ELIχR: Eliminating Computation Redundancy in CNN-Based Video Processing

Jordan Schmerge, Daniel Mawhirter, Connor Holmes, Jedidiah McClurg, Bo Wu
Dept. of Computer Science
Colorado School of Mines
Golden, CO, USA

***Abstract*—Video processing frequently relies on applying convolutional neural networks (CNNs) for various tasks, including object tracking, real-time action classification, and image recognition. Due to complicated network design, processing even a single frame requires many operations, leading to low throughput and high latency. This process can be parallelized, but since consecutive images have similar content, most of these operations produce identical results, leading to inefficient usage of parallel hardware accelerators. In this paper, we present *ELIχR*, a software system that systematically addresses this computation redundancy problem in an architecture-independent way, using two key techniques. First, ELIχR implements a lightweight change propagation algorithm to automatically determine which data to recompute for each new frame based on changes in the input. Second, ELIχR implements a dynamic check to further reduce needed computations by leveraging special operators in the model (e.g., ReLU), and trading off accuracy for performance. We evaluate ELIχR on two real-world models, Inception V3 and Resnet-50, and two video streams. We show that ELIχR running on the CPU produces up to 3.49X speedup (1.76X on average) compared with frame sampling, given the same accuracy and real-time processing requirements, and we describe how our approach can be applied in an architecture-independent way to improve CNN performance in heterogeneous systems.**

I. Introduction

Image processing using convolutional neural networks (CNNs) is an increasingly popular technique to address the growing amount of video data obtained from a variety of camera sources. Its accuracy and versatility have been noted for a wide variety of applications [1], [2], [3]. However, a CNN typically consists of tens or hundreds of layers, requiring a significant number of operations for processing even a single input frame. This often makes the approach impractical for real-time processing, especially as the quality and quantity of cameras increases. Ultimately, new techniques that support efficient video processing using CNNs are needed to efficiently extract meaningful results from these large data streams.

Most video applications use CNNs in a naïve way by running the model repeatedly on each frame (or a selected subset of frames). However, in practice, similarities in adjacent frames can be significant, e.g., in an input stream generated from a surveillance video. In some extreme cases, no changes may have occurred, meaning all computation is redundant, resulting in poor performance even when utilizing parallel hardware accelerators. More commonly, *most* parts of adjacent frames stay the same, with changes only affecting small regions (e.g., a pedestrian moves to a slightly different location). Since there are typically many operations in the CNN model that only traverse small local regions of the input (e.g., 2D convolution), the computations on the unchanged regions produce identical results, and do not need to be repeated.

This redundancy in video streams is well-known, and is one of the key drivers of techniques such as compression. Finding differences between images is not conceptually difficult, but significant challenges arise from identifying changes that are large enough to impact the inner layers of a CNN, eventually affecting the overall output of the model. Part of this difficulty stems from the fact that the inner workings of the model parameters may not be well understood, leading CNNs to often be viewed as "black boxes". For example, pixel values often vary due to input image noise, even if the image itself does not appear to have changed to the human eye. This makes it difficult to distinguish important changes from irrelevant ones, as we would not expect variations caused by typical image noise to change the final result, but we would expect the appearance of a new object to be recognized. However, despite this distinction being straightforward for a human, noise can lead to significant computational overhead as redundant computations will propagate through the layers of the CNN if not addressed. The input image is the input to the first layer, and each output from a layer is the input to the next, so the chain of dependencies is clear, but any inefficiency in finding the important changes for the input image will lead to additional overhead on each subsequent layer. This motivates our need to better understand how local changes propagate through the layers of the CNN.

The most basic approach for handling redundancy is to simply reduce the frame rate at which input images are sampled, in order to reduce the total amount of computation done for an input stream, and increase the likelihood that a meaningful change has occurred. For the most part, this sidesteps the issue, leaving many opportunities for performance improvement. Beyond this, prior work has already identified the importance of efficiently retaining intermediate results, which are internal computations produced by a CNN layer for use by the next. Existing techniques include approximation approaches, such as linear quantization [4] or vector centroids [5] to identify similar intermediate results, but these can suffer from inaccuracy. Another approach tracks changes at each layer, introducing more overhead [6]. What these approaches lack is

a lightweight and precise way to accurately track changes in the intermediate layers of the network.

Instead of rounding intermediate values or modifying the network, we propose a new approach called *ELIχR* to keep track of all relevant changes with a set of bounding boxes. By using a threshold to determine what changes are meaningful enough to track, ELIχR can trade accuracy for throughput in a controlled manner. In this approach, we perform an initial complete CNN inference, and save the output values of each layer for future use. With these intermediate results saved, ELIχR can make small updates in place without needing to compute or modify values that are unchanged. Because each layer expects a certain size input on each run of the model, all of the values must be passed whether they have changed or not, so we cannot simply pass only the changed values. For this reason, it is critical to save intermediate results from the prior run, as this is the only way to maintain a properly-functioning CNN while enabling local updates.

ELIχR's key insight is its lightweight approach to bound the changes, which are captured by statically-computed bounding boxes. Equally important is its ability to track those changes through the layers by a propagation step. Once the input bounding boxes and the network layers are known, the size and location of the bounding boxes on every layer can be calculated based on the operators that make up the network. This ensures that all computations depending on changed results will be computed and updated, while the rest can be ignored. Given a perfect set of bounding boxes, computing the results they contain leads to no loss in accuracy compared to a full run containing extensive redundant computation.

In addition to our static bounding boxes and their propagation across layers, we also develop a dynamic technique that attempts to reduce the size of these boxes at runtime. Because the static propagation is conservative by nature, it is fruitful to check whether a box may actually be larger than necessary by comparing its values to the stored intermediate results that are already saved. We carefully chose to implement this technique after a rectified linear unit (ReLU) layer, because the ReLU function can only return 0 if the input is negative, or the original value if it is positive. Because of this many-to-one relationship, negative values that may have varied substantially all become zero, increasing the chances of being able to shrink a bounding box. If many intermediate results are identical, they will lead to redundant computation, so the bounding box can shrink to include only the relevant values. This technique helps ELIχR adapt to a given input at runtime and contributes significantly to the overall achieved speedup.

We test ELIχR with two real world CNNs, Inception V3 and Resnet-50, and use two real world video streams as their input. These experiments are able to show a clear tradeoff based on the required accuracy and desired sampling rate. We show up to 3.49x speedup running on the CPU, with a controllable loss in accuracy compared to frame sampling. Even with busy input streams and complex real world models, ELIχR still shows a significant improvement over the basic uniform sampling approach. Our approach is not architecture-specific, and offers further improvement on heterogeneous architectures, especially when parallel hardware accelerators are available—not only can many frames be processed in parallel, but our approach can ensure that batches of non-redundant operations are supplied to the accelerators.

Fig. 1: Two sample frames from a "town plaza" scene. Even though many objects are moving, the overall change between images is still relatively small (third image shows difference).

Our contributions are as follows. We introduce ELIχR, an inference framework that can take a user-defined accuracy loss and sampling rate, and automatically process the stream. We make use of our new static and dynamic recompute techniques to calculate the relevant changes, while automatically ignoring the redundant computation. We also give insight into trading accuracy for performance, and characterize the situations where our approach can offer speedup.

II. CNN Background

Before describing our approach, we begin with some background on CNNs and video processing.

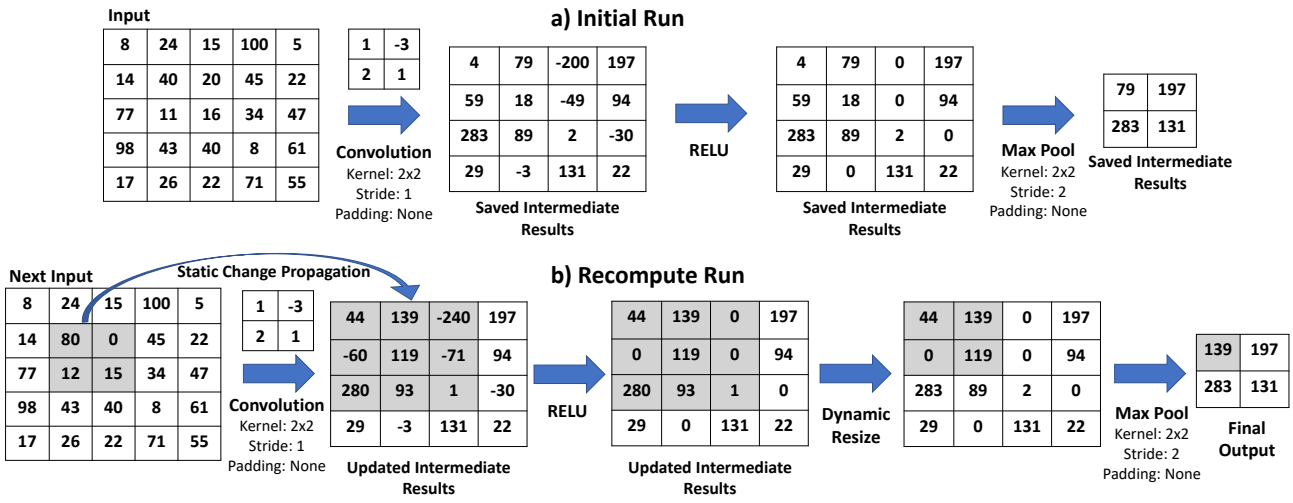

Fig. 2: In (a), the input matrix is on the left, and the 2x2 kernel will be applied to produce the next input for ReLU, which is then passed to the max pooling layer to produce the final output. In (b), the changed area (shaded region) changes size based on the kernel, then passes through ReLU, before the dynamic recompute changes the area again, this time shrinking it—because of some 0's that are identical, and a several values that are within a threshold, the area can be shrunk before moving to max pooling, and ultimately producing the final output with only a fraction of the computation.

A. Convolutional Neural Networks

CNNs are a subset of neural networks that are frequently used for image processing, due to their versatility and ability to build spatial understanding for an input. CNNs are typically constructed from convolutions, using filters, pooling layers, and non-linear activation functions. Convolutional layers have a trained kernel of fixed dimension (typically in two dimensions for image processing), which is multiplied element-wise with spatially local pieces of the input. Dimensionality of the output is determined by the size of the kernel, and the stride at which the convolution is performed. Pooling layers are frequently used to reduce dimensionality of intermediate results by distilling the local information so that future convolutions can determine relationships across disparate portions of the input more efficiently. Common pooling operations include maximums or averages of a region of the input. Activation functions are used to introduce non-linearity into the CNN. Common activation functions include ReLU, sigmoid, hyperbolic tangent, and derivative variations of these. In this paper, we will primarily consider ReLU, which returns $max(0, input)$. A simple example of a portion of a CNN may be found in Figure 2a.

Execution of CNNs is dominated by time spent on the convolutional layers. These are typically executed using a two-stage process of data reorganization and a dense matrix-matrix multiplication. In the data reorganization stage, the current filter window in the convolution is unrolled from its two-dimensional organization into a row-vector in an input matrix. Once all filter windows have been processed, the constructed matrix may be multiplied directly against the filter matrix, where each column-vector constitutes a filter, producing the layer output. This process is demonstrated in Figure 3.

B. Video Processing

With the emergence of the Internet of Things (IoT), using cameras for surveillance and other tasks has become a major application of CNNs. CNN functionalities include object detection, where the network is trained to build bounding boxes around specific types of objects in each frame, and semantic segmentation, in which each pixel is classified as belonging to some output class. CNNs are often used to process each frame completely, and are either applied to every frame or a subset of frames based on a sampling rate. We expect the number and size of CNN use-cases to increase, motivating the development of advanced systems to execute them more efficiently.

III. Motivating Example

A. Redundancy in Video Streams

Modern cameras frequently produce high resolution (e.g., 1080p) streams at high frame rates (typically 30 fps or greater). In this context, changes to the *entire image* tend to occur infrequently. Consider the two example input images from a test data stream in Figure 1. These images, taken about a second apart, and having dozens of intermediate frames, are highly similar, with the majority of pixels not having undergone significant change—in practice, these pixels are often part of the background or a stationary object, so no new information is needed about them. Pixels primarily change values when (1) a new object is being represented in the image, or (2) the value has modulated due to image capture noise. which is a common phenomenon due to imperfect analog input

cameras. While changes due to noise are often not noticeable to the human eye, the actual pixel values can vary significantly.

Regardless of the reasons for differing pixel values, these must be considered when performing an inference for the model. If the first layer of the model were fully connected, meaning each input value affected every output value, then even a single pixel changing by a small amount could account for significant changes, which could propagate through the layers and result in large changes later in the network. However, as described above, CNNs perform computations and understand relationships locally over the dimension of a kernel, meaning that regions having no changes will give the same results as the previous run, and will have no cascading effects. Therefore, computing them is entirely redundant, as seen in Figure 2b. The kernel will be convolved across the input, once for each value in the output matrix. Regions of change will clearly need to be recomputed with the new values, and can be seen in the shaded regions, but if these regions are sparse, a significant amount of computation can be saved. In theory, if all of the changed regions are fully tracked and recomputed, there would be no loss in accuracy while still reducing computation.

B. Opportunities

To reduce computation, the framework only needs to unroll patches if they have been impacted by a change in input value. This will be determined initially by the bounding boxes created on the input image, and later by the propagated boxes from our static recompute technique. We will also refer to this as change propagation, an example of which can be seen on the convolution layer in Figure 2b as the box changes size. If a large portion of the input has remained unchanged, we can significantly reduce the size of the input matrices. In Figure 2b, there would only be 9 rows in the input matrix instead of the 16 that would otherwise be needed to compute the output of the layer, cutting the number of operations by nearly 50%. This can be reduced further after a ReLU layer, as some negative values may have become zero, allowing us to further shrink the box to save computation later.

C. Challenges

Some practical engineering problems exist that complicate implementation. It is necessary to develop rigorous methods to correctly propagate the changes across layers, prevent bounding boxes from growing incrementally larger, and discriminate input image noise from the change in signal. If the process to determine which patches need to be computed were to introduce significant overhead, general matrix multiply (GEMM) savings may be eliminated. Since CNNs do reduce spatial dimensionality with strides and pooling layers, minimizing the growing effects of this process can help ensure that speedup occurs in all layers of the network. Since camera noise is a common occurrence, poor determination of which pixels have changed due to noise can force significant redundant computation, dramatically reducing the speedup.

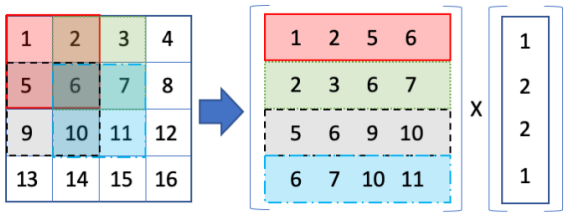

Fig. 3: Convolutions implemented as an unrolled matrix multiplication. This figure shows all patches that touch the upper left 2x2 area—as the kernel is convolved, it will unroll each of these patches and add the rows the matrix. Then these row vectors will be multiplied by a column vector that represents the values of the kernel filters to produce the overall output.

IV. ELIχR: Eliminating Redundancy

We first present our system at a high level by discussing the main components before explaining each in more detail. When the system first starts on an input stream, it executes a full run using the standard CNN computation. For the first image examined, there is nothing to compare to, and no computation is redundant. During this run, the first image is saved along with all intermediate results, so that they can be utilized on subsequent runs.

When processing the next image, new opportunities arise. First, change detection will occur to understand what areas need to be recomputed. This will be manifested as a set of bounding boxes that contain all the relevant changes. Any computation outside these bounding boxes is now known to be redundant and can be ignored. Instead of wasting cycles to compute these results, the intermediate results that were saved before can be used instead. These boxes will move through the layers based on our static recompute method that propagates them across layers, in turn showing each layer the computation that needs to be done. These updates will then locally be inserted into the existing intermediate results to produce the new output.

These bounding boxes can change size based on the convolution and pooling layers, and may also be shrunk by our dynamic recompute method that occurs after a ReLU layer. In addition to taking a CNN model and input frames from the user, our system can also take a maximum loss and maximum stride. This will guide the system to profile some sample frames and select a stride that will be used to process all the frames in the input. After discussing each piece in detail, we will analyze and discuss the results of our experiments.

A. Change Detection

Having done a full run for the first image, only the changed regions are of concern for subsequent runs. Using the previous and current images, the goal of our change detection algorithm is to create a set of bounding boxes that capture all the changes

with minimal excess. We care about these changes exclusively, because only computations within these bounds can cause updates to previous intermediate results.

In order to do this, the image is first treated as a series of smaller boxes. In order to mitigate the effects of noise on one pixel, all pixels in the box will be considered together. If, on average, all the pixels in the box have changed more than a threshold, that box will be marked as needing an update. To further mitigate noise, a 3x3 Gaussian blurring kernel is used, so that nearby pixels can be factored into deciding how much a single pixel has changed. The fact that the whole box is being considered together ensures that this approach effectively handles noise.

The threshold to decide if a box should be updated is based on the average change of all pixels in the image. This is quickly computed beforehand for each new image, so that it can be adaptive to different levels of change. We make a general assumption that important changes will exceed this threshold. When this step is complete, there is a grid of boxes with some marked for updating. Next, we attempt to merge some of these together to minimize the number of boxes we must track. Boxes must be square or rectangular, as they are defined with an x-y coordinate for their top left corner and dimensions for that box. We use a straightforward algorithm to merge nearby boxes, trading simplicity for optimal merging. Once this is complete, our system takes this set of bounding boxes and the input image, and begins inference.

B. Static Recompute

Our static recompute technique was motivated by the need to fully understand how changes could move through the network. By quantifying what computations these changes could affect, we can properly do partial computation on each layer, even as these changes propagate. To see how this works, first recall the bounding boxes that were created in the change detection step above. If all operations were simple mappings like ReLU that did not change the layer size, then the boxes created previously would be acceptable to use for the whole run. However, operations that have kernels, like conv, max pool, and avg pool, frequently change the size of the layer, thus changing the size of the bounding boxes. This is shown in Figure 2b, where the shaded area represents the changed area. Note the size and location are not the same after the kernels are applied for convolution or max pooling. To visualize how this algorithm works, consider the set of all patches that would be produced by convolving the kernel across the entire input. The algorithm selects only the patches that would have at least one value in the area of a bounding box. These patches map directly to the output values that will be produced as input to the next layer, so they must be computed and inserted into the prior intermediate results. Additionally, because they were computed with new values, they should become part of the set of boxes that propagate to the next layer.

These boxes frequently grow depending on the filter size, stride, and padding of the layer, so the algorithm must be able to map any set of boxes to the corresponding input patches. Since the next layer can be a different size, the boxes should also change size to capture all values that were updated because of the bounding boxes on the last layer. Our work modified the operators to do the partial computation with these boxes, and built a function to propagate these boxes across the layer. This propagation is what CBInfer [6] calls "worst case change propagation", essentially considering every possible patch that could touch a changed area. A key part of our reasoning for developing this algorithm and accepting its overhead is that these static boxes and change propagation give a much faster way to determine where our layers should focus, even allowing all later boxes to be computed statically based only on the input change detection and given layers of the model. This tradeoff allows us to avoid performing real time change detection on each layer, which is the only other alternative. Our static technique is key to obtaining speedup, and is crucial for enabling our next technique.

C. Dynamic Recompute

When developing our system, we understood that our static recompute technique meant the bounding boxes were being propagated in a worst-case manner, and were possibly leaving some performance on the table. Without any intervention, the boxes will generally continue to grow as execution moves deeper through the network. Larger boxes means more computation, and less speedup. Our insight is that in some cases, we can shrink the perimeter of these boxes without introducing a significant amount of error, by using previous intermediate results again. By comparing values to their counterparts on the last run, we can see how much they have changed over the course of the current run, to see if shrinking the box would be possible.

One of the main reasons that values may be similar is that noise in the input image can force our early boxes to be larger than needed. The effects of noise are often reduced further into the network after many kernels have been applied. Another possibility is that many values may have been negative, but these will become zero after a ReLU layer. Regardless of the magnitude of the negative value, ReLU causes both to become zero. With this mind, we use another threshold to shrink a box to its minimum size. This threshold is necessary because floating point values may naturally vary, and forcing values to be identical removes much of the potential.

A box can only be shrunk by an entire row or column at a time, since each box must maintain its rectangular shape. Because of this shrinking, dynamic recompute can contribute speedup on top of what the static idea could do on its own. Carefully shrinking boxes will allow subsequent matrix multiplications to be smaller in future convolutional or pooling layers. Stated another way, the static boxes tell our recompute engine where to work, while the dynamic method allows us to keep that box as small as possible to further reduce overhead.

D. New Operations

In the base system, all operations were implemented to compute on the full input of the layer, with no regard to

whether that computation was producing fresh or redundant results. Therefore, modifications needed to be made at the operator level to take advantage of the bounding boxes and do the local updates correctly. At minimum, we still need the ability to do the full computation on the very first run, so the original compute method is clearly still needed. Therefore, both methods coexist, the first doing the computation as the original system did, and the new mode doing recomputation given a set of bounding boxes. For the recompute mode, the layer can map the inner areas of those boxes to the relevant input values. For example, on a convolution layer, this means finding all the input patches that touch a particular bounding box (see Figure 3). This is repeated for each box, as there will frequently be more than one. Only these patches need to be considered—the rest can safely be ignored.

Since there is a one-to-one mapping between input patches and output boxes, our system actually uses the boxes from the next layer to determine the relevant patches. Max and average pool are implemented very similarly because they do their computation with similar kernels. In terms of mapping the boxes to the input values, the mapping is computed on the fly to avoid the memory overhead of storing a list of patches. This also allows parallelization of the loops, as each worker can manage a chunk of tasks, and compute its patch number on the fly, removing any dependencies. ReLU is simple, as it is only applied to individual values. Therefore, it applies the ReLU function only to the values contained in the bounding boxes, as they are the only ones that could have changed. ReLU will also implement the dynamic recompute technique for reasons discussed previously. It is important to have the intermediate results from the last run, because the final step of these modified operations is to do a partial update. Specifically, the newly-computed values are overwritten into the old intermediate results, and then all are passed together to the next layer.

E. Other Considerations

There may be many bounding boxes that are moving between the layers. If at some point a single box fills the entire layer, it is clearly redundant to keep track of all the other boxes, so we eliminate them. In the case of 1x1 kernels with a stride of 1, the function for change propagation does not need to be called, as the layer and box sizes do not change, just like for ReLU. Additionally, if enough of the input image has changed, the overhead of the recompute strategy would be too great. In that case, our system simply falls back on the original method. Experimentally, this cutoff corresponds to the current image having more than 50% of its pixels changed. As other works have noted, subtle but continuing changes can be a problem, as they may never exceed the threshold, but their effect can still accumulate over time. To combat this and other error accumulation, our system can periodically do a full run between recompute runs to refresh its ground truth and ensure that errors do not grow unchecked. This is currently set at every 15 input images, but this can be tuned based on the input stream, and the level of acceptable loss.

V. Evaluation

We evaluate ELIχR using two real world CNN models: Inception [7] and Resnet-50 [8], on frames from two different input streams [9] [10]. We believe the structure of these networks and the noise in the video data make this evaluation representative of real-world workloads.

The baseline comes from a uniform sampling approach which processes a subset of the frames using a fixed-time stride. Both uniform sampling and partial recomputation introduce error, and this evaluation will test a range of sampling strides and recompute thresholds to explore the trade-offs between them. We run all of our experiments on a system running Ubuntu 16.04 with an Intel Xeon E3-1286 v3 and 32GB of memory. We focus on CPU performance based on the real use cases described in [11], [12].

A. Approaches

When running a full image model on every frame of a video becomes too expensive, the simplest way around the problem is to run on only a portion of the frames. It is reasonable to expect a maximum bound on how many frames can safely be skipped—we call this a realtime requirement. By performing the computation on every n^{th} frame, uniform sampling assumes the result for every skipped frame is equal to the result of its most recently sampled frame. As long as the image sequence always changes slowly, this can be a viable strategy. CBInfer [6] uses an approach which is a combination of our change propagation and dynamic recompute steps, but our static approach and combination of techniques is unique in the literature, to the best of our knowledge. Our system's approach leverages the static approach of uniform sampling alongside a dynamic approach which detects and recomputes changing parts of the stream. This hybrid approach is expected to decrease the average processing time per frame while maintaining high accuracy, which uniform sampling can struggle to accomplish. The reported execution times include all overhead for change detection as well as the static and dynamic computations to make an end-to-end comparison. We parameterize our system according to an error target and maximum stride to enable user configuration.

B. Theoretical Performance

Given a maximum acceptable error target, selecting a stride for uniform sampling is still difficult, as it depends on the input data. To obtain the theoretical peak performance of uniform sampling, we allow an oracle to run the input with all possible strides, and record the error, noting that the highest stride achieving a particular error target has the highest performance.

Comparing to this theoretical peak for sampling, Figures 4 through 7 show the trade-off for each model and input stream combination according to the recompute parameters. The error targets on the y-axis come from the error range that our recompute sampling can achieve, the x-axis bounds the stride to a maximum value, and the darkest areas mark conditions under which recomputation cannot offer speedup. Such conditions occur in Figures 4, 6, and 7 only in the lower-right area when

the error target is tight, but a relatively high stride can still achieve the target. This limitation is expected, as skipping large amounts of computation offers substantial performance advantages, and even the process of difference checking can become a performance limitation. Any time recomputation is triggered, our system will spend some time on that frame, but in the case where changes are insignificant, the result may still be comparable. The lighter horizontal band in the middle of Figure 5 has a more interesting cause. Since the peak error of uniform sampling can vary substantially with small changes in stride, a small change in error target can substantially change the feasible strides for a particular input. These jumps can produce bands in the data for which uniform sampling can perform very well, and the recomputation cannot keep up. Even with these scenarios considered, our system produces an average of 1.76X speedup (up to 3.49X speedup) over the best case for uniform sampling. Additionally, we note a stride of 1 in the uniform sampling approach would correspond to an unmodified CNN executing on all frames and in all cases we can see a strong improvement indicated by the bright yellow column on the left of all plots.

C. Real-World Performance

Our theoretical analysis cannot lead to a feasible approach for a real system, as the oracle will not be available to make perfect decisions. To make this system usable, we added the ability for our system to select strides automatically. For both approaches, our framework allows a user to specify an acceptable error and a maximum stride. This stride should be based on the expected rate of changes in the stream, as it should be frequent enough that nothing can go undetected between inference runs. Given this, our system will automatically select a stride that is less than or equal to the maximum stride. The goal is to pick the largest stride that will satisfy the accuracy requirement, as fewer runs will require less execution time. To make this decision, we test over a range of strides and then measure the relative error of the output vectors compared to the ground truth.

Skipping images introduces some error for the frames that are not processed. In order to select a stride, our system takes a sample of frames, ideally as representative of the stream as possible, and computes the ground truth frames as well as the output from the uniform or recompute approach. It quickly computes the relative error between the output vectors, using the most recently computed output in cases where a frame was skipped. This error is averaged over all the frames and compared to the user specified requirement. If it is below, we increment the stride and try again. Once the error is exceeded, we select the previous stride. Processing images more frequently than the user-specified maximum stride is acceptable, if necessary to meet the accuracy requirement. While our loss estimate cannot have guaranteed accuracy, we believe it can be tuned in a real system to provide reasonable results. It will become more accurate as the sample frames become more representative, and with this selected stride, our system can proceed to process the entire stream.

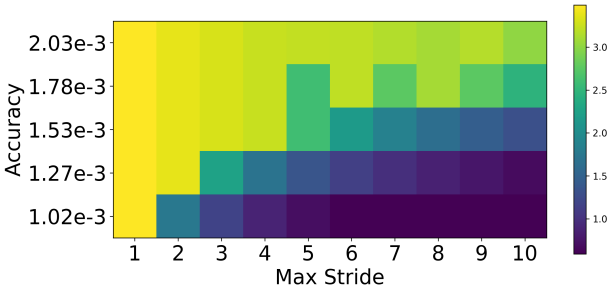

Fig. 4: This plot shows the tradeoff between stride and accuracy for the Inception Model on the Parking Lot scene.

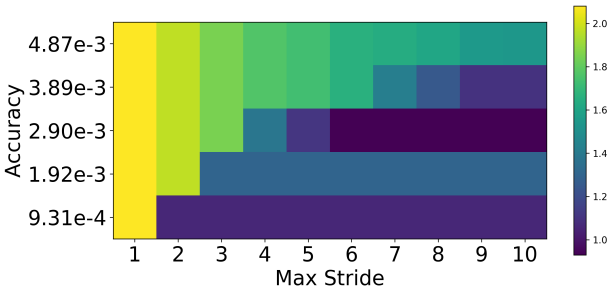

Fig. 5: This plot shows the tradeoff between stride and accuracy for the Inception Model on the Town scene.

D. Convolution Layers

As part of our analysis, we wanted to carefully understand where our speedup was coming from, and if our intuition was correct that our methods could be effectively implemented on convolution operators. As previously discussed, convolution layers dominate execution time for inference on CNNs. In our experience, they account for about 85% of the execution time and other sources in the current literature show similar results [6]. With this in mind, it is clearly most fruitful to focus on improving the speedup of the part of our system that dominates execution time. We measure the time to execute each convolution operator in the network and consider the speedup gained due to our static recompute technique. This can be seen in Figure 8. The idea clearly shows substantial improvement on the early layers, but because of the boxes growing relative to the layers, a clear downward trend of the speedup for each of the 94 convolutions can be seen.

This is where the motivation for our dynamic recompute technique arises. In Figure 9 a more sustained speedup can be seen, due to shrinking the boxes as the layers progress. Figures 10 and 11 show the same plots for Resnet, and a very similar trend appears. This shows that the idea is effective

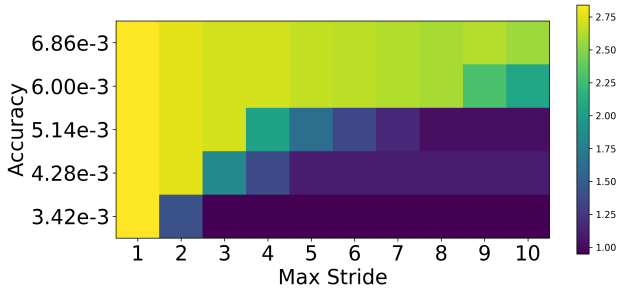

Fig. 6: This plot shows the tradeoff between stride and accuracy for the Resnet-50 Model on the Parking Lot scene.

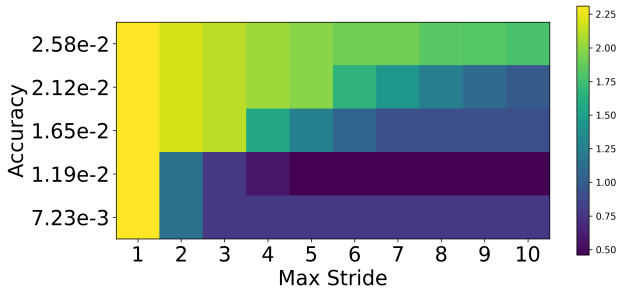

Fig. 7: This plot shows the tradeoff between stride and accuracy for the Resnet-50 Model on the Town scene.

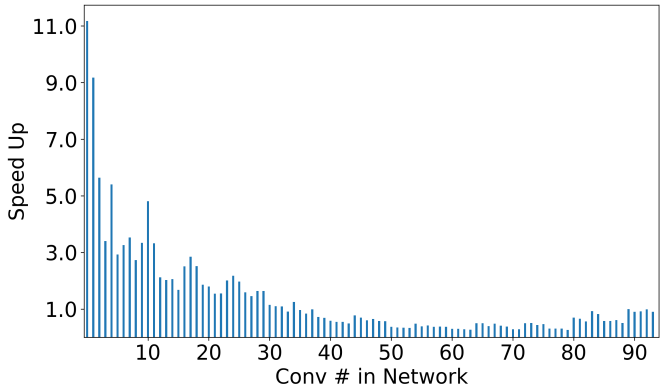

Fig. 8: This plot shows the speedup of each individual convolution layer in the Inception network for the parking lot scene. Only the static recompute technique was used.

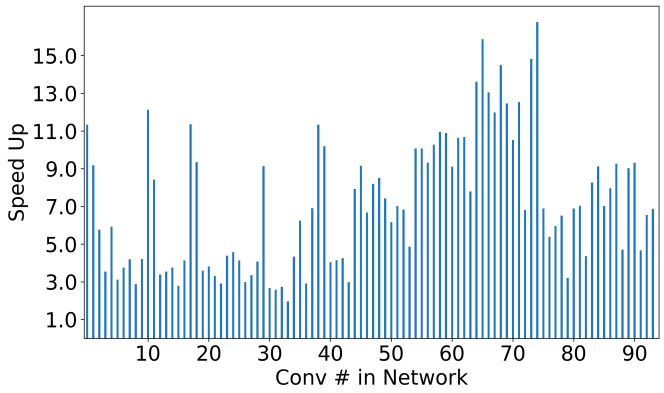

Fig. 9: This shows the speedup of each layer again with both the static and dynamic recompute techniques enabled.

for the layers that dominate execution time, and can therefore contribute to speeding up the execution of the model overall.

The static recompute is also implemented for the max pooling, average pooling, and ReLU layers. Because max and average pooling do local computation with a kernel sized chunk of input and ReLU is very easy to localize, these layers will also benefit from these techniques. Because of this, a speedup is also expected in these layers, but it will clearly be less significant overall since all other operators are only about 15% of the total execution time. Therefore we do not show layer-by-layer speedups separately due to space. It is important to consider that these speedups may become more significant, as the already-noted improvement of the convolutional layers means they will no longer dominate execution time as much, causing speedups in other areas to become more significant.

VI. Future Work

We believe that this work is an important step towards mitigating problems related to redundancy in video processing, but there are several potential directions in which this work could be extended.

- There may be a better way to detect and merge boxes on the input layer. The performance of our system is heavily tied to the size and count of the bounding boxes, and having fewer and smaller boxes can improve speedup significantly. Because of input noise and the complexity of non-rectangular bounding boxes, we leave this topic for future exploration. One disadvantage of tracking bounding boxes is the associated space overhead. However, if reducing computation is more important than memory usage, this tradeoff may be acceptable.
- There are many experimentally-chosen parameters in our system that could possibly be tuned in a more rigorous way. This includes the cut-off threshold that decides when to do full recompute, the static threshold used to detect changed boxes on the input layer, the size of those boxes, and the dynamic threshold used when shrinking boxes. That being said, achieving our accuracy targets while still providing speedup strongly indicates that we have used reasonable values for our experimental setup.
- We hope to further integrate the ideas described in this paper into other applications, like an automated compiler or full object detection framework. While our prototype system is approaching this stage, more operators are

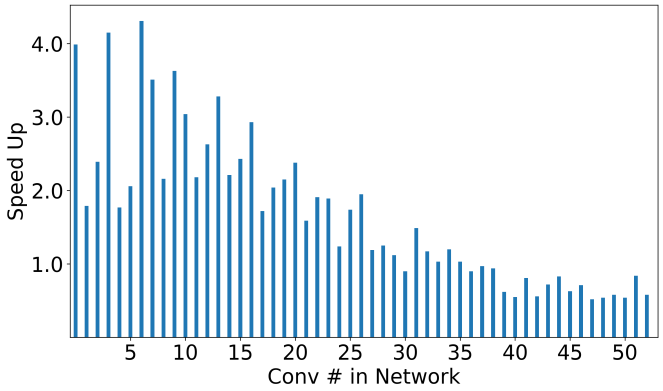

Fig. 10: This plot shows the speedup of each individual convolution layer in the Resnet-50 network on the town center scene. Only the static recompute technique was used.

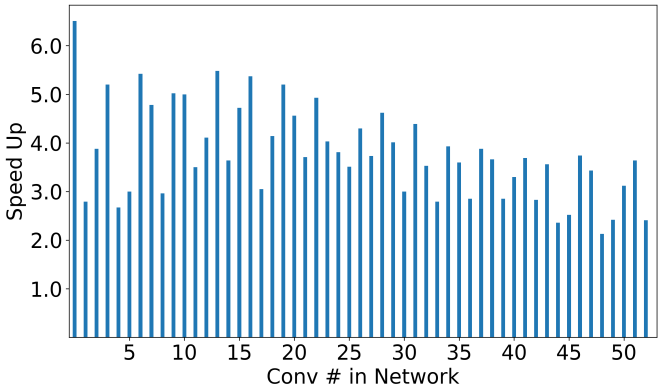

Fig. 11: This shows the speedup of each layer again with both the static and dynamic recompute techniques enabled.

needed to make the system fully general. Object detection, as seen in [13], [14], [1] makes heavy use of CNN models, and our approach could be integrated into a system like those.

- While our experimental evaluation focused on performance of our approach on a CPU, our techniques also extend naturally to heterogeneous architectures. Specifically, we currently perform a full computation every 15 frames to mitigate error propagation. All inferences done between full computes are recompute runs that leverage the intermediate results of the previously-processed image. If parallel hardware accelerators are available for inference, then there is an opportunity to do all intermediate recompute runs in parallel by dispatching frames to different coprocessors in realtime. The central compute node would be responsible for doing the sparsely-needed full runs, as well as dispatching tasks. A dispatch would consist of an input image on which to run inference, as well as the most recently computed intermediate results. This would mean that all recompute runs use the first computation as their baseline. The main consequence of this is that it may cause some of the bounding boxes to be larger than in our experiments, but the effects of this should be offset by the parallel processing it enables. In summary, this would allow the benefits of a heterogeneous environment to mesh with the redundancy elimination enabled by ELIχR.

VII. Related Work

In this work, we have developed a purely-software approach. There are other existing software approaches to address the problem of redundancy elimination and performance improvement of CNN inference. Some approaches use techniques from more than one area, just as our approach leverages the compilers-focused redundancy elimination technique. Finally, we briefly discuss the works that are most similar to ours, and explain how our approach differs.

A. Compiler Approaches

Early work has been done using compiler approach Halide [15] to better understand and exploit the locality and redundancy in image pipelines. Because of the complexity associated with optimizing these kinds of workloads, the compiler has been able to outperform even expert hand tuning. More specific to deep learning, Latte [16] introduces a domain specific language to allow users to express layers in a more natural way without sacrificing performance. This work is primarily concerned with expressing the model, whereas we focus on inference of an already-trained model. Because specialized hardware is becoming more prevalent, the ability to compile with that hardware's strengths and constraints in mind is important, as in [17]. Since in this paper we focus on CPU execution, most of the precision and limited functionality concerns are different from ours, but the motivation of improving performance for CNNs is similar. TVM [18] and DLVM [19] also address the need to perform optimizations and produce code for varied hardware backends in a safe and efficient manner. They introduce techniques like operator fusion, and can make changes to the computation graph. Intel has also developed a compiler with similar goals that is specifically for deep learning called nGraph [20]. These works apply to similar workloads, but perform optimizations in ways that are mostly orthogonal to our work.

B. Network Modification and Approximations

Another popular way to improve performance is via modifications to the network. Approaches vary, but examples include changing the precision of floating point values in the network or even changing the model itself. For example, a spiked neural net [21] changes the weights and activations to be more similar to the human brain, and may require much fewer total operations to get its output. Some models [22] require weights to be binary to save significant computation costs by approximation convolutions. An approach leveraging spatial and temporal redundancy is outlined in [23]. Here the authors want to pass changes between layers, and use rounding instead of needing to recompute each frame. In a paper on recurrent

scale approximations [24], the authors identify redundancy when doing object detection. They develop a technique to compute a feature map once, and use it to approximate future computations. A unique way to approximate a CNN is to not evaluate the CNN on all of the frames. In [25], the authors choose to index each frame using a cheaper CNN and then when querying the video stream, pull the frames corresponding to that index, and run a more expensive CNN only on the relevant frames. There have also been efforts to change certain operators in the network to improve performance while maintaining accuracy, like [26], which replaces certain bottleneck convolutions with a redesigned one. [27] makes changes to the training and execution of the network to improve performance by deciding when (or if) a certain convolution kernel should be applied to a given input. While all of these approaches work to save computation and execution time, they tend to introduce much more error than our approach, and may also require changes to the network itself.

C. Hardware Approaches

Another area of research has been to focus directly on hardware implementations to realize performance improvement for CNNs. While high level goals may often be similar, the implementation and results can vary considerably. EVA^2 [28] proposes to use activation motion compensation to do incremental updates based on the changes detected in related images. In [4], they propose to reuse values that have not changed significantly, but to improve the amount of computation that can be reused, they apply linear quantization to the intermediate results. This results in some error, but greatly increases the amount of reusable values. In [29], the authors realized that many values would be negative, and further operations would be wasted because ReLU would set the values to 0. They propose to reorder computations so that a simple sign check could determine if computation should continue. They also introduce an exact mode and an approximate mode, and discuss the tradeoffs of introducing error to gain performance. It is also possible to design hardware to improve the performance of CNN inference, as in [30]. Finally, [31] proposes to evaluate CNN layers on an FPGA, and allocate resources to different processors that are best used for a certain type of layer. This leads to higher efficiency and higher throughput when evaluating a complete model.

D. Software Approaches

The following works are the most similar to our software approach. An early work that investigated the performance of CNNs for vision tasks is DeepMon [32]. The authors used GPUs to process certain layers in early CNN models. However, they are more concerned with improving performance for mobile applications than eliminating any inherent redundancy in the inputs. NoScope [33] takes a different approach—the authors identify that there may not be a need to evaluate a complex model when certain situations may only see a limited set of the possible classifications. They create a framework to automatically train smaller and faster models and explore how to decide which model to run. In a similar spirit, [34] chooses to focus on a lower resolution image first and only use the full high resolution image if needed, allowing for a possible early exit from the network if classification confidence is high. CNNCache [35] and DeepCache [36] choose to use a cache-like mechanism that can identify similar regions and propagate cached results to later layers. This can avoid the redundant task of recomputing the outputs of those regions as the cached values can quickly be accessed and used instead. [37] also chooses to change how convolution is implemented, but with a focus on redundancy across the color channels in an image, while [38] learns to focus on only specific regions of the input using trainable masks. In Deep Reuse [5], the authors also look for similarities in intermediate vectors and propose to use clusters and cluster centroids to save computation instead of using the full vectors for all computations. The technique most similar to ours is CBInfer [6]. CBInfer acknowledges what we call static recompute as worst case change propagation. Because of their focus on GPU computation of the layers, they do not further investigate this idea. Instead, they do change detection between the layers by parallelizing it with many GPU threads. Our static and dynamic recompute methods allow us more opportunities to quickly propagate the changed areas and shrink them to avoid some of the overhead that the CBInfer identifies as prohibitive.

VIII. Conclusion

In this paper, we have introduced ELIχR, a new framework for performing inference on CNNs. With the introduction of a complete system that implements a static recompute technique to propagate bounding boxes across the layers, and a dynamic technique to resize the boxes, we are able to track all relevant changes in the network, and significantly reduce redundant computation. Based on our experiments with two real world CNNs and two real world input streams, we are able to show up to 3.49X speedup (1.76X on average), and give greater insights into trading accuracy for performance when executing CNN inference.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the National Science Foundation under awards CCF-1823005, an NSF CAREER Award (CNS-1750760), and CCF-2018910.

References

[1] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[2] A. Bhandare, M. Bhide, P. Gokhale, and R. V. Chandavarkar, "Applications of convolutional neural networks," 2016.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[4] M. Riera, J.-M. Arnau, and A. González, "Computation reuse in dnns by exploiting input similarity," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, pp. 57–68.

[5] L. Ning and X. Shen, "Deep reuse: Streamline cnn inference on the fly via coarse-grained computation reuse," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. ACM, 2019, pp. 438–448.

[6] L. Cavigelli and L. Benini, "Cbinfer: Exploiting frame-to-frame locality for faster convolutional network inference on video streams," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 30, no. 5, pp. 1451–1465, 2020.

[7] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1–9.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[9] S. Oh, A. Hoogs, A. Perera, N. Cuntoor, C. Chen, J. T. Lee, S. Mukherjee, J. K. Aggarwal, H. Lee, L. Davis, E. Swears, X. Wang, Q. Ji, K. Reddy, M. Shah, C. Vondrick, H. Pirsiavash, D. Ramanan, J. Yuen, A. Torralba, B. Song, A. Fong, A. Roy-Chowdhury, and M. Desai, "A large-scale benchmark dataset for event recognition in surveillance video," in *CVPR 2011*, June 2011, pp. 3153–3160.

[10] B. Benfold and I. Reid, "Guiding visual surveillance by tracking human attention," in *Proceedings of the 20th British Machine Vision Conference*, September 2009.

[11] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 620–629.

[12] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USENIX Association, 2018, pp. 951–965.

[13] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 1, pp. 142–158, Jan 2016.

[14] R. Girshick, "Fast r-cnn," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ser. ICCV '15. IEEE Computer Society, 2015, pp. 1440–1448.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. ACM, 2013, pp. 519–530.

[16] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman, "Latte: A language, compiler, and runtime for elegant and efficient deep neural networks," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. ACM, 2016, pp. 209–223.

[17] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. ACM, 2018, pp. 448–460.

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USENIX Association, 2018, pp. 579–594.

[19] V. A. Richard Wei, Lane Schwartz, "DLVM: A modern compiler infrastructure for deep learning systems," 2018.

[20] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, "Intel ngraph," *SYSML*, 2018.

[21] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, 04 2018.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Springer International Publishing, 2016, pp. 525–542.

[23] P. O'Connor and M.Welling, "Sigma delta quantized networks," *ICLR*, 2017.

[24] Y. Liu, H. Li, J. Yan, F. Wei, X. Wang, and X. Tang, "Recurrent scale approximation for object detection in cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017, pp. 571–579.

[25] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu, "Focus: Querying large video datasets with low latency and low cost," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USENIX Association, 2018, pp. 269–286.

[26] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *CVPR*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 6848–6856.

[27] N. Fragoulis, I. Theodorakopoulos, V. K. Pothos, and E. Vassalos, "Dynamic pruning of CNN networks," in *IISA*. IEEE, 2019, pp. 1–5.

[28] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson, "Eva2: Exploiting temporal redundancy in live computer vision," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, pp. 533–546.

[29] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, pp. 662–673.

[30] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello, "An efficient implementation of deep convolutional neural networks on a mobile coprocessor," in *MWSCAS*. IEEE, 2014, pp. 133–136.

[31] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, 2017, pp. 535–547.

[32] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. ACM, 2017, pp. 82–95.

[33] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1586–1597, Aug. 2017.

[34] L. Yang, Y. Han, X. Chen, S. Song, J. Dai, and G. Huang, "Resolution adaptive networks for efficient inference," in *CVPR*. Computer Vision Foundation / IEEE, 2020, pp. 2366–2375.

[35] P. Wang and J. Cheng, "Accelerating convolutional neural networks for mobile applications," in *Proceedings of the 24th ACM International Conference on Multimedia*, ser. MM '16. ACM, 2016, pp. 541–545.

[36] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '18. ACM, 2018, pp. 129–144.

[37] J. Liang, T. Zhang, and G. Feng, "Channel compression: Rethinking information redundancy among channels in CNN architecture," *IEEE Access*, vol. 8, pp. 147 265–147 274, 2020.

[38] T. Verelst and T. Tuytelaars, "Dynamic convolutions: Exploiting spatial sparsity for faster inference," in *CVPR*. Computer Vision Foundation / IEEE, 2020, pp. 2317–2326.