Edge Computing for Smart Buildings

By

Elaina Chai & Boris Murmann

CIFE Technical Report #TR236
November 2019

STANFORD UNIVERSITY
1 Overview

The astonishing progress in deep learning and inference techniques has opened up new capabilities and possibilities for vision-based smart buildings. However, state of the art in computer hardware has not kept up with the increasing computational and data requirements of these vision algorithms. As a result, existing real-time systems are not scalable due to the unmanageable amounts of raw data that need to be piped between local/edge sensors and remote servers in smart building applications. We propose a custom solution for vision-based smart buildings. By leveraging edge processing, the solution can use cascaded classifiers to reduce this data deluge, as well as open up the possibilities of online training and adaptation of the smart vision algorithms at the edge. Due to the resource constraints, such as power and latency, in processing so close to the sensor, this solution will also require an assessment of the practicality of vision-based smart building systems, and a study on how to optimally partition and distribute these increasingly more powerful vision-based machine learning algorithms between local and server processing.

1.1 Engineering Background

In recent work at Stanford, [1] (Albert Haque, Serena Yeung, Arnold Milstein, Fei-Fei Li et al.) looked at vision-based smart hospitals and demonstrated a system for monitoring of the staff’s hand hygiene as a specific example. While the achieved results are very impressive, due to the data deluge from transmitting all of the raw remote sensor data, taking such a system from a proof-of-concept demonstrator to scalable real-time solutions for the real world will require more than raw improvements in hardware speed/efficiency due to the end of Moore’s law (and the less well-known Dennard’s law). Instead, it is widely agreed that major improvements in hardware will need to come from application-centric optimization in the future [2], necessitating considerable progress in hardware/software co-design.

A critical next step is to move the algorithmic compute local to the sensors, i.e. to the “edge” of the system. This will enable scaling both to large, full hospitals, as well as diverse types of assisted living and home/work spaces where more powerful remote compute is either unavailable or undesired for privacy reasons. Through local processing, we expect to achieve a data reduction of more than 100x, an energy consumption reduction of a similar magnitude, and simultaneously improve the performance of the algorithms due to the resulting low latency and improved resolution.

Laying the ground work for these improvements through advanced hardware/software co-design was the main objective of the work of the last year.
1.2 Theoretical Background

Figure 2 shows a generic vision system, highlighting the tradeoff between local and remote data analysis. Local processing is more efficient, private, and responsive, at the expense of a reduction in analysis scope. However, when local processing is judiciously combined with remote processing given a full understanding of the application goals, this penalty can be mitigated. We aim to show that a collection of “smart” cameras with moderate amounts of local inference can be networked to form a highly efficient solution for smart building applications. Similar work is currently being explored in a variety of labs across EE and CS departments, however, this type of research is still at its infancy. By specializing on a concrete application (smart building), and by leveraging prior research results (from Stanford CS and SoM), we are uniquely positioned to generate impact as this new concept of mixed local/remote processing emerges.

Figure 2: Generic vision system, highlighting the tradeoffs between local and remote data analysis. Image by Chris Rowen

In almost all vision systems, the cameras see rather uninteresting data that has no bearing on the sensed state or action trajectory for most of the time. An approach that leverages this situation
toward massive data reduction is the idea to cascade increasingly powerful algorithms on demand (see Figure 3). For example, one might design a smart camera such that it only “wakes” up powerful and data hungry algorithms for fine classification only once a face is detected by a simple, bare-bone algorithm. The Murmann group has recently demonstrated custom chips that achieve record performance on such wake-up tasks, with a power dissipation that allows the processors to operate on small batteries for several months [3].

![Figure 3: Cascaded inference approach. Image by Bert Moons.](image)

The smart hospital vision system described in [1] follows the brute-force approach where the classifiers are not cascaded and all of the raw data is sent to the cloud (see Figure 2), therefore leading to a prototype that is unsuitable for practical deployment at scale. However, this choice was merely driven by the fact that both the application requirements and algorithms were not known in this first iteration of research. In the second research iteration that was the subject of this work, much more is known about the specific algorithmic needs, and we exploit this to develop the core concepts for custom smart building hardware.

2 Research Methods

Over the course of the last year, we met with our collaborator Serena Yeung to detail the design requirements of the Smart Hospital Project. We analyzed the datasets and algorithms used in her work in [1] and studied current hardware capabilities. Then we outlined hard performance requirements for real-time deployment, and began the hardware/software co-design process for current and future algorithms and tasks.

The co-design process consists of two parts. The first part is an algorithm exploration and assessment, to identify areas of optimization and constraints. Such an exploration also informs the process of how to design and evaluate the hardware solutions. The second part is hardware exploration, in which we identify edge constraints and hardware capabilities. Using these insights, we can begin development of an edge hardware solution.

We chose to approach solving the underlying engineering problem using a co-design process, because insights from both algorithm and hardware explorations will in turn inform the developments of the other. As Yeung increases her model capabilities, this will place additional constraints on the hardware. As the hardware is improved to meet these expanded capabilities, fixed resource constraints will motivate and drive the work on algorithm optimization and model compression.
3 Algorithm Exploration

The Smart Hospital project targets tasks for "rich visual recognition". The core tasks under this group that we are interested in are:

1. Activity Recognition [1],[4]
2. Object Detection
3. Dense Human Pose Estimation [5]

At a high level, the state of the art algorithms for these tasks are comprised of two parts: a front-end deep neural network (DNN) for feature extraction of the camera images, and a second DNN on the back-end with a recurrent structure. We analyzed state of the art algorithms for these tasks([1],[4],[5],[6]), looking for areas of application-centric optimization, specifically areas of algorithm commonality. These areas of commonality are obvious targets for applying parameter sharing techniques among the core tasks. Parameter sharing techniques are often seen in multi-task learning (MTL)[7]. In MTL, the underlying idea is that by learning shared features between similar tasks, model generalization on the original task is improved. In our case, we would instead use these ideas to reduce any redundant feature extraction between DNNs. This can potentially lead to dramatic reductions in overall computation costs at the edge as we expand the core tasks for the project.

Indeed, for each of the state-of-the-art algorithms for the core tasks above, a central commonality is the backbone architecture of the feature extraction front-end. All use a form of a standard convolutional neural network, ResNet[8] as the backbone. Currently, Serena Yeung’s networks use the 18-layer ResNet, known as "ResNet18", as the backbone architecture. Other algorithms use the much larger "ResNet50" architecture as the backbone (see Figure 4). Since identifying this commonality in the feature extraction backbone architecture, one of the ongoing research steps is to explore to what extent these ResNet-based front-ends can be successfully combined into a single front-end to be shared among all core tasks of interest.

3.1 Current Model Analysis

The first model we are analyzing from Yeung is an expanded variant of the model in [1]. This variant detects the following list of activities:

1. getting in bed
2. getting out of bed
3. getting in chair
4. getting out of chair

The model uses ResNet18 for the front-end, and a second DNN with a recurrent structure, called a Long-Short-Term-Memory (LSTM), on the back-end. The front-end produces 512 features per frame. The LSTM combines these features taken from 64 image frames to predict the activity.
Figure 4: Standard 18-layer and 50-layer ResNet Architectures (ResNet18 and ResNet50 respectively).

Ongoing work on this model is to first understand performance using standard model reduction techniques, such as quantization, layer fusion, bias correction and model pruning. Due to current limitations on access to the patient training data, we are limited to simulation data, and thus are focusing our efforts on techniques that do not require fine-tuning of the reduced model.

4 Hardware Exploration

In the activity recognition tasks described in [1], hardware capabilities were as follows. 50 depth cameras captured image data that was then sent over the hospital network. The network fed this data to a back-end server (see Figure 5) that processed the images offline. The bandwidth of this network was determined to be one of the key roadblocks to realizing a scalable real-time solution.

<table>
<thead>
<tr>
<th>Layer Name</th>
<th>Output Size</th>
<th>ResNet18</th>
<th>ResNet50</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>112x112</td>
<td>7x7, 64, stride 2</td>
<td>3x3 max pool, stride 2</td>
</tr>
<tr>
<td>conv2-x</td>
<td>56x56</td>
<td>[3x3, 64] x2</td>
<td>1x1, 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 64] x2</td>
<td>3x3, 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1x1, 256] x3</td>
<td>1x1, 256</td>
</tr>
<tr>
<td>conv3-x</td>
<td>28x28</td>
<td>[3x3, 128] x2</td>
<td>1x1, 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 128] x2</td>
<td>3x3, 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1x1, 512] x4</td>
<td>1x1, 512</td>
</tr>
<tr>
<td>conv4-x</td>
<td>14x14</td>
<td>[3x3, 256] x2</td>
<td>1x1, 256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 256] x2</td>
<td>3x3, 256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1x1, 1024] x6</td>
<td>1x1, 1024</td>
</tr>
<tr>
<td>conv5-x</td>
<td>7x7</td>
<td>[3x3, 512] x2</td>
<td>1x1, 512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3x3, 512] x2</td>
<td>3x3, 512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1x1, 2048] x3</td>
<td>1x1, 2048</td>
</tr>
<tr>
<td>FLOPs</td>
<td>1.6x10^7</td>
<td>3.8x10^7</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Hardware Processing Used in [1].
For the proposed scalable real-time hardware solution, we decided to partition the back-end algorithms between local/edge processing and server processing. Instead of sending raw camera images over the hospital network, we instead send the much smaller extracted features. This has two key benefits. The first is that we will have mitigated the data deluge, and therefore increased the scalability of the hardware solution. The second is that the extracted features will have less identifying information of individuals, leading to increased privacy in the proposed solution in the event of a network breach. The partitioning of the algorithms is shown in Figure 6, using a Xilinx Device as an example of edge processing hardware. To enable real-time performance, we need to ensure an edge processing latency of at most 100 ms per image.

The amount of features we can extract will be limited at the edge by the latency and power constraints. Therefore we began an hardware exploration of the entire system from the camera front-end to back-end processor to understand the roles these constraints play in the projected capabilities of our system, with an initial goal of developing a single room demo.

![Proposed Hardware Solution](image)

Figure 6: Proposed Hardware Solution.

### 4.1 Building Constraints: Power

The first step is to understand the resource constraints on the edge device. In her earlier work, Yeung identified network bandwidth as a constraint. However as we push compute to the edge, an increasingly important constraint will be power. At Yeung’s older test sites, wall power is available. However, this is not the case at future sites. Indeed, at one of the future sites at Stanford Hospital, the devices will need to be powered via Power over Ethernet (PoE).

There are many PoE standards but the most basic guarantees at most 12.95W at the powered device (PD). While the highest power standard (802.3bt Type 4) guarantees power up to 71W at the PD, even this is still too low for most, if not all commercial desktop and server GPUs, which easily consume over 100W. This not only further motivates the need for specialized hardware designed for edge computing, but provides a specific power constraint which we can use to evaluate the potential edge hardware.

### 4.2 Edge Compute Hardware Assessment

Due to the large growth of commercial opportunities for deep learning at the edge, many companies are dedicating considerable amounts of engineering efforts to provide compute solutions in this space. Examples include Xilinx Zynq MPSoCs, Nvidia Jetson SoC, and most recently the Google Edge
TPU. For any of these edge solutions to be practical, they must be hardware-software co-designed solutions in order to both manage the heavy workloads and adopt the latest developments in the deep neural network algorithmic space. That is, solutions must fundamentally have two parts:

1. Dedicated hardware to run core operations of DNN at low power and low latency.

2. Software compilers to quickly and seamlessly convert DNN models from popular development frameworks such as Tensorflow [9] and PyTorch [10] into a form optimized for specific target edge hardware.

In light of this rapidly developing space of hardware-software co-designed solutions, we first established a baseline performance survey of off-the-shelf hardware solutions. An experiment was constructed to compare the offering of two of the largest companies in this edge hardware space: Xilinx and Nvidia. We targeted chips that had readily-available evaluation boards. Additionally, we targeted solutions with publicly available compilers that could deploy our standard version of ResNet50. Due the commonalities in the backbone architecture that we identified in the previous section, we determined that the standard ResNet50 architecture provided the most representative algorithm for evaluating the off-the-shelf hardware solutions.

While we evaluated multiple hardware solutions, the rest of this section will detail the results on the lowest power hardware solutions publicly available at the time of testing: The Ultra96 board featuring a Xilinx ZU3 chip and using the Xilinx/DeepPhi Edge AI Compiler, and the Nvidia Jetson TX2 Evaluation board, featuring the Jetson TX2 SoC and using TensorRT1.0. For the Jetson TX2, we evaluated the system in its Max-Q mode, the most power efficient setting available.

The two metrics for comparing hardware at the edge is latency and power. For latency, we classified objects from images drawn from the ImageNet dataset [11] using ResNet50 for loop counts of 1, 10, 100 and 1000 and measured runtimes. While the average latencies of the 1000-loop case for both hardware solutions matched reported benchmarks, we found a high variance for runtimes for smaller loop counts for the Jetson hardware. For example, over 100 10-loop measurements, the average latency per loop was 40.8ms, with a standard deviation of 8.5ms. In comparison, over a 100 100-loop measurements, the average latency per loop was 32.6ms, with a standard deviation of 0.77ms. The exact source of this high runtime variance is still unknown.

For power measurements, we chose to compare steady-state power consumption. To keep the hardware running in steady-state so that we could measure power, we also classified objects from images drawn from the ImageNet dataset [11] using ResNet50 on a loop of 1000 iterations. Power measurements were captured directly from the board power supply input. More specifically, each device under test was connected directly to a Keysight power supply using custom power connectors. By measuring board power directly with the Keysight power supply, via a GPIB connection, we could capture regularly sampled measurements of voltage and current over the steady-state run. We used these measurements to calculate board power consumption. An example of the power measurement setup is shown in Figure 7.

The final power and latency measurements are below:

<table>
<thead>
<tr>
<th></th>
<th>Power (W)</th>
<th>Avg Image Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra96</td>
<td>7.07</td>
<td>45</td>
</tr>
<tr>
<td>Jetson TX2</td>
<td>7.15</td>
<td>32</td>
</tr>
</tbody>
</table>

In conclusion, we found that both hardware solutions easily met the 100 ms latency requirement. Both solutions have a similar steady-state power consumption of a little over 7 W, placing them
well under the limits of the most power constrained PoE (at 12.95W). However, the more consistent single-image latency performance of the Ultra96 board made this board the more desirable solution with which to begin development of the single room experimental system. Additionally the Xilinx/DeepPhi compiler provided practically turn-key compilation of target models from standard ML frameworks for the Ultra96 board.

4.3 Camera Assessment

An important component for the demo is the camera. The original cameras (ASUS Xtion Pro Live) used in [1] have since been discontinued, which means that for the demo, we had to use a different camera. The first choice was the ASUS Xtion 2, but we soon discovered that it was unsuitable for our single room edge hardware demo.

The core problem with the ASUS Xtion 2 is its incompatibility with ARM devices. Virtually all edge devices are designed with a specialized accelerator to compute the target algorithm, and an ARM core processor on the front end to manage communications and other interfaces with the rest of the system. The underlying driver for the camera chip is only compatible with Intel processors, and the source code is not open. Because desktop processors are not an option for edge devices, the ASUS Xtion 2 is not an option at the edge.

We are currently assessing options for the depth camera. Given the problem with the closed source nature of the driver, we are targeting cameras with open source drivers. A promising option we have identified is the Microsoft Azure Kinect Depth Camera. It uses the Time-of-Flight (ToF) method for measuring depth, the same method used by the original ASUS Xtion. Additionally, it is completely open source, and so we will be able to compile the driver for ARM.

While the camera choice currently does not directly impact the power or latency constraints, it will have an effect on the final model performance. How much accuracy can be recovered with this change in camera is ongoing work.
5 Conclusion and Ongoing Research Objectives

In the past year, we have completed a basic assessment of both the basic underlying models and current hardware capabilities. In the algorithm assessment, we identified areas of further study for model compression and optimization. In the hardware assessment, we completed a base survey of off-the-shelf commercial solution and identified further constraints beyond the original bandwidth constraint identified by Yeung.

Both assessments laid the groundwork for the (currently ongoing) development of a single room demo, in which to demonstrate a real-time deployment of front-end architectures of the algorithms for the core tasks outlined above. This work is focused on combining the two sets of insights into this demo, which will be used to not just validate current ideas but also further study the limits of the system to provide large compute power locally on demand. This will provide further insight into the fundamental limits of data and power reduction for front-end DNN architectures at the edge, and how we can approach these limits with future implementations on custom silicon chips.

References


