Delivering Deep Learning to Mobile Devices via Offloading

Xukan Ran
University of California, Riverside
xukan.ran@email.ucr.edu

Haoliang Chen
University of California, Riverside
haoliang.chen@email.ucr.edu

Zhenming Liu
College of William and Mary
zliu@cs.wm.edu

Jiasi Chen
University of California, Riverside
jiasi@cs.ucr.edu

ABSTRACT

Deep learning has the potential to make Augmented Reality (AR) devices smarter, but few AR apps use such technology today because it is compute-intensive, and front-end devices cannot deliver sufficient compute power. We propose a distributed framework that ties together front-end devices with more powerful back-end "helpers" that allow deep learning to be executed locally or to be offloaded. This framework should be able to intelligently use current estimates of network conditions and back-end server loads, in conjunction with the application's requirements, to determine an optimal strategy.

This work reports our preliminary investigation in implementing such a framework, in which the front-end is assumed to be smartphones. Our specific contributions include: (1) development of an Android application that performs real-time object detection, either locally on the smartphone or remotely on a server; and (2) characterization of the tradeoffs between object detection accuracy, latency, and battery drain, based on the system parameters of video resolution, deep learning model size, and offloading decision.

CCS CONCEPTS

• **Networks** → *Network experimentation*; *Mobile networks*; • **Computing methodologies** → *Neural networks*;

KEYWORDS

Wireless, Offloading, Neural networks

ACM Reference format:
Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. 2017. Delivering Deep Learning to Mobile Devices via Offloading. In *Proceedings of VR/AR Network '17, Los Angeles, CA, USA, August 25, 2017,* 6 pages.
https://doi.org/10.1145/3097895.3097903

1 INTRODUCTION

Deep learning holds the promise to make Augmented Reality (AR) devices smarter. For example, real-time object recognition tools can improve a user's shopping experience in large malls [23, 31], facilitate rendering of animations in AR apps (*e.g.,* detect a table, and overlay a game of Minecraft on top of it), and assist visually impaired people in navigation [15]. Deep learning-based face recognition tools can also be used in authentication systems [30]. Increasing sophistication of deep learning models, such as the one used to play Go [29], have the potential to provide powerful intelligence in the near future.

However, few AR apps use deep learning today because there is not enough infrastructure support. Deep learning algorithms are computationally intensive, and the front-end devices are often ill-equipped to execute them with acceptable latencies for the end user. For example, Tensorflow's Inception deep learning model can process less than one video frame per second on a typical Android phone; that is, the video streams cannot be analyzed in real time [2]. Even with speedup from the mobile GPU [13], typical processing times are approximately 600 ms, which is equivalent to less than 1.7 frames per second and is still not acceptable for real-time processing. In industry, although there are a few applications that run deep learning locally on the phone, *e.g.,* Apple Photo Album, these are lightweight models that do not run in real-time. The solution widely used to run deep learning at the front-end is to transfer all the input data to more powerful back-ends and execute deep learning algorithms there. Existing voice-based intelligent personal assistants (*e.g.,* Alexa, Cortana, Google Assistant, and Siri) use this approach. Such cloud-based solutions are applicable when network access is reliable, and is further encouraged by the development of powerful server hardware for machine learning (*e.g.,* Google Cloud Tensor Processing Unit).

Our observation. AR apps relying on deep learning have different accuracy/latency requirements. For example, an app helping visually impaired people navigate on a street may need low latencies but may tolerate a high number of false positives, *i.e.,* false alarms are fine but missing any potential threats on the street is costly. ARs used in shopping malls for recommending products may tolerate longer latencies (fine to let users to wait a second or two) but have a higher accuracy requirement. Finally, in an authentication system that uses deep learning, users can wait even longer but they expect ultra-high accuracy.

Thus, there is a need to develop a flexible architecture that can make intelligent tradeoffs between accuracy, latency, and other infrastructural resources. We propose a distributed infrastructure that ties together computationally weak front-end devices with more powerful back-end "helpers", to allow deep learning the choice of local or remote execution. The back-end helpers could be any devices that provide more computational power such as a cloud

VR/AR Network '17, August 25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5055-6/17/08…$15.00
https://doi.org/10.1145/3097895.3097903

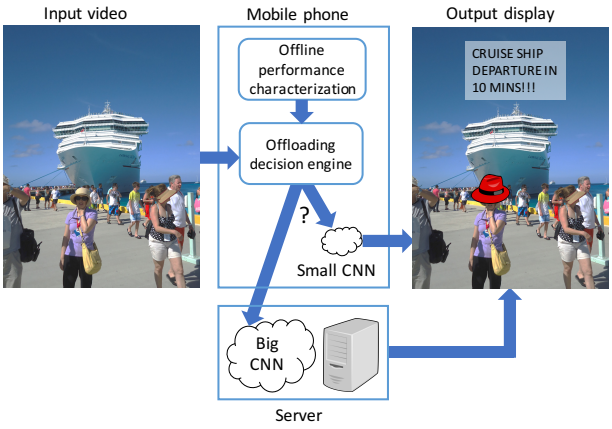


Figure 1: System overview. The smartphone chooses where to process the input video for real-time display.

server, the user's home router, or even the user's laptop carried in her backpack. For example, wearable devices such as smartwatches or head-mounted displays with camera capabilities and video processing requirements [8] could offload to a nearby device such as the user's smartphone.

Unlike the architecture for Alexa/Cortana/Google Assistant/Siri that always offloads machine learning tasks to the back-end, our final goal is to develop a framework that intelligently uses current estimates of network conditions and back-end server loads, in conjunction with the application's requirements, to determine an optimal offload strategy. For example, a Minecraft game relying on deep learning to render games could reduce the resolution of the video feed uploaded to the back-end server when the Internet performance is not good, and if the connection further degrades, would choose not to offload and switch to a smaller, although less accurate, neural net model to run locally.

Our contribution. This work reports our preliminary investigation in implementing the aforementioned architecture, in which the front-end is assumed to be smartphones. We focus on an executing a convolutional neural network (CNN) optimized for detecting objects in real-time for augmented reality applications. We specially aim to understand how the changes of key resources (*e.g.*, network bandwidth, neural network model size, image resolutions, battery level) in the system impact the offloading decision. An example of our system is illustrated in Fig. 1. We make the following contributions:

- Development of an Android application that performs real-time object detection, either locally on the smartphone or remotely on a server. The object's location in the scene is returned and displayed to the user.
- Characterization of the tradeoffs between object detection accuracy, latency, and battery drain, based on the system parameters of image resolution, CNN model size, and offloading decision. We perform these measurements in both controlled network environments and in public locations in-the-wild.

In the remainder of this work, we discuss the relevant background on CNNs (§2), our setup (§3), and our measurements (§4). We conclude with related work (§5) and future directions (§6).

2 DEEP LEARNING FOR OBJECT DETECTION

This section provides relevant background on CNNs for object detection. Our objective is not to develop new object detection algorithms, but to understand the performance of state-of-the-art deep learning algorithms on smartphones and the potential benefits from offloading.

A standard CNN works as follows (see also [9]): A frame of the video is fed to the CNN as input. Filters (known as convolutions) are applied to the RGB layers of the frame, and then averaging and thresholding functions (known as pooling) are applied. These stages of convolutions and pooling are applied repeatedly, with the order and specific functions depending on the particular CNN model. Finally, the features output by the CNN are probabilistically classified to one of the trained categories.

The object detection task places an additional burden on CNNs to locate the object in the image, in addition to object classification. Many existing neural nets for object recognition builds a pipeline solution, *i.e.,* they use one neural net to detect the boundaries of objects and a second net to inspect contents within each bounding box. In this work, we evaluate a particular object detection CNN called Yolo [27]. (In our future work, we intend to evaluate other popular neural nets like faster-RCNN and MultiBox [18, 28].) Yolo is optimized for processing video streams in real-time and possesses the following two salient features: 1. *One neural net for boundary detection and object recognition.* Observing that using multiple neural nets unnecessarily consumes more resources, Yolo trains one single neural network that predicts boundaries and recognizes objects simultaneously. 2. *Scaling with resolution.* Yolo handles images with different resolutions, *e.g.,* when there is a change in the dimension of an input to a convolutional layer, Yolo does not change the kernels and their associated learnable parameters – this results in a change of output dimensions. Thus, the computation time of a neural network scales directly with input's resolution, *e.g.,* lower resolution images require less computation.

3 METRICS AND DEGREES OF FREEDOM

This section first discusses the key metrics of front-end apps, and second, the other factors that affect the system design. Thirdly, we discuss the possible degrees-of-freedom in our system.

Key metrics. AR apps often require service guarantee on two important metrics: (see §1 for more discussion):

1. Latency/frame rate: Latency is the total time needed to get an output on one frame of the video stream. When the deep learning is executed locally, this is the time used for executing deep learning. When the deep learning is executed at a backend server, this time is the total of communication and execution time at the backend.

2. Accuracy: The mean average precision (mAP) is a measurement of a classifier's effectiveness. Specifically, the average precision (AP) for standard image datasets [5] is defined for each class as the fraction of correct classifications above a given rank, averaged across ranks. The mAP is the mean AP across classes.

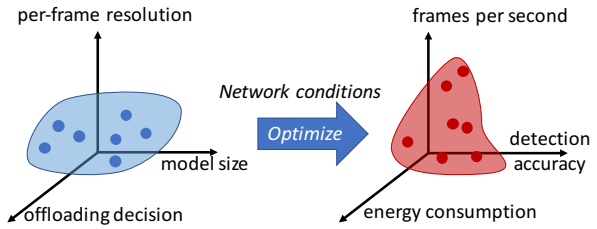


Figure 2: Input parameters and outcomes.

Being a responsible citizen. While purely focusing on these key metrics may maximize the performance of the deep learning module, other potential impacts on the front-end device should be considered. For example, running more powerful deep learning models may consume more CPU cycles, disrupting other background processes, and also draining the device battery. If the client communicates with the server, the network transmission also uses battery; moreover, if the data transfer is over LTE, the cost to the user in terms of data quota must also be considered. These factors of battery consumption and network data usage must be considered holistically alongside the performance metrics.

Degrees of freedom. There are several degrees of freedom we may consider in our system.

1. Adjust the frame resolution. By adjusting the resolution of each frame, the execution time for a deep learning model may change. For example, increasing the frame resolution increases the processing time, but may also improve detection accuracy.

2. Offload to backend. By offloading the problem to a backend server, we can substantially reduce computation burden at the front-end devices, possibly at the expense of transmission energy and monetary cost.

3. Use smaller deep learning models. We may also use a smaller neural network to reduce the run time, at the cost of reducing the prediction accuracies.

Each of these operations may impact one or more of the metrics described above. Furthermore, we may employ multiple operations simultaneously, *e.g.*, we can reduce the resolution and use smaller models at the same time. In fact, any subset of these operations defines a legitimate strategy, although not necessarily optimal.

4 EXPERIMENTS

In this section, we describe our initial efforts towards measuring and understanding the performance tradeoffs of running deep learning algorithms on smartphones. We measure these tradeoffs under both controlled conditions in the lab, as well as in-the-wild at various public locations such as coffee shops and personal homes.

4.1 Testbed Setup

We set up a server at UCR which runs Yolo using GPU speedup. The server is equipped with a quad-core Intel processor at 2.7 GHz with 8 GB of RAM, and with an NVIDIA GeoForce GTX970 graphics card with 4GB of RAM. We use powerful phones (OnePlus 3T and Google Pixel) to provide an upper bound on performance (in the future, we intend to study performance on lower-end smartphones).

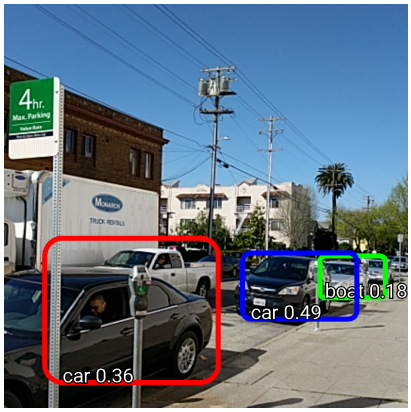


Figure 3: Screenshot from our Android app which can run in "local" or "offload" mode and logs latency, battery, and data usage. The bounding boxes indicate detected objects and the numbers are the class probabilities. In this example, "boat" is a false positive.

We experiment with two neural network models: a smaller model with 9 convolutional layers that can detect 20 object classes, and a larger model with 22 convolutional layers and can detect 80 object classes. The Android implementation is based on Tensorflow [1] and runs the smaller model, while the server implementation is based on a custom framework called Darknet [25] and runs the larger model. We extend the Android app to enable offloading of the camera feed to the server. To avoid having the server process stale frames, the app sends the current frame to the server and waits to receive the result before sending the next frame for processing. The client also logs battery drain reported by the Android OS, the cellular data usage, and the time elapsed between sending the frame and receiving the detection result from the server. To reduce the communication time and bandwidth usage between the server and the client, the server only transmits the location of the bounding box, instead of sending the full video frame. On the server, we modify Yolo to load the neural network configuration only once, listen for frames sent by the Android app, and process each frame in sequence.

4.2 Deep learning on the phone

Tradeoff between resolution and frame rate: We first examine the case where the phone runs the CNN locally, without offloading, to see the tradeoff between frame resolution and latency/frame rate. Firstly, we vary the image resolution from 160x160 to 544x544 pixels. Recall that Yolo is dynamically able to adapt to different image resolutions by adjusting the neural network model size. In Fig. 4, we plot the tradeoff between frame resolution and frame rate. When the CNN is run on the phone, the highest achievable frame rate is about 5 FPS for a low resolution, while the lowest frame rate is about 1 FPS for a high resolution. The phone must decide between analyzing a high resolution with high latency, and a low resolution with low latency.

We next compare against the scenario where the CNN is run on the server with a 10 Mbps link to the phone. In this case, the frame

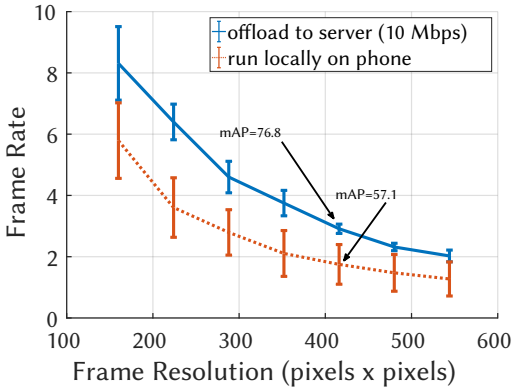


Figure 4: Impact of frame resolution. The phone has lower accuracy than the server, and trades off resolution for frame rate.

rates are significantly higher; and moreover, the accuracy [27] of the larger model on the server is better than the small model on the phone. The phone's decreased computational power and smaller model result in performance degradation in terms of both speed and accuracy; therefore, offloading may be helpful, even given the additional time spent to transmit the data.

Battery: We measure the battery usage of running the CNN locally on the phone for 30 minutes. In Fig. 5d, we show the results for the OnePlus and Google Pixel phones. With the phones using 20-25% of their battery for 30 minutes of usage, running the CNN for 2-3 hours would result in complete depletion of the user's battery. For comparison, this is significantly more expensive than video streaming, which uses about 5% of the battery for 30 minutes of usage [20, 21].

4.3 Impact of variable network conditions

Next, we study how the (communication) network conditions impact the performance of our system. For these experiments, the phone is located in the same subnet as the server, and chooses to offload. We measure the time it takes for each frame to be processed, as well as the frame rate, while modifying the bandwidth and latency between the client and the server using the `tc` traffic control tool. The latency with unconstrained bandwidth between the client and the server is about 30 ms. These experiments use the OnePlus phone, and each trial is repeated 10 times, with a frame resolution of 416x416 pixels, and a per-frame size of about 200 kB.

Latency breakdown: We first measure the time taken for each frame to be processed, which is the sum of the network transmission time plus the time taken by the server CNN to process to process the frame. We plot the results in Fig. 5a. The majority of the processing time is taken by the network transmission time, which the server generally running the CNN in < 30 ms. This indicates that network transmission is the key driver of latency and frame rate, rather than the CNN processing time, and the offloading decision will highly depend on the network conditions. Therefore, we dig deeper into the impact of network conditions.

Offloading based on network bandwidth: In Fig. 5b, we vary the network bandwidth and measure the frame rate. We see that frame rate increases roughly linearly with network bandwidth, which makes sense because frame rate is roughly proportional to data transferred. We also show on the same plot the average frame rate from running the CNN locally on the phone. This is a horizontal line, because local processing speed does not depend on network bandwidth. In this particular scenario, a simple offloading policy may work to maximize frame rate: below a threshold of about 4 Mbps, run the CNN on the phone, and above the threshold, offload the CNN computation to the server.

Offloading based on network latency: In Fig. 5c, we fix the network bandwidth at 10 Mbps and add additional network latency using the `tc` tool. We expect an inversely proportional relationship, since frame rate is calculated as data transferred per time, and this is what we see. Again, a simple offloading policy may work here: when the network latency is below 100 ms, offload to the server, and when the network latency is above 100 ms, run the computation locally. Note that the processing time is dependent on the specific phone model and server processing power/GPU, so some calibration phase will be necessary in the final system to determine these bandwidth and latency threshold values. Moreover, the bandwidth-based and latency-based offloading policies must be combined to determine an offloading policy for any tuple of (network bandwidth, network latency). We intend to investigate these offloading policies in future work.

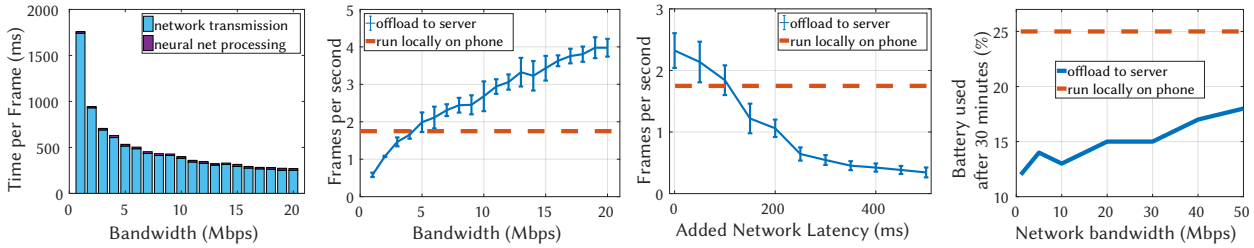
Tradeoff between battery and frame rate: Finally, we also examine the impact of offloading on battery usage. In Fig. 5d, we plot the battery usage as the network bandwidth increases. We run each trial for 30 minutes and plot the drop in battery reported by the Android OS. When there is more bandwidth, the phone tends to transfer more data, and use up the battery of the phone for wireless transmission. Combined with the bandwidth-based offloading policy discussed, we can see a tradeoff between battery and frame rate: while the phone would prefer to offload to the server in order to save battery, for network bandwidths below 4 Mbps, the optimal offloading policy that maximizes frame rate is to run the CNN locally, at the expense of high battery usage. Therefore, the decision to offload cannot solely consider maximizing frame rate, but most also consider the holistic effect of battery drain.

Finally, we note that across all of our experiments, even for high bandwidth, the frame rate caps off at about 5 FPS. This suggests that additional optimizations, such as batching the frame transmissions and object tracking, will be needed to reach higher frame rates suitable for real-time applications, which we intend to investigate.

4.4 Performance in-the-wild

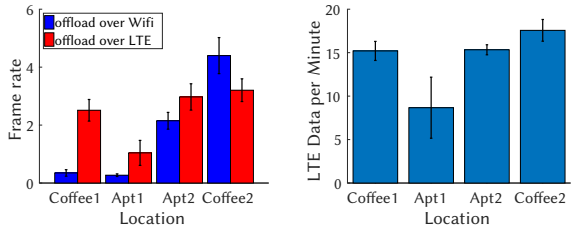
In this section, we measure the offloading performance in-the-wild at various locations, including a home, coffee shop, and university campus. These locations are realizations of the controlled network conditions discussed in Section 4.3. Our eventual goal is to compare this offloading performance with the on-phone performance discussed in Section 4.2, in order to make the best offloading decision.

To evaluate edge computing versus a more distributed scenario, we consider several test locations: when the client is in the same subnet as the server, a different subnet but in the same city, and finally



(a) Breakdown of latency **(b) Bandwidth** **(c) Network latency** **(d) Battery**

Figure 5: Performance with offloading. Network conditions are the bottleneck to increasing frame rate and informs the offloading decision. However, increased frame rate comes at the expense of increased battery usage.



(a) Latency **(b) LTE Data usage**

Figure 6: Performance in-the-wild. Using LTE to increase the frame rate comes at the expense of data quota usage.

when the client is in a different city than the server. Specifically, our test locations are:

- Coffee 1 (different subnet, different city): A coffee shop in Berkeley, CA. The camera is pointed towards a window and detects cars and people on the street.
- Apt 1 (different subnet, different city): An apartment in Berkeley, CA with cable Internet. The scene is a fairly static home environment and mainly detects computer monitors, cups, and potted plants.
- Apt 2 (different subnet, same city): An apartment in Riverside, CA. The main objects detected are chairs, refrigerators, and TV monitors.
- Coffee 2 (same subnet, same city): An on-campus coffee shop in Riverside, CA. It mainly detects chairs, tables, and umbrellas.

In Fig. 6a, we plot the frame rate of the client in these locations, when the client offloads using WiFi or LTE. Each trial lasts 60 seconds, and repeated 3 times. In general, the frame rate with WiFi in a city far from the server (Coffee1 and Apt1) show quite a low frame rate, with each frame taking more than 500 ms to process. When the client is located in the same city as the server (Coffee 2 and Apt 2), the frame rate over WiFi seems to be slightly better, particularly if the client is located in the same subnet as the server. Qualitatively, we observe that in high-latency environments such as the coffee shop, the detection boxes become inaccurate and are drawn in incorrect locations on the screen. The reason is because if the camera or the object is moving and the network is slow, the result returned from the server is stale. In general, the performance (both in terms of quantitative frame rate and qualitative observations) over LTE seems to be better; however, there is a tradeoff here. Although LTE may provide a higher frame rate, the typical constraint is data usage, which is limited and costs money. To delve further into the monetary costs to the user, Fig. 6b shows the LTE data usage in the same scenarios. We can see that the average data usage is about 15 MB/minute, which is fairly high. Assuming a 2GB costs $35, this mean that each minute of usage costs $0.25. Another possible tradeoff is with accuracy: if LTE has higher bandwidth, the user can upload higher resolution video frames for higher accuracy, at the expense of paying more for more data transfers.

5 RELATED WORK

Object detection: Traditional object detection frameworks such as SIFT [19] and Viola-Jones [32] have been proposed for general and domain-specific detection. Recently, applying CNNs to object classification has shown good accuracy[17]. [7] first proposed combining region proposals and CNNs for object detection, while [6, 28] improved accuracy and running time. [26, 27] also used CNNs to perform object detection with an emphasis on real-time performance. [12] compares the speed and accuracy tradeoffs of various CNN models. However, none of these works have considered the performance of CNNs on mobile phones.

Mobile offloading: [4] developed a general framework for task offloading, while [24] specifically study interactive visual applications. These frameworks cannot directly be applied to our scenario because our set of tasks may be reduced if executed locally as opposed to remotely. [2, 10, 16] explore offloading for real-time video analysis. [2] considered modifying the data (offloading a subset of frames) to reduce latency, which is complementary to our approach of compressing the model to reduce latency. [16] developed a system based on SIFT, whereas we focus on more recent CNNs. [10] offloads processing from Google Glass to nearby cloudlets, whereas we also consider running the code locally. The closest to our work is perhaps [11], which decides whether to offload CNNs to the cloud; however, they do not explicitly consider the current network conditions, which can greatly impact the offloading decision. Finally, [3] proposed traffic steering towards nearby mobile clouds for AR applications, whereas we also modify the AR application itself to improve performance.

Deep learning on smartphones Recently, several works have studied model compression of CNNs running on mobile phones. Specifically, [14] uses the GPU to speed up latency, while [22] considers hardware-based approaches for deciding important frames. We believe our approach is complementary to these works in that we can leverage these speedups to local processing, while also considering the option to offload to the cloud.

6 CONCLUSIONS AND FUTURE WORK

In this work, we evaluated an architecture of running deep learning algorithms on computationally-limited front-end devices such as smartphones. The hope is that such architectures will eventually enable applications such as real-time augmented reality. We found that there are various tradeoffs between frame rate, accuracy, battery usage, and data usage, depending on system variables such as model size, offloading decision, and video resolution. Overall, the results suggest that offloading can at least improve the frame rate and accuracy, depending on the network conditions. However, while we evaluated the impact of network conditions in isolation, a combination of factors will result in complex interactions that need to be further investigated in order to fully develop an intelligent offloading strategy.

Other future work includes evaluation of offloading decisions under a greater variety of network conditions, development of a more advanced offloading framework incorporating partial or parallel processing on the phone or server, and using tracking algorithms such as optical flow to improve processing times.

REFERENCES

[1] The TensorFlow Authors. 2017. TensorFlow Android Camera Demo. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android. (2017).

[2] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. *ACM SenSys* (2015).

[3] Junguk Cho, Karthikeyan Sundaresan, Rajesh Mahindra, Jacobus Van der Merwe, and Sampath Rangarajan. 2016. ACACIA: Context-aware Edge Computing for Continuous Interactive Applications over Mobile Networks. In *ACM CoNEXT*.

[4] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. *ACM MobiSys* (2010).

[5] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. *International journal of computer vision* 88, 2 (2010), 303–338.

[6] Ross Girshick. 2015. Fast r-cnn. *IEEE ICCV* (2015).

[7] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. *IEEE CVPR* (2014).

[8] GLIDE. 2017. The Camera Band for Apple Watch. http://getcmra.com/. (2017).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. Book in preparation for MIT Press.

[10] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. *ACM MobiSys* (2014).

[11] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *ACM Mobisys*.

[12] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. 2017. Speed/accuracy trade-offs for modern convolutional object detectors. *IEEE CVPR* (2017).

[13] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. 2016. DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices. In *ACM WearSys*.

[14] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. *ACM MobiSys* (2017).

[15] Michael Irving. 2016. Horus wearable helps the blind navigate, remember faces and read books. http://newatlas.com/horus-wearable-blind-assistant/46173/. (2016).

[16] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2016. Low Bandwidth Offload for Mobile AR. *ACM CoNEXT* (2016).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *NIPS* (2012).

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. *ECCV* (2016). http://arxiv.org/abs/1512.02325

[19] David G Lowe. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision* 60, 2 (2004), 91–110.

[20] John McCann. 2017. Google Pixel Review. http://www.techradar.com/reviews/google-pixel-review/4. (2017).

[21] John McCann. 2017. OnePlus 3T Review. http://www.techradar.com/reviews/oneplus-3t-review/3. (2017).

[22] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. 2017. Glimpse: A Programmable Early-Discard Camera Architecture for Continuous Mobile Vision. *ACM MobiSys* (2017).

[23] Thomas Olsson, Else Lagerstam, Tuula Kärkkäinen, and Kaisa Väänänen-Vainio-Mattila. 2013. Expected User Experience of Mobile Augmented Reality Services: A User Study in the Context of Shopping Centres. *Personal Ubiquitous Comput.* 17, 2 (Feb. 2013), 287–304.

[24] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. 2011. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *ACM MobiSys*.

[25] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/. (2013–2016).

[26] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. *IEEE CVPR* (2016).

[27] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016). http://arxiv.org/abs/1612.08242

[28] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *NIPS* (2015).

[29] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489.

[30] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *IEEE CVPR*.

[31] Gabriel Takacs, Vijay Chandrasekhar, Natasha Gelfand, Yingen Xiong, Wei-Chao Chen, Thanos Bismpigiannis, Radek Grzeszczuk, Kari Pulli, and Bernd Girod. 2008. Outdoors Augmented Reality on Mobile Phone Using Loxel-based Visual Feature Organization. In *ACM International Conference on Multimedia Information Retrieval*.

[32] Paul Viola and Michael Jones. 2001. Rapid object detection using a boosted cascade of simple features. *IEEE CVPR* (2001).