. Fully-Connected
- Transformations that operate at the granularity of individual spatial locations.
  - E.g. ReLU, Batch Normalization
- Transformations that operate at the granularity of a local spatial context.
  - E.g. Convolution, Pooling

Transformations that operate at the granularity of a global context. These transformations operate on a global context and does not take into account the spatial information. Fully-Connected, which is the only global transformation layer in a CNN, takes in a 1-D tensor or a flattened 3-D tensor and performs a matrix-vector multiplication with a weight matrix and produces an output 1-D tensor. Since it performs one bulk transformation, there is almost no substantial opportunity for exploiting redundancies with incremental computations. The computational cost of a Full-Connected transformation is proportional to the product of the size of the input and output 1-D tensors. Fully-Connected layers are used as the last or last few layers in a CNN and only accounts for a small fraction of the total computational cost. Thus we ignore them henceforth.

Transformations that operate at the granularity of individual spatial locations. These transformations essentially perform a *map*(.) function on each individual element in the tensor (see Figure 2 (b)). Hence, the output will have the same dimensions as input. The computational cost incurred by these transformations is proportional to the volume of the input (or output). However, with incremental spatially localized updates in the input, such as placing an occlusion patch, only the updated region needs to be recalculated. Extending these transformations to become change aware is straightforward since they are element-wise operations. The computational cost of the change aware incremental transformation is proportional to the volume of only the modified region.

Transformations that operate at the granularity of a local spatial context. With incremental spatially localized updates in the input, transformations that operate at the granularity of a local spatial context also provide opportunities for exploiting redundancy and can be made change aware. However, with local context transformations, such as Convolution and Pooling, this extension is non-trivial due to the overlapping nature of the spatial contexts.

Each Convolution layer can have $C_{O:l}$ 3-D "filter kernels" organized into a 4-D array $\mathcal{K}_{conv:l}$, with each having a smaller spatial width $W_{\mathcal{K}:l}$ and height $H_{\mathcal{K}:l}$ compared to the width $W_{\mathcal{I}:l}$ and height $H_{\mathcal{I}:l}$ of the input tensor $\mathcal{I}_{:l}$, but has the same depth $C_{\mathcal{I}:l}$. During inference, c^{th} filter kernel is "strided" along the width and height dimensions of the input and a 2-D "activation map" $A_{:c} = (a_{y,x:c}) \in \mathbb{R}^{H_{O:l} \times W_{O:l}}$ is produced by calculating element-wise products between the kernel and the input and adding a bias value as per Equation (7). The computational cost of each of these individual element-wise products is proportional to the volume of the filter kernel. Finally, these 2-D activation maps are stacked together along the depth dimension to produce an output tensor $\mathcal{O}_{:l} \in \mathbb{R}^{C_{O:l} \times H_{O:l} \times W_{O:l}}$ as per Equation (8). A simplified representation of Convolution transformation is shown in Figure 2 (a).

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

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

Pooling can also be thought of as a Convolution operation with a fixed (i.e., not learned) 2-D filter kernel $\mathcal{K}_{pool:l}$. But unlike Convolution, Pooling operates independently on each depth slice of the input tensor. A Pooling layer takes a 3-D activation tensor $\mathcal{O}_l$ having a depth of $C_{\mathcal{I}:l}$, width of $W_{\mathcal{I}:l}$, and height of $H_{\mathcal{I}:l}$ as input and produces another 3-D activation tensor $\mathcal{O}_{:l}$ with the same depth of $C_{O:l} = C_{\mathcal{I}:l}$, width of $W_{O:l}$, and height of $H_{O:l}$ as the output. Pooling kernel is generally strided with more than one pixel at a time and hence $W_{O:l}$ and $H_{O:l}$ are generally smaller than $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$. A simplified representation of Pooling transformation is shown in Figure 2 (c).

It can be seen that Convolution and Pooling transformations can be cast into a form of applying a filter along the spatial dimensions of the 3-D input tensor. However, how each transformation operates along the depth dimension is different. For our purpose, since we are only interested in finding the spatial propagation of the patches in the input through the consecutive layers, both of these transformations can be treated similarly.

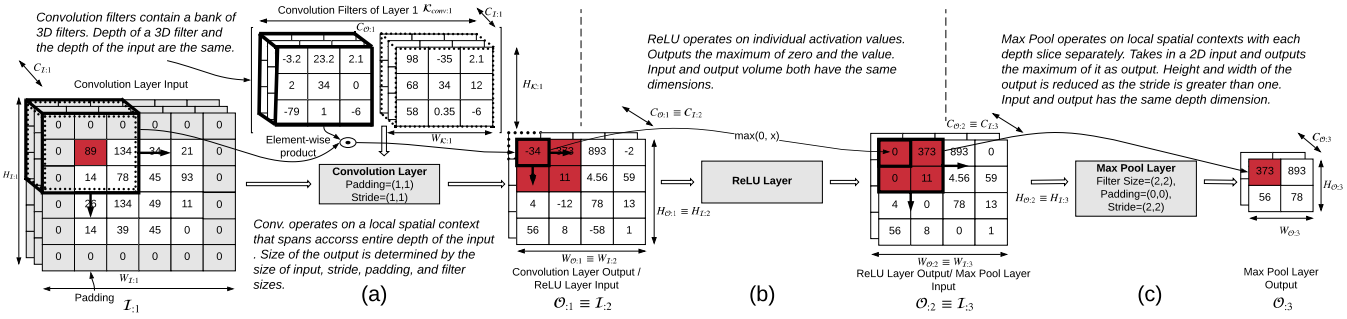


Figure 2: Simplified representation of selected layers of a Deep CNN. The values marked in red show how a small spatial update in the first input would propagate through subsequent layers. (a) Convolution layer (for simplicity addition of bias is not shown in the Convolution transformation), (b) ReLU layer, and (c) Pool layer. Notation is explained in Table 1.

Relationship between Input and Output Dimensions. The output tensor's dimensions $W_{O:l}$ and $H_{O:l}$ are determined by the dimensions of the input tensor $W_{I:l}$ and $H_{I:l}$, dimensions of the filter kernel $W_{\mathcal{K}:l}$ and $H_{\mathcal{K}:l}$ and two other parameters: **stride** $S_{:l}$ and **padding** $P_{:l}$. Stride is the number of pixel values used to move the filter kernel at a time when producing a 2-D activation map. It is possible to have two different values, with one for the width dimension $S_{x:l}$ and one for the height dimension $S_{y:l}$. Generally $S_{x:l} \leq W_{\mathcal{K}:l}$ and $S_{y:l} \leq H_{\mathcal{K}:l}$. In Figure 2, Convolution transformation uses a stride value of 1 and Pool transformation uses a stride value of 2 for both dimensions. Sometimes, in order to control the dimensions of the output tensor to be same as the input tensor, one needs to pad the input tensor with zeros around the spatial border. Padding $P_{:l}$ captures the number of zeros that needs to be added. Similar to the stride $S_{:l}$, it is possible to have two separate values for padding, with one for the width dimension $P_{x:l}$ and one for the height dimension $P_{y:l}$. In Figure 2, Convolution transformation pads the input with one line of zeros on both dimensions. With these parameters defined, the width (similarly height) of the output tensor can be defined as follows:

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

Computational Cost of Deep CNNs. Deep CNNs are highly compute-intensive. Of all the types of layers, Convolution layers almost always contribute to 90% (or more) of the computation. Hence, we can roughly estimate the computational cost of a Deep CNN by counting the number of fused multiply-add (FMA) floating point operations (FLOPs) required by Convolution layers for a single forward pass for inference.

For example, applying a Convolution filter having the dimensions of $(C_{I:l}, H_{\mathcal{K}:l}, W_{\mathcal{K}:l})$ to compute a single value in the output tensor will require $C_{I:l} \times H_{\mathcal{K}:l} \times W_{\mathcal{K}:l}$ FLOPs, each corresponding to a single element-wise multiplication. Thus, the total amount of computations $Q_{:l}$ required by that layer in order to produce an output tensor $O_{:l}$ with dimensions $C_{O:l} \times H_{O:l} \times lW_{O:l}$, and the total amount of computations Q required to process the entire set of convolution layers L in the CNN can be calculated as per Equation (10) and (11).

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

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

3 INCREMENTAL INFERENCE OPTIMIZATIONS

In this section, we first present the resulting theoretical speedups that can be achieved with incremental inference for popular deep CNN architectures. We then explain the mechanics of the *incremental inference* operator for a single layer. Next, we explain how the individual *incremental inference* operator can be used to propagate changes across multiple layers in a deep CNN. Finally, we explain how we combine *incremental inference* along with *multi-query optimizations* (MQO) to improve the performance of occlusion experiment workload which requires multiple CNN inference requests.

3.1 Expected Speedups

We now explain why there is a potential for incremental computation and estimate the upper-bounds for the expected speedups. In the case of incremental updates only a smaller spatial context having a width $W_{\mathcal{P}:l}$ ($W_{\mathcal{P}:l} <= W_{O:l}$) and height $H_{\mathcal{P}:l}$ ($H_{\mathcal{P}:l} <= H_{O:l}$) is needed to be recomputed. The amount of computations required for the incremental computation $Q_{inc:l}$ and total amount of incremental computations Q_{inc} required for the entire set of convolution layers L will be smaller than the full computation values explained

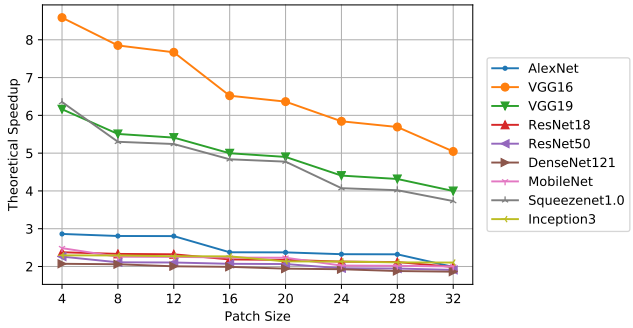


Figure 3: Theoretical speedup for popular CNN architectures with *incremental inference*.

in Section 3, and can be calculated as per Equation (12) and (13).

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

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

Notice that in both full and incremental formulations, computational cost calculation takes into account the dimensions of the output or the output patch. The output dimensions can be determined using the dimensions of the input and other parameters as explained in Section 3.2. In Section 4.2 we show that the dimensions of the output patch can also be predetermined using the input patch dimensions. Using the above quantities we define a new metric named *theoretical speedup* , which is the ratio between the total full computational cost Q and total incremental computation cost Q_{inc} (see Equation (14)). This ratio essentially acts as a surrogate for the theoretical upper-bound for computational and runtime savings that can be achieved by applying incremental computations to Deep CNNs.

$$\text{theoretical speedup} = \frac{Q}{Q_{inc}} \quad (14)$$

We analyze the theoretical speedup that can be achieved with the *incremental inference* approach when a square occlusion patch is placed on the center[1] of the input image. Figure 3 shows the results. VGG16 model results in the maximum theoretical speedup and DenseNet121 model has the lowest theoretical speedup. Most CNN architectures result in a theoretical speedup between 2-3. The theoretical speedup for a CNN with *incremental inference* is determined by the characteristics of its architecture such as the number of layers, the sizes of the filter kernels, and the filter stride values. For example, VGG16 which uses small Convolution filter kernels and strides incurs a very high computational cost (15 GFLOPs) for a single full inference. However, as the change propagation rate with small filter sizes and strides is small, significant computational savings and runtime speedups can be achieved with incremental inference. These speedups are significant specially in the human in the loop interactive mode. For example, with VGG-16 if the original approach would take 1 minute on GPU to execute, it will now take roughly 10 seconds to execute and hence is more amenable for interactive diagnosis of CNN predictions. On the CPU environment, this implies if a VGG-16 occlusion experiment would take 6 minutes to execute, it will now take only 1 minute to execute and hence will enable significant computational savings and runtime reductions. Even though for most CNNs the theoretical speedup with incremental inference varies between 2-3, incremental inference approach lays the foundation for the other approximate inference optimizations in Krypton which combined together yields higher speedups. We next show how to get as close to the above calculated theoretical speedup as possible by combining incremental inference and multi-query optimization.

[1]If the occlusion patch is placed towards a corner of the input image the theoretical speedup will be slightly higher. But placing the occlusion patch on the center gives us a worst-case estimate.

3.2 Incremental Inference: Of a Single Layer

As explained earlier, occlusion experiments in its naive form perform many redundant computations. In order to avoid these redundancies, layers in a CNN have to be change-aware and operate in an incremental manner, i.e., reuse previous computations as much as possible and compute only the required ones. In this section, we focus only on transformations that operate at the granularity of a local spatial context (i.e. Convolution and Pooling) as other types either have no redundancies (global context transformations) or are trivial to make change-aware (point transformations) as explained in Section 3.2.

Calculating the Coordinates and Dimensions of Propagating Patches. The coordinates and the dimensions (i.e. height and width) of the modified patch in the output tensor of l^{th} layer caused by a modified patch in the input tensor are determined by the coordinates and the dimensions of the input patch, sizes of the filter kernel $H_{\mathcal{K}:l}$ and $W_{\mathcal{K}:l}$, padding values $P_{x:l}$ and $P_{y:l}$, and the strides $S_{x:l}$ and $S_{y:l}$. For example, consider the simplified demonstration showing a cross-section of input and output in Figure 4. We use a coordinate system whose origin is placed at the top left corner of the input. A patch marked in red is placed on the input starting at $x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l}$ coordinates and has a height of $H^{\mathcal{I}}_{\mathcal{P}:l}$ and width of $W^{\mathcal{I}}_{\mathcal{P}:l}$. The updated patch in the output starts at $x^{O}_{\mathcal{P}:l}, y^{O}_{\mathcal{P}:l}$

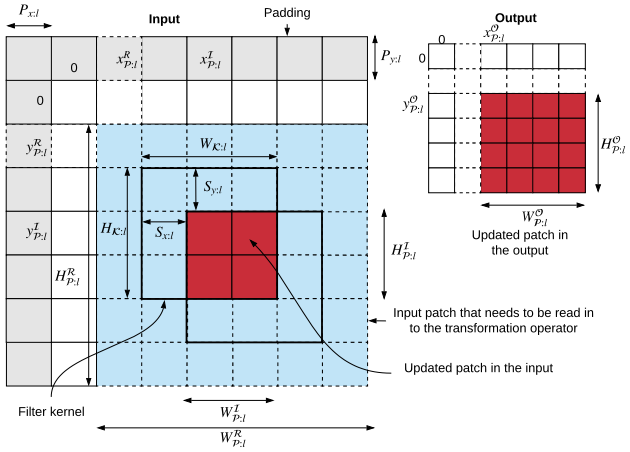


Figure 4: Simplified representation of input and output patch coordinates and dimensions of Convolution and Pool transformations.

and has a height of $H^O_{\mathcal{P}:l}$ and width of $W^O_{\mathcal{P}:l}$. Note that due to the overlapping nature of filter positions, to compute the output patch, transformations have to read a slightly larger context than the updated patch. This "read-in context" is shown by the blue shaded area in Figure 4. The starting coordinates of this read-in patch are denoted by $x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l}$ and the dimensions are denoted by $W^{\mathcal{R}}_{\mathcal{P}:l}, H^{\mathcal{R}}_{\mathcal{P}:l}$. Table 2 summarizes these additional notation. The relationship between the coordinates and dimensions in the horizontal axis (similarly on the vertical axis) can be expressed as follows:

$$x^O_{\mathcal{P}:l} = max\Big(\lceil (P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\Big) \quad (15)$$

$$W^O_{\mathcal{P}:l} = min\Big(\lceil (W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W_{O:l}\Big) \quad (16)$$

$$x^{\mathcal{R}}_{\mathcal{P}:l} = x^O_{\mathcal{P}:l} \times S_{x:l} - P_{x:l} \quad (17)$$

$$W^{\mathcal{R}}_{\mathcal{P}:l} = W_{\mathcal{K}:l} + (W^O_{\mathcal{P}:l} - 1) \times S_{x:l} \quad (18)$$

Equation (15) calculates the starting coordinates of the output patch. Use of padding effectively shifts the coordinate system and therefore $P_{x:l}$ is added to correct it. Due to the overlapping nature of filter kernels, the affected span of the updated patch in the input will be increased by $W_{\mathcal{K}:l} - 1$ amount and hence 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 the new change at $x^{\mathcal{I}}_{\mathcal{P}:l}$). Equation (16) calculates the width of the output patches. Once the output patch coordinate and width are calculated it is straightforward to calculate the read-in patch coordinate as per Equation (17) and the width as per Equation (18).

Incremental Inference Operation. For layer l, given the original transformation function $T_{:l}$, pre-materialized input tensor $\mathcal{I}_{:l}$, updated input patch $\mathcal{P}^O_{:l}$, and the above calculated coordinates and dimensions of the input, output, and read-in patches, the output patch $\mathcal{P}^O_{:l}$ can be calculated as follows:

Table 2: Additional symbols used in the Section 3 and Section 4

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

$$\mathcal{U} = \mathcal{I}_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}:l} : x^{\mathcal{R}}_{\mathcal{P}:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l} : y^{\mathcal{R}}_{\mathcal{P}:l} + H^{\mathcal{R}}_{\mathcal{P}:l}] \quad (19)$$

$$\mathcal{U} = \mathcal{U} \circ_{(x^{\mathcal{I}}_{\mathcal{P}:l} - x^{\mathcal{R}}_{\mathcal{P}:l}), (y^{\mathcal{I}}_{\mathcal{P}:l} - y^{\mathcal{R}}_{\mathcal{P}:l})} \mathcal{P}^{\mathcal{I}}_{:l} \quad (20)$$

$$\mathcal{P}^O_{:l} = T_{:l}(\mathcal{U}) \quad (21)$$

Equation (19) reads the read-in context $\mathcal{U}$ from the pre-materialized input $\mathcal{I}_{:l}$. Equation (20) superimposes the input patch $\mathcal{P}^{\mathcal{I}}_{:l}$ on it. Finally, Equation (21) calculates the modified output patch $\mathcal{P}^O_{:l}$ by invoking T on $\mathcal{U}$.

3.3 Propagating Patches across Layers

CNNs with a Sequence of Layers. In a CNN composed of a sequence of layers, the updated output patch $\mathcal{P}^O_{:l}$ of layer l is fed as the input to layer $l + 1$ along with the pre-materialized input $\mathcal{I}_{:l+1}$ from the original image. However, at a boundary of a local context transformation and a global context transformation, such as in Convolution → Fully-Connected or Pool → Fully-Connected, the full updated output has to be materialized instead of propagating only the updated patches.

Extending to DAG like CNNs. In most CNNs, especially the newer and popular ones, the layers are organized into the more generic directed acyclic graph (DAG) structure instead of a simple sequence. In addition to the different CNN layers, theses CNNs contain two linear algebra operators: element-wise addition and depth-wise concatenation which are used to merge two branches in the DAG. Element-wise addition operator requires the two input tensor to have

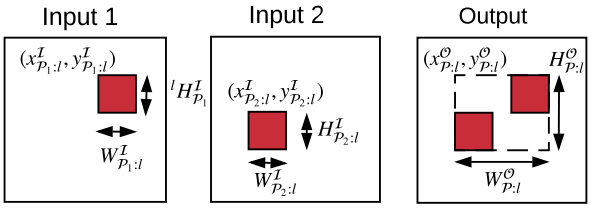


Figure 5: Input-output coordinate and dimension mapping for element-wise addition and depth-wise concatenation.

exact same dimensions. Depth-wise concatenation requires them to have the same height and width dimensions.

Consider a situation for these operators as shown in Figure 5 where the first input has an incremental update starting at coordinates $x^{\mathcal{I}}_{\mathcal{P}_1:l}, y^{\mathcal{I}}_{\mathcal{P}_1:l}$ with dimensions of $H^{\mathcal{I}}_{\mathcal{P}_1:l}$ and $W^{\mathcal{I}}_{\mathcal{P}_1:l}$ and for the second input starting at coordinates $x^{\mathcal{I}}_{\mathcal{P}_2:l}, y^{\mathcal{I}}_{\mathcal{P}_2:l}$ with dimensions of $H^{\mathcal{I}}_{\mathcal{P}_2:l}$ and $W^{\mathcal{I}}_{\mathcal{P}_2:l}$. In this case, the coordinates and dimensions of both output and read-in contexts will be the same, computing them essentially requires finding the bounding box of the two input patches. For the horizontal axis (similarly for the vertical axis) this can be expressed as follows:

$$x^{\mathcal{O}}_{\mathcal{P}:l} = \texttt{min}(x^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l})$$
$$W^{\mathcal{O}}_{\mathcal{P}:l} = \texttt{max}(x^{\mathcal{I}}_{\mathcal{P}_1:l} + W^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l} + W^{\mathcal{I}}_{\mathcal{P}_2:l}) - \texttt{min}(x^{\mathcal{I}}_{\mathcal{P}_1:l}, x^{\mathcal{I}}_{\mathcal{P}_2:l}) \quad (22)$$

3.4 Multi-Query Inference

As explained in Section 2.1, occlusion experiments require performing multiple CNN inferences with each corresponding to a specific occlusion patch position. Each of these individual inference request is analogous to a *query* in the RDBMs context. Coupled with *incremental inference* optimization, these multiple inference requests provide opportunities to apply *multi-query optimization* style optimization to the occlusion experiment workload. We materialize all intermediate tensors in the CNN once for the unmodified image and reuse it for all remaining queries with incremental computations. To the best of our knowledge, this is the first instance of applying incremental view maintenance combined with multi-query optimization to improve performance in a data system. We also apply system specific optimizations to improve performance in the high throughput GPU environment.

Batched Incremental Inference. In Krypton *incremental inference* optimization is applied on top of the currently dominant approach of performing CNN inference on batches of images, with batch size selected to optimize hardware utilization. Each image in the batch corresponds to an occluded instance of the original image. Batched inference is important as it reduces the per-image inference time by amortizing the fixed overheads. In our experiments we found that this simple optimization alone can give up to 1.4X speedups in the CPU environment and 2X speedups on the GPU environment compared to the per-image inference approach. Algorithm 1 presents the *batched incremental inference* operator for the l^{th} layer formally.

Algorithm 1 Batched Incremental Inference Algorithm

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*
Output:
$[\mathcal{P}^{\mathcal{O}}{}_{1:l}, ..., \mathcal{P}^{\mathcal{O}}{}_{n:l}]$: *Output patches*
$[(x^{\mathcal{O}}_{\mathcal{P}_1:l}, y^{\mathcal{O}}_{\mathcal{P}_1:l}), ..., (x^{\mathcal{O}}_{\mathcal{P}_n:l}, y^{\mathcal{O}}_{\mathcal{P}_n:l})]$: *Output patch coordinates*
$W^{\mathcal{O}}_{\mathcal{P}:l}, H^{\mathcal{O}}_{\mathcal{P}:l}$: *Output 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* $\mathcal{U} \in \mathbb{R}^{n \times \texttt{depth}(\mathcal{I}_l) \times H^{\mathcal{R}}_{\mathcal{P}:l} \times W^{\mathcal{R}}_{\mathcal{P}:l}}$
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: $[(x^{\mathcal{O}}_{\mathcal{P}_1:l}, y^{\mathcal{O}}_{\mathcal{P}_1:l}), ..., (x^{\mathcal{O}}_{\mathcal{P}_n:l}, y^{\mathcal{O}}_{\mathcal{P}_n:l})]$, $(W^{\mathcal{O}}_{\mathcal{P}:l}, H^{\mathcal{O}}_{\mathcal{P}:l})$

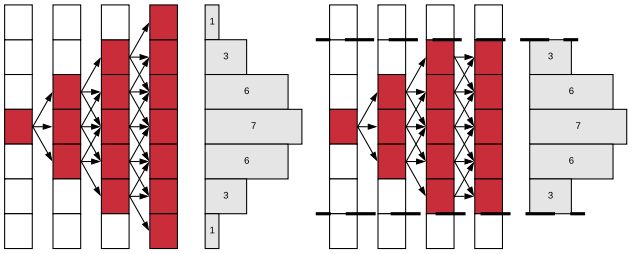
Notice that for the first layer, all the elements in the batch of updated patches will be identical as the same occlusion patch (in RGB format) is used. However, at latter layers they will be different as each input patch will subject to different computations due to different read-in contexts across patches. First, BatchedIncrementalInference calculates the geometric properties of the output and the read-in contexts. A temporary input tensor U is initialized to hold the input patches with their read-in contexts. The `for` loop iteratively populates U with corresponding patches. Finally, $T_{:l}$ is applied to U to compute the output patches.

GPU Optimized Implementation. Through our experiments, we found that even though a straight-forward implementation of *batched incremental inference* approach as shown in Algorithm 1 produces expected speedups for the CPU environment, it performs poorly on the GPU environment. The `for` loop on the line 7 of Algorithm 1 is essentially preparing the input for $T_{:l}$ by copying values from one part of the memory to another sequentially. However, in the *multi-query* setting this sequential operation becomes a bottleneck for the GPU implementation, since it cannot exploit the available parallelism of the GPU efficiently. To overcome this we have extended PyTorch by adding a custom kernel written in CUDA language which performs the input preparation more efficiently by parallelly copying the memory regions for all items in the batch and then invoke the CNN transformation function $T_{:l}$. More details on the custom GPU kernel implementation is explained in Appendix. The original transformation function $T_{:l}$, which is provided as an input to the Algorithm 1 is already optimized to use the GPU efficiently for a batch of inputs when performing inference. TODO: Refer to hardware works, e.g. Li Tseng, Lin, Swanson, Yannis on hardware accelerators

3.5 Putting it all together

The high-level steps taken by the end-to-end incremental inference approach for the occlusion experiment can be summarized as follows:

(1) Take in CNN f, image $\mathcal{I}_{:img}$, predicted class label L, occlusion patch $\mathcal{P}$, stride $S_{\mathcal{P}}$ for the $\mathcal{P}$, and the set of occluding patch placement positions G as input.
(2) Pre-materialize output of all the transformations in f by feeding in $\mathcal{I}_{:img}$.
(3) Prepare the occluded images ($\mathcal{I}'_{x,y:img}$ s) corresponding to all positions in G.
(4) For batches of $\mathcal{I}'_{x,y:img}$ as the input, invoke the transformations in f in topological order and calculate the corresponding values of heat map M.
   - For local-context transformation, invoke BatchedIncrementalInference.
   - For local-context transformation, that feeds in a global context transformation additionally materialize the full updated output.
   - For all others, invoke the original transformation function.
(5) Return M as the final output.



(a) Projective Field (b) Projective field thresholding

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

4 APPROXIMATE INFERENCE OPTIMIZATIONS

In this Section, we explain *projective field thresholding* and *adaptive drill-down*, which are two *approximate inference* optimizations used in Krypton. We also explain how Krypton tunes its internal system configuration parameters for these *approximate inference* optimizations.

4.1 Projective Field Thresholding

Projective field [17, 18] of a CNN neuron is the local region (including the depth) of the output 3-D tensor which is connected to it. The term is borrowed from the Neuroscience field where it is used to describe the spatiotemporal effects exerted by a retinal cell on all of the outputs of the neuronal circuitry [19]. For our work, the notion of projective field is useful as it determines the change propagation path for incremental changes. The three types of CNN transformations affect the size of the projective field differently. Point transformations do not change the projective field size while global context transformations increase it to the maximum. Transformations that operate on a local spatial context increase it gradually. The amount of increase in a local context transformation is determined by the filter size and stride parameters. At every transformation, the size of the projective field will increase linearly by the filter size and multiplicatively by the stride value.

Because of the projective field growth, even though there will be many computational redundancies in the early layers, towards the latter layers it will decrease or even have no redundancies. However, we empirically found that the projective field growth can be restrained up to a certain extent without significantly sacrificing the accuracy. For a more intuitive understanding of why this would work consider the simplified 1-D Convolution example shown in Figure 6 (a). In the example, a single neuron is modified (marked in red)

and a filter of size three is applied with a stride of one repeatedly four times. Since the filter size is three, each updated neuron will propagate the change to three neurons in the next output layer causing the projective field to grow linearly. The histogram at the end of the fourth layer shows the number of unique paths that are available between each output neuron and the originally updated neuron in the first layer. It can be seen that this distribution resembles a Gaussian where many of the paths are connected to the central region. The amount of change in the output layer is determined by both the number of unique paths and also the individual weights of the connections. It can be shown that the distribution of change in the output layer will converge to a Gaussian distribution provided certain conditions are met for the weight values of the filter kernel (more details in Appendix X).

As most of the change will be concentrated on the center we introduce the notion of a projective field threshold τ ($0 < \tau \leq 1$) which will be used to restrict the growth of the projective field. It determines the maximum size of the projective field as a fraction of the size of the output. Figure 6 (b) demonstrates the application of projective field thresholding with a τ value of 5/7. From the histogram generated for the projective field thresholding approach, we can expect that much of the final output change is maintained by this approach.

In Krypton, *projective field thresholding* is implemented on top of *incremental inference* approach by applying set of additional constraints on input-output coordinate mappings. For the horizontal dimension (similarly for vertical dimension) the new set of calculations can be expressed as follows:

$$W^O_{\mathcal{P}:l} = \texttt{min}\Big(\lceil (W^I_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W^O_{\mathcal{P}:l}\Big) \tag{23}$$

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

$$W^O_{\mathcal{P}:l} = \texttt{round}(\tau \times W^O_{:l}) \tag{25}$$

$$W^I_{\mathcal{P}_{new}:l} = W^O_{\mathcal{P}:l} \times S_{x:l} - W_{\mathcal{K}:l} + 1 \tag{26}$$

$$x^I_{\mathcal{P}:l} \mathrel{+}= (W^I_{\mathcal{P}:l} - W^I_{\mathcal{P}_{new}:l})/2 \tag{27}$$

$$W^I_{\mathcal{P}:l} = W^I_{\mathcal{P}_{new}:l} \tag{28}$$

$$x^O_{\mathcal{P}:l} = \texttt{max}\Big(\lceil (P_{x:l} + x^I_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\Big) \tag{29}$$

Equation (23) calculates output width assuming no thresholding. But if the output width exceeds the threshold defined by τ, output width is set to the threshold value as per Equation (25). Equation 26 calculates the input width that would produce an output of width $W^O_{\mathcal{P}:l}$ (think of this as making $W^I_{\mathcal{P}:l}$ the subject of equation 23). If the new input width is smaller than the original input width, the starting x coordinate should be updated as per Equation (27) such that the updated coordinates correspond to a center crop from the original. Equation (28) set the input width to the newly calculated input width and Equation (29) calculates the x coordinate of the output patch from the updated values.

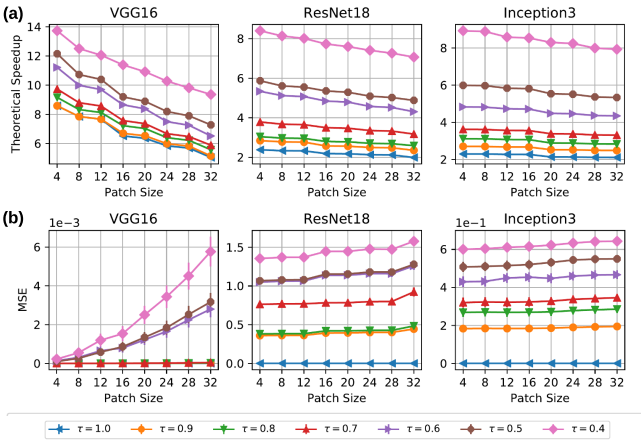


Figure 7: (a) Theoretical speedup ratio with projective field thresholding. (b) Mean Square Error between exact and approximate output of the final Convolution or Pool transformation.

Theoretical Speedup with Projective Field Thresholding. We analyze the theoretical speedup that can be achieved with *projective field thresholding* approach when a square occlusion patch is placed on the center of the input image. Figure 7 (a) presents the results. It can be seen that with increasing τ attainable theoretical speedup also increases. We also analyze the mean square error (MSE) between the exact and approximate output tensors produced by the final Convolution or Pool transformation with a black occlusion patch placed on the center of the input image. The results are shown in Figure 7 (b). With increasing τ and increasing patch size we see that the MSE is also increasing.

4.2 Adaptive Drill-Down

Adaptive drill-down approach, which is only applicable in the non-interactive mode, is based on the observation that in many occlusion based explainability workloads, such as in medical imaging, the regions of interest will occupy only a small fraction of the entire image. In such cases, it is unnecessary to inspect the entire image at a higher resolution with a small stride value for the occlusion patch. In *adaptive-drill-down* the final occlusion heat map will be generated using a two-stage process. At the first stage, a low-resolution heat map will be generated by using a larger stride which we call stage one stride S_1. From the heat map generated at

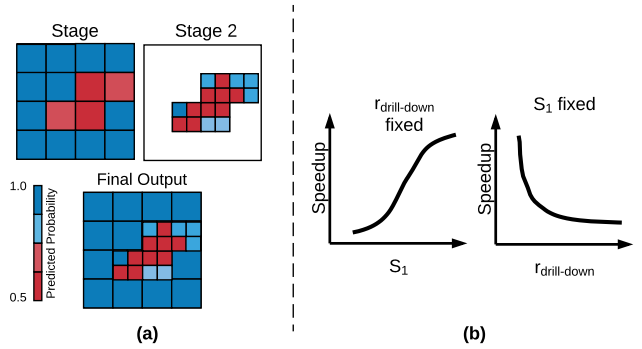


Figure 8: (a) Schematic representation of *adaptive drill-down*. (b) Conceptual diagram showing the effect of S_1 and $r_{drill-down}$ on speedup.

stage one, a predefined drill-down fraction $r_{drill-down}$ of regions with highest probability drop for the predicted class is identified. At stage two a high-resolution occlusion map is generated using the original user provided stride value, also called stage two stride S_2, only for the selected region. A schematic representation of *adaptive drill-down* is shown in Figure 8 (a).

The amount of speedup that can be obtained from *adaptive drill-down* is determined by both $r_{drill-down}$ and S_1. If the $r_{drill-down}$ is low, only a small region will have to be examined at a higher resolution and thus it will be faster. However, this smaller region may not be sufficient to cover all the interesting regions on the image and hence can result in losing important information. Larger S_1 also reduces the overall runtime as it reduces the time taken for stage one. But it has the risk of misidentifying interesting regions especially when the granularity of those regions are smaller than the occlusion patch size. The speedup obtained by *adaptive drill-down* approach is equal to the ratio between the number of individual occlusion patch positions generated for the normal and *adaptive drill-down* approaches. Number of individual occlusion patch positions generated with a stride value of S is proportional to $1/S^2$ (total number of patch positions is equal to $\frac{H_{I_{img}}}{S} \times \frac{W_{I_{img}}}{S}$). Hence the speedup can be expressed as per Equation 30. Figure 8 (b) conceptually shows how the speedup would vary with S_1 when $r_{drill-down}$ is fixed and vice versa.

$$\text{speedup} = \frac{S_1^2}{S_2^2 + r_{drill-down} \times S_1^2} \tag{30}$$

4.3 System Tuning

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

Tuning projective field threshold. The inaccuracies incurred when applying *projective field thresholding* can cause quality degradation in the generate approximate heat map all the way from indistinguishable changes major structural changes. To measure this quality degradation we use Structural Similarity (SSIM) Index [20], which is one of the widely used approaches to measure the *human perceived difference* between two similar images. When applying SSIM index, we treat the original heat map as the reference image with no distortions and the perceived image similarity of the approximate heat map is calculated with reference to it. The generated SSIM index is a value between -1 and 1, where 1 corresponds to perfect similarity. Typically SSIM index values in the range of $0.90 - 0.95$ are used in practical applications such as image compression and video encoding as at the human perception level they produce indistinguishable distortions.

Tuning *projective field threshold* τ is done during a special initial tuning phase. During this tuning phase Krypton takes in a sample of images (default 30) from the operational workload and evaluates SSIM value of the approximate heat map (compared to the exact heat map) for different τ values (default values are 1.0, 0.9, 0.8, ..., 0.4). These τ versus SSIM data points are then used to fit a second-degree curve. At the operational time, Krypton requires the user to provide the expected level of quality for the heat maps in terms of a SSIM value. τ is then selected from the curve fit to match this target SSIM value. Figure 9 (a) shows the SSIM variation and degree two curve fit for different τ values and three different CNN models for a tuning set (n=30) from OCT dataset. From the plots, it can be seen that the distribution of SSIM versus τ lies in a lower dimensional manifold and with increasing τ SSIM also increases. Figure 9 (b) shows the cumulative percentage plots for SSIM deviation for the tune and test sets (n=30) when the system is tuned for a target SSIM of 0.9. For a target SSIM of 0.9 system picks τ values of 0.5, 0.7, and 0.9 for VGG16, ResNet18, and Inception3 models respectively. It can be seen that approximately more than 50% of test cases will result in an SSIM value of 0.9 or greater. Even in cases where it performs worse than 0.9 SSIM, significant $(95\% - 100\%)$ portion of them are within +0.1 deviation.

Tuning adaptive drill-down. As explained in section 4.2 the speedup obtained by *adaptive drill-down* approach is determined by two factors; stage one stride value S_1 and drill-down fraction $r_{drill-down}$. For configuring *adaptive*

drill-down KRYPTON requires the user to provide $r_{drill-down}$ and a target speedup value. $r_{drill-down}$ should be selected based on the user's experience and understanding on the relative size of interesting regions compared to the full image. This is a fair assumption and in most cases such as in medical imaging, users will have a fairly good understanding on the relative size of the interesting regions. However, if the user is unable to provide this value KRYPTON will use a default value of 0.25 as $r_{drill-down}$. The speedup value basically captures user's input on how much faster the occlusion experiment should run. Higher speedup values will sacrifice the quality of non-interesting (1-$r_{drill-down}$) regions for faster execution. The default value for speedup value is three. The way how KRYPTON configures *adaptive drill-down* is different to how it configures *projective field thresholding*. The reason for this is, unlike in *projective field thresholding* in *adaptive drill-down* users have more intuition on the outcomes of $r_{drill-down}$ and target speedup parameters compared to the SSIM quality value of the final output. Given $r_{drill-down}$, target speedup value, and original occlusion patch stride value S_2 (also called stage two stride) KRYPTON then calculates the stage one stride value S_1 as per equation 31. As S_1 cannot be greater than the width W_{img} (similarly height H_{img}) of the image and with the mathematical constraint of $(1 - r_{drilldown} \times \text{speedup})$ being positive, it can be seen that possible values for the speedup value is upper-bounded as per equation 32.

$$S_1 = \sqrt{\frac{\text{speedup}}{1 - r_{drill-down} \times \text{speedup}}} \times S_2 \tag{31}$$

$$S_1 = \sqrt{\frac{\text{speedup}}{1 - r_{drill-down} \times \text{speedup}}} \times S_2 < W_{img}$$
$$\text{speedup} < \texttt{min}\left(\frac{W_{img}^2}{S_2^2 + r_{drill-down} \times W_{img}^2}, \frac{1}{r_{drill-down}}\right) \tag{32}$$

5 EXPERIMENTAL EVALUATION

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

Datasets. We use three real-world datasets: *OCT*, *Chest X-Ray*, and a sample from *ImageNet*. *OCT* has about 84,000 optical coherence tomography retinal images categorized into four categories: CNV, DME, DRUSEN, and NORMAL.

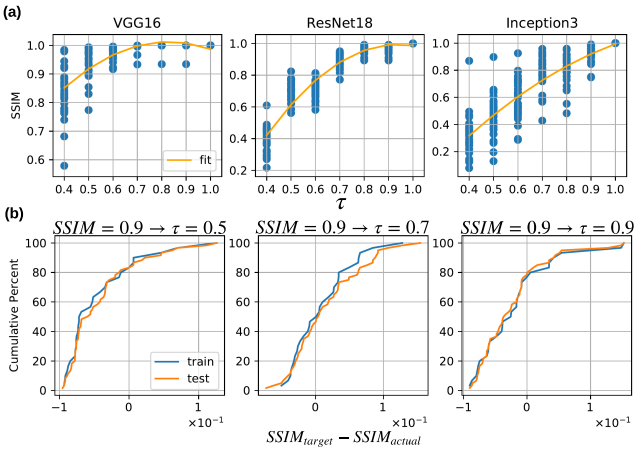


Figure 9: (a) SSIM variation and degree two curve fit for a sample of OCT dataset. (b) CDF plot for the SSIM deviation for the τ values picked from the curve fit for a target SSIM of 0.9.

CNV (choroidal neovascularization), DME (diabetic macular edema), and DRUSEN are three different varieties of Diabetic Retinopathy. NORMAL corresponds to healthy retinal images. *Chest X-Ray* has about 6,000 X-ray images categorized into three categories: VIRAL, BACTERIAL, and NORMAL. VIRAL and BACTERIAL categories correspond to two varieties of Pneumonia. NORMAL corresponds to chest X-Rays of healthy people. Both *OCT* and *Chest X-Ray* datasets are obtained from an original scientific study [6] which uses CNNs for predicting Diabetic Retinopathy and Pneumonia from radiological images. *ImageNet* sample dataset contains 1,000 images corresponding to two hundred categories selected from the original thousand categorical dataset [21].

Workloads. We use three popular ImageNet-trained Deep CNNs: VGG16 [2], ResNet18 [3], and Inception3 [4], obtained from [22]. They complement each other in terms of model size, computational cost, amount of theoretical redundancy that exist for occlusion experiments, and the level of architectural complexity of the CNN model. For *OCT* and *Chest X-Ray* datasets the three CNN models are fine-tuned by retraining the final fully-connected layer with hyperparameter tuning as per standard practice. More details on the fine-tuning process are included in the Appendix. Heat map for the predicted probabilities is generated using Python Matplotlib library's `imshow` method using the `jet_r` color scheme. For the heat map, the maximum threshold value is set to $\texttt{min}(1, 1.25 \times p)$ and minimum threshold value is set to $0.75 \times p$ where p is predicted class probability for the unmodified image. Original images were resized to the size required by the CNNs (224×224 for VGG16 and ResNet18 and 299×299 for Inception3) and no additional

pre-processing is done. For GPU experiments a batch size of 128 and for CPU experiments a batch size 16 is used. CPU experiments are executed with a thread parallelism of 8. All of our datasets, fine-tuning, experiment, and system code will be made available on our project web page.

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

5.1 End-to-End Evaluation

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

We see that Krypton improves the efficiency of the occlusion based explainability workload across the board. Krypton-Approximate for *OCT* results in the highest speedup with VGG16 on both CPU and GPU environments (16X for CPU and 34.5X for GPU). Speedups obtained by Krypton-Exact for all the datasets are same for all three CNN models. However, with Krypton-Approximate they result in different speedup values. This is because with *approximate inference* each dataset uses different system configuration parameters. *OCT* which is configured with a low $r_{drill-down}$ of 0.1, high target speedup of 5, and a *projective field threshold* value of 0.5 results in the highest speedup. Speedup obtained by Krypton-Exact on GPU with Inception3 model (0.7X) is slightly lower than one. However ResNet18 which has roughly the same theoretical speedup (see Figure 3) results in a higher speedup value (1.6X). The reason for this is Inception3's internal architecture is more complex compared ResNet18 with more branches and depth-wise stacking operations. Thus Inception3 requires more memory copying operations whose overheads are not captured by our theoretical speedup calculation. Overall compared to GPU environment Krypton results in higher speedups on the CPU environment though the actual runtimes are much slower. GPUs enable higher parallelism with thousands of processing cores compared to CPUs with several cores. Hence computations are much cheaper on GPU. Memory operations required by Krypton throttles the overall performance on GPU and hinders it from achieving higher speedups. On CPU environment as computational cost dominates the overall runtime, the additional overhead introduced by the memory operations does not matter much. Therefore on CPU Krypton achieves higher speedups which are closer to the theoretical speedup value. Overall Krypton offers the best efficiency on these workloads. This confirms the benefits of different optimizations performed by Krypton for improving the efficiency of the workload and thereby to reduce the computational and runtime costs. Bringing down the runtimes also make occlusion experiments more amenable for interactive diagnosis of CNN predictions.

5.2 Lesion Study

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

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

Speedups from Projective Field Thresholding. We vary *projective field threshold* τ from 1.0 (no thresholding) to 0.4

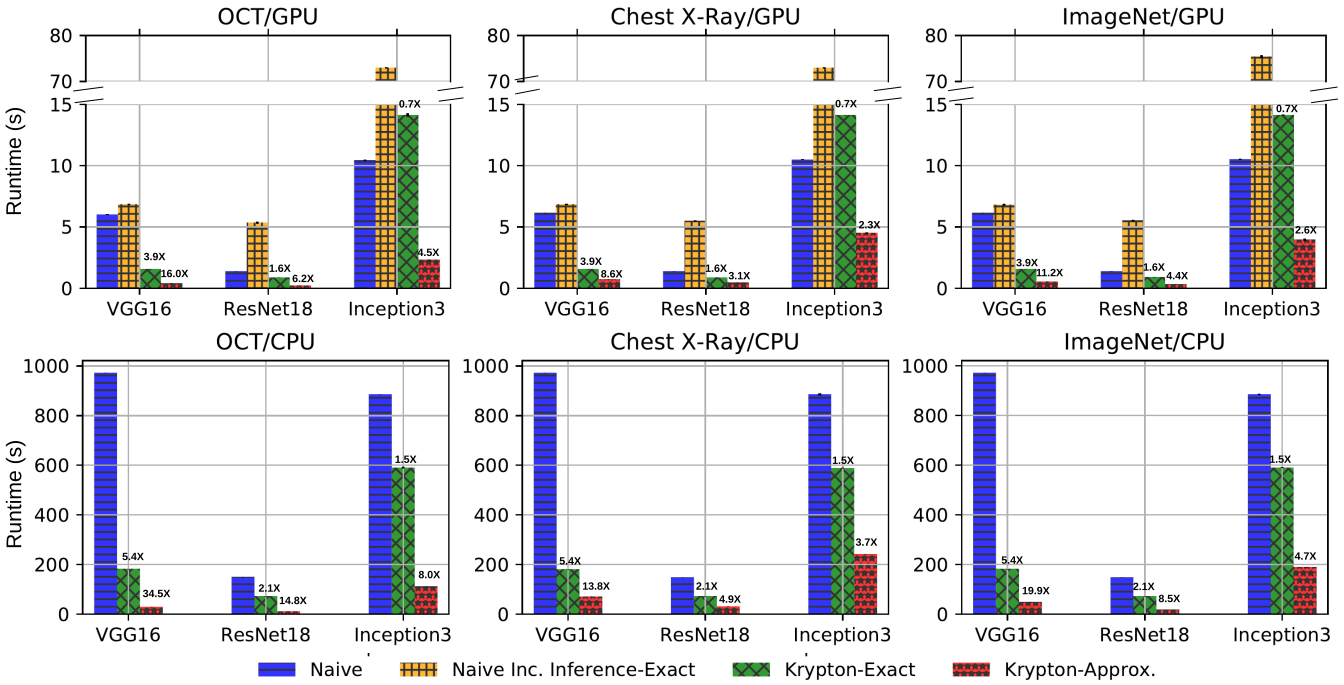


Figure 10: End-to-end efficiency achieved by KRYPTON over naive approaches.

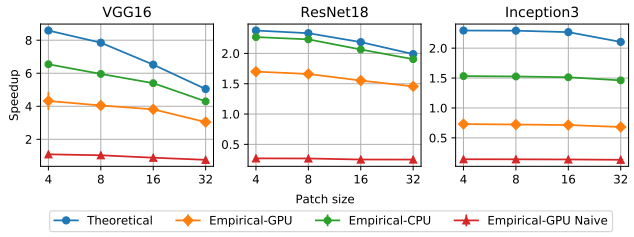


Figure 11: Theoretical versus empirical speedup for *incremental inference* (Occlusion patch stride $S = 4$).

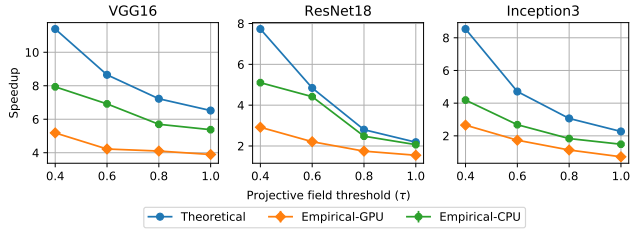


Figure 12: Theoretical versus empirical speedup for *incremental inference* with *projective field thresholding* (Occlusion patch size = 16×16, stride $S = 4$).

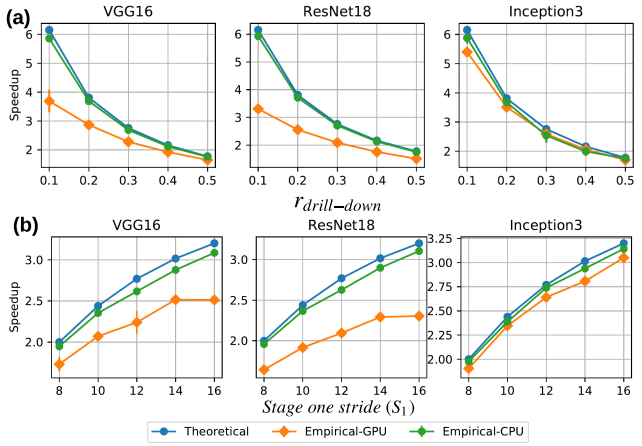


Figure 13: Theoretical versus empirical speedup for *adaptive drill-down* (Occlusion patch size = 16×16, stage two stride $S_2 = 4$, projective field threshold $\tau = 1.0$. For (a) S_1=16 and for (b) r_{drill_down}=0.25).

and evaluate the speedups. The occlusion patch size used is 16 and the stride is 4. The results are shown in Figure 12. Empirical-CPU and Empirical-GPU both results in higher speedups with Empirical-CPU being closer to the theoretical speedup value. When τ decreases the speedups increase as the amount of computational savings increase.

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

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

6 OTHER RELATED WORK

Incremental View Maintenance. Incremental view maintenance [23–25] is a well studied concept in the relational data model context. It is used in databases to support incremental updates to materialized views by applying differential updates without re-evaluating the complete view definitions. The IVM approach presented in this paper falls into the broader category of *"lazy maintenance"* strategy [23]. While there are many work in this context, most relevant to our work are [26] and [27]. [26] extended the IVM concept to support IVM of linear algebra operators as opposed to classical database queries. [27] extended the IVM concept to support multi-dimensional array data model with support for both relational style functions (e.g. filter and join) and also array style functions (e.g. smoothing and cross-matching). The spatially localized transformations explained in Section 2.2 can be thought of as a form of *"spatial array join"* explained in [27]. However, the focus of [27] is on supporting efficient IVM on distributed sparse arrays by reducing the communication and chunk reassignment where as the focus of this paper is on formalizing the exact semantics of the IVM operators for spatially localized transformations in the emerging workload of occlusion based deep CNN predictions. We also analyze the theoretical speedups that can be achieved for occlusion experiments using our IVM operator formulations.

Multi Query Optimization.

7 CONCLUSIONS AND FUTURE WORK

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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] 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.
[18] 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.
[19] Saskia EJ de Vries et al. The projective field of a retinal amacrine cell. *Journal of Neuroscience*, 31(23):8595–8604, 2011.
[20] Zhou Wang et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.
[21] 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.
[22] torch vison models. https://github.com/pytorch/vision/tree/master/torchvision/models. Accessed September 31, 2018.
[23] Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.
[24] Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
[25] 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.
[26] 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.
[27] 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.
[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[29] Steffen Eger. Restricted weighted integer compositions and extended binomial coefficients. *J. Integer Seq*, 16(13.1):3, 2013.

A SPECIAL SITUATIONS WITH INCREMENTAL INFERENCE

It is important to note that there are special situations under which the actual output patch size can be smaller than the values calculated in Section 3.2. Consider the simplified one dimensional situation shown in Figure 14 (a), where the stride value[2] (3) is same as the filter size (3). In this situation, the size of the output patch is one less than the value calculated by Equation 16. However, it is not the case in Figure 14 (b) which has the same input patch size but is placed at a different location. Another situation arises when the input patch is placed at the edge of the input as shown in Figure 14 (c). In this situation, it is not possible for the filter to move freely through all filter positions as it hits the input boundary compared to having the input patch on the middle of the input as shown in Figure 14 (c). In Krypton we do not treat theses differences separately and use the values calculated by Equation 16 and **??** as they act as an upper

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

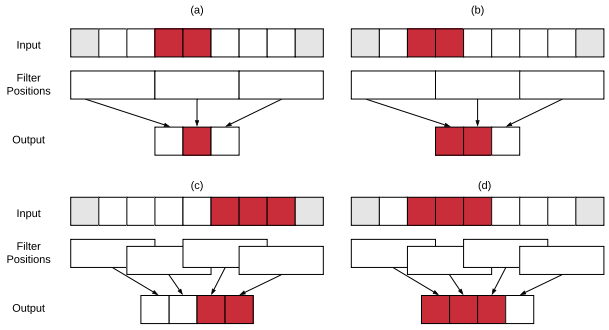


Figure 14: One dimensional representation showing special situations under which actual output size will be smaller than the values calculated by Equations 15 and ??. (a) and (b) shows a situation with filter stride being equal to the filter size. (c) and (d) shows a situation with input patch being placed at the edge of the input.

bound. In case of a smaller output patch, Krypton simply reads off and updates slightly bigger patches to preserve uniformity. This also requires updating the starting coordinates of the patches as shown in Equations 33 and 34. Such uniform treatment is required for performing batched inference operations which out of the box gives significant speedups compared to per image inference.

If $x^O_{\mathcal{P}} + W^O_{\mathcal{P}} > W_O$:

$$\begin{aligned} x^O_{\mathcal{P}} &= W_O - W^O_{\mathcal{P}} \\ x^I_{\mathcal{P}} &= W_I - W^I_{\mathcal{P}} \\ x^{\mathcal{R}}_{\mathcal{P}} &= W_I - W^{\mathcal{R}}_{\mathcal{P}} \end{aligned} \tag{33}$$

If $y^O_{\mathcal{P}} + H^O_{\mathcal{P}} > H_O$:

$$\begin{aligned} y^O_{\mathcal{P}} &= H_O - H^O_{\mathcal{P}} \\ y^I_{\mathcal{P}} &= H_I - H^I_{\mathcal{P}} \\ y^{\mathcal{R}}_{\mathcal{P}} &= H_I - H^{\mathcal{R}}_{\mathcal{P}} \end{aligned} \tag{34}$$

B FINE-TUNING CNNS

For *OCT* and *Chest X-Ray* datasets the three ImageNet pretrained CNN models are fine-tuned by retraining the final layer. We use a train-validation-test split of 60-20-20 and the exact numbers for each dataset are shown in Table 3. Cross-entropy loss with L2 regularization is used as the loss function and Adam [28] 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.

C VISUAL EXAMPLES

Figure 15 presents occlusion heat maps for a sample image from each dataset with (a) *incremental inference* and (b) *incremental inference* with *adaptive drill-down* for different *projective field threshold* values. The predicted class label for *OCT*, *Chest X-Ray*, and *ImageNet* are DME, VIRAL, and OBOE respectively.

D EFFECTIVE PROJECTIVE FIELD SIZE (ONE DIMENSIONAL SCENARIO)

What follows formalizes the effective projective field growth for the one dimensional scenario with n convolution layers (assuming certain conditions).

The input is $u(t)$ where

$$u(t) = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \tag{35}$$

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

$$v(t) = \sum_{m=0}^{k-1} w(m)\delta(t-m) \tag{36}$$

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

$$\begin{aligned} U(\omega) &= \sum_{t=-\infty}^{\infty} u(t)e^{-j\omega t} = 1,\ V(\omega) \\ &= \sum_{t=-\infty}^{\infty} v(t)e^{-j\omega t} = \sum_{m=0}^{k-1} w(m)e^{-j\omega t} \end{aligned} \tag{37}$$

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

$$\begin{aligned} \mathcal{F}(o) &= \mathcal{F}(u * v * \ldots * v)(\omega) = U(\omega).V(\omega)^n \\ &= \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t} \right)^n \end{aligned} \tag{38}$$

With inverse Fourier transform

$$o(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} \left(\sum_{m=0}^{k-1} w(m)e^{-j\omega t} \right)^n e^{j\omega t} d\omega \tag{39}$$

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

$$\begin{aligned} o(t) &= p(S_n = t) \\ \text{where } S_n &= \sum_{i=1}^{n} X_i \text{ and } p(X_i = m) = w(m) \end{aligned} \tag{40}$$

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

$$\mathbb{E}[S_n] = n \sum_{m=0}^{k-1} mw(m) \tag{41}$$

$$Var[S_n] = n\left(\sum_{m=0}^{k-1} m^2 w(m) - \left(\sum_{m=0}^{k-1} mw(m) \right)^2 \right) \tag{42}$$

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

$$\sqrt{Var[S_n]} = \sqrt{nVar[X_i]} = O(\sqrt{n}) \tag{43}$$

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 $O(1/\sqrt{n})$.

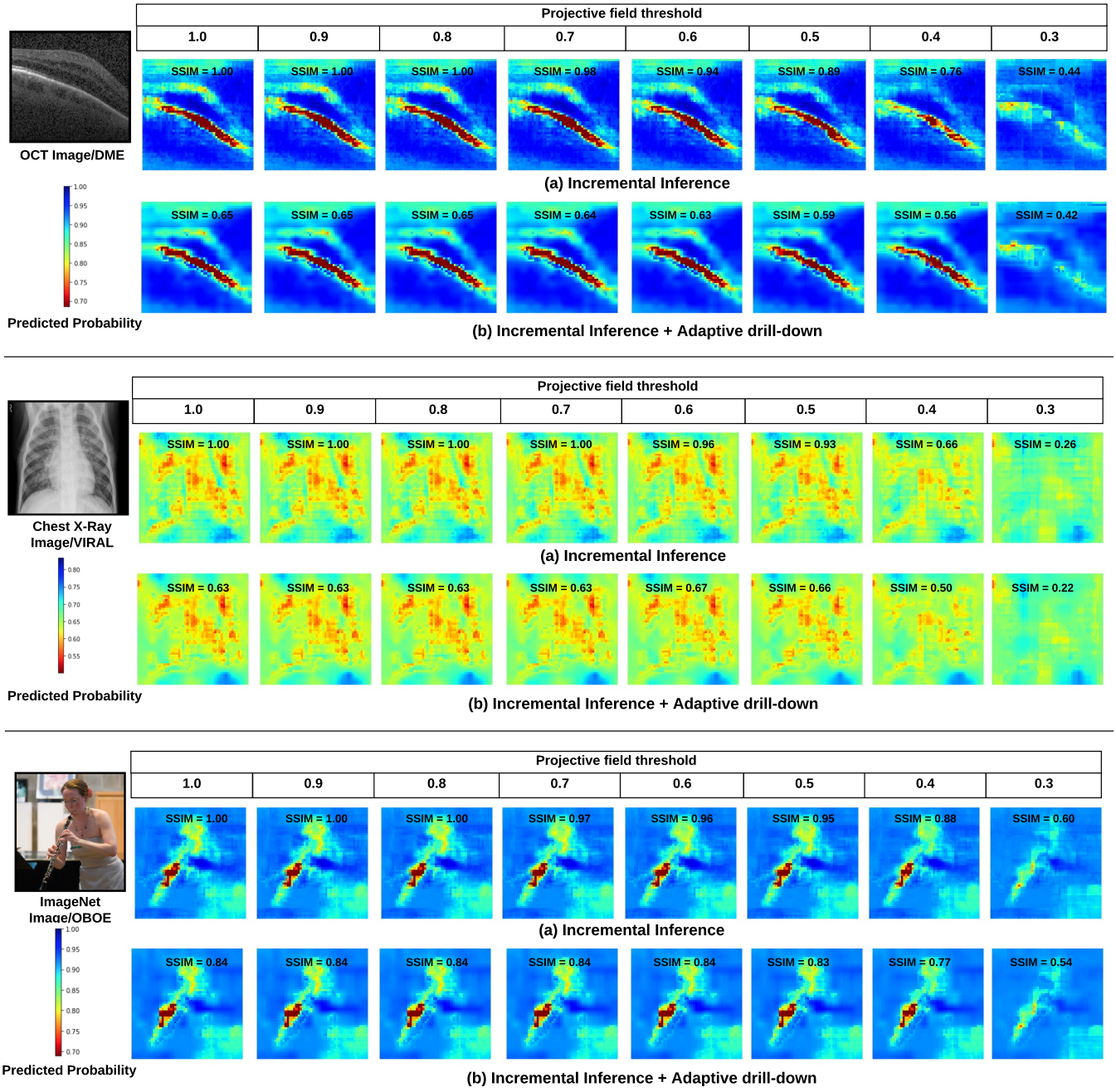


Figure 15: Occlusion heat maps for sample images (CNN model = VGG16, occlusion patch size = 16, patch color = black, occlusion patch stride (S or S_2) = 4. For *OCT* $r_{drill\_down} = 0.1$ and target speedup=5. For *Chest X-Ray* $r_{drill\_down} = 0.4$ and target speedup=2. For *ImageNet* $r_{drill\_down} = 0.25$ and target speedup=3).