Kryton: A System for Accelerating Occlusion based Deep CNN Explainability Workloads

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

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 such as ImageNet challenge. Thus they are now being deployed in many real-world use cases using a paradigm called *transfer learning*. 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 has been proposed and one of the widely used method in image classification tasks is occlusion experiments. In occlusion experiments one would mask the regions of the input image using a small grey 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 very time consuming. We present Krypton, the first data system to elevate occlusion experiments to a declarative level and enable automated *incremental* and *approximate* inference optimizations. Experiments with real-world datasets and deep CNNs show that Krypton can enable up to 10x speedups.

1 INTRODUCTION

Deep convolutional neural networks (CNNs) [6, 8] have revolutionized the computer vision field with even resulting near-human accuracies for some image recognition challenges. Many of these successful pre-trained CNNs from computer vision challenges have been successfully repurposed to be used in other real-world image recognition tasks, using a paradigm called *transfer learning* [2]. In transfer learning, instead of training a CNN from scratch, one uses a pre-trained Deep CNN, e.g., ImageNet trained VGG, and fine tune it for the target problem using the target training dataset. This approach avoids the need for a large training datasets and computational power and time for training which is otherwise a bottleneck for training a CNN from scratch. As a result, this paradigm has enabled the wide adoption of deep CNN technology in variety of real world image recognition tasks in several domains including health care [3, 5], agriculture [7], security [1], and sociology [9].

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 [4]. In order to improve the explainability of deep CNN predictions several approaches has been proposed. One of the most widely used approach used in image classification tasks is occlusion experiments [10]. In occlusion experiments, a square patch, usually of gray or black color, is used to occlude parts of the image and record the variation in the predicted label probability. By systematically striding this patch horizontally and vertically one or two pixels at a time over the image, a sensitivity map for the predicted label can be generated. Using this heatmap, the regions in the image which are highly sensitive (or highly contributing) to the predicted class can be identified. This localization of highly sensitive regions then enables the practitioners to get an idea on the the prediction process of the deep CNN (see Figure. **??**).

However, occlusion experiments are highly compute intensive and time consuming as each patch super imposition is treated as a new image and requires a separate CNN inference. In this work our goal is to apply database style optimizations to the occlusion based explainability approach to reduce both the computational cost and runtime taken for an experiment. 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 lot of redundant computations which can be avoided. To avoid redundant computations we introduce the notion of *incremental inference* of deep CNNs. Due to the overlapping nature of how a convolution kernel would operate, 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, as we propagate the patch to the deeper layers the size of the patch will start growing. However at deeper layers the effect over patch coordinates which are radially further away from the center of the patch position will be diminishing. Our second optimization is based on this observation where we apply a form of approximate inference which applies a *propagation threshold* to limit the growth of the updating patch. We show that by applying propagation thresholds, a significant level of computation redundancy can be retained without significantly affecting the quality of the generate sensitivity heatmap. 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 of interest is contained in a relative small region of the image. In such situations it is unnecessary to inspect the original image at the same high resolution of striding the patch one or two pixels at a time at all locations. In adaptive drill-down approach, first a low resolution heatmap is generated with a larger stride with relatively low computational cost and only the interested regions will be inspected further with a lower stride generating a higher resolution output. This two stage process also reduces

the runtime of occlusion experiments without affecting the accuracy of the occlusion experiment output significantly.
Outline. The rest of this paper is organized as follows.

2 BACKGROUND

Deep CNNs. 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; *non-linearity* to apply a non-linear function (e.g., ReLU) to all features; and *fully connected*, which is a multi-layer perceptron. A "deep" CNN just stacks such layers many times over. Popular deep CNN model architectures include AlexNet [6], VGG [8], Inception, ResNet, SqueezeNet, and MobileNet. In this work, the discussion and evaluation is focused on VGG, ResNet and SqueezeNet which are three widely used CNN models in real world use cases. Nevertheless, our work is orthogonal to how CNNs are designed and the proposed approaches can be easily extended to any architecture.

Deep CNN Explainability With image classification models, natural question is if the model is truly identifying objects in the image or just using surrounding or other objects for making false predictions. The various approaches used to explain CNN predictions can be broadly divided into two categories, namely gradient based and perturbation based approaches. Gradient based approaches generate a sensitivity map 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. [10]. Even though gradient approaches require only a single forward inference and a single backpropagation to generate the sensitivity map, the output may not be very intuitive and hard to understand because the salient pixels tend to spread over a very large are of the input image. 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 as they produce high quality fine grained sensitivity maps despite being time consuming.

Over the years there has been several modifications proposed to the original occlusion experiment approach. More recently Zintgraf. et. al. [11] proposed a variation to the original occlusion experiment approach named *Prediction Difference Analysis*. In this method instead of masking with a grey or black 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.

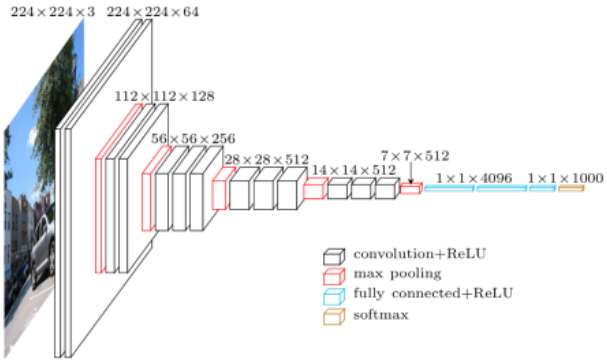

Figure 1: VGG16 CNN Architecture

3 PRELIMINARIES AND OVERVIEW

In this section we formalize the internals of a Deep CNN which will be later used to propose our incremental inference approach in Section 3. We then state the problem studied, explain our assumptions, and given an overview of Krypton.

3.1 Deep CNN Internals

CNNs are composed of a number of different layers: **convolution** layer, **pooling** layer (average or max pooling), **activation** layer, and **fully-connected** layer.

The main difference of CNNs compared to other neural networks is that they make the explicit assumption that the input is always going to be an image. This assumption enables them to incorporate several architectural properties into the CNN architecture which reduce the amount of computations required for inference and the amount of learnable parameters. Neurons in a particular layer of a CNN are organized into 3D volumes with width, hight, and depth dimensions. For example the images in ImageNet dataset can be treated as an input volume of activations having dimensions of $224 \times 224 \times 3$. Unlike in typical neural networks, a neuron in a convolution layer is only connected to a small region of the layer before it, instead of all the neurons in a full-connected manner. However this is changed in the last layer (or last few layers) of a CNN in which a neuron is connected to all the neurons in the layer below in a fully connected manner.

A Deep CNN is composed of several different types of layers. The main types used to build a CNN are: Convolutional Layer, ReLu Layer, Pooling Layer, Fully-Connected Layer and Softmax Layer. Usually Convolution and Fully-Connected are always followed by a ReLu layer which performs a activation wise function which outputs the max of zero of the weighted sum of activations of the connected neurons. Convolution and Pooling layers are stacked alternately to form a cascade of layers and at the end Full-Connected layers are used. A Softmax layer is used at the end of Fully-Connected Layers to output the class probabilities. Generally in CNNs the activation volumes and filter maps, height and width of the spatial resolution is set to be the same producing square matrices. Figure. 1 demonstrates how these layers are stacked together to form VGG16 CNN architecture. Next we will look into Convolutional and Pooling Layers in more detail.

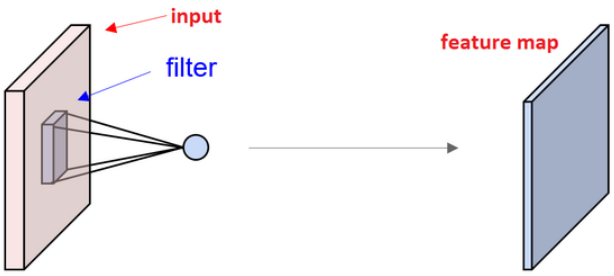

Figure 2: Output feature map is computed by performing a dot product between filter values and the input activations

3.1.1 Convolutional Layer. Convolutional Layer (Conv) is the most important type of layer in a CNN architecture. Also it is the most computationally intensive layer which accounts up to 90% (or more) of the total computations in a Deep CNN model. Conv layer parameters consist of set of learnable filters each of which has small spatial dimensions but extends across the depth of the input volume. For example the first Conv layer of VGG16 model has 64 filter and each has a spatial dimension of 3×3 and a depth of 3 (corresponding to the color depth of the image). In contrast the second Conv layer has 64 filters with the spatial dimension of 3×3 but with a depth of 64 (the depth of the activation volume of layer 1). A filter is convolved across the height and width dimensions of the input volume producing a 2D feature map where each activation is calculated by the dot product between the filter parameters and the activations of the input volume. Note that here, only a small portion of the total activations in the input volume, which are spatially collocated together, are used to calculate the activation of an output neuron (see Figure. 2). Stacking the 2D feature maps produced for all the filter(e.g VGG16 first Conv layer has 64 filters) a output 3D activation map is created. At the training time the parameters of the filters are updated using backpropagation and stochastic gradient descent such that these parameters learn to identify different visual concepts (eg. edges, color blotches, and high level objects such human faces) from the activation volume of the layer below [10].

In addition to performing spatial convolutions, Conv layers can optionally reduce the spatial resolution of features volumes. Two hyper-parameters control the size of the output feature volume: the **stride**, and **zero-padding**.

stride: When moving the filter across the input volume a stride value has to be specified. When the stride is set to one, the filter is moved one pixel at a time. If the stride is larger than 1, for example setting it to two, the filter is moved 2 pixels at a time producing a smaller output spatial resolution.

zero-padding: Sometimes before performing the filter convolution it is beneficial to pad the input volume with zeros around the border to obtain the required output size. Also, by padding with zeros we can ensure that the input and output volumes will have the same spatial resolution when the stride is set to one.

Given the size of the input volume (**W**), size of the filter (**F**), amount of zero-padding (**P**), and the stride (**S**) the size of the output volume can be computed by (**W** - **F** + 2**P**)/**S** + 1. For example in VGG16 first Conv layer, the input and output size is **W** = 224 and the filter size is set to **F** = 3 and is stride is set to **S** = 1. To produce the output of same size the padding value should be set to **P** = 1. Also note that the potential values for **W**, **F**, **P**, and **S** has mutual constraints as the output size has to be an integer.

3.1.2 Pooling Layer. CNNs architectures periodically insert Pooling Layers to mainly to reduce the spatial resolution of output features maps and also to introduce translational invariance to the image predictions. The pooling layer operated independently on each input feature map along the depth dimension and applies a local filter such as **max(...)**. The most typical form of Max Pooling is to apply a filter map of size 2×2 with a stride of 2 which reduces the height and width of the input feature map by a factor of 2 and discard 75% of the activations. In general when applying a Pooling filter of size **F** on an input feature volume of size **W** with a stride of **S** the produced output volume will have a size of (**W** - **F**)/**S** + 1.

3.2 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 way to estimate the computational cost of a Deep CNN is to estimate the number of fused multiply add (FMA) floating point operations (FLOPs) required for a single forward inference.

Conv layer producing an output feature map of size $(W_2 \times W_2 \times D_2)$ from an input feature map of $(W_1 \times W_1 \times D_1)$ using a filter of size $(F \times F \times D_1)$ will require $F \times F \times D_1 \times W_2 \times W_2 \times D_2$ FMA operations. For example in VGG16, computing a single activation of the first Conv volume requires 27 ($3 \times 3 \times 3$) FMA operations and computing the whole output Conv volume requires 84 ($3 \times 3 \times 3 \times 224 \times 224 \times 64$) Mega FMA operations. A Fully-Connected Layer reading N_1 input activations and producing N_2 output activations requires $WN1 \times N_2$ FMA operations. For example, the first Fully-Connected Layer in VGG16 model requires 98 $((7 \times 7 \times 512) \times 4096)$ Mega FMA operations. The floating points operations performed by other layers (e.g. ReLu and Pooling) are relatively very smaller than that performed by Conv and Full-Connected Layers and hence neglected in compute cost analyses.

Alternatively one could also evaluate the computational cost of a CNN model by actually performing a CNN inference and recording the wall clock time. For making CNN inference (and in generally for deep neural networks) one could use either CPUs or GPUs. GPUs are generally an order of magnitude faster than CPUs. However the overhead of data transfer to the GPUs is higher than that of CPUs. Hence by batching multiple input images the overhead can be amortized. Figure 3 shows the theoretically floating point operations required and actual per image inference time on CPU and GPU for several widely used Deep CNN models (AlexNet, VGG16, ResNet50, and MobileNet).

4 OPTIMIZATIONS

4.1 Incremental Inference of Convolution and Pooling Layers

As explained earlier, occlusion experiments are performed by performing CNN inference on large amounts of images which are generated by masking a small square region of the input image. When

	AlexNet	VGG16	ResNet50	MobileNet
MFLOPs	714	15470	4089	569
CPU Runtime(batch size=1)	37.0 ms	409.0 ms	158.0 ms	92.0 ms
CPU Runtime(batch size=16)	19.0 ms	339.0 ms	135.0 ms	56.0 ms
GPU Runtime(batch size=1)	2.7 ms	7.0 ms	7.3 ms	3.3 ms
GPU Runtime(batch size=16)	1.5 ms	3.7 ms	2.7 ms	1.7 ms

Figure 3: Compute cost of widely used Deep CNNs. (CPU: Intel(R) Core(TM) i7-6700 CPU 3.40GHz machine with 32 GB Ram, GPU: Nvidia Titan Xp, Deep Learning Library: PyTorch 0.3.1)

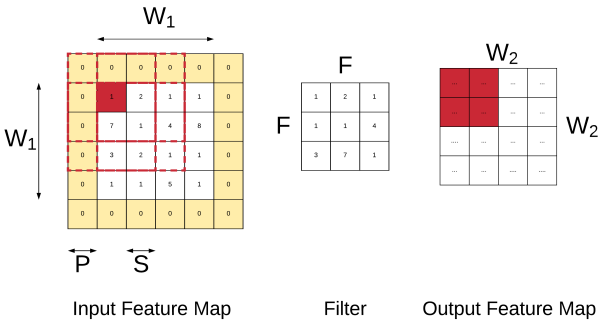

Figure 4: Small perturbations in the input feature map only affect small regions in the output feature map after convolution

we compare the original image and a single masked image most of the pixel values are not changed. For example for an input image of size 224×224 pixels when using a occlusion patch of size 16×16 only 0.5% of the pixels are different between the two images. As a result of this performing a full convolution inference on the masked image introduces lot of redundant computations which could potentially be saved. For example consider the simple 2D convolution example shown in Figure. 4. The input feature map is square map with size $W_1 = 4$ and is convolved by a 2D square filter kernel of size $F = 3$ with a stride of $S = 1$ to produce an output feature map of size $W_2 = 4$. The input feature map is also padded with zeros with a pad width of $P = 1$ to ensure that both the input feature map size and the output feature map size is the same (this step is optional). Now if we update the top left corner value in the input feature map (marked in red), it will only update 4 output values at the top left corner of the output feature map which corresponds to filter map positions on the input feature map with some overlapping with the updated input value. Even though in this example the amount of updated values in the output feature maps is 25% of the total output values, in general with larger feature map sizes the portion of updated values will be much smaller. This analysis similarly applies to pooling layers.

More generally the propagation changes in the output feature map of a convolution or pooling layer caused by updating a patch

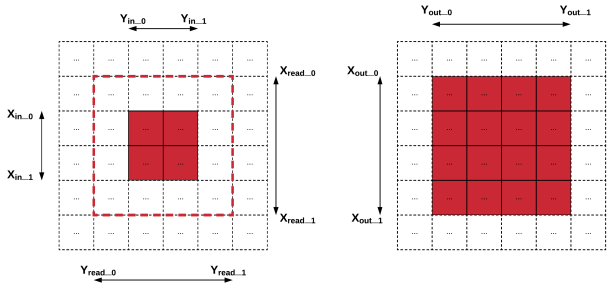

Figure 5: Occlusion patch propagation in Conv layers

in the input feature map is determined by the filter size F and the stride S of the Conv filter. For example consider the situation shown in Figure. 5. Assume a modified patch is placed on the input feature map which spans across $x_{in\_0} \rightarrow x_{in\_1}$ in x dimension and $y_{in\_0} \rightarrow y_{in\_1}$ in y dimension. Then the coordinates of the modified patch in the output feature map, $x_{out\_0} \rightarrow x_{out\_1}$ in x dimension and $y_{out\_0} \rightarrow y_{out\_1}$ in y dimension, can be expressed as a function of filter size F and stride S of the Conv/Pool filter as follows:

$$x_{out\_0} = \texttt{max}(\texttt{ceil}((x_{in\_0} - F + 1)/S), 0) \quad (1)$$

$$x_{out\_1} = \texttt{min}(\texttt{floor}((x_{in\_1} - 1)/S) + 1, W_2) \quad (2)$$

$$y_{out\_0} = \texttt{max}(\texttt{ceil}((y_{in\_0} - F + 1)/S), 0) \quad (3)$$

$$y_{out\_1} = \texttt{min}(\texttt{floor}((y_{in\_1} - 1)/S) + 1, W_2) \quad (4)$$

Also not that to compute these values in the output feature map we need to read a slightly bigger region from the input feature map due to the overlapping with the filter positions (marked in red dashed lines in Figure. 5). The coordinates of this read input patch, $x_{read\_0} \rightarrow x_{read\_1}$ in x dimension and $y_{read\_0} \rightarrow y_{read\_1}$ in y dimension, is also a function filter size F and stride S of the Conv filter and can be expressed as follows:

$$x_{read\_0} = \texttt{max}(\texttt{ceil}((x_{in\_0} - F + 1)/S) \times S, 0) \quad (5)$$

$$x_{read\_1} = \texttt{min}(\texttt{floor}((x_{in\_1} - 1)/S) \times S + F, W_1) \quad (6)$$

$$y_{read\_0} = \texttt{max}(\texttt{ceil}((y_{in\_0} - F + 1)/S) \times S, 0) \quad (7)$$

$$y_{read\_1} = \texttt{min}(\texttt{floor}((y_{in\_1} - 1)/S) \times S + F, W_2) \quad (8)$$

4.1.1 Estimating the maximum attainable theoretical speedup. Important thing to notice with incremental inference of Conv and Pool layers for occlusion experiments is that the size of the updated patch in the output layer is larger than the updated patch in the input layer. The growth is determined by the filter size and stride. Higher the filter size and stride higher the propagation rate of the modified patch. With this observation, it is interesting to find out what percentage of computations can be saved by performing incremental inference of Conv and Pooling layers. This can be easily estimated by iteratively calculating the updated patch sizes for Conv and Pooling layers based on the starting occlusion patch size W_{patch} on the input image for all possible patch locations based on

Table 1: Notation used in Section. 4.1

Symbol	Description
W_1	Width of the input feature map to the Conv/Pool operator
D_1	Depth of the input feature map to the Conv operator
F	Width of filter kernel of the Conv/Pool operator
W_2	Width of the output feature map produced by the Conv/Pool operator
D_2	Depth of the output feature map produced by the Conv operator
x_{in_0}	Starting x coordinate of the updated patch in the input feature map
x_{in_1}	Ending x coordinate of the updated patch in the input feature map
y_{in_0}	Starting y coordinate of the updated patch in the input feature map
y_{in_1}	Ending y coordinate of the updated patch in the input feature map
x_{out_0}	Starting x coordinate of the updated patch in the output feature map
x_{out_1}	Ending x coordinate of the updated patch in the output feature map
y_{out_0}	Starting y coordinate of the updated patch in the output feature map
y_{out_1}	Ending y coordinate of the updated patch in the output feature map
x_{read_0}	Starting x coordinate of the input feature map that need to be used for computing the updated output
x_{read_1}	Ending x coordinate of the input feature map that need to be used for computing the updated output
y_{read_0}	Starting y coordinate of the input feature map that need to be used for computing the updated output
y_{read_1}	Ending y coordinate of the input feature map that need to be used for computing the updated output

stride used S_{patch}. Algorithm 1 shows how this can be calculated programmatically. For sake of simplicity this algorithm assumes that the CNN architecture is a simple chained style architecture instead of more general style of directed-acyclic-graph (DAG). However the algorithm can be easily extended support more general DAG style architectures. It takes an object *CNN* which is a nested information object containing information about different layers of the CNN and their properties, the size of the occlusion patch and the size of the stride for occlusion patch. It then calculates the floating point operations required for incremental inference versus full inference for each possible location of the occlusion patch and computes the theoretical speedup which is the ratio between operation required for full inference and incremental inference. It also computes the overall speedup which is the aggregation for all possible positions of the occlusion map and return this value along with 2D array containing individual position wise speedups as the output.

We have applied this algorithm to analyze the theoretical maximum attainable speedup for many widely used CNN architectures by performing static analyzing on the CNN models defined using PyTorch framework (see Figure. 6). With an occlusion patch of size 16, moved with a stride of 1, for most CNN architectures we can achieve an average speedup of around 2. However VGG and Squeezenet1_0 CNN architectures can produce higher speedups than this. The reason for this is VGG (16 and 19 layer versions) and Squeezenet1_0 architectures use smaller Conv filter kernels (3×3). Therefore the rate of propagation of the occlusion patch is slower than other CNN architectures. Thus more redundant computations can be saved from those architectures by applying incremental inference approach.

Algorithm 1 Estimate Maximum Theoretical Speedup

procedure EstimateMaxSpeedup(CNN, W_{patch}, S_{patch})
  $flops_{inc} \leftarrow 0$
  $flops_{full} \leftarrow 0$
  $tmp \leftarrow floor((CNN.image.W - W_{patch} + 1)/S_{patch})$
  $speedup \leftarrow ARRAY[tmp][tmp]$
  for `i in range(0, tmp,` S_{patch}`)` **do**
    for `i in range(0, tmp,` S_{patch}`)` **do**
      $tmp\_flops_{full} \leftarrow 0$
      $tmp\_flops_{inc} \leftarrow 0$
      $x_{in\_0} \leftarrow i$
      $x_{in\_0} \leftarrow i + W_{patch}$
      $y_{in\_0} \leftarrow j$
      $y_{in\_0} \leftarrow j + W_{patch}$
      for `k in range(0, size(CNN.layers), 1)` **do**
        $layer \leftarrow CNN.layers[k]$
        if $layer.type$ *in* $[conv, pool]$ **then**
          $F \leftarrow layer.filter.F$
          $S \leftarrow layer.filter.S$
          $x_{out\_0} = \texttt{max(ceil}((x_{in\_0} - F + 1)/S), 0)$
          $x_{out\_1} = \texttt{min(floor}((x_{in\_1} - 1)/S) + 1, W_2)$
          $y_{out\_0} = \texttt{max(ceil}((y_{in\_0} - F + 1)/S), 0)$
          $y_{out\_1} = \texttt{min(floor}((y_{in\_1} - 1)/S) + 1, W_2)$
          if $layer.type = conv$ **then**
            $W_1 \leftarrow layer.input.W$
            $D_1 \leftarrow layer.input.D$
            $W_2 \leftarrow layer.output.W$
            $D_2 \leftarrow layer.output.D$
            tmp_flops_{full} += $F^2 \times D_1 \times W_2^2 \times D_2$
            tmp_flops_{inc} += $F^2 \times D_1$
                $\times (x_{out\_1} - x_{out\_0})$
                $\times (y_{out\_1} - y_{out\_0})$
                $\times D_2$
          $(x_{in\_0}, x_{in\_1}, y_{in\_0}, y_{in\_1})$
              $\leftarrow (x_{out\_0}, x_{out\_1}, y_{out\_0}, y_{out\_1})$
        else if $layer.type = fully - connected$ **then**
          $W_1 \leftarrow layer.input.W$
          $W_2 \leftarrow layer.outputs.W$
          tmp_flops_{full} += $W_1 \times W_2$
          tmp_flops_{inc} += $W_1 \times W_2$
      $flops_{inc}$ += tmp_flops_{inc}
      $flops_{full}$ += tmp_flops_{full}
      $speedup[i][j] \leftarrow tmp\_flops_{full}/tmp\_flops_{inc}$
  return $(flops_{full}/flops_{inc}, speedup)$

4.1.2 Occlusion Experiment with Incremental Inference. With all the necessary core ideas explained, we now explain how incremental inference approach can be used to implement occlusion experiment (see Algorithm. 2). On a high-level the structure of this algorithm is similar to the theoretical speedup calculation algorithm. But the following differences can be noted. The algorithm takes

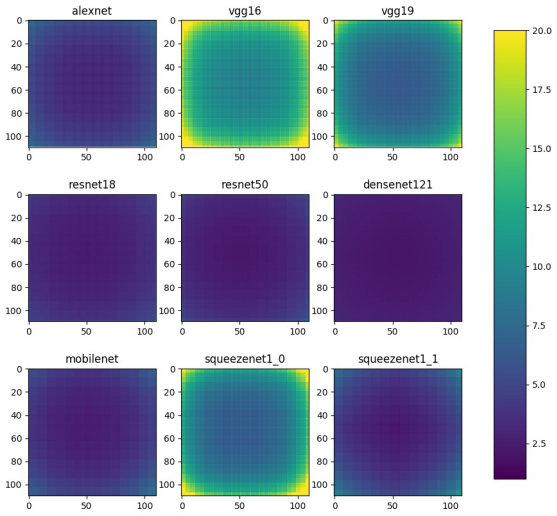

Figure 6: Maximum attainable theoretical speedup for incremental inference approach for an occlusion experiment with patch of size 16 pixels stride of 2

an input image I as input for the occlusion experiment in addition to the CNN model, patch width W_{patch}, and stride S_{patch}. It then performs a full inference on the image and obtain a list containing activations for all the layers, including input image, by calling the method `PerformFullCNNInference(CNN, I)`. It then finds the index of the predicted class label by finding the index of the maximum activation in the softmax layer, which is the last layer in the CNN. Next similar to Algorithm. 1, it iterates through all the possible positions of the occlusion patch on the input image and perform incremental inference for Conv and Pool layers based on the updated patch locations. For fully-connected and softmax layers usual full-inference is performed as there is no redundancy in computations. When performing incremental inference for Conv and Pooling layers, the updated output of the previous layers is stitched together with pre-materialized values obtained from full inference (M) corresponding to that layer to create the input patch for the incremental inference operator.

Similar to speedup calculations, for simplicity, the algorithm here assumes the CNN architecture is a simple chain styled architecture. However it can be easily extended to support more general DAG style CNN architectures. Another point to notice is that this algorithm performs inference for single occlusion position at a time. Alternatively one could batch together multiple occlusion patch positions together and perform batched inference. However since the patch sizes are not guaranteed to be the same at each layer, they may be need to be padded to transform them into the same size. Batching multiple inferences together can reduce the runtime of occlusion experiments as it can amortize the overheads specially when using GPUs for inference.

Algorithm 2 Occlusion Experiment with Incremental Inference

procedure OcclusionWithIncrementalInference(I,CNN, W_{patch}, S_{patch})
  $M \leftarrow$ `PerformFullCNNInference`(CNN, I)
  $label_{index} \leftarrow argmax(M[-1])$
  $tmp \leftarrow floor((CNN.image.W - W_{patch} + 1)/S_{patch})$
  $P \leftarrow ARRAY[tmp][tmp]$
  for `i in range(0, tmp,` S_{patch}`)` **do**
    for `i in range(0, tmp,` S_{patch}`)` **do**
      $x_{in\_0} \leftarrow i$
      $x_{in\_0} \leftarrow i + W_{patch}$
      $y_{in\_0} \leftarrow j$
      $y_{in\_0} \leftarrow j + W_{patch}$
      $x \leftarrow Zero[x_{in\_1} - x_{in\_0}][y_{in\_1}][y_{in\_0}]$
      for `k in range(0, size(CNN.layers), 1)` **do**
        $layer \leftarrow CNN.layers[k]$
        if $layer.type$ *in* $[conv, pool]$ **then**
          $x_{read\_0} \leftarrow \texttt{max(ceil}((x_{in\_0} - F + 1)/S) \times S, 0)$
          $x_{read\_1} \leftarrow \texttt{min(floor}((x_{in\_1} - 1)/S) \times S + F, W_1)$
          $y_{read\_0} \leftarrow \texttt{max(ceil}((y_{in\_0} - F + 1)/S) \times S, 0)$
          $y_{read\_1} \leftarrow \texttt{min(floor}((y_{in\_1} - 1)/S) \times S + F, W_2)$
          $tmp \leftarrow M[k]$
          $tmp[x_{in\_0} : x_{in\_1}][y_{in\_0} : y_{in\_1}] \leftarrow x$
          $x \leftarrow tmp[x_{read\_1} - x_{read\_0}][y_{read\_1} - y_{read\_0}]$
          $x \leftarrow layer.transform(x)$
          $F \leftarrow layer.filter.F$
          $S \leftarrow layer.filter.S$
          $x_{out\_0} = \texttt{max(ceil}((x_{in\_0} - F + 1)/S), 0)$
          $x_{out\_1} = \texttt{min(floor}((x_{in\_1} - 1)/S) + 1, W_2)$
          $y_{out\_0} = \texttt{max(ceil}((y_{in\_0} - F + 1)/S), 0)$
          $y_{out\_1} = \texttt{min(floor}((y_{in\_1} - 1)/S) + 1, W_2)$
          $(x_{in\_0}, x_{in\_1}, y_{in\_0}, y_{in\_1}) \leftarrow (x_{out\_0}, x_{out\_1}, y_{out\_0}, y_{out\_1})$
        else if $layer.type = fully - connected$ **then**
          $tmp \leftarrow CNN.layers[k-1].type$
          if tmp != $fully - connected$ **then**
            $tmp \leftarrow M[k-1]$
            $tmp[x_{out\_0} : x_{out\_1}][y_{out\_0} : y_{out\_1}] \leftarrow x$
            $x \leftarrow tmp$
          $x \leftarrow fully - connected(x, layer.weights)$
        else if $layer.type = softmax$ **then**
          $x \leftarrow softmax(x)$
      $P[i][j] \leftarrow x[label_{index}]$
  return $(label_{index}, P)$

5 EARLY EXPERIMENTAL RESULTS

In this section we summarize the early experimental results obtained by implementing the incremental inference for occlusion experiments.

Experimental Setup. The experiments was performed on an Intel(R) Core(TM) i7-6700 CPU 3.40GHz machine with 32 GB RAM. The machine is also equipped with a Nvidia Titan Xp GPU. We use PyTorch 0.3.1 library as the deep learning toolkit library.

Workload. We use a popular ImageNet pre-trained VGG 16 layer CNN model and subject it to occlusion experiments. The performance of the occlusion experiment with naive approach and incremental inference approach is benchmarked on CPU and GPU separately. An occlusion patch of size 16×16 was placed on the center of the image of size 224×224 and runtime for full inference approach and incremental inference approach average over 5 iterations is recorded. From these values the speedup is calculated. The experiment was repeated with a batch size of 1 and 16 (see Table. 2).

Batch Size	Theoretical Speedup	Experimental Speedup	
		CPU	GPU
1	7.6	5.4	1.3
16	7.6	5.4	1.6

Table 2: Theoretical versus empirical speedup achievable with incremental inference

Our implementation of incremental inference of Conv and Pool layers could achieve a speedup of 5.4 on CPU for a batch size of 1 and 16 compared to the theoretical maximum speedup of 7.4. However, the GPU implementation could achieve a speedup of only 1.3 and this can be increased upto 1.6 with a batch size of 16. We suspect that the random memory operations introduced by the incremental inference approach due to stitching of output of incremental Conv and Pool layers with pre-materialized activations of the full inference is throttling the GPU performance in incremental approach.

6 CONCLUSIONS & FUTURE WORK

In this work explore applying incremental inference of Conv and Pooling layers of CNN models to reduce the runtime of occlusion experiments. We formalize the incremental inference approach for CNNs and evaluate the theoretical upper-bound of speedup achievable for different CNN architectures by statically analyzing the CNN architecture definitions specified in PyTorch framework. For most CNN architectures we can achieve a speedup higher than 2 and for some CNN architectures, such as VGG and Squezenet1_0, this can be higher than 7. Our implementation of incremental inference of Conv and Pool layers could achieve a empirical speedup of 5.4 on CPU and 1.6 on GPU with a batch size of 16. We suspect the relatively low speedup of GPU implementation is attributable to the random memory operations caused by incremental inference throttling the GPU performance.

As future work we are looking in to developing a more efficient GPU implementation which can attain higher speedup. This will require implementing GPU kernels which can perform incremental Conv and Pooling operations in place without additional memory copying. We also plan to explore several other optimizations including explore and exploit style approaches on localizing the most sensitive image regions and approximate CNN inference for reducing the runtime of occlusion based CNN explainability workloads. Explore and exploit style approach is an algorithmic optimization motivated by the specific use cases of occlusion experiments. In most applications such as medical imaging the objects of interest in an image occupies a relatively small portion and are located together. In such settings rather using a patch of small size we can start with a large patch with a relatively large stride and then iteratively focus into smaller regions which appears to be sensitive for the predicted class label. Approximate inference of CNN approach is based on a general observation of deep CNN inference. Even though in theory the projective field of an input pixel grows linearly in practice the effective projective field does not grow in that rate. This mean most of the local changes in the input space are affecting localized changes in the output space of a convolution operation. Therefore we plan to experiment the possibility of constraining the growth of the projective field of an input pixel and there by reduce runtime using our *incremental inference* approach.

REFERENCES

[1] F. Arbabzadah, G. Montavon, K.-R. Müller, and W. Samek. Identifying individual facial expressions by deconstructing a neural network. In *German Conference on Pattern Recognition*, pages 344–354. Springer, 2016.

[2] H. Azizpour, A. S. Razavian, J. Sullivan, A. Maki, and S. Carlsson. Factors of transferability for a generic convnet representation. *IEEE transactions on pattern analysis and machine intelligence*, 38(9):1790–1802, 2016.

[3] M. T. Islam, M. A. Aowal, A. T. Minhaz, and K. Ashraf. Abnormality detection and localization in chest x-rays using deep convolutional neural networks. *arXiv preprint arXiv:1705.09850*, 2017.

[4] K.-H. Jung, H. Park, and W. Hwang. Deep learning for medical image analysis: Applications to computed tomography and magnetic resonance imaging. *Hanyang Medical Reviews*, 37(2):61–70, 2017.

[5] D. S. Kermany, M. Goldbaum, W. Cai, C. C. Valentim, H. Liang, S. L. Baxter, A. McKeown, G. Yang, X. Wu, F. Yan, et al. Identifying medical diagnoses and treatable diseases by image-based deep learning. *Cell*, 172(5):1122–1131, 2018.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] S. P. Mohanty, D. P. Hughes, and M. Salathé. Using deep learning for image-based plant disease detection. *Frontiers in plant science*, 7:1419, 2016.

[8] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] Y. Wang and M. Kosinski. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. 2017.

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

[11] L. M. Zintgraf, T. S. Cohen, T. Adel, and M. Welling. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.