Query Optimization for Faster Deep CNN Explanations

Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou
University of California, San Diego
{snakanda,arunkk,yannis}@eng.ucsd.edu

ABSTRACT

Deep Convolutional Neural Networks (CNNs) now match human accuracy in many image prediction tasks, resulting in a growing adoption in e-commerce, radiology, and other domains. Naturally, "explaining" CNN predictions is a key concern for many users. Since the internal workings of CNNs are unintuitive for most users, *occlusion-based explanations* (OBE) are popular for understanding which parts of an image matter most for a prediction. One occludes a region of the image using a patch and moves it around to produce a *heatmap* of changes to the prediction probability. This approach is computationally expensive due to the large number of re-inference requests produced, which wastes time and raises resource costs. We tackle this issue by casting the OBE task as a new instance of the classical incremental view maintenance problem. We create a novel and comprehensive algebraic framework for incremental CNN inference combining materialized views with multi-query optimization to reduce computational costs. We then present two novel approximate inference optimizations that exploit the semantics of CNNs and the OBE task to further reduce runtimes. We prototype our ideas in a tool we call KRYPTON. Experiments with real data and CNNs show that KRYPTON reduces runtimes by up to 5x (resp. 35x) to produce exact (resp. high-quality approximate) results without raising resource requirements.

1. INTRODUCTION

Deep Convolutional Neural Networks (CNNs) are now the state-of-the-art machine learning (ML) method for many image prediction tasks [25]. Thus, there is growing adoption of deep CNNs in many applications across healthcare, domain sciences, enterprises, and Web companies. Remarkably, even the US Food and Drug Administration recently approved the use of deep CNNs to assist radiologists in processing X-rays and other scans, cross-checking their decisions, and even mitigating the shortage of radiologists [1].

©ACM 2019. This is a minor revision of the paper entitled "Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations," published in SIGMOD'19, ISBN 978-1-4503-5643-5/19/06, June 30-July 5, 2019, Amsterdam, Netherlands. DOI: https://doi.org/10.1145/3299869.3319874
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

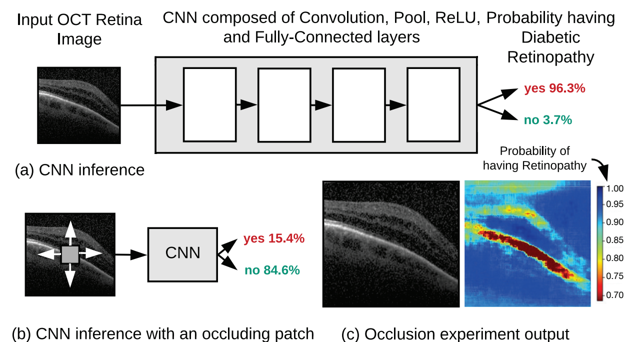

Figure 1: (a) Using a CNN to predict diabetic retinopathy in an OCT image/scan. (b) Occluding a part of the image changes the prediction probability. (c) By moving the occluding patch, a sensitivity heatmap can be produced.

Despite their successes, a key criticism of CNNs is that their internal workings are unintuitive to non-technical users. Thus, users often seek an "explanation" for why a CNN predicted a certain label. Explanations can help users trust CNNs, especially in high stakes applications such as radiology [10], and are a legal requirement for machine learning applications in some countries [27]. How to explain a CNN prediction is still an active research question, but in the practical literature, an already popular mechanism for CNN explanations is a simple procedure called *occlusion-based explanations* [29], or OBE for short.

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

However, OBE is highly computationally expensive. Deep CNN inference is already expensive; OBE just amplifies it by issuing a large number of CNN re-inference requests (even thousands). For example, [31] reports 500,000 re-inference

requests for 1 image, taking 1 hour even on a GPU! Such long wait times can hinder users' ability to consume explanations and reduce their productivity. One could use more compute hardware, if available, since OBE is embarrassingly parallel across re-inference requests. However, this may not always be affordable, especially for domain scientists, or feasible in all settings, e.g., in mobile clinical diagnosis. Extra hardware can also raise monetary costs, especially in the cloud.

To mitigate the above issue, we use a database-inspired lens to formalize and accelerate OBE. We start with a simple but crucial observation: *occluded images are not disjoint but share most of their pixels; so, most of the re-inference computations are redundant.* This observation leads us to connect OBE with two classical data management concerns: *incremental view maintenance* (IVM) and *multi-query optimization* (MQO). Instead of treating a CNN as a "blackbox," we open it up and formalize *CNN layers* as "queries." Just like how a relational query converts relations to other relations, a CNN layer converts *tensors* (multidimensional arrays) to other tensors. A deep CNN stacks many types of such layers to convert the input (represented as a tensor) to the prediction output, as Figure 1(a) illustrates. So, we reimagine OBE as *a set of tensor transformation queries* with incrementally updated inputs. With this fresh database-inspired view, we devise several *novel CNN-specific query optimization techniques* to accelerate OBE.

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

We then introduce two novel *approximate inference* optimizations that allow users to tolerate some degradation in visual quality of the heatmaps produced to reduce runtimes further. These optimizations build upon our incremental inference optimization and use our IVM framework. Our first approximate optimization, *projective field thresholding*, draws upon an idea from neuroscience and exploits the internal semantics of how CNNs work. Our second, *adaptive drill-down*, exploits the semantics of the OBE task and the way users typically consume the heatmaps produced. We also present intuitive automated parameter tuning methods to help users adopt these optimizations. Our optimizations operate largely at the logical level and are complementary to more physical-level optimizations such as low-precision computation and model pruning.

We prototype our ideas in the popular deep learning framework PyTorch to create a tool we call KRYPTON. It works on both CPU and GPU. We perform an empirical evaluation of KRYPTON with multiple CNNs and real-world image datasets from recent radiology and ML papers. KRYPTON yields up to 35x speedups over the current dominant practice of running re-inference with just batching for producing high-quality approximate heatmaps, and up to 5x speedups for producing exact heatmaps.

This paper is a shortened version of our paper titled "Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations" that appeared in ACM SIGMOD 2019 [20]. More details about the techniques discussed in this paper and more experimental results can be found in that SIGMOD paper, as well as in the associated extended version published in ACM TODS [21].

2. SETUP AND PRELIMINARIES

We now state our problem formally and explain our assumptions. We then formalize the dataflow of the layers of a CNN, since these are required for understanding our techniques in Sections 3 and 4. Table 1 lists our notation.

2.1 Problem Statement and Assumptions

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

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

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

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

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

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

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

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

2.2 Dataflow of CNN Layers

CNNs are organized as *layers* of various types, each of which transforms a tensor (multidimensional array, typically 3-D) into another tensor: *Convolution* uses image

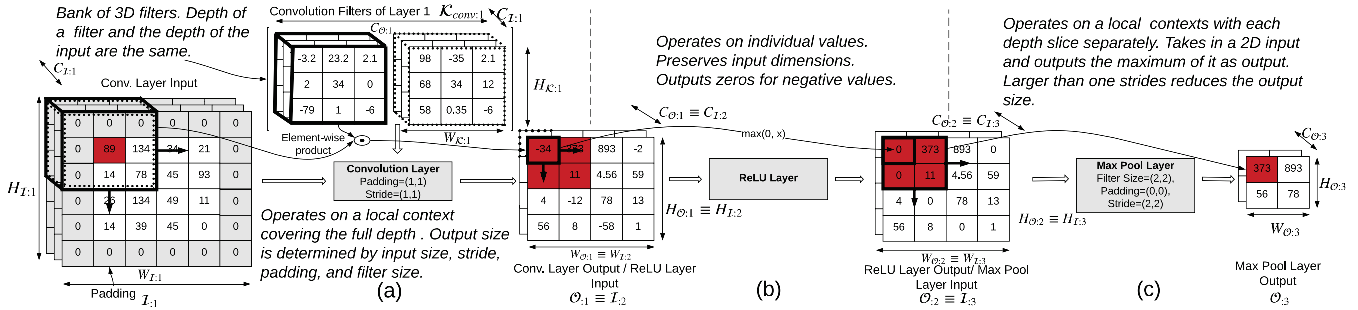

Figure 2: Simplified illustration of the key layers of a typical CNN. The highlighted cells (dark/red background) show how a small local spatial context in the first input propagates through subsequent layers. (a) Convolution layer (for simplicity sake, bias addition is not shown). (b) ReLU Non-linearity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

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

Table 1: Notation used in this paper.

filters from graphics to extract features, but with parametric filter weights (learned during training); *Pooling* subsamples features in a spatial-aware manner; *Batch-Normalization* normalizes the output tensor; *Non-Linearity* applies an element-wise non-linear function (e.g., ReLU); *Fully-Connected* is an ordered collection of perceptrons [9]. The output tensor of a layer can have a different width, height, and/or depth than the input. An image can be viewed as a tensor, e.g., a 224×224 RGB image is a 3-D tensor with width and height 224 and depth 3. A Fully-Connected layer converts a 1-D tensor (or a "flattened" 3-D tensor) to another 1-D tensor. For simplicity of exposition, we group CNN layers into 3 main categories based on the *spatial locality* of how they transform a tensor: (1) Transformations with a *global context*; (2) Transformations at the granularity of *individual elements*; and (3) Transformations at the granularity of a *local spatial context*.

Global context granularity. Such layers convert the input tensor into an output tensor using one global transformation. Since, every element of the output will likely be affected by a point change in the input, such layers do not offer a major opportunity for incremental computations. Fully-Connected is the only layer of this type. They typically arise only as the last layer(s) in deep CNNs (and never in some recent deep CNNs), and typically account for a negligible fraction of the total computational cost.

Individual element granularity. Such layers apply a "map()" function on the elements of the input tensor, as Figure 2 (b) illustrates. Non-Linearity (e.g., ReLU) falls under this category. If the input is incrementally updated, only the corresponding region of the output will be affected. Thus, incremental inference for such layers is straightforward.

Local spatial context granularity. Such layers perform weighted aggregations of slices of the input tensor, called *local spatial contexts*, by multiplying them with a *filter kernel* (a tensor of weights). If the input is incrementally updated, the region of the output that will be affected is not straightforward to ascertain–this requires non-trivial and careful calculations due to the overlapping nature of how filters get applied to local spatial contexts. Both Convolution and Pooling fall under this category. Since such layers typically account for the bulk of the computational cost of deep CNN inference, enabling incremental inference for such layers in the OBE context is a key focus of this paper (Section 3). The rest of this section explains the machinery of the dataflow in such layers using our notation.

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

Dataflow of Pooling Layers. Such layers behave essentially like Convolution layers with a fixed (not learned) 2-D

filter kernel $\mathcal{K}_{pool:l}$. These kernels aggregate a local spatial context to compute its maximum or average element. However, unlike Convolution, Pooling operates on the depth slices of the input tensor independently. Figure 2(c) presents a simplified illustration of this layer. Since OBE only concerns the width and height dimensions of the image and subsequent tensors, we treat both these types of layers in a unified manner for our optimizations.

Relationship between Input and Output Dimensions. For Convolution and Pooling layers, $W_{\mathcal{O}:l}$ and $H_{\mathcal{O}:l}$ are determined by $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$, $W_{\mathcal{K}:l}$ and $H_{\mathcal{K}:l}$, and two other parameters that are specific to that layer: *stride* $S_{:l}$ and *padding* $P_{:l}$. Stride is the number of pixels by which the filter kernel is moved at a time. For some layers, to help control the dimensions of the output to be the same as the input, one "pads" the input with zeros. *Padding* $P_{:l}$ captures how much such padding extends these dimensions. Both stride and padding values can differ along the width and height dimensions; $S_{x:l}$ and $S_{y:l}$ and $P_{x:l}$ and $P_{y:l}$, respectively. In Figure 2, the Convolution layer has $S_{x:l} = S_{y:l} = 1$, while the Pooling layer has $S_{x:l} = S_{y:l} = 2$. Convolution layer also has $P_{x:l} = P_{y:l} = 1$. Given these parameters, width (similarly height) of the output tensor is given by the following formula:

$$W_{\mathcal{O}:l} = (W_{\mathcal{I}:l} - W_{\mathcal{K}:l} + 1 + 2 \times P_{x:l})/S_{x:l} \qquad (7)$$

Computational Cost of Inference. Convolution layers typically account for a bulk of the cost (90% or more). Thus, we can roughly estimate the computational cost of inference by counting the number of *fused multiply-add* (FMA) floating point operations (FLOPs) needed for the Convolution layers. The amount of computations performed by a single application of a Convolution filter kernel $\mathcal{K}_{:l}$ is equal to the volume of the filter in FLOPs, with each FLOP corresponding to one FMA. Thus, the total computational cost $Q_{:l}$ of a layer that produces output $\mathcal{O}_{:l}$ and the total computational cost Q of the entire set of Convolution layers of a given CNN f can be calculated as per Equations (8) and (9).

$$Q_{:l} = (C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{\mathcal{O}:l} \cdot H_{\mathcal{O}:l} \cdot W_{\mathcal{O}:l}) \qquad (8)$$

$$Q = \sum_{l \ in \ f} Q_{:l} \qquad (9)$$

3. INCREMENTAL CNN INFERENCE

In relational IVM, when a part of the input relation is updated, we recompute only the part of the output that changes. We bring this notion to CNNs; a CNN layer is our "query" and a materialized feature tensor is our "relation." OBE updates only a part of the image. So, only some parts of the tensors need to be recomputed. We call this *incremental inference*. We create an algebraic framework to determine which parts of a CNN layer must be updated and how to propagate updates across layers. We then combine our incremental inference framework with an MQO-style technique and characterize theoretical upper bounds on the speedups possible with these ideas.

3.1 Single Layer Incremental Inference

As per the discussion in Section 2.2, we focus only on the non-trivial layers that operate at the granularity of a local spatial context (Convolution and Pooling). Table 2 lists some extra notation for this section.

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

Table 2: Additional notation for Sections 3 and 4.

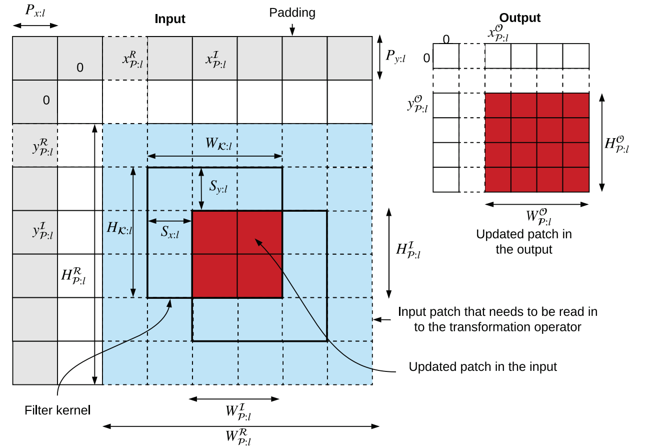

Figure 3: Simplified illustration of input and output update patches for Convolution/Pooling layers.

Determining Patch Update Locations. We first explain how to calculate the coordinates and dimensions of the *output update patch* of layer l given the *input update patch* and layer-specific parameters. Figure 3 illustrates these calculations. Our coordinate system's origin is at the top left corner. The input update patch is shown in red/dark color and starts at $(x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l})$, with height $H^{\mathcal{I}}_{\mathcal{P}:l}$ and width $W^{\mathcal{I}}_{\mathcal{P}:l}$. The output update patch starts at $(x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l})$ and has a height $H^{\mathcal{O}}_{\mathcal{P}:l}$ and width $W^{\mathcal{O}}_{\mathcal{P}:l}$. Due to overlaps among filter kernel positions during inference, computing the output update patch requires reading a slightly larger spatial context than the input update patch–we call this the "read-in context," and it is illustrated by the blue/shaded region in Figure 3. The read-in context starts at $(x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l})$, with its dimensions denoted by $W^{\mathcal{R}}_{\mathcal{P}:l}$ and $H^{\mathcal{R}}_{\mathcal{P}:l}$. The relationship between these quantities along the width dimension can be expressed as follows (likewise for the height dimension):

$$x^{\mathcal{O}}_{\mathcal{P}:l} = max\big(\lceil (P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\big) \qquad (10)$$

$$W^{\mathcal{O}}_{\mathcal{P}:l} = min\big(\lceil (W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W_{\mathcal{O}:l}\big) \qquad (11)$$

$$x^{\mathcal{R}}_{\mathcal{P}:l} = x^{\mathcal{O}}_{\mathcal{P}:l} \times S_{x:l} - P_{x:l} \qquad (12)$$

$$W^{\mathcal{R}}_{\mathcal{P}:l} = W_{\mathcal{K}:l} + (W^{\mathcal{O}}_{\mathcal{P}:l} - 1) \times S_{x:l} \qquad (13)$$

Equation (10) calculates the coordinates of the output update patch. As shown in Figure 3, padding effectively shifts the coordinate system and thus, $P_{x:l}$ is added to correct it. Due to overlaps among the filter kernels, the affected region of the input update patch (blue/shaded region in Figure 3) will be increased by $W_{\mathcal{K}:l} - 1$, which needs to be subtracted from the input coordinate $x^{\mathcal{I}}_{\mathcal{P}:l}$. A filter of size $W_{\mathcal{K}:l}$ that is placed starting at $x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1$ will see an update starting from $x^{\mathcal{I}}_{\mathcal{P}:l}$. Equation (11) calculates the width of the output update patch, which is essentially the number of filter kernel stride positions on the read-in input context. However, this value cannot be larger than the output size. Given these, a start coordinate and width of the read-in context are given by Equations (12) and (13); similar equations hold for the height dimension (skipped for brevity).

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

$$\mathcal{U} = \mathcal{I}_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}:l} : x^{\mathcal{R}}_{\mathcal{P}:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l} : y^{\mathcal{R}}_{\mathcal{P}:l} + H^{\mathcal{R}}_{\mathcal{P}:l}] \quad (14)$$

$$\mathcal{U} = \mathcal{U} \circ_{(x^{\mathcal{I}}_{\mathcal{P}:l} - x^{\mathcal{R}}_{\mathcal{P}:l}),(y^{\mathcal{I}}_{\mathcal{P}:l} - y^{\mathcal{R}}_{\mathcal{P}:l})} \mathcal{P}^{\mathcal{I}}_{:l} \quad (15)$$

$$\mathcal{P}^{\mathcal{O}}_{:l} = T_{:l}(\mathcal{U}) \quad (16)$$

Equation (14) slices the read-in context $\mathcal{U}$ from the pre-materialized input tensor $\mathcal{I}_{:l}$. Equation (15) superimposes the input update patch $\mathcal{P}^{\mathcal{I}}_{:l}$ on it. This is an in-place update of the array holding the read-in context. Finally, Equation (16) computes the output update patch $\mathcal{P}^{\mathcal{O}}_{:l}$ by invoking $T_{:l}$ on $\mathcal{U}$. Thus, we avoid performing inference on all of $\mathcal{I}_{:l}$, thus achieving incremental inference and reducing FLOPs.

3.2 Propagating Updates across Layers

Unlike relational IVM, CNNs have many layers, often in a sequence. This is analogous to a sequence of queries, each requiring IVM on its predecessor's output. This leads to a new issue: correctly and automatically configuring the update patches across layers of a CNN. While this seems simple, it requires care at the boundary of a local context transformation and a global context transformation. In particular, we need to materialize the full updated output, not just the output update patches, since global context transformations lose spatial locality for subsequent layers. Some recent deep CNNs have a more general directed acyclic graph (DAG) structure for layers. They have two new kinds of layers that "merge" two branches in the DAG: *element-wise addition* and *depth-wise concatenation*. To address such cases, we propose a simple unified solution: compute the *bounding box* of the input update patches. While this will potentially recompute parts of the output that do not get modified, we think this trade-off is acceptable because the gains are likely to be marginal for the additional complexity introduced.

3.3 Multi-Query Incremental Inference

OBE issues $|G|$ re-inference requests *in one go*. Viewing each request as a "query" makes the connection with MQO [26] clear. The $|G|$ queries are also *not disjoint*, as the occlusion patch is small, which means most pixels are the same. We now briefly explain how we extend our IVM framework with an MQO-style optimization fusing multiple re-inference requests. An analogy with relational queries is many concurrent incremental updates on the same relation.

Batched Incremental Inference. Our optimization works as follows: materialize all CNN tensors *once* and *reuse* them for incremental inference across all $|G|$ queries. Since the occluded images share most of their pixels, parts of the tensors will likely be identical too. Thus, we can amortize the materialization cost. Batched execution is standard practice on high-throughput compute hardware like GPUs, since it amortizes CNN set up costs, data movement costs, etc. Batch sizes are tuned to optimize hardware utilization. Thus, we combine both these ideas to execute incremental inference in a batched manner. We call this approach "batched incremental inference." Empirically, we found that batching alone yields limited speedups (under 2X), but batched incremental inference amplifies the speedups.

GPU Optimized Implementation. Empirically, we found a dichotomy between CPUs and GPUs: batched incremental inference yielded expected speedups on CPUs, but it performed dramatically poorly on GPUs. In fact, a naive implementation on GPUs was *slower* than full re-inference! The reason for this was the overheads incurred during read-in context preparation step, which throttles the GPU throughput. To overcome this issue, we created a custom CUDA kernel to perform read-in context preparation more efficiently by *copying memory regions in parallel* for all items in the batched inference request.

3.4 Expected Speedups

We extend our framework to perform "static analysis" on a given CNN f to find how much FLOPs can be saved using incremental inference, yielding us an upper bound on speedups. The computational cost of incremental inference for a layer is proportional to the volume of the individual filter kernel times the total volume of the updated output. The total computational cost for incremental inference, denoted Q_{inc}, is the sum of incremental inference cost across all layers. Q_{inc} can be much smaller than Q in Equation (9). We define the *theoretical speedup* as the ratio $\frac{Q}{Q_{inc}}$. This tells us how beneficial incremental inference can be in the best case *without* running the actual inference itself.

We calculated the theoretical speedups for many popular CNNs for occlusion patches with varying sizes placed at the center of the image. For an occlusion patch of size 16×16, VGG-16 sees the highest theoretical speedups of 6x; DenseNet-121 sees a speedup of 2x, the lowest. Most CNNs fall in the 2x–3x range. The differences arise due to the specifics of the CNNs' architectures: VGG-16 has small Convolution filter kernels and strides, which means full re-inference is costlier. While speedups of 2x–3x may sound "not that significant" in practice, we find they are indeed significant for two reasons. First, *users often wait in the loop* for OBE when performing interactive diagnoses. Thus, even such speedups can improve their productivity. Second, our IVM is the *foundation for our approximate inference* optimizations (Section 4), which amplify the speedups.

4. APPROXIMATE CNN INFERENCE

Since incremental inference is *exact*, i.e., it yields the same heatmap as full inference, it does not exploit a capability of human perception: tolerance of some degradation in visual quality. We now briefly explain how we build upon our IVM

framework to create two novel heuristic approximate inference optimizations that trade off the heatmap's quality in a user-tunable manner to accelerate OBE further.

4.1 Projective Field Thresholding

The *projective field* of a CNN neuron is the slice of the output tensor that is connected to it. It is a term from neuroscience to describe the effects of a retinal cell on the output of the eye's neuronal circuitry [7]. This notion sheds light on the *growth of the size* of the update patches through the layers of a CNN. The 3 kinds of layers (Section 2.2) affect the projective field size growth differently. Individual element transformations do not alter the projective field size. Global context transformations increase it to the whole output. However, local spatial context transformations, which are the most crucial, increase it *gradually* at a rate determined by the filter kernel's size and stride: additively in the size and multiplicatively in the stride. The growth of the projective field size implies the amount of FLOPs saved by IVM decreases as we go to the higher layers of a CNN. Eventually, the output update patch becomes as large as the output tensor. This growth is illustrated by Figure 4(a).

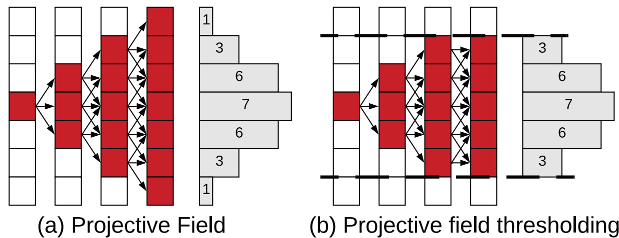

Figure 4: (a) Projective field growth for 1-D Convolution (filter size 2, stride 1). (b) Projective field *thresholding*; $\tau = 5/7$.

Our above observation motivates the main idea of this optimization, which we call projective field thresholding: *truncate* the projective field from growing beyond a given *threshold fraction* τ $(0 < \tau \leq 1)$ of the output size. This means inference in subsequent layers is approximate. Figure 4(b) illustrates the idea for a filter size 3 and stride 1. This approximation can alter the accuracy of the output values and the heatmap's visual quality. Empirically, we find that modest truncation is tolerable and does not affect the heatmap's visual quality too significantly.

To provide intuition on why the above happens, consider histograms shown in Figures 4(a,b) that list the number of unique "paths" from the updated element to each output element. It resembles a Gaussian distribution. Thus, for most of the output patch updates, truncation will only discard a few values at the "fringes" that contribute to an output element. This optimization is only feasible *in conjunction with* our incremental inference framework (Section 3) to reuse the remaining parts of the tensors and save FLOPs.

4.2 Adaptive Drill-Down

This heuristic optimization is based on our observation about a peculiar semantics of OBE that lets us modify how G (the set of occlusion patch locations) is specified and handled, especially in the non-interactive specification mode. We explain our intuition with an example. Consider a radiologist explaining a CNN prediction for diabetic retinopathy on a tissue image. The region of interest typically occupies only a tiny fraction of the image. Thus, it is not necessary to perform regular OBE for *every* patch location: most of the (incremental) inference computations are effectively "wasted" on uninteresting regions. In such cases, we modify the OBE workflow to produce an approximate heatmap using a two-stage process, illustrated by Figure 5.

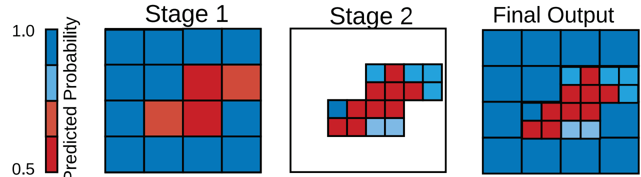

Figure 5: Schematic representation of *adaptive drill-down*.

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

4.3 Automated Parameter Tuning

We also devise automated parameter tuning methods for easily configuring the approximate inference optimizations. For projective field thresholding, mapping a threshold value (τ) to visual quality directly is likely to be unintuitive for users. Thus, to measure visual quality more intuitively, we adopt a cognitive science-inspired metric called Structural Similarity (SSIM) Index, which is widely used to quantify human-perceptible differences between two images [28]. During an offline phase, we learn a function that maps the heatmap visual quality to a τ value using a sample of workload images. During the online phase, we use the learned function to map the user given SSIM value to a target τ value. For adaptive drill-down, we expect the user to provide the drill-down ratio ($r_{drill-down}$) based on her understanding of the size of the region of interest in the OBE heatmap and on how much speedup she wants to achieve. We set the stage one stride (S_1) using these two user-given settings.

5. EXPERIMENTAL EVALUATION

We integrated our techniques with the popular deep learning tool PyTorch to create a system we call Krypton. We now present a snapshot of our key empirical results with Krypton on different CNNs and datasets.

Datasets. We use 2 real-world image datasets: *OCT* and *Chest X-Ray*. *OCT* has about 84,000 optical coherence tomography retinal images with 4 classes. *Chest X-Ray* has about 6,000 X-ray images with 3 classes. Both *OCT* and *Chest X-Ray* are from a recent radiology study that applied deep CNNs to detect the respective diseases [11].

Workloads. We use 3 diverse ImageNet-trained [25] deep CNNs: VGG16, ResNet18 and Inception3. They comple-

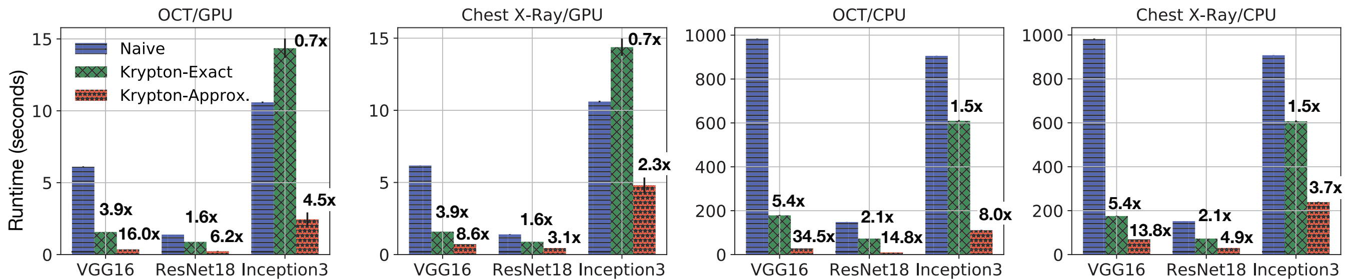

Figure 6: End-to-end runtimes of KRYPTON and the baseline on 2 datasets and 3 CNNs on GPU and CPU.

ment each other in terms of model size, architectural complexity, computational cost, and our predicted theoretical speedups. CNNs were fine-tuned by retraining their final Fully-Connected layers using the *OCT* and *Chest X-Ray* datasets, as per standard practice. The GPU-based experiments used a batch size of 128; for CPUs, the batch size was 16. All CPU-based experiments were executed with a thread parallelism of 8.

Experimental Setup. We use a machine with 32 GB RAM, Intel i7 3.4GHz CPU, and NVIDIA Titan X (Pascal) GPU with 12 GB memory. The machine runs Ubuntu 16.04 with PyTorch version 0.4.0, CUDA version 9.0, and cuDNN version 7.1.2. All reported runtimes are the average of 3 runs, with 95% confidence intervals shown.

5.1 End-to- End Runtimes

We focus on perhaps the most common scenario for OBE: produce the whole heatmap for automatically created G ("non-interactive" mode). The occlusion patch size is set to 16; stride, 4. We compare two variants of KRYPTON: KRYPTON-Exact uses only incremental inference, while KRYPTON-Approximate uses our approximate inference optimizations too. The baseline is *Naive*, which runs full re-inference with only batching to improve hardware utilization. We set the approximate inference parameters based on the semantics of each dataset's prediction task. Figure 6 presents the results. More details about the parameters and visual examples of the heatmaps are available in the longer version of this paper [20].

Overall, we see that KRYPTON offers significant speedups across the board on both GPU and CPU, with the highest speedups seen by KRYPTON-Approximate on *OCT* with VGG16: 16x on GPU and 34.5x on CPU. The highest speedups of KRYPTON-Exact are also on VGG16: 3.9x on GPU and 5.4x on CPU. The speedups of KRYPTON-Exact are identical across datasets for a given CNN, since it does not depend on the image semantics, unlike KRYPTON-Approximate due to its parameters. KRYPTON-Approximate sees the highest speedups on *OCT*.

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

Finally, the speedups are higher on CPU than GPU; this is because CPU suffers less from memory stalls during incremental inferences. However, the *absolute* runtimes are much lower on GPU, as expected. Overall, KRYPTON reduces OBE runtimes substantially for multiple datasets and deep CNNs.

5.2 Other Experimental Results

We also perform ablation studies to evaluate the impact of each of our optimization techniques for varying configuration parameters for OBE. The patch size and stride have an inverse effect on speedups because they reduce the sheer amount of FLOPs in the re-inference requests. The parameters of the approximate optimizations also affect speedups significantly, and our automated tuning methods help optimize the accuracy-runtime tradeoffs effectively. The memory overhead of our batched incremental inference approach is also significantly lower (about 2x) compared to full re-inference.

5.3 Demonstration and Extensions

In follow-on work, we extended KRYPTON and demonstrated support for human-in-the-loop OBE [19, 24]. The user can interactively select a sub-region of the image (to specify G) and iteratively refine it. We also showed that KRYPTON can help accelerate OBE on time-series data out of the box and can also help accelerate object recognition in fixed-angle camera videos when combined with new approximate inference techniques [21].

6. OTHER RELATED WORK

Explaining CNN Predictions. Perturbation-based and gradient-based are the two main kinds of methods for explaining CNN predictions. Perturbation-based methods observe the output of the CNN by modifying regions of the input image. OBE belongs to this category. In practice, however, OBE is usually more popular among domain scientific users, especially in radiology [10], since it is easy to understand for non-technical users and typically produces high-quality heatmaps.

Faster CNN Inference. EVA^2 [3] and CBInfer [4] use approximate change detection for faster CNN inference over video data. While one can map OBE to a "video," our IVM

and MQO techniques are complementary to such systems, while our approximate inference optimizations are also novel and exploit specific properties of CNNs and OBE.

Query Optimization. Our work is inspired by the long line of work on relational IVM [6, 16], but ours is the first to use the IVM lens for OBE with CNNs. Our algebraic IVM framework is closely tied to the dataflow of CNN layers, which transform tensors in non-trivial ways. Our work is related to the IVM framework for linear algebra in [23]. They focus on bulk matrix operators and incremental addition of rows. The focus of our work is on more fine-grained CNN inference computations. Our work is also inspired by relational MQO [26], but our focus is CNN inference, not relational queries. MQO for ML systems is a growing area of research [2, 14, 15], both for classical statistical ML (e.g., [5, 12, 13, 17, 30]) and deep learning (e.g., [18, 22]). Our work adds to this direction, but ours is the first work to combine MQO with IVM for ML systems. Our approximate inference optimizations are inspired by AQP [8], but unlike statistical approximations of SQL aggregates, our techniques are novel CNN-specific and human perception-aware heuristics tailored to OBE.

7. CONCLUSIONS AND FUTURE WORK

Deep CNNs are popular for image prediction tasks, but their internal workings are unintuitive for most users. Occlusion-based explanation (OBE) is a popular mechanism to explain CNN predictions, but it is highly compute-intensive. We formalize OBE from a data management standpoint and present several novel database-inspired optimizations to speed up OBE. Our techniques span incremental inference and multi-query optimization for CNNs to human perception-aware approximate inference. Overall, our ideas yield over an order of magnitude speedups for OBE on both GPU and CPU. As for future work, we plan to extend our ideas to other deep learning workloads and data types. More broadly, we believe database-inspired query optimization techniques can help reduce resource costs of deep learning systems significantly, thus enabling a wider base of application users to benefit from modern ML.

Acknowledgments. This work was supported in part by a Hellman Fellowship and by the NIDDK of the NIH under award number R01DK114945. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH. We thank NVIDIA Corporation for donating the Titan Xp GPU used for this work. We thank the members of UC San Diego's Database Lab for their feedback on this work.

8. REFERENCES

[1] AI Device for Detecting Diabetic Retinopathy Earns Swift FDA Approval. https://bit.ly/36300H9. Accessed April 30, 2020.
[2] M. Boehm et al. *Data Management in Machine Learning Systems.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.
[3] M. Buckler et al. EVA^2: Exploiting Temporal Redundancy in Live Computer Vision. In *ISCA*, 2018.
[4] L. Cavigelli et al. CBInfer: Change-based Inference for Convolutional Neural Networks on Video Data. In *International Conference on Distributed Smart Cameras*, 2017.
[5] L. Chen et al. Towards Linear Algebra over Normalized Data. In *VLDB*, 2017.
[6] R. Chirkova and J. Yang. *Materialized Views.* Now Publishers Inc., 2012.
[7] S. E. de Vries et al. The Projective Field of a Retinal Amacrine Cell. In *Journal of Neuroscience*, 2011.
[8] M. N. Garofalakis and P. B. Gibbons. Approximate Query Processing: Taming the TeraBytes. In *VLDB*, 2001.
[9] I. Goodfellow et al. *Deep Learning.* MIT press Cambridge, 2016.
[10] K.-H. Jung et al. Deep Learning for Medical Image Analysis: Applications to Computed Tomography and Magnetic Resonance Imaging. *Hanyang Medical Reviews*, 2017.
[11] D. S. Kermany et al. Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning. *Cell*, 2018.
[12] P. Konda et al. Feature Selection in Enterprise Analytics: A Demonstration Using an R-Based Data Analytics System. In *VLDB*, 2013.
[13] A. Kumar et al. Learning Generalized Linear Models Over Normalized Data. In *ACM SIGMOD*, 2015.
[14] A. Kumar et al. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *ACM SIGMOD Rec.*, 2016.
[15] A. Kumar et al. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *ACM SIGMOD*, 2017.
[16] A. Y. Levy et al. Answering Queries Using Views. In *PODS*, 1995.
[17] S. Li et al. Enabling and Optimizing Non-Linear Feature Interactions in Factorized Linear Algebra. In *ACM SIGMOD*, 2019.
[18] S. Nakandala et al. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *ACM SIGMOD DEEM Workshop*, 2019.
[19] S. Nakandala et al. Demonstration of Krypton: Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations. In *SysML*, 2019.
[20] S. Nakandala et al. Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations. In *ACM SIGMOD*, 2019.
[21] S. Nakandala et al. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM TODS*, 2020.
[22] S. Nakandala and A. Kumar. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *ACM SIGMOD*, 2020.
[23] M. Nikolic et al. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *ACM SIGMOD*, 2014.
[24] A. Ordookhanians et al. Demonstration of Krypton: Optimized CNN Inference for Occlusion-Based Deep CNN Explanations. In *VLDB*, 2019.
[25] O. Russakovsky et al. Imagenet Large Scale Visual Recognition Challenge. In *International Journal of Computer Vision*, 2015.
[26] T. K. Sellis. Multiple-Query Optimization. In *ACM TODS*, 1988.
[27] P. Voigt and A. Von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide.* Springer Publishing Company, Inc., 2017.
[28] Z. Wang et al. Image Quality Assessment: From Error Visibility to Structural Similarity. In *IEEE Transactions on Image Processing*, 2004.
[29] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision*, 2014.
[30] C. Zhang et al. Materialization Optimizations for Feature Selection Workloads. In *ACM SIGMOD*, 2014.
[31] L. M. Zintgraf et al. Visualizing Deep Neural Network Decisions: Prediction Difference Analysis. In *ICLR*, 2017.