KRYPTON: Accelerating Occlusion based Deep CNN Explainability Workloads

Anonymous Author(s)

ABSTRACT

Deep Convolution Neural Networks (CNN) have revolutionized the field of computer vision with even surpassing human level accuracy in some of the image recognition tasks. Thus they are now being deployed in many real-world use cases ranging from health care, autonomous vehicles, and e-commerce applications. However one of the major criticisms pointed against Deep CNNs is the black-box nature of how they make predictions. This is a critical issue when applying CNN based approaches to critical applications such as in health care where the explainability of the predictions is also very important. For interpreting CNN predictions several approaches have been proposed and one of the widely used method in image recognition tasks is occlusion experiments. In occlusion experiments one would mask the regions of the input image using a small gray or black patch and record the change in the predicted label probability. By systematically changing the position of the patch location, a sensitivity map can be generated from which the regions in the input image which influence the predicted class label most can be identified. However, this method requires performing multiple forward passes of CNN inference for explaining a single prediction and hence is very time consuming. We present KRYPTON, the first data system to elevate occlusion experiments to a declarative level and enable database inspired automated *incremental* and *approximate* inference optimizations. Experiments with real-world datasets and deep CNNs show that KRYPTON can enable speedup over 10x in certain cases.

ACM Reference Format:
Anonymous Author(s). 2018. KRYPTON: Accelerating Occlusion based Deep CNN Explainability Workloads. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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

1 INTRODUCTION

Deep convolutional neural networks (CNNs) [1–4] have revolutionized the computer vision field with even surpassing human level accuracies in some of the image recognition challenges such as ImageNet [5]. As a result, there is wide adoption of deep CNN technology in variety of real world image recognition tasks in several domains including health care [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].

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 health care, 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 systematically striding this patch horizontally and vertically few pixels pixels at a time over the image a sensitivity heat map for the predicted label can be generated (similar to one shown in Figure 1 (C)). 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

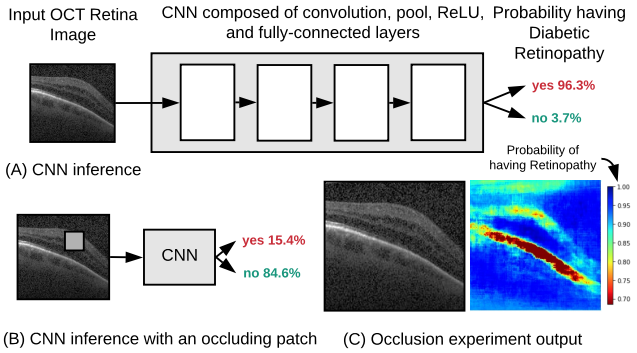

Figure 1: (A) Simplified representation depicting how OCT images are used for predicting Diabetic Retinopathy using CNNs. (B) Occluding parts of the OCT image changes the predicted probability for the disease. (C) By systematically moving patch along vertical and horizontal axes an output heat map is generated where each individual value corresponds to the predicted disease probability when the occlusion patch was placed on that position.

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 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 patch position, there are significant portion of redundant computations which can be avoided. To avoid redundant computations we introduce the notion of *incremental inference* of deep CNNs which is inspired by the incremental view maintenance technique used in the context of relational databases.

Due to the overlapping nature of how a convolution kernel would operate (details to follow in Section 3), the size of the modified patch will start growing as it progress through more layers in a CNN and reduce the amount of redundant computations. 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 this optimization as *projective field thresholding*.

The third optimization is also a form of *approximate inference* which we refer as *adaptive drill-down*. 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 image locations. In adaptive drill-down approach, first a low resolution heatmap is generated using a larger occluding patch and a larger stride with relatively low computational cost. Only the interesting regions will be then inspected further with a smaller stride to produce a higher resolution output.

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 will 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. We use Structural Similarity Index (SSIM) to quantify the quality degradation caused by *approximate inference* and provide user configurable system tuning parameters for easily picking an operational point on the speed-quality trade-off space.

Finally, we have implemented Krypton on top of PyTorch deep learning toolkit by extending it by adding custom implementations of incremental and approximate inference operators. It currently supports VGG16, ResNet18, and InceptionV3 both on CPU and GPU environments, which are three widely used deep CNN architectures for transfer learning applications. 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 up to 10x speedups. While we have implemented Krypton on top of PyTorch toolkit, our work is largely orthogonal to 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 BACKGROUND

Deep CNNs. Deep CNNs are a type of neural networks specialized for image data. They exploit spatial locality of information in image pixels to construct a hierarchy of parametric feature extractors and transformers organized as layers of various types: *convolutions*, 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 the building block of a multi-layer perceptron; and *softmax*, which emits predicted probabilities to each class label. In most "deep" CNN architectures, above layers are simply stacked together with ones output is simply fed as the input to the other, while adding multiple layers element-wise or stacking multiple layers together depth-wise to produce a new layer is also present in some architectures. Popular deep CNN model architectures include AlexNet [1], VGG [2], Inception [4], ResNet [3], SqueezeNet [15], and MobileNet [16].

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 heatmap by computing the partial derivatives of model output with respect to every input pixel via back propagation. In perturbation based approaches the output of the model is observed by masking out regions in the input image and there by identify the sensitive regions. The most popular perturbation based approach is occlusion experiments which was first introduced by Zeiler et. al. [14]. Even though gradient approaches require only a single forward inference and a single backpropagation to generate the sensitivity heatmap, the output may not be very intuitive and hard to understand because the salient pixels tend to spread over a very large area of the input image. Also as explained in [17], the back-propagation based methods are based on the AI researchers´ intuition of what constitutes a "good" explanation. But if the focus is on explaining decision to a human observer, then the approach used to produce the explanation should have a structure that humans accept. As a result in most real world use cases such as in medical imaging, practitioners tend to use occlusion experiments as the preferred approach for explanations despite being time consuming, as they produce high quality fine grained sensitivity heatmaps [13].

Over the years there has been several modifications proposed to the original occlusion experiment approach. More recently Zintgraf. et. al. [18] proposed a variation to the original occlusion experiment approach named *Prediction Difference Analysis*. In their method instead of masking with a black or gray patch, samples from surrounding regions in the image are chosen as occlusion patches. In our work we mainly focus on the original occlusion experiment method. But, the methods and optimizations proposed in our work are readily applicable to more advanced occlusion based explainability approaches.

Table 1: Symbols used in the Preliminaries Section

Symbol	Meaning
$\mathcal{I}(\mathcal{I}_{img})$	Input activation volume (Input Image)
$\mathcal{O}$	Output activation volume
C_I, H_I, W_I	Depth, height, and width of Input
C_O, H_O, W_O	Depth, height, and width of Output
$\mathcal{K}_{conv}$	Convolution filter kernels
$\mathcal{B}_{conv}$	Convolution bias value vector
$\mathcal{K}_{pool}$	Pooling filter kernel
$H_{\mathcal{K}}, W_{\mathcal{K}}$	Height and width of filter kernel
$S(S_x, S_y)$	Filter kernel or occlusion patch striding amount (S_x and S_y corresponds to width and height dimensions)
$P(P_x, P_y)$	Padding amount (P_x and P_y corresponds to padding along width and height dimensions)
$\mathcal{P}$	Occluding patch
$\mathcal{I}_{img}^{x,y}$	Modified image by superimposing $\mathcal{P}$ on top of $\mathcal{I}_{img}$ such that the top left corner of $\mathcal{P}$ is positioned at x, y location of $\mathcal{I}_{img}$
M	Heat map produced by the occlusion experiment
H_M, W_M	Height and width of M
f	Fine-tuned CNN
C	Class label predicted by f for the original image $\mathcal{I}_{img}$
$\circ_{x,y}$	Superimposition operator. $A \circ_{x,y} B$, superimposes B on top of A starting off at (x, y) position

3 PRELIMINARIES AND OVERVIEW

In this section first we formally state the problem and explain our assumptions. Then we formalize the internals of some of the layers in a Deep CNN for the purpose of proposing our incremental inference approach in Section 4. Finally we briefly explain the Structural Similarity Index (SSIM) which is used to quantify the quality of the generated sensitivity heatmaps.

3.1 Problem Statement and Assumptions

We are given a fine-tuned CNN f, an image $\mathcal{I}_{img}$ on which the occlusion experiment needs to be run, the predicted class label C for the image $\mathcal{I}_{img}$, an occluding patch $\mathcal{P}$ in

RGB format, and patch striding amount S which will be used to move the occlusion patch on the image at a time in horizontal and vertical directions. The occlusion experiment workload is to generate a 2D heat map M where each individual value in M corresponds to the predicted probability for C by f when $\mathcal{P}$ is superimposed on $\mathcal{I}_{img}$ corresponding to the position of that particular value. More precisely, we can state the workload using the following set of logical statements:

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

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

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

$$\forall\, x, y \in [0, H_M] \times [0, W_M] : \quad (4)$$

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

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

Step (1), and (2) calculates the dimensions of the generated heat map M which is dependent on the dimensions of $\mathcal{I}_{img}$, P, and S. 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 $\mathcal{I}_{img}^{x,y}$ using f and taking the predicted probability for the label C.

We assume that f is a CNN from a roaster of well-known CNNs (currently, VGG 16 layer version, ResNet 18 layer version, and Inception version 3). This is a reasonable start, since most recent CNN based image recognition applications use only such well-known CNNs from model zoos [19, 20]. Nevertheless, our work is orthogonal to the specifics of a particular architecture and the proposed approaches can be easily extended to any architecture. We leave support for arbitrary CNNs to future work.

3.2 Deep CNN Internals

Input and output of individual layers in a Deep CNN except for fully-connected ones are arranged into three dimensional volumes which has a width, height, and depth. For example an RGB input image of 224×224 spatial size can be considered as an input volume having a width and height of 224 and a depth of 3 (corresponding to 3 color channels). Every non fully-connected layer will take in an input volume and transform it into another volume. A fully-connected layer will transform an input volume into an output vector. For our purpose these transformations can be broadly divided into three categories based on how they operate spatially:

- Transformations that operate on individual (point) spatial locations.
  - E.g. ReLU, Batch Normalization
- Transformations that operate on a local spatial context.
  - E.g. Convolution, Pooling
- Transformation that operate on the global spatial context.
  - E.g. Fully-Connected

With incremental spatially localized updates in the input, such placing an occlusion patch, both types of transformations that operate on individual spatial locations and transformations that operate on a local spatial contexts provide opportunities for exploiting redundancy. Extending the transformations that operate on individual spatial locations to become redundancy aware is straightforward. However, with transformations that operate on a local spatial context, such as Convolution and Pooling, this extension is non-trivial due to the overlapping nature of the spatial contexts of individual transformations. A simplified representation of how different types of layers would propagate a spatially localized change is shown in Figure 2. We next formally define the transformations of Convolution and Pooling layers and also the relationship between input and output dimensions. These will be later used in Section 4 to introduce our incremental inference approach.

Transformations that operate on a local spatial context. The two types of transformations that operate on a local spatial context in a deep CNN are Convolution and Pooling layers. Each convolutional layer can have several (say C_O) three dimensional filter kernels organized into a four dimensional array $\mathcal{K}_{conv}$ with each having a smaller spatial width $W_{\mathcal{K}}$ and height $H_{\mathcal{K}}$ compared to the width $W_{\mathcal{I}}$ and height $H_{\mathcal{I}}$ of the input volume $\mathcal{I}$, but has the same depth $C_{\mathcal{I}}$. During inference, each filter kernel is slided along the width and height dimensions of the input and a two dimensional activation map is produced by taking element-wise product between the kernel and the input and adding a bias value $\mathcal{B}[c]$ for some c ∈ $[0, C_O - 1]$ (see Figure 2). These two dimensional activation maps are then stacked together along the depth dimension to produce an output volume $\mathcal{O}$ having the dimensions of (C_O,H_O,W_O). This can be formally defined as follows:

$$\text{Input Volume} : \mathcal{I} \in \mathbb{R}^{C_{\mathcal{I}} \times H_{\mathcal{I}} \times W_{\mathcal{I}}} \quad (7)$$

$$\text{Conv. Filters} : \mathcal{K}_{conv} \in \mathbb{R}^{C_O \times C_{\mathcal{I}} \times H_{\mathcal{K}} \times W_{\mathcal{K}}} \quad (8)$$

$$\text{Conv. Bias Vector} : \mathcal{B}_{conv} \in \mathbb{R}^{C_O} \quad (9)$$

$$\text{Output Volume} : \mathcal{O} \in \mathbb{R}^{C_O \times H_O \times W_O} \quad (10)$$

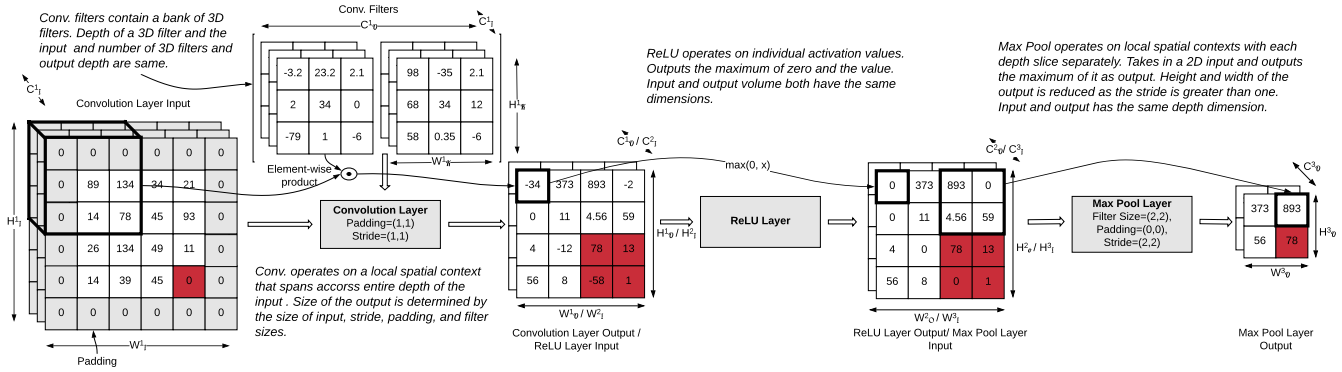

Figure 2: Simplified representation of selected layers of a Deep CNN. For simplicity sake addition of bias is not shown in the Conv. transformation. The values marked in red shows how a small spatial update in the first input would propagate through subsequent transformations. Notation used is explained in Table 1.

$$O[c, y, x] = \sum_{k=0}^{C_I} \sum_{j=0}^{H_\mathcal{K}-1} \sum_{i=0}^{W_\mathcal{K}-1} \mathcal{K}_{conv}[c, k, j, i] \times \mathcal{I}[k, y - \lfloor \frac{H_\mathcal{K}}{2} \rfloor + j, x - \lfloor \frac{W_\mathcal{K}}{2} \rfloor + i] + \mathcal{B}[c] \tag{11}$$

Pooling which reduces the spatial size of an activation volume can also be thought as a Convolution operation with a fixed (i.e. not learned) two dimensional filter kernel $\mathcal{K}_{pool}$. But unlike Convolution, Pooling operates independently on each depth slice of the input volume. A Pooling layer takes a three dimensional activation volume O having a depth of C, width of W_I, and height of H_I as input and produces another three dimensional activation volume O which has the same depth of C, width of W_O, and height of H_O as the output.

Relationship between Input and Output Spatial Sizes. The output volume's spatial sizes (W_O and H_O) are determined by the spatial sizes of the input volume (W_I and H_I), spatial sizes of the filter kernel ($W_\mathcal{K}$ and $H_\mathcal{K}$) and two other parameters: **stride** S and **padding** P. Stride is the amount of pixel values used to slide the filter kernel at a time when producing a two dimensional activation map. It is possible to have two different values with one for the width dimension S_x and one for the height dimension S_y. Generally $S_x \leq W_\mathcal{K}$ and $S_y \leq H_\mathcal{K}$. Sometimes in order to control the spatial size of the output activation map to be same as the input activation map, one needs to pad the input feature map with zeros around the spatial border. Padding (P) captures the amount of zeros that needs to be added. Similar to the stride S, it is possible to have two separate values for padding with one for the width dimension P_x and one for the height dimension P_y. With these parameters defined the width and the height of the output activation volume can be defined as follows:

$$W_O = (W_I - W_\mathcal{K} + 1 + 2 \times P_x)/S_x$$
$$H_O = (H_I - H_\mathcal{K} + 1 + 2 \times P_y)/S_y \tag{12}$$

Estimating the Computational Cost of Deep CNNs. Deep CNNs are highly compute intensive and out of the different types of layers Conv. layers contributes to 90% (or more) of the computations. One of the widely used ways to estimate the computational cost of a Deep CNN is to estimate the number of fused multiply add (FMA) floating point operations (FLOPs) required by convolution layers for a single forward inference.

For example, applying a convolution filter having the dimensions of (C^l_{in}, $H^l_\mathcal{K}$, $W^l_\mathcal{K}$) to a single spatial context will require $C^l_I \times H^l_\mathcal{K} \times W^l_\mathcal{K}$ many 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 O having dimensions $C^l_O \times H^l_O \times W^l_O$, 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 13 and Equation 14. However, in the case of incremental updates only a smaller spatial patch having a width $W^l_\mathcal{P}$ ($W^l_\mathcal{P}$ <= W^l_O) and height $H^l_\mathcal{P}$ ($H^l_\mathcal{P}$ <= H^l_O) is needed to be recomputed. The amount of computations required for the incremental computation Q^l_{inc} and total amount of incremental computations Q_{inc} required for the entire set of convolution layers L will be smaller than the above full computation values and can be calculated as per Equation 15 and Equation 16.

Based on the above quantities we define a new metric named **theoretical speedup (R)**, which is the ratio between total full computational cost Q and total incremental

computation cost Q_{inc} (see Equation 17). 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.

$$Q^l = (C^l_{\mathcal{I}} \times H^l_{\mathcal{K}} \times W^l_{\mathcal{K}}) \times (C^l_{\mathcal{O}} \times H^l_{\mathcal{O}} \times W^l_{\mathcal{O}}) \tag{13}$$

$$Q = \sum_{l \in L} Q^l \tag{14}$$

$$Q^l_{inc} = (C^l_{\mathcal{I}} \times H^l_{\mathcal{K}} \times W^l_{\mathcal{K}}) \times (C^l_{\mathcal{O}} \times H^l_{\mathcal{P}} \times W^l_{\mathcal{P}}) \tag{15}$$

$$Q_{inc} = \sum_{l \in L} Q^l_{inc} \tag{16}$$

$$R = \frac{Q}{Q_{inc}} \tag{17}$$

3.3 Estimating the Quality of Generated Approximate Heat Maps

When applying approximate inference optimizations, Krypton sacrifices the the accuracy/quality of the generated heat map in favor of faster execution. To measure this drop of accuracy we use Structural Similarity (SSIM) Index [21], 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 Krypton generated 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 above $0.95 - 0.80$ are used in practical applications such as image compression and video encoding as at the human perception level they produce indistinguishable distortions. For more details on SSIM Index method, we refer the reader to the original SSIM Index paper [21].

4 OPTIMIZATIONS

In this section we explain *incremental inference* and the two *approximate inference* approaches, *projective field thresholding* and *adaptive drill-down* in detail. In Krypton these optimizations are applied on top of the current dominant approach of performing CNN inference on batches of images where each image corresponds to an occluded instance of the original image. Batched inference is important as it reduces 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 on CPU environments and 2X speedups on the GPU environment compared to the per image inference approach. Finally we explain how Krypton configures it's internal system configurations for *approximate inference*.

Table 2: Additional symbols used in the Optimizations Section

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

4.1 Incremental Inference

As explained earlier, occlusion experiments in it's naive form performs 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 on transformations that operate on a local spatial context (i.e. Convolution and Pooling) as other types either has no redundancies (global context transformations) or is trivial to make incremental (point transformations).

Incremental Convolution and Pooling.

In Section 3 we explained that both convolution and polling transformations can be cast into a form of applying a filter along the spatial dimensions of the input volume. However, how each transformation operate along the depth dimension is different. For our purpose we are interested in finding the propagation of the patches in the input through the consecutive layers and hence both these transformations can be treated similarly. The coordinates and the dimensions (i.e. height and width) of the modified patch in the output volume caused by a modified patch in the input volume are determined by the coordinates and the dimensions of the input patch, sizes of the filter kernel ($H_{\mathcal{K}}$ and $W_{\mathcal{K}}$), padding values (P_x and P_y), and the strides (S_x and S_y). For example consider simplified demonstration showing a cross-section of input and output in Figure 3. 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 off at $x^{\mathcal{I}}_{\mathcal{P}}, y^{\mathcal{I}}_{\mathcal{P}}$ coordinates and has a height of $H^{\mathcal{I}}_{\mathcal{P}}$ and width of $W^{\mathcal{I}}_{\mathcal{P}}$. The updated patch in the output starts off at $x^{\mathcal{O}}_{\mathcal{P}}, y^{\mathcal{O}}_{\mathcal{P}}$ and has a height of $H^{\mathcal{O}}_{\mathcal{P}}$ and width of

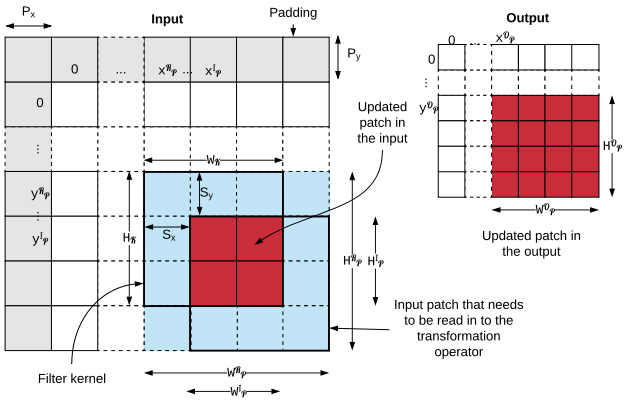

Figure 3: Simplified representation of input and output patch coordinates and dimensions of Conv. and Pool transformations.

$W^O_{\mathcal{P}}$. 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 3. The starting coordinates of this read-in patch are denoted by $x^{\mathcal{R}}_{\mathcal{P}}, y^{\mathcal{R}}_{\mathcal{P}}$ and the dimensions are denoted by $W^{\mathcal{R}}_{\mathcal{P}}, H^{\mathcal{R}}_{\mathcal{P}}$. The relationship between the coordinates and dimensions can be expressed as follows:

$$x^O_{\mathcal{P}} = max\Big(\lceil (P_x + x^{\mathcal{I}}_{\mathcal{P}} - W_{\mathcal{K}} + 1)/S_x \rceil, 0\Big) \tag{18}$$

$$y^O_{\mathcal{P}} = max\Big(\lceil (P_y + y^{\mathcal{I}}_{\mathcal{P}} - H_{\mathcal{K}} + 1)/S_y \rceil, 0\Big) \tag{19}$$

$$W^O_{\mathcal{P}} = min\Big(\lceil (W^{\mathcal{I}}_{\mathcal{P}} + W_{\mathcal{K}} - 1)/S_x \rceil, W_O\Big) \tag{20}$$

$$H^O_{\mathcal{P}} = min\Big(\lceil (H^{\mathcal{I}}_{\mathcal{P}} + H_{\mathcal{K}} - 1)/S_y \rceil, H_O\Big) \tag{21}$$

$$x^{\mathcal{R}}_{\mathcal{P}} = x^O_{\mathcal{P}} \times S_x - P_x \tag{22}$$

$$y^{\mathcal{R}}_{\mathcal{P}} = y^O_{\mathcal{P}} \times S_y - P_y \tag{23}$$

$$W^{\mathcal{R}}_{\mathcal{P}} = W_{\mathcal{K}} + (W^O_{\mathcal{P}} - 1) \times S_x \tag{24}$$

$$H^{\mathcal{R}}_{\mathcal{P}} = H_{\mathcal{K}} + (H^O_{\mathcal{P}} - 1) \times S_y \tag{25}$$

Equation 18 and 19 calculates the starting coordinates of the output patch. Use of padding effectively shifts the coordinate system and therefore P_x and P_y values are added to correct it. Due to the overlapping nature of filter kernels, the maximum affected span of the updated patch in the input will be increased by $W_{\mathcal{K}} - 1$ and $H_{\mathcal{K}} - 1$ amounts and hence needs to be subtracted from the input coordinates $x^{\mathcal{I}}_{\mathcal{P}}$ and $y^{\mathcal{I}}_{\mathcal{P}}$ (a filter of size $W_{\mathcal{K}}$ which is placed starting at $x^{\mathcal{I}}_{\mathcal{P}} - W_{\mathcal{K}} + 1$ will see the new change at $x^{\mathcal{I}}_{\mathcal{P}}$). Dividing the above values by the stride values S_x and S_y and taking the `ceil` gives the starting coordinates of the output patch (essentially calculates the number of strides). Towards the left side edge of the input, where the affected span of the input cannot be extended by $W_{\mathcal{K}} - 1$ or $H_{\mathcal{K}} - 1$ amounts this value will be negative. Therefore the maximum of zero or the above value should be taken as the final. Equation 20 and 21 calculates the width and height of the output patches. Similar to output coordinates calculations, the span of the input patch is increased by $W_{\mathcal{K}} - 1$ and $H_{\mathcal{K}} - 1$ amounts. Dividing the above values by the stride values S_x and S_y and taking the `ceil` gives the width and height of the output patch. Since the output patch cannot grow beyond the size of the output, minimum of the output dimension or the above value should be taken as the final. Once the output patch coordinates and dimensions are calculated, it is straight forward to calculate the read-in patch coordinates as per Equations 22 and 23 and the dimensions as per Equations 24 and 25.

With all the geometric mappings defined, we now explain the complete incremental inference approach for a single transformation. Algorithm 1 presents it formally. The IncrementalInference procedure takes in the original transformation T, pre-materialized input for T corresponding to original image, a batch of updated patches and their geometric properties as input. First it calculates geometric properties of the output and read-in patches. A temporary input volume R is initialized to hold the input patches with their read-in contexts. The for loop iteratively populates R with corresponding patches. Finally T is applied on R to compute the output patches. In a CNN which has multiple such transformations chained together, the outputs of the IncrementalInference procedure are fed as input again for the incremental inference of the next transformation along with the unchanged pre-materialized input corresponding to the new transformation. However at a boundary of local context transformation and a global context transformation, such as in Conv. → Fully-Connected or Pool → Fully-Connected, full updated output has to be created as per IncrementalToFullProjection procedure instead of propagating only the updated patches. The high-level steps taken by the end-to-end incremental inference approach can be summarized as follows:

(1) Take in CNN f, image $\mathcal{I}_{img}$, predicted class label C, occlusion patch $\mathcal{P}$, and stride S for the $\mathcal{P}$ as input.
(2) Pre-materialize output of all the transformations in f by feeding in $\mathcal{I}_{img}$.
(3) Calculate all possible $\mathcal{P}$ positions on $\mathcal{I}_{img}$.
(4) Prepare the occluded images ($\mathcal{I}^{x,y}_{img}$ s) corresponding to all positions.

Algorithm 1 Incremental Inference Transformation

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

Output:
$[\mathcal{P}^{\mathcal{O}}{}_1, ..., \mathcal{P}^{\mathcal{O}}{}_n]$: *Output patches*
$[(x^{\mathcal{O}}_{\mathcal{P}_1}, y^{\mathcal{O}}_{\mathcal{P}_1}), ..., (x^{\mathcal{O}}_{\mathcal{P}_n}, y^{\mathcal{O}}_{\mathcal{P}_n})]$: *Output patch coordinates*
$W^{\mathcal{O}}_{\mathcal{P}}, H^{\mathcal{O}}_{\mathcal{P}}$: *Output patch dimensions*

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

Input:
$\mathcal{O}$: *Pre-materialized output from original image*
$[\mathcal{P}^{\mathcal{O}}_1, ..., \mathcal{P}^{\mathcal{O}}_n]$: *Output patches*
$[(x^{\mathcal{O}}_{\mathcal{P}_1}, y^{\mathcal{O}}_{\mathcal{P}_1}), ..., (x^{\mathcal{O}}_{\mathcal{P}_n}, y^{\mathcal{O}}_{\mathcal{P}_n})]$: *Output patch coordinates*

Output:
O' : *Updated output*

1: **procedure** IncrementalToFullProjection
2: *Initialize* $\mathcal{O}' \in \mathbb{R}^{n \times \mathtt{depth}(\mathcal{O}) \times \mathtt{height}(\mathcal{P}^{\mathcal{O}}_1) \times \mathtt{width}(\mathcal{P}^{\mathcal{O}}_1)}$
3: **for** `i in [1,...,n]` **do**
4: $T \leftarrow \mathtt{copy}(\mathcal{O})$
5: $\mathcal{O}'[i, :, :] \leftarrow \mathcal{P}^{\mathcal{O}}_i \circ_{x^{\mathcal{O}}_{\mathcal{P}_i}, y^{\mathcal{O}}_{\mathcal{P}_i}} T$
6: **return** $\mathcal{O}'$

(5) For batches of $\mathcal{I}^{x,y}_{img}$ as the input traverse the transformations in f in a topological order and calculate the corresponding values of heatmap M.
- For local context transformation invoke IncrementalInference.
- For local context transformation that feeds in a global context transformation additionally invoke IncrementalToFullProjection
- For all others invoke the original transformation.

(6) Return M as the final output.

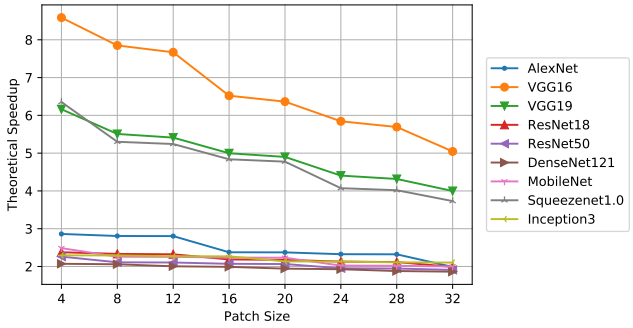

Figure 4: Theoretical speedup for popular CNN architectures with *incremental inference* when a square occlusion patch is placed on the center of the image.

We analyze the theoretical speedup that can be achieved with *incremental inference* approach when a square occlusion patch is placed on the center[1] of the input image. Figure 4 shows the results. VGG16 model results in the maximum theoretical speedup and DenseNet121 model has the lowest theoretical speedup. Most CNN architectures results in a theoretical speedup between 2-3. The theoretical speedup for a CNN with *incremental inference* is determined by the characteristics of it's architecture such as number of layers, the sizes of the filter kernels, and the filter stride values.

CPU versus GPU implementation concerns. Through our experiments we found that even though a straightforward implementation of *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 5 of Algorithm 1 is essentially preparing the input for T by copying values from one part of the memory to another sequentially. This sequential operation becomes a bottleneck for the GPU implementation as 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.

Element-wise addition and depth-wise concatenation. Element-wise addition and depth-wise concatenation are two widely used linear algebra operators in CNNs. Element-wise addition operator requires the two input volumes to have exact same dimensions and the depth-wise concatenation requires them to have same height and width dimensions. Consider a situation for these operators as shown in

[1]If the occlusion patch is placed towards to 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.

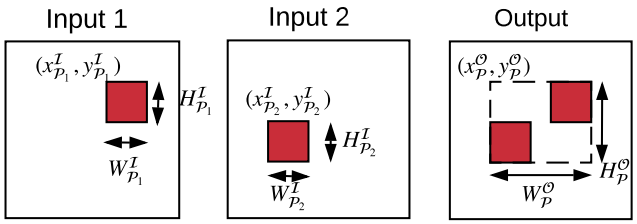

Figure 5: Input-Output coordinate and dimension mapping for element-wise addition and depth-wise concatenation.

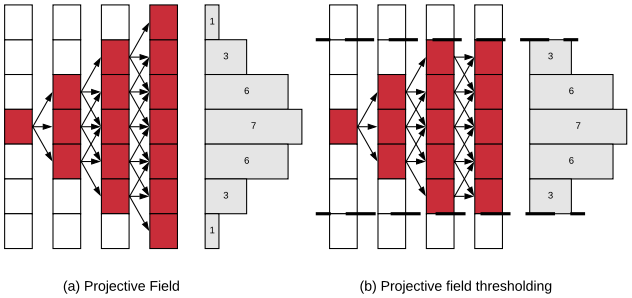

Figure 6: (a) One dimensional Convolution demonstrating projective field growth (filter size = 2, stride = 1). (b) *Projective field thresholding* with $\tau = 5/7$. Histograms denote the number of unique change propagation paths.

Figure 5 where the first input has incremental spatial update starting at $x^{\mathcal{I}}_{\mathcal{P}_1}, y^{\mathcal{I}}_{\mathcal{P}_1}$ coordinates with dimensions of $H^{\mathcal{I}}_{\mathcal{P}_1}$ and $W^{\mathcal{I}}_{\mathcal{P}_1}$ and for the second input starting at $x^{\mathcal{I}}_{\mathcal{P}_2}, y^{\mathcal{I}}_{\mathcal{P}_2}$ coordinates with dimensions of $H^{\mathcal{I}}_{\mathcal{P}_2}$ and $W^{\mathcal{I}}_{\mathcal{P}_2}$. Then computing the coordinates and the dimensions of the output and read-in patches is essentially finding the bounding box for the two patches and can be expressed as follows:

$$x^{\mathcal{O}}_{\mathcal{P}} = x^{\mathcal{R}}_{\mathcal{P}} = \texttt{min}(x^{\mathcal{I}}_{\mathcal{P}_1}, x^{\mathcal{I}}_{\mathcal{P}_2}) \tag{26}$$

$$y^{\mathcal{O}}_{\mathcal{P}} = y^{\mathcal{R}}_{\mathcal{P}} = \texttt{min}(y^{\mathcal{I}}_{\mathcal{P}_1}, y^{\mathcal{I}}_{\mathcal{P}_2}) \tag{27}$$

$$W^{\mathcal{O}}_{\mathcal{P}} = W^{\mathcal{R}}_{\mathcal{P}} = \texttt{max}(x^{\mathcal{I}}_{\mathcal{P}_1} + W^{\mathcal{I}}_{\mathcal{P}_1}, x^{\mathcal{I}}_{\mathcal{P}_2} + W^{\mathcal{I}}_{\mathcal{P}_2}) - \texttt{min}(x^{\mathcal{I}}_{\mathcal{P}_1}, x^{\mathcal{I}}_{\mathcal{P}_2}) \tag{28}$$

$$H^{\mathcal{O}}_{\mathcal{P}} = H^{\mathcal{R}}_{\mathcal{P}} = \texttt{max}(y^{\mathcal{I}}_{\mathcal{P}_1} + H^{\mathcal{I}}_{\mathcal{P}_1}, y^{\mathcal{I}}_{\mathcal{P}_2} + H^{\mathcal{I}}_{\mathcal{P}_2}) - \texttt{min}(y^{\mathcal{I}}_{\mathcal{P}_1}, y^{\mathcal{I}}_{\mathcal{P}_2}) \tag{29}$$

4.2 Projective Field Thresholding

Projective field [22, 23] of a CNN neuron is the local region (including the depth) of the output volume which is connected to it. The term is borrowed from 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 [24]. For our work the notion of projective field is useful as it essentially determines the change propagation path for incremental changes. The three types of CNN transformations affects the size of the projective field differently. Point transformations does not change the projective field size while global context transformations increases 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 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 much 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 truncated up to a certain extent without significantly affecting the accuracy. For a more intuitive understanding on why this would work consider the simplified one dimensional 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. At the end of the fourth layer, the histogram shows the number of unique paths that are available between each output neuron and the original 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 actual change in the output layer is determined by both the number of unique paths and also the individual weights of the connections. But the actual change in the output will converge to a Gaussian in distribution over all possible weight values. For more details we refer the reader to [25], where a similar theoretical result has been proved for the receptive field[2] of a deep CNN.

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 essentially 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 will be maintained by this approach.

[2]Receptive field of a CNN neuron is the local region (including the depth) of the input volume which is connected to it.

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, the new set of calculations can be expressed as follows:

$$W_{\mathcal{P}}^{O} = \texttt{min}\Big(\lceil (W_{\mathcal{P}}^{I} + W_{\mathcal{K}} - 1)/S_x \rceil, W_{\mathcal{P}}^{O}\Big) \tag{30}$$

$$\text{If } W_{\mathcal{P}}^{O} > \texttt{round}(\tau \times W^{O}) : \tag{31}$$

$$W_{\mathcal{P}}^{O} = \texttt{round}(\tau \times W^{O}) \tag{32}$$

$$W_{\mathcal{P}_{new}}^{I} = W_{\mathcal{P}}^{O} \times S_x - W_{\mathcal{K}} + 1 \tag{33}$$

$$x_{\mathcal{P}}^{I} \mathrel{+}= (W_{\mathcal{P}}^{I} - W_{\mathcal{P}_{new}}^{I})/2 \tag{34}$$

$$W_{\mathcal{P}}^{I} = W_{\mathcal{P}_{new}}^{I} \tag{35}$$

$$x_{\mathcal{P}}^{O} = \texttt{max}\Big(\lceil (P_x + x_{\mathcal{P}}^{I} - W_{\mathcal{K}} + 1)/S_x \rceil, 0\Big) \tag{36}$$

Equation 30 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 32. Equation 33 calculates the input width that would produce an output of width $W_{\mathcal{P}}^{O}$ (think of this as making $W_{\mathcal{P}}^{I}$ the subject of equation 30). If the new input width is smaller than the original input width, the starting x coordinate should be updated as per Equation 34 such that the updated coordinates correspond to a center crop from the original. Equation 35 set the input width to the newly calculated input width and Equation 36 calculates the x coordinate of the output patch from the updated values. Coordinates and dimensions of the vertical dimension can also be computed similarly.

4.3 Adaptive Drill-Down

Adaptive drill-down approach 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 heatmap will be generated using a two stage process. At the first stage a low resolution heatmap will be generated by using a larger stride which we call stage one stride (S_1). From the heatmap generated at 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 9.

The amount of speedup that can be obtained from *adaptive drill-down* is determined by both $r_{drill-down}$ and S_1. If

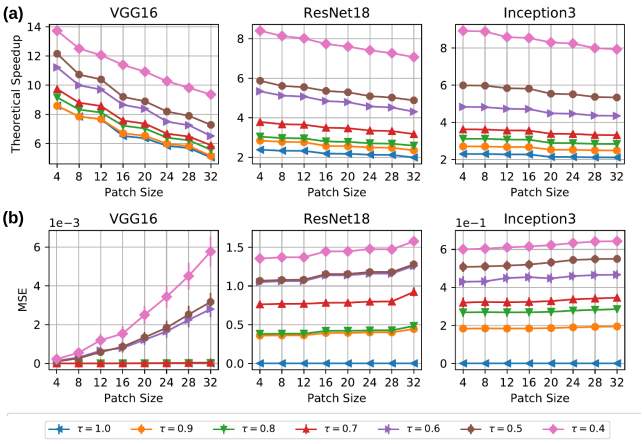

Figure 7: (a) Theoretical speedup ratio with projective field thresholding for different occlusion patch sizes and CNN models. (b) Mean Square Error between exact and approximate output of the final activation volume produce by a conv. or a pool layer for different occlusion patch sizes and CNN models on a sample (n=30) of OCT dataset. In both (a) and (b) occlusion patches are placed at the center of the image.

$r_{drill-down}$ is low, only a small region has to be examined at a higher resolution and thus 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 loosing 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 mis-identifying interesting regions specially when the granularity of those regions is 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* approach. 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 37. Figure 10 shows conceptually how the speedup would vary with S_1 when $r_{drill-down}$ is fixed and with $r_{drill-down}$ when S_1 is fixed.

$$\text{speedup} = \frac{S_1^2}{S_2^2 + r_{drill-down} \times S_1^2} \tag{37}$$

4.4 System Tuning

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

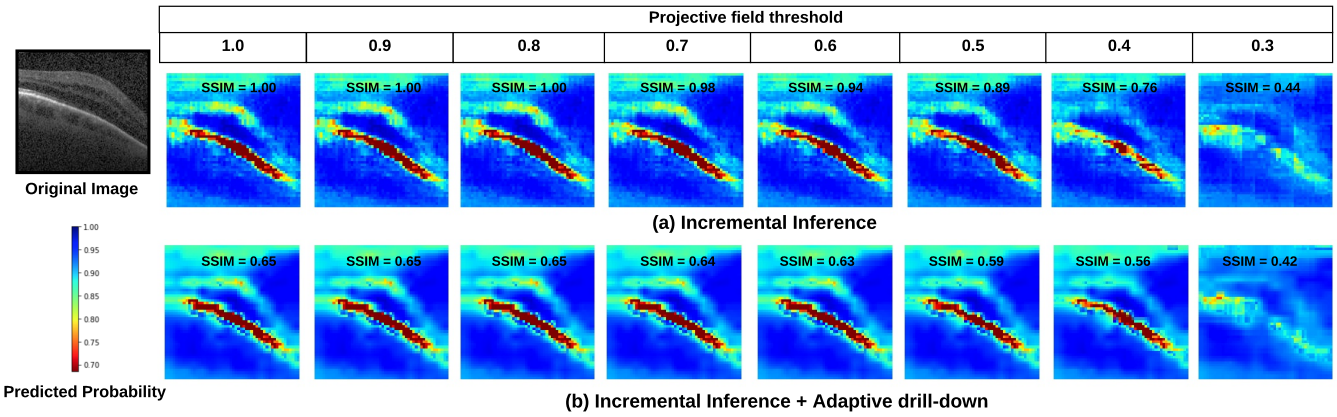

Figure 8: Occlusion heatmaps from a sample OCT image with (a) *incremental inference* and (b) *incremental inference* with *adaptive drill-down* for different *projective field threshold* values (CNN model = VGG16, occlusion patch size = 16, patch color = black, occlusion patch stride (S or S_2) = 4, r_{drill_down} = 0.1, target speedup=5).

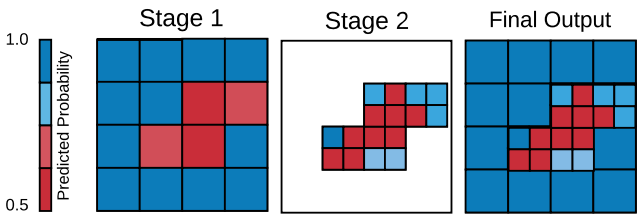

Figure 9: Schematic representation of *adaptive drill-down*

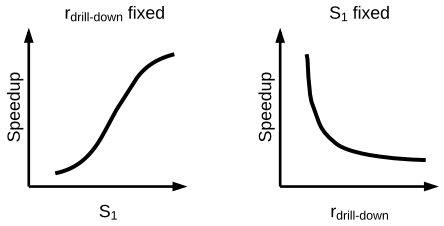

Figure 10: Conceptual diagram showing the effect of S_1 and $r_{drill-down}$ on speedup.

Tuning projective field threshold. Tuning *projective field threshold* (τ) requires 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 heatmap (compared to the exact heatmap) 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 operational time, Krypton requires the user to provide the expected level of quality for the heatmaps in terms of a SSIM value. τ is then selected from the curve fit to match this target SSIM value. Figure 11 (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 11 (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 4.3 the speedup obtained by *adaptive drill-down* approach is determined by two factors, stage one stride value (S_1) and drill-down fraction ($r_{drill-down}$). Figure 12 shows how the SSIM value of approximate heat maps would change when changing $r_{drill-down}$ and S_1 with all other configurations kept fixed. A general trend of increasing SSIM with increasing $r_{drill-down}$ and decreasing S_1 can be observed from the plots. 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, currently 0.25, as $r_{drill-down}$. The speedup value basically captures user's input on how much faster the occlusion experiment

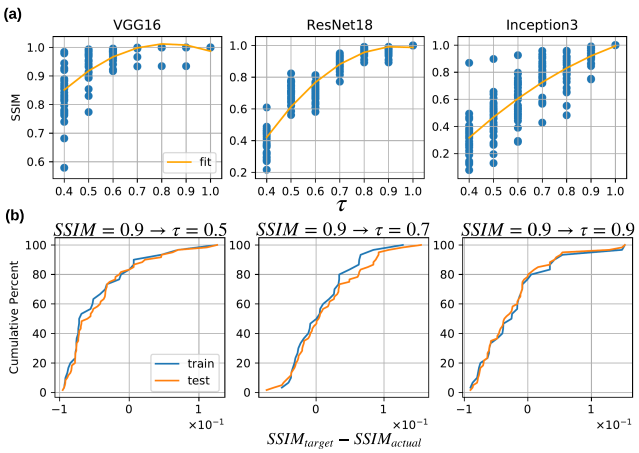

Figure 11: (a) SSIM variation and degree two curve fit for different τ values for a sample of OCT dataset. (b) Cumulative distribution plot for the SSIM deviation for the τ values obtained from the curve fit for a SSIM value of 0.9.

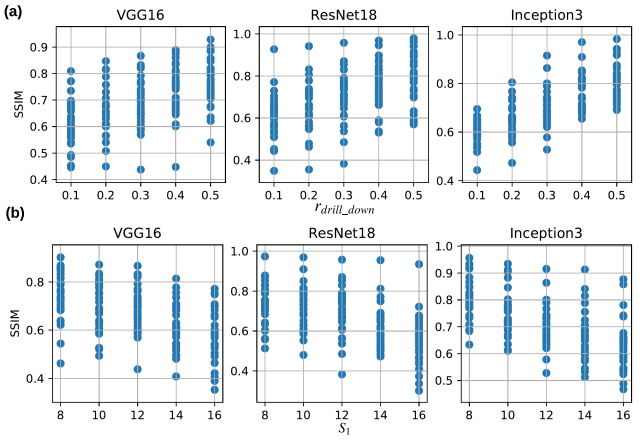

Figure 12: (a) SSIM variation for changing r_{drill_down} fixing $\tau = 1$, $S_1 = 12$, and $S_2 = 4$. (b) SSIM variation for changing S_1 fixing $\tau = 1.0$, $S_2 = 4$, and $r_{drill\_down} = 0.3$ for a sample ($n = 30$) of OCT dataset.

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 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 38. As S_1 cannot be greater than the height (H_{img}) or width (W_{img}) of the image it can be seen that possible values for the speedup value are upper-bounded as per equation 39.

$$S_1 = \sqrt{\frac{\text{speedup}}{1 - r_{drill-down} \times \text{speedup}}} \times S_2 \tag{38}$$

$$S_1 = \sqrt{\frac{\text{speedup}}{1 - r_{drill-down} \times \text{speedup}}} \times S_2 < W_{img}$$
$$\text{speedup} < \frac{W_{img}^2}{S_2^2 + r_{drill-down} \times W_{img}^2} \tag{39}$$

5 EXPERIMENTAL EVALUATION

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

Datasets. We use three real-world datasets: *OCT*, *Chest X-Ray*, and a sample from *ImageNet*. *OCT* has about 84,000 optical coherence tomography retinal images categorized into four categories: CNV, DME, DRUSEN, and NORMAL. CNV (choroidal neovascularization), DME (diabetic macular edema), and DRUSEN are three different varieties of Diabetic Retinopathy. NORMAL corresponds to healthy retinal images. *Chest X-Ray* has about 6,000 X-ray images categorized into three categories: VIRAL, BACTERIAL, and NORMAL. VIRAL and BACTERIAL categories corresponds 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 [26].

Workloads. We use three popular ImageNet-trained deep CNNs: VGG16 [2], ResNet18 [3], and Inception3 [4], obtained from [27]. They complement each other in terms of model size, computational cost, amount of theoretical redundancy that exist for occlusion experiments, and the level of architectural complexity of the CNN model. For *OCT* and *Chest X-Ray* datasets, the three CNN models are fine-tuned by retraining the final fully-connected layer with hyper-parameter tuning as per standard practice. More details on the fine-tuning process are included in the Appendix. Heat map for the predicted probabilities is generated using Python Matplotlib library's `imshow` method using the `jet_r` color scheme. For the heatmap, 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 preprocessing 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. Figure 13 presents the 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 4) 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 there by to reduce the computational and runtime costs. Brining 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 4. The speedup values are calculated compared to the runtime taken by the current dominant practice of performing full inference for batches of modified images.

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 14 shows the results. Empirical-GPU Naive results in the worst performance for all three CNN models. Empirical-GPU and

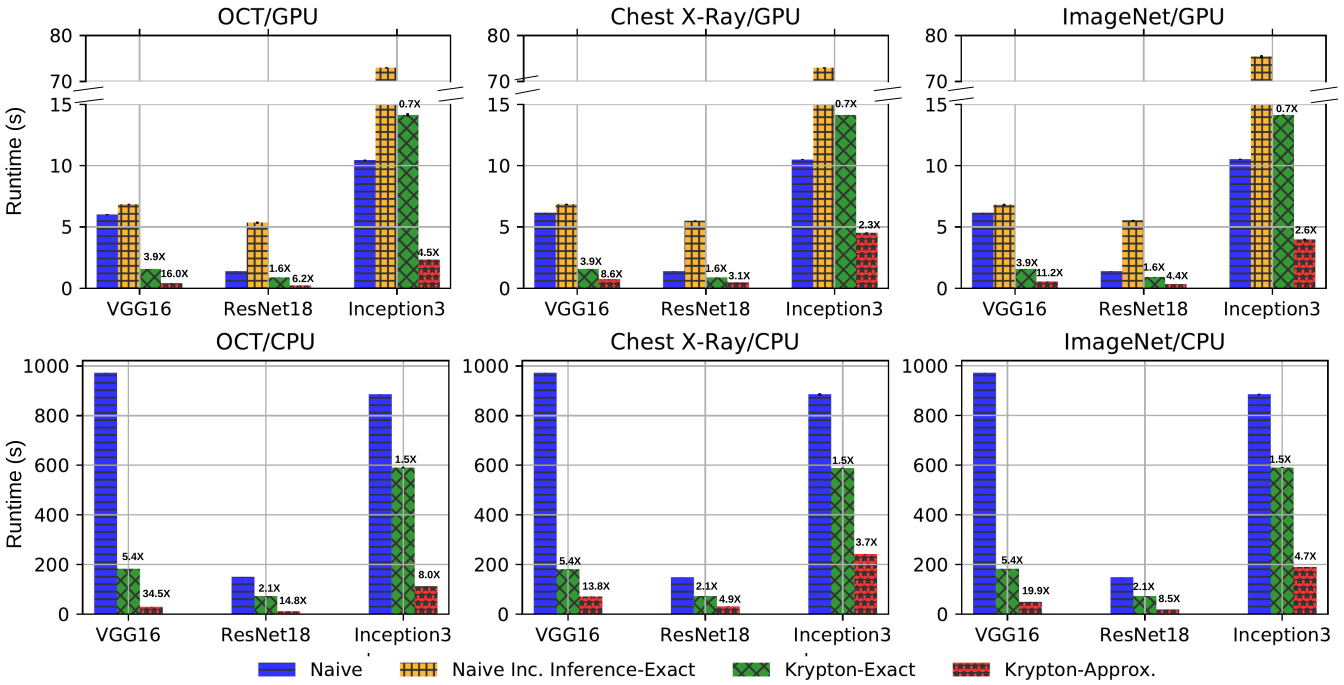

Figure 13: End-to-end efficiency achieved by KRYPTON over naive approaches.

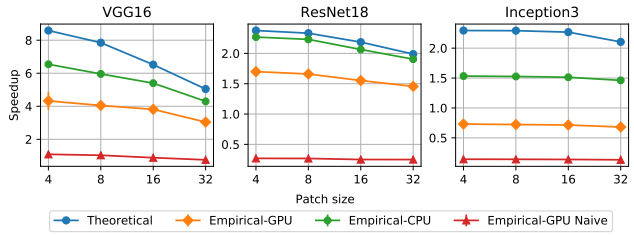

Figure 14: Theoretical versus empirical speedup for *incremental inference* with varying occlusion patch sizes and different CNN models (Occlusion patch stride $S = 4$).

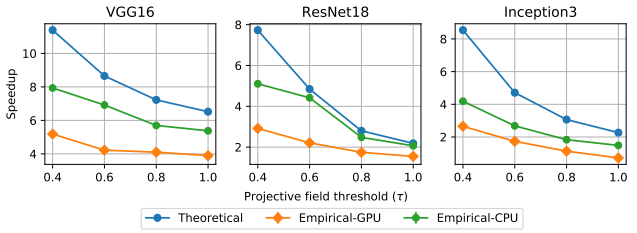

Figure 15: Theoretical versus empirical speedup for *incremental inference* with *projective field thresholding* with varying threshold values (τ) and different CNN models (Occlusion patch size = 16×16, stride $S = 4$).

Empirical-CPU implementations result in higher speedups with Empirical-CPU being closer to the theoretical speedup value. As the occlusion patch size increases the speedups decrease.

Speedups from Projective Field Thresholding.

We vary *projective field threshold* (τ) from 1.0 (no thresholding) to 0.4 and evaluate the speedups. The occlusion patch size used is 16 and the stride is 4. The results are shown in Figure 15. 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 16 (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 similar to the previous case. Figure 16 (b) presents the results. In both cases we see Empirical-GPU and Empirical-CPU achieve higher speedups with Empirical-CPU being very close to

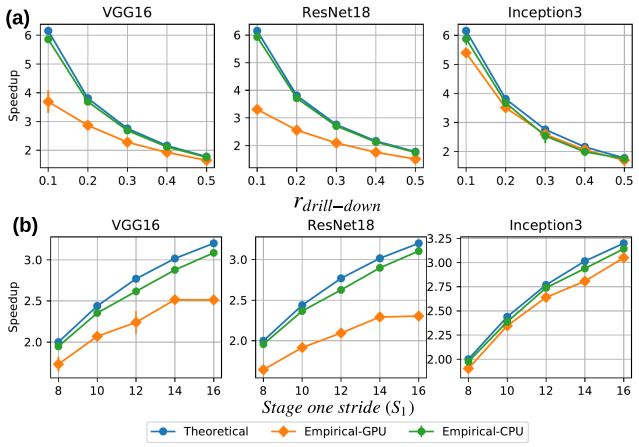

Figure 16: Theoretical versus empirical speedup for *adaptive drill-down* with (a) varying drill-down ratios ($r_{drill-down}$) and (b) varying stage one stride S_1 values for different CNN models (Occlusion patch size = 16×16, stage two stride $S_2 = 4$, projective field threshold $\tau = 1.0$).

the theoretical speedup. On the CPU environment the relative cost of other overheads are 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. Further more 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

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] Forrest N Iandola et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[16] Andrew G Howard et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

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

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

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

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

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

[22] 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.

[23] 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.

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

[25] Wenjie Luo et al. Understanding the effective receptive field in deep convolutional neural networks. In *Advances in neural information*

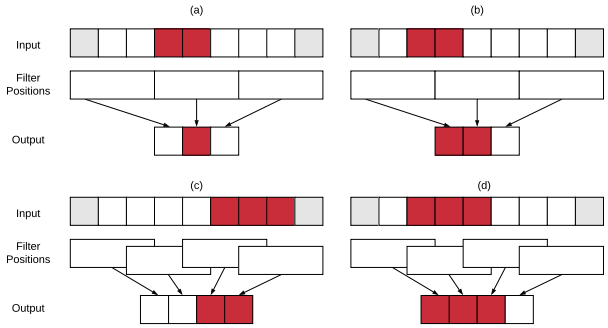

Figure 17: One dimensional representation showing special situations under which actual output size will be smaller than the values calculated by Equations 18 and 19. (a) and (b) shows a situation with filter stride being equal to the filter size. (c) and (d) shows a situation with input patch being placed at the edge of the input.

processing systems, pages 4898–4906, 2016.

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

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

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 4.1. Consider the simplified one dimensional situation shown in Figure 17 (a), where the stride value[3] (3) is same as the filter size (3). In this situation the the size of the output patch is one less than the value calculated by Equation 20. However it is not the case in Figure 17 (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 17 (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 17 (c). In Krypton we do not treat theses differences separately and use the values calculated by Equation 20 and 21 as they act as an upper bound. In case of a smaller output patch, Krypton simply reads off and updates slightly bigger patches to preserve uniformity. This also requires updating the starting coordinates of the patches as shown in Equations 40 and 41. Such uniform treatment is required for performing batched inference operations which out of the box gives significant speedups compared to per image inference.

$$\text{If } x^O_{\mathcal{P}} + W^O_{\mathcal{P}} > W_O:$$
$$\begin{aligned} x^O_{\mathcal{P}} &= W_O - W^O_{\mathcal{P}} \\ x^I_{\mathcal{P}} &= W_I - W^I_{\mathcal{P}} \\ x^{\mathcal{R}}_{\mathcal{P}} &= W_I - W^{\mathcal{R}}_{\mathcal{P}} \end{aligned} \tag{40}$$
$$\text{If } y^O_{\mathcal{P}} + H^O_{\mathcal{P}} > H_O:$$
$$\begin{aligned} y^O_{\mathcal{P}} &= H_O - H^O_{\mathcal{P}} \\ y^I_{\mathcal{P}} &= H_I - H^I_{\mathcal{P}} \\ y^{\mathcal{R}}_{\mathcal{P}} &= H_I - H^{\mathcal{R}}_{\mathcal{P}} \end{aligned} \tag{41}$$

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.

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