Enabling Cooperative Inference of Deep Learning on Wearables and Smartphones

Mengwei Xu, Feng Qian, and Saumay Pushp

Abstract

Deep Learning (DL) algorithm is the state-of-the-art algorithm of many computer science fields and applied on many intelligent mobile applications. In this paper, we propose a system called CoINF, a practical, adaptive, and flexible deep learning framework that enables cooperative inference between wearable devices (e.g., smartwatches and smart glasses) and handhelds. Our framework accelerates the processing and saves the energy consumption of generic deep learning models inference on wearables via judiciously offloading the workloads to paired handhelds at fine granularity in considering of the system environment, the application requirements, and user preference. Deployed as a user-space library, CoINF offers developer-friendly APIs that are as simple as those in traditional DL libraries such as TensorFlow, with all complicated offloading details hidden. We have implemented a prototype of CoINF on Android OS, and used real deep learning models to evaluate its performance on commercial off-the-shelf smartphone and smartwatches. The experimental results show that our framework can achieve substantial execution speedup and energy saving compared to wearable-only and handheld-only strategies.

I. INTRODUCTION

In recent years, both the academia [58], [62], [51], [35], [23], [63], [52], [41], [74], [56], [48], [71] and industry [6], [5], [8], [12] have made tremendous efforts on bringing Deep Learning (DL) to mobile devices such as smartphones. Indeed, the fusion of DL and mobile computing makes today’s mobile gadgets unprecedentedly intelligent: DL has enabled numerous exciting applications such as intelligent personal assistant, handwriting recognition, real-time language translation, and smart activity tracking, on COTS (commercial off-the-shelf) smartphones, smartwatches, and so on.

In this paper, we explore how to effectively and efficiently apply DL on wearable devices. Our study is motivated by three key observations. First, wearable devices are becoming increasingly popular. For example, a recent CNET article [1] estimated that smartwatch sales will jump from 30 million in 2015 to 67 million in 2017. By contrast, Switzerland exported only 28 million timepieces in 2015, and is experiencing a steep decline in watch sales [3], [2]. Second, DL on wearable devices enables new usage scenarios. Due to their on-body and ubiquitous nature, wearables can collect a wide spectrum of data such as body gesture, heartbeat reading, fitness tracking, eye tracking, and vision (through a smart glass). Such unique groups of sensor data create countless opportunities for DL. Third, despite a plethora of work on general mobile DL, there exist much fewer studies focusing specifically on the interplay between DL and the wearable ecosystem.

Adding DL support to wearable devices is quite challenging. Due to their small form factors and heat dissipation restrictions, most wearables have quite weak computation power compared to a typical smartphone. Wearables also have very limited energy constraints due to their small battery capacity. On the other hand, DL usually requires heavy computation: regardless of their types (Deep/Convolutional/Recurrent Neural Networks), a DL model could consist of hundreds of connected layers, each of which can have a collection of processing elements (i.e., neurons) executing a non-trivial function. Therefore, modern DL requires considerable computation and energy resources in particular during streamed data processing (e.g., continuous speech translation). Intuitively, running DL tasks only locally thus is not a good option for most wearables. Then a natural idea, which is used by many computation-intensive applications on smartphones, is offloading [32], [56]. At a high level, we can also adopt the offloading approach. However, instead of offloading computation to the remote cloud, we instantiate the Edge Computing concept [39] by offloading DL tasks to a nearby mobile device (e.g., a smartphone or a tablet) that has local connectivity with the wearable. This is indeed feasible because (1) as to be demonstrated in our study, today’s handheld devices such as smartphones are powerful enough (with multi-core CPU, fast GPU, and GBs of memory) to run DL tasks; (2) the vast majority of wearables (e.g., smartwatches and smart glasses) are by default paired with a handheld device and using it as a “gateway” to access the external world. For example, a crowd-sourced user study [35] shows that a smartwatch is paired with a smartphone during 84% of the day time. (3) users routinely carry both wearables and the handheld devices, making the offloading opportunity ubiquitously feasible.

Compared to those traditional remote-cloud-based offloading, our local (edge) offloading approach offers two key advantages. First, offloading to a handheld does not require the Internet connectivity that is often expensive in terms of energy and monetary cost (e.g., when cellular networks are used). Instead, the communication between the wearable and the handheld can be realized by cheap short-range radio such as Bluetooth or Bluetooth Low Energy (BLE). Second, offloading to a handheld also minimizes risks of privacy breach because the potentially sensitive data (e.g., medical data) generated from wearables is never leaked to the Internet.

Motivated by the above observations, we design, implement, and evaluate a system called CoINF, a holistic DL framework providing offloading support for wearable-side DL applications. Tailored to the wearable ecosystem as well as the phone-wearable interaction, it has several salient features as described below.
Making an appropriate offloading decision involves scrutinizing a wide range of factors including the DL model structure, the application’s latency requirement, and the connectivity condition. Also, our offloading target (the handheld) introduces additional complexities: despite being more powerful than a wearable, a handheld still has limited battery and processing power; as a personal computing device, a handheld also runs other apps that consume system resources by incurring bursting workload. Therefore, CoINF further takes into account the status of the handheld. We incorporate the preceding considerations into a lightweight online scheduling algorithm that judiciously determines which, when, and how to offload.

Instead of making a binary decision of offloading the whole DL model vs. executing the model locally, CoINF supports partial and selective offloading. More specifically, CoINF splits a model into two sub-models that are separately executed first on the wearable and then on the handheld. We found that in some scenarios, partial offloading even outperforms the binary decision, because an internal layer inside the model may yield a small intermediate output compared to the original input size, thus reducing the data-transmission delay. The optimal splitting point is strategically determined by the scheduling algorithm based on a candidate set of split points.

Prior remote-cloud-based DL offloading schemes \[48\] require the DL model to satisfy some certain properties (e.g., a linear structure). In contrast, CoINF can work with any DL models (DNN, CNN, RNN) with any structure, i.e., its topology can be a general directed graph without any preliminary assumptions. A challenge here is that enumerating all split-point candidates (for partial offloading) on a general graph takes exponential execution time w.r.t. the graph size. We thus propose an efficient heuristic-based algorithm that performs effective pruning during the split point search.

CoINF introduces additional optimization for streaming input that consists of a stream of data such as video frames and audio snippets continuously fed into the same model. For such streaming input, CoINF exploits pipelining to improve the overall throughput.

Finally, we integrate the above design into a holistic framework. Deployed as a user-space library, CoINF offers some developer-friendly APIs that are as simple as those in traditional DL libraries such as TensorFlow. The offloading details are transparently handled by CoINF. In addition, end users can conveniently specify offloading policies (e.g., giving a higher preference to offloading when the wearable is running out of battery).

We have implemented the CoINF prototype on Android OS (for handheld) and Android Wear OS (for wearable). We evaluate our prototype on COTS smartphones and smartwatches using real DL models. We highlight some key results below.

- **CoINF** is able to identify the best partition choice under various combinations of devices and contexts (96.9%).
- **CoINF** can gain up to 15.9X and 23.0X execution speedup, as well as 81.3% and 85.5% energy saving compared to wearable-only and handheld-only strategies, respectively.
- **CoINF** can adapt its offloading strategies to diverse environment factors, and ensures its performance always better than (or equal to) both wearable-only and handheld-only strategies.

**Contributions.** To summarize, this paper makes the following contributions.

- We conduct in-depth measurements to understand the DL execution performance on wearables, as well as various key tradeoffs that an offloading decision may incur (Section II-B).
- We propose CoINF, a practical, adaptive, and flexible DL framework designed specifically for wearables and their paired handheld devices. It examines the contexts including system environment, the application requirements, and user preference to judiciously make offloading decisions at a fine granularity. To the best of our knowledge, we are not aware of any existing system that can offer similar functionalities for wearable devices. (Section III).
- We prototype CoINF and thoroughly evaluate it using various real DL models on COTS wearable devices along with their paired handheld devices (Section IV and V). We plan to release our framework as well as related tools in near future.

**Paper Organization.** The remainder of this paper is organized as follows. We present some background knowledge about DL and motivational experiments in Section II. We then propose the design principle of CoINF in Section III and describe the implementation details of the CoINF prototype in Section IV. We comprehensively evaluate CoINF and demonstrate the effectiveness and efficiency in Section V. We discuss some limitations and possible future directions in Section VI. We survey the related work in Section VII and end up the paper in Section VIII.

## II. Background And Motivations

Deep learning (DL) is now having transformative impacts on how mobile sensor data are utilized and interpreted. In recent years, Many new mobile applications have been developed with the capability of deep learning \[76\] \[62\] \[53\] \[24\] \[59\]. Also many DL frameworks and specialized hardware devices for speeding up DL have been proposed \[58\] \[41\] \[74\] \[56\] \[25\]. Some of DL’s representative use cases are shown in Table I. However, running such DL tasks on wearables is quite challenging due to wearables’ unique characteristics such as weak computation power, tight energy constraints, and short-range pairing with smartphones.

### A. DL Models

Deep Learning (DL) models are essentially a directed graph where each node represents a processing unit that applies certain operations to its input and generates output. To utilize a DL model, developers need to first design and construct a specific
model graph, and then use data to train it (training stage). Once trained, this model can be applied in applications to run its tasks (inference stage). Our work focuses on optimizing the inference stage of DL processing, since DL models are mostly trained on cloud once and for ever. We then present some basic knowledge about DL model structures as following.

**DNN** models are the most basic deep learning models. As shown in Figure 1(a), a typical DNN model such as SENNA is composed of a series of fully-connected layers with each layer comprised by a collection of units (or neurons). By inserting proper activation layers between those fully-connected layers (e.g., \( \tanh \) function for SENNA), DNN models can quickly become more predictive and powerful for certain application domains. It’s worth mentioning that though neuron is the basic unit when describing DL algorithm, in practice a programmable unit is often layers.

**CNN** models are categorized for their extensive adoption of convolution operations. As shown in Figure 1(b), different from fully-connected layers that connect each units together, convolution layers only extract simple features from input and capture local data properties. This is achieved by applying various convolutional filters (with small kernel width) on the input feature map. CNN models are most popular for computer vision tasks, e.g., image classification (Inception, MobileNet), digital recognition (MNIST), and audio processing such as speech recognition (WaveNet).

**RNN** is another important type of deep learning algorithm, which can make use of sequential information input. The basic unit of RNN models, as shown in Figure 1(c), is called cell. Each cell is usually a combination of layers, and a number of cells can form a directed cycle so that RNN is capable of process sequential input. RNN models are widely used in NLP tasks, such as language modeling (LSTM) and document classification (TextCNN). It is worth mentioning that deep learning models are often generalized and can be used in many different tasks with very few customization efforts. For example, the LSTM model used for language modeling can also be applied to machine translation, question answering, and handwriting generating. Importantly, our proposed framework CoINF does not assume any specific DL model structure, and can work with all three types of DL models (DNN, CNN, and RNN).

### B. Offloading DL Tasks from Wearables to Handhelds

We envision that deep learning will become an essential part in the wearable ecosystem due to wearables’ unique sensing capabilities. However, running deep learning tasks on wearable devices faces two challenges. First, deep learning algorithms are computation-intensive. Second, wearables often suffer from limited processing capabilities. Therefore, a straightforward idea would be offloading the workload from a wearable to its paired handheld. The reason we choose handheld over cloud as the offloading target is multifolded. First, today’s personal smartphones and tablets are computationally powerful enough (e.g., with multi-core CPU, fast GPU, and GBs of memory) to run many DL tasks. Second, offloading to handheld does not require Internet connectivity, which often has high energy and monetary cost (e.g., when cellular is used). Third, very importantly, offloading computation locally to users’ personal mobile devices minimizes risks of privacy breach because the potentially sensitive data on wearables is never leaked to the Internet. Note that there are several prior work targeting at wearable offloading for better performance (see Section VII). However, none of them studies deep learning tasks, thus leaving some important questions unanswered.

**Q1:** Can offloading to a paired handheld device really help speed up DL tasks for a wearable?  

\( ^1 \)The term DNN can refer to any neural network with several hidden layers, with CNN and RNN as special cases. In this work, we use it to represent the basic deep models without convolution or recurrent structures.

### Table I: 8 popular deep learning models used in this work, as well as their usage in applications and input data format.

| Model      | App                | Input         | Model       | App                | Input         |
|------------|--------------------|---------------|-------------|--------------------|---------------|
| MNIST [10] | digit recognition  | image         | LSTM [75]   | language modeling  | word vectors  |
| Inception [66] | image classification | image         | TextCNN [19] | document classification | word vectors |
| MobileNet [42] | image classification | image         | TextRNN [20] | document classification | word vectors |
| WaveNet [67] | speech recognition | mfcc features | SENNA [30]  | word chunking      | word vectors  |

Fig. 1: Typical structures for Deep Neural Network (DNN), Convolution Neural Network (CNN), and Recurrent Neural Network (RNN) models.
TABLE II: Hardware specifications for wearables and smartphones used in this work.

| Device          | CPU                        | Memory                | GPU            | System        |
|-----------------|----------------------------|-----------------------|----------------|---------------|
| Nexus 6         | Quad-core 2.7 GHz Krait 450| 3 GB RAM              | Adreno 420     | Android 7.0   |
| LG Watch Urbane | Quad-core 1.2 GHz Cortex-A7| 512MB RAM             | Adreno 305     | Android Wear OS|
| Galaxy S2       | Dual-core 1.2 GHz Cortex-A9 | 1GB RAM               | Mali-400MP4    | Android 4.4   |

Fig. 2: End-to-end latency breakdown under different offloading scenarios (wearable CPU-only, handheld CPU-only, and handheld GPU-only). The upper percentage indicates the proportion of computation time among the overall latency. Offloading to the handheld is often slower than wearable execution due to the high data transfer delay via Bluetooth.

Q2: If yes to Q1, how much benefit can offloading to a handheld bring in consideration of execution latency reduction and energy consumption reduction?

To answer these two questions, we carry out a set of experiments on 8 popular DL models and various hardware setups. Our experiment results show that whether and how much users can benefit from offloading depends on a multiple factors. In particular, we will reveal that in some cases partitioning the DL models into two parts and run them separately on the wearable and the handheld would be a better option. We call such a scheme “partial offloading”.

Experimental setup. We use a Nexus 6 smartphone running Android 7.0 as the handheld device, and an LG Watch Urbane as the wearable device. Also, we use an old device, Galaxy S2 released in 2011, to emulate the hardware of a head-mount device such as Vuzix M1000 [21] who shares the same CPU (Cortex-A9), RAM size (1GB), and even the operating system (Android ICS 4.X) with those of Galaxy S2. Table II elaborates the hardware specifications of these three devices used in this study.

We use TensorFlow [17], one of the most popular machine learning frameworks, to run deep learning tasks on both wearables and handhelds. To allow TensorFlow to use mobile GPU for performance acceleration, we use the open-source library RSTensorFlow [23] that extends TensorFlow onto mobile GPU. We use Bluetooth for the data transfer between wearable and handheld due to Bluetooth’s wide availability on wearable and its energy efficiency. Also Google has recommended it as the proper way of performing data communication on wearable devices [22]. To ensure reproducibility, all experiments are carried out by fixing the distance between the wearable and handheld (0.5m) unless otherwise specified.

Which factors determine offloading decisions? Figure 2 and Figure 3 show the latency breakdown of four popular deep learning models under different offloading scenarios. In each plot, the two left columns present the latency of executing the whole model on different wearable devices (LG Urbane and S2), while the two right columns show the latency of offloading them to handheld processors (CPU and GPU respectively). For the two offloading scenarios, we further breakdown the overall latency into the time spent on network communication and handheld-side computation. Our key observation is that although offloading to handheld CPU and GPU can dramatically reduce the computation time, e.g., more than 10 times for Inception model.

Fig. 3: End-to-end latency breakdown under different handheld CPU governor (interactive or powersave). The upper percentage indicates the proportion of computation time among the overall latency. The device status such as current CPU governor can have key impacts on making choice about offloading.
model, the end-to-end latency is often not reduced due to the high data transfer latency over Bluetooth. The results show that making a judicious offloading decision can have significant impacts on the user experience. For example, running the TextCNN model locally on LG Urbane can save up to 71% of end-to-end latency compared to running on handheld CPU, while for the LSTM model, running locally leads to 2.1 seconds more delay compared to offloading to a handheld. The optimal decision depends on several factors described below.

- **Device heterogeneity.** There exist diverse wearable devices with highly heterogeneous hardware, ranging from a tiny smart ring to a large head-mount device for virtual reality. For example, our experiments show that for LG Urbane and Galaxy S2, they often need to adopt different offloading strategies: to achieve the lowest end-to-end latency for the Inception model, LG Urbane should offload the task to Nexus 6 while Galaxy S2 does not need to do so according to Figure 2(a).
- **Model structure.** Different deep learning models can vary a lot in terms of computational overhead and input size. Models with high computational overhead and small input such as LSTM and WaveNet are more suitable to be offloaded to handheds, while other models may not benefit from offloading such as TextCNN and SENNA. We observe that there is a non-trivial round-trip time around 62 ms between the handheld and wearable over Bluetooth. Due to such communication latency, even for small DL models, offloading them may incur a longer delay than running them locally despite powerful handheld processors. For example, for TextCNN, the communication latency contributes to 97.7% and 99.2% of the overall end-to-end latency when offloading to handheld CPU and GPU, respectively, as shown in Figure 2(b).
- **Processor status.** In real-world usage scenarios, handheld CPUs often run under different governors that control the clock frequency. The default governor of Android is interactive, which boosts CPU to the max frequency when running computation-intensive tasks. However, the system may also keep the CPU to run at low frequency (powersave) under certain scenarios, e.g., screen turned off, high device temperature, and low remaining battery. Figure 3 plots the end-to-end latency for four DL models. Similar to Figure 2, the two left columns are the latency of executing the whole model on a wearable, and the two right columns correspond to the latency of offloading them to Nexus 6 CPU with different governors (interactive vs. powersave). The results indicate that the CPU status can have substantial impacts on the end-to-end latency as well as the offloading strategy. Take WaveNet as an example. It takes almost 7X more time to process the model under the powersave governor than the interactive governor, with the former rendering offloading no longer beneficial. There are other processor status such as the load level (incurred by other concurrently running CPU/GPU-intensive processes) that can also affect the performance of both CPU and GPU [72], [43], [82], [51].
- **Latency vs energy preference.** Besides end-to-end latency, energy consumption is another key metric to consider as wearable devices have smaller battery capacities compared to their handheld counterparts. Figure 4 plots the energy consumption under different scenarios for two DL models. As shown, offloading deep learning models such as TextCNN to the handheld may consume more energy on wearables due to the network transmission overhead. For some other models such as Inception, though offloading can help save wearable battery, it will also cause non-trivial energy consumption for the handheld (around 2.9 J for Nexus 6 CPU). The results indicate that a challenge of making judicious offloading decisions is to balance the tradeoff among three factors: end-to-end latency, energy of wearable, and energy of handheld, based on user-specified preferences or policies. In real-world scenarios, a static policy may not always satisfy users’ requirements. For instance, when user’s handheld (wearable) is low on battery, ColNF needs to focus on saving the energy for the handheld (wearable). Therefore it is beneficial to adjust the offloading decisions dynamically based on external factors such as battery life, network condition, and CPU/GPU workload.

**Partial offloading.** The above pilot experiments only consider two scenarios: offloading the whole DL model to the handheld or executing it locally. Our further investigation indicates that partial offloading, i.e., dividing the deep learning model into two sub-models and executing them separately on the wearable and the handheld as shown in Figure 5, can sometimes achieve even better results.
Fig. 5: Three wearable DL execution approaches: offloading nothing, offloading everything, and partial offloading. Offloading nothing means executing all DL task on wearable. Offloading everything means offloading all DL task to handheld. Partial offloading, which is adopted in CoINF, means partitioning computation among wearable and handheld.

![Offloading Approaches](image)

Fig. 6: Energy consumption of running Inception model on LG Urbane and Nexus 6 under different partition scenarios. We only select 20 partition points to present the figure. X-axis presents the layer names that we select as partition point, after which output data are sent to handheld for continuous processing. The left-most bar represents handheld-only processing and the right-most bar represents wearable-only processing. The energy consumption of Nexus 6 in this figure is normalized as $E = \frac{\text{original } E}{\text{Nexus6_capacity}} \times \text{Urbane_capacity}$, where $\text{original } E$ is the real consumed energy at handheld, while $\text{Nexus6_capacity}$ and $\text{Urbane_capacity}$ is the battery capacity of Nexus 6 (3220 mAh) and LG Urbane (410 mAh).

We confirm the benefit of partial offloading through controlled experiments. Figure 6 plots the energy consumption (broken down into the wearable and the handheld parts) with different partition points for the Inception model. The left-most and right-most bars correspond to handheld-only and wearable-only processing, respectively. As shown, executing the model locally without offloading is the most energy-efficient for the handheld, while offloading the whole task to the handheld consumes the least mount energy for the wearable. However, users often care about the battery life of both devices, therefore we need to find a optimal partition to achieve the least total energy consumption. In this case, the overall optimal partition point resides in an internal layer ($\text{AvgPool}_0a$/AvgPool) that is neither input nor output. Doing such a partial offloading helps save around 84% and 29% of energy compared to the wearable-only and handheld-only strategies, respectively. Similarly, we found sometimes performing partial offloading helps minimize the overall latency, because an internal layer may yield a small intermediate output compared to the original input size, thus reducing the network transmission delay. Also sometimes partial offloading incurs better tradeoffs between energy and latency compared to the wearable/handheld-only schemes. Therefore, a key design decision we make for CoINF is to support partial offloading.

In conclusion, developing an offloading framework for wearable devices with various aforementioned factors considered is very challenging. We thus argue that flexible and efficient DL offloading support should be provided as a ready-made service to all applications, as opposed to being handled by app developers in an ad-hoc manner. To this end, we propose a holistic framework called CoINF. It is provided as a library with simple APIs exposed, and helps applications optimally determine whether or not to offload, how to offload, and what to offload with user preferences, partial offloading, and offloading policies taken into account. We next describe the design details of CoINF.

III. CoINF: DESIGN

We now present the design principles of CoINF. There are following goals that our framework aims to achieve.

- **Latency-aware**: Different DL applications have diverse end-to-end latency requirements. For example, interactive augmented reality apps always require a very short response time (no more tan tens of milliseconds), while background activity tracking
and analysis can tolerate a much longer delay. As a result, our CoINF should meet the appropriate user-perceived latency requirement, which is given by app developers, as the foremost goal to satisfy.

- **Working with off-the-shelf DL Models.** CoINF should not require the developers’ additional efforts to re-train the deep learning models. This is important as most app developers today only utilize off-the-shelf models in a “as-it-is” style. It makes no sense to ask them to manually tailor the models to the wearable contexts.
- **High Accuracy.** CoINF inherently should not sacrifice or compromise the accuracy when running DL models under diverse settings. In other words, CoINF should maintain consistently adequate accuracy results regardless of the offloading decision.
- **Trade-off-flexible.** CoINF should flexibly balance the tradeoff between the latency and energy based on external factors such as the device battery life on both the wearable and handheld devices. The policies can be specified and customized by users.
- **Developer-friendly.** CoINF should provide developers with simple API, as simple as the facilities provided off-the-shelf deep-learning frameworks/libraries such as TensorFlow, Caffe2, PyTorch, etc. More specifically, CoINF should abstract wearable and handheld devices as one entity by shielding low-level details such as offloading decisions.

### A. Overall Architecture

The overall architecture of CoINF is shown in Figure 7. To use CoINF, there are two main steps involved. 1) **The offline training phase** involves one-time effort of constructing the latency and energy prediction models, i.e., given a deep learning model structure, what is the end-to-end latency and energy consumption to run this model on a given device. Those prediction models are device-specific, indicating that the training process needs to be carried out on every single device of interest. To facilitate this process, we develop a suite of tools to automatically run deep learning models offline for multiple times along with varied parameters, and construct the prediction models by combining the extracted latency and energy information with the parameter setting. More details can be found in Section II-C.

2) In the **runtime phase**, DL applications rely on CoINF to perform adaptive offloading for DL tasks. There are following major components.

- **Decision Maker** is the core part of CoINF runtime. Given a DL model to run, the decision maker identifies the optimal model partition point based on the latency/energy prediction models and the current device’s running status provided by the System Profiler. The decision made indicates which part of the model should be executed locally and which part should be offloaded to the nearby paired handheld, including two special cases of offloading none or the entire task. A key logic of Decision Maker is to balance the tradeoff between the latency and the energy consumption. We will elaborate such a trade-off in Section II-D.

- **System Profiler** periodically profiles the system status including the pairing state, CPU governor (frequency) and CPU loads on both devices, GPU loads on handheld, Bluetooth bandwidth, etc. Such system dynamics can intuitively affect the decisions made by the Decision Maker.

- **DL Algorithms Driver** is the library that implements the deep learning algorithms. Currently, CoINF directly employs the unmodified TensorFlow library.

- **Data Trans Manager** deals with the communication and data transmission between the wearable and its paired handheld. It is realized using the standard Wearable Data Layer API in Android.

# Fig. 7: Overview of CoINF. Grey parts constitute a library provided for deep learning application developers.
| Model      | Conv | Fc  | Pooling | Act   | Total  | Model      | Conv | Fc  | Pooling | Act   | Total  |
|------------|------|-----|---------|-------|--------|------------|------|-----|---------|-------|--------|
| MNIST      | 39.0%| 54.2%| 1.1%    | 3.1%  | 97.4%  | TextCNN   | 71.6%| 1.1%| 1.9%    | 16.1% | 90.7%  |
| Inception  | 80.2%| 0.1%| 7.5%    | 8.1%  | 95.7%  | LSTM       | /    | 98.5%| /       | 0.7%  | 99.2%  |
| MobileNet  | 45.4%| /   | /       | 51.1% | 96.5%  | TextRNN   | /    | 16.0%| /       | 79.1% | 95.1%  |
| WaveNet    | 82.6%| /   | /       | 11.6% | 94.1%  | SENNA     | /    | 92.6%| /       | 7.2%  | 99.8%  |

TABLE III: The major latency composition of some popular deep learning models tested on Nexus 6. Conv, Fc, and Act are the abbr for convolution layers, fully-connected layers, and activation layers.

| Layer Type       | Prediction Model                                                                 |
|------------------|----------------------------------------------------------------------------------|
| Convolution, Pooling | decision tree input: \textit{filter\_size}, linear regression input: \textit{batch} \cdot \textit{input\_width} \cdot \textit{input\_height} \cdot \textit{channel} \cdot \textit{kernel\_number} ÷ \textit{stride}² |
| Fully-connected  | linear regression input: \textit{a\_width} \cdot \textit{a\_height} \cdot \textit{b\_width}, \textit{a\_width} \cdot \textit{a\_height}, \textit{b\_width} \cdot \textit{b\_height} |
| Activation       | decision tree input: \textit{activation\_function\_type}, linear regression input: \textit{input\_size} |

TABLE IV: Our latency & energy prediction models (decision tree + linear regression) for different kinds of deep learning layers and the prediction results. We use coefficient of determination $R^2$ as the metric to evaluate the accuracy of our prediction models (best possible score is 1.0).

- **Developer API Wrapper** is the developer interface through which DL applications can be easily developed to use the deep learning libraries with transparent offloading support. We present the design details in Section III-B.

### B. Developer APIs

ColINF exposes a set of easy-to-use APIs for developers for running model inference, as listed in the code snippet in List[1].

The high-level design principle of such APIs is to minimize the developers’ additional overhead including learning curve and programming efforts. Therefore, low-level details of whether/when/how to offload should be completely transparent to developers. As a result, the exposed interfaces are almost the same as a conventional DL library such as TensorFlow. The only new knob ColINF provides is a hint function for specifying the latency requirement (Line 3 in List[1]), which helps ColINF make offloading decisions.

Listing 1: A code sample of using ColINF

```java
CoINFInference infer = new CoINFInference("/path/to/model"); // Initialize a runtime from pre-trained model
infer.set_expected_latency(100); // Set the developer expected latency to 100 ms
infer.feed(input_name, input_data); // Feed the input data for computation
infer.run(); // Run the deep learning graph
float[] result = infer.fetch(output_name); // Get the computed results
```

As exemplified in Listing[1], using the APIs provided by ColINF is quite similar to using the standard Java APIs [18] provided by TensorFlow. It typically consists of four steps: loading pre-trained model, feeding the input, executing the graph, and finally fetching the output.

### C. Prediction Models

Now we consider how to construct the prediction model of the latency and energy for a given DL model. A simple approach is to treat a DL model as a “black box” and directly measure its execution latency and energy on both the wearable and the handheld, respectively. This approach however suffers from two limitations. First, since ColINF supports partial offloading that essentially breaks down a DL model into two parts, treating the DL model as a single black box does not apply. Second, each layer may have various parameters that need to be considered by the prediction models; the overall parameter space may thus be prohibitively very large for a whole model consisting of multiple layers.

To overcome the preceding challenges, our high-level approach is to model each layer individually and then to combine the prediction models across all layers into the final prediction model. To prove this approach is feasible and practical, we first carried out a simple controlled experiment via running DL models and logging the latency/energy in total as well as for each layer[3]. Through this controlled experiment, we found that to compute the latency/energy consumption of a given (possibly partial) DL model, we can compute the incurred latency/energy for every single layer and then sum them up. More

[3] Built-in TensorFlow functionality to log individual layer performance: [https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/stat_summarizer.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/stat_summarizer.h)
specifically, the result made by summing up latency/energy for each layer has no more than 1.82% deviation compared to the overall latency/energy. But we still need to deal with a practical problem: there are a large number of layer types inside a DL model (e.g., more than 100 types supported in TensorFlow). As a result, making a prediction model for each of them can thus incur considerable substantial training overhead. Fortunately, we find that among those hundreds of layer types, only a small number of them are responsible for typical workloads on wearables: 1) **Convolution layers** are used to convolve an image with a set of learned filters to produce a set of feature maps, 2) **FC (fully-connected) layers** are used to exhaustively connect neurons of two layers, 3) **Pooling layers** are used to apply a pre-defined function such as max or average over regions of input feature maps to group features together, and 4) **Activation layers** are used to apply a linear or non-linear function to each of its input data individually and produce the same amount of data as output. The most commonly used activation functions include Tanh, RELU, Sigmoid, add, mul, etc. As shown in Table III, these four layer types constitute up to more than 90% of the inference latency of popular DL models. It is important to note that although current CoINF considers only these layers, other layer types can be easily incorporated into our framework.

We next describe the methodology of building a prediction model of latency/energy for a given layer.

**Latency Prediction.** We observe that even for the same layer type, there is a large latency variation across different layer parameters (e.g., the size of input feature map and the kernel for convolution layers). Thus, we vary the configurable parameters of the layer and measure the latency for each parameter combination. We use the collected latency data to train and test our prediction models. As shown in Table [V], we use a combination of decision tree and linear regression to model the latency. The former is used to classify some types (i.e., convolution, pooling, and activation) into sub-types based on several metrics (i.e., kernel size and activation function). Then, we apply a linear-regression model to each of those sub-types to get the final predicted results. As shown in Table [V], our latency prediction models perform well, especially for the two most computation-intensive layers: convolution and fc, with a high variance score of 0.993 and 0.945, respectively. Here, we use the coefficient of determination $R^2$ to measure the accuracy.

**Energy Prediction.** We use a similar approach to predicting the energy consumption of a layer. Our model construction technique can directly work with existing power models that are provided by the device vendor or 3rd-party measurement efforts. In our study, we typically build the power models for the smartphone by using the Monsoon Power Meter [13] (following a high-level approach of component-based power modeling [78]), or obtain them from the literature [55] for smartwatch. As shown in Table [IV], our energy prediction model can have a satisfactory accuracy (> 92%) for 3 out of 4 layer types. Although prediction result of Pooling layer is relatively lower (0.772), fortunately, as shown in Table 4 this layer contributes little to total latency & energy compared to other layers.

**D. Offloading Decision**

Utilizing the layer performance of prediction models described above, CoINF dynamically selects the optimal partition point. We take two extreme cases: offloading nothing but making the entire DL models run locally on the wearable, and offloading the entire DL model to the handheld. The decision making procedure involves two major steps: finding a set of possible partitions for a given graph, and identifying the optimal one among them.

**Dynamic Partition.** A deep-learning model can be abstracted as a Directed Acyclic Graph (DAG) with the source (input) and the sink (output) nodes, where each node represents a layer and each edge represents the data flow among those layers. A partition equals to a cut [9] of the DAG and requires the source and the sink to be placed in different subsets. However, finding all cuts of a given graph shall need the $O(2^n)$ complexity where $n$ is the number of nodes (layers in fact). For a large deep-learning model, e.g., the Inception model with 1,096 nodes, such a complexity is prohibitive and cannot be accepted in practice. Previous work such as [48] that also tries to partition DL graphs and simply assumes these graphs are linear. Hence, each single edge represents a valid partition point. However, we observe that such an assumption is not always reliable for many popular DL models (e.g., Inception). One core reason is that there can be branches and intersections in the graph. This observation motivates us to design a heuristic-based algorithm that efficiently computes a set of “representative” cuts for a general graph of a DL model, as to be described below.

Our algorithm works as follows. We begin with simplifying the graph based on two observations. First, most operations (nodes) are rather lightweight in computation and contribute quite little to the end-to-end latency (shown in Table [V]) along with the energy consumption. Second, one graph often contains repeated subgraphs connected. This pattern is common in CNN and RNN models where developers often use a looper to construct the model as exemplified in Figure [8]. Based on the above observations, as shown in Figure [9] our graph simplification works in two steps: pruning low-computation nodes and mining frequent subgraphs. Currently, CoINF identifies all nodes except convolution, fc, pooling, and activation as low-computation, based on our findings in Section III-C, again, this low-computation node set can be flexibly changed based on the profiling

---

4We observe that there are only limited kinds of kernel size used in current CNN models, which is 1X1, 3X3, 5X5, 7X7, and 11X11.

5Although we mention in Section II that neuron is the atomic unit in a DL model graph, in practice the programmable graph unit provided by frameworks such as TensorFlow is layer (or called operation). Thus, in the rest of this paper we use “node” to represent “layer”, and it is the atomic unit that CoINF makes partition decision.

6If we take the deep-learning graph as a flow network [11], then a partition is intuitively an s-t cut.
results). After identifying the low-computation nodes, CoINF removes them and connects their inputs and outputs. To mine frequent subgraphs, the most straightforward way is utilizing the node namespace: nodes under the same namespace are often in the same subgraph. However, the idea of namespace is not supported in all deep-learning frameworks; more importantly, setting the namespaces is rather subjective and optional, and requires developers’ additional support. We thus utilize the GRAMI [34], a fast and versatile algorithm that automatically mines frequent subgraphs. After the graph mining, we group each subgraph into one node so that we do not have to split this subgraph further when cutting the graph.

After the graph is simplified, there will be much fewer nodes (e.g., 1, 096 to 35 for Inception). Additionally, the graph exhibits a mostly linear structure. This allows us to apply a brute-force approach to identify all cuts. These cuts are then delivered to the next step. In addition, this simplification result will be cached for every single DL model and can be reused, so that CoINF does not bother to run above algorithm repeatedly.

**Optimal Partition Selection.** The algorithm for determining the optimal partition is demonstrated in Algorithm III.1. Taking as input possible partitions generated by the previous step, CoINF analyzes the partitioned subgraphs on the wearable and the handheld, and uses the prediction models (Section II-C) to estimate the corresponding latency and energy consumption (line 2~11), respectively. Note that the overall energy consumption metric is a weighted mean from the energy consumed on both the wearable and handheld. If the DL app integrated with CoINF is latency-sensitive, or more specifically, developers require that the deep-learning inference should be processed as fast as possible, we select the partition with the best end-to-end latency (line 12~13). In contrast, if the app is latency-insensitive, then we select the partition with the lowest energy consumption (line 14~15). In a more general case, the developer is able to quantitatively specify the latency requirement (Section III-B). We then select the most energy-efficient partition that satisfies this latency requirement (line 16~18).

The models and parameters in Algorithm III.1 are obtained from various sources and can be classified into four types: 1) offline-training models, including the latency and energy prediction models (f, g), as well as the power model of Bluetooth data transfer (PR, PT), 2) runtime-profiling parameters gathered by the System Profiler module (Section III-A), including the handheld status (S) and Bluetooth bandwidth (B), 3) application-specified parameters, including the expected end-to-end latency of deep learning inference (PropT) (Section III-B), 4) trade-off parameters that can be configured by end users, such as the energy consumption weights on wearable and handheld (W_w, W_p).

**End-user Interface.** For the fourth type mentioned above, CoINF provides several pre-defined rules for different usage contexts. For example, the rule R1 (W_w ← 100 − wear_battany_level) takes into account the current battery, and ensures that CoINF focuses on saving the energy when the wearable is in low battery. A similar rule can be applied for the handheld battery. In another example, the rule R2 (W_w ← 1, W_p ← 0 when handheld being charged = true) can be applied when the smartphone connects to an external charger. CoINF provides interfaces allowing end-users to pick some high-level “profiles”, which will then be translated to the above low-level rules, as illustrated in Figure 9.

**E. Optimization for Streaming Data**

Deep learning tasks such as video stream analysis for augmented reality and speech recognition will become common on wearable devices. In these tasks, the input consists of a stream of data such as video frames and audio snippets that are
Input: 
\[ G: \text{pre-trained graph to be executed} \]
\[ p(G): \text{binary-partition function, returns a list of valid partitions } (G_{\text{wear}}, G_{\text{handheld}}, dt), \text{ where } dt \text{ is the size of data needs to be transferred among the two partial graphs} \]
\[ f(G, S), g(G, S): \text{pre-trained models for predicting the latency/energy of executing } G \text{ under device status } S \]
\[ S_w, S_p: \text{current device running status for wearable and handheld, including CPU frequency, CPU loads, etc} \]
\[ B: \text{current Bluetooth uplink bandwidth} \]
\[ PR, PT: \text{power consumption of receiving/sending data via Bluetooth} \]
\[ PropT: \text{proper latency that the application is supposed to run on} \]
\[ W_w, W_p: \text{weights of battery for wearable and handheld} \]

Output: Optimal partition choice 
\[ \text{partitions} \leftarrow p(G), \text{latency}_\text{lst} = \text{energy}_\text{lst} = \emptyset \]

for each \( G_{\text{wear}}, G_{\text{handheld}}, dt \) ∈ partitions do
  if streaming_opt_on then
    Streaming data optimization turned on
    \[ \text{latency} \leftarrow \max(f(G_{\text{wear}}, S_w), f(G_{\text{handheld}}, S_p) + dt/B) \]
    \[ \text{energy}_w \leftarrow g(G_{\text{wear}}, S_w) + dt * PT \]
    \[ \text{energy}_p \leftarrow g(G_{\text{handheld}}, S_p) + dt * PR \]
  else
    \[ \text{latency} \leftarrow f(G_{\text{wear}}, S_w) + f(G_{\text{handheld}}, S_p) + dt/B \]
  end

end if

\[ \text{if PropT} = 0 \text{ or } \min(\text{latency}_\text{lst}) > PropT \] then \text{Latency-sensitive applications}
  \[ \text{opt}_\text{index} \leftarrow \arg \min (\text{latency}_\text{lst}[i]) \]
  \[ i \in \{1..N\} \]
else \text{Latency-insensitive applications}
  \[ \text{opt}_\text{index} \leftarrow \arg \min (\text{energy}_\text{lst}[i]) \]
  \[ i \in \{1..N\} \]
end if

Applications for trade-offs between latency and energy
\[ R \leftarrow \text{list of index } i \text{ that satisfies latency}_\text{lst}[i] \leq PropT \]
\[ \text{opt}_\text{index} \leftarrow \arg \min (\text{energy}_\text{lst}[i]) \]
\[ i \in R \]

return \[ \text{partitions}[\text{opt}_\text{index}] \];

Algorithm III.1: CoINF Optimal Partition Algorithm.

continuously fed into the same model. Here we use “frame” to indicate an input unit for a DL model, e.g., an image or an audio snippet.

A major difference between streaming and non-streaming data processing is that the former cares more about the overall throughput, e.g., how many frames can be processed per time unit, rather than the end-to-end latency for every single frame. Therefore, CoINF introduces a key optimization for streaming data: pipelined processing on wearable and handheld.

For non-streaming input, the data dependency between two partitioned sub-models makes pipelined or parallel processing impossible: when the wearable is processing the first part of model, the handheld has to wait for its output that serves as the input to the second part of the model to be executed on the handheld. For streamed input, we can leverage the opportunity of pipelining the processing of two consecutive frames on wearable and handheld. Specifically, when the \( n \)-th frame is finished computing on the wearable and being sent to the handheld, the wearable can immediately start processing the \( (n+1) \)-th frame, and so on. Pipelining intuitively makes both devices busy and thus effectively improves the overall throughput. To better integrate the pipelining support into our partition-decision algorithm, we revise the end-to-end latency calculation in Algorithm 3.1 as the maximum of the wearable computation delay and the handheld computation delay along with the data transmission delay (Line 4). In other words, due to pipelining, the amortized end-to-end latency is determined by the processing delay on either device, whichever is longer. CoINF will automatically turn on the pipelining mode (Line 3) when it detects multiple input instances arriving back-to-back.

IV. CoINF: IMPLEMENTATION

We have implemented CoINF on commodity Android smartphone and smartwatches. Our prototyping efforts consist of around 3,200 lines of code written in Java, excluding the scripts for constructing and analyzing prediction models. We next highlight several significant implementation details.

At the wearable-side where DL tasks are requested, developers need to only import CoINF library and follow the APIs mentioned in Section II-B. At the handheld-side, CoINF is deployed as an Android system-wide service. Developers need to only import CoINF library into the handheld-side app and declare the components in the manifest file. In the one-time initialization phase when the app is being installed, CoINF will also locate other necessary components such as the DL models (stored at both the wearable and the handheld side) and the latency/energy prediction models (stored at the wearable side only). The handheld-side library also provides a console allowing users to configure offloading policies as described in Section III-D.
Currently, CoINF uses TensorFlow [17] as the DL algorithms driver (Figure 7) due to TensorFlow’s good support for mobile environment and partial execution. Other popular frameworks such as Caffe2 [8] and PyTorch [14] can also be easily integrated into CoINF with very small adaptation. To realize the System Profiler, CoINF obtains the processor status via the sysfs interface of operating system. More specifically, the CPU information can be obtained from /sys/devices/system/cpu/ on both the smartphone and the smartwatch. For GPU on smartphones, the hardware driver exposes the information such as the total running time and busy time. On the Nexus 6 model, such information can be obtained from /sys/class/kgsl/kgsl-3d0/.

The data communication between wearable and handheld is realized by the standard Wearable Data Layer API [22] in Android platform. Specifically, we use the Message API [7] for the message exchange in the control channel, and use DataItem & Asset APIs for transferring computation results and intermediate data (when the DL model is partitioned across the two devices). The Bluetooth bandwidth is profiled by transferring a fixed-size file between the wearable and the handheld. The measurement is triggered periodically (by default every 5 minutes) as well as when the Bluetooth signal strength changes on either side.

V. EVALUATION

We then comprehensively evaluate CoINF using 8 popular deep-learning models aforementioned as our benchmark under different handheld device status (CPU-interactive, CPU-powersave, and GPU) and configuration (\(W_w\) and \(W_p\)). We first present whether CoINF can accurately find the optimal partition points under complex contexts and configuration described in Section V-A. We then demonstrate that CoINF is able to achieve significant execution speedup and energy improvements over two naive strategies: wearable-only (no offloading) and handheld-only (offloading everything) approaches mentioned in Section V-B. CoINF’s capacity to satisfy developers’ expected latency is demonstrated in Section V-C. We also evaluate CoINF’s performance under variations of device status, e.g., battery level, Bluetooth network connectivity, and the workloads on handhelds in Section V-D. In addition, we demonstrate how our techniques presented in Section III-E can further improve the performance of CoINF in Section V-E. Finally, we present the system overhead of CoINF in Section V-F.

The experimental setup is the same as our motivating experiments explained in Section II-B. We directly logged the latency in an application. The energy consumption is retrieved by the Monsoon Power Meter [13] (for Nexus 6 and Galaxy S2) or previously established power model [55] (for Urbane LG). Each experiment is repeated for 20 times in order to make sure the results are statistically meaningful.
TABLE V: $CoINF$ partition point selections under different configuration, devices, and models. Red blocks indicate $CoINF$ fails to make the optimal partition choice and white block means the optimal partition point is picked. On average, $CoINF$ achieves 96.9% accuracy of selecting best partition point. The energy consumption of handheld is normalized as in Figure 6. Here, we consider only two extreme configuration: latency-sensitive (chasing for the smallest end-to-end delay) and latency-insensitive (chasing for the lowest energy consumption).

A. Partition Selection Accuracy

Table[V] summarizes the partition points selected by $CoINF$ under different configurations, devices, and deep learning models. Each cell represents the DL layer name at which $CoINF$ performs partition, indicating that the output data of this layer shall be offloaded to the handheld. $PropT$ is the developer-specified expected latency, while $W_w$ and $W_p$ are the importance weights of wearable and handheld energy, respectively, as defined in Section III-D. The 3 red blocks indicate that $CoINF$ fails to make the optimal partition choice while other white blocks mean the optimal partition point is picked. Here, an “optimal” partition choice means that it outperforms all other partition choices for the specified goal, e.g., end-to-end latency when $PropT$ equals to 0, or normalized energy consumption when $PropT$ equals to $+\infty$.

In summary, $CoINF$ is able to select the best partition point for 93 out of 96 (96.9%). The mis-predictions occur because of two reasons. First, our prediction models used in $CoINF$ consider only a subset of layer types as explained in Section III-C. However, the ignored layers still contribute to the end-to-end delay and the energy consumption at runtime. Second, those prediction models themselves cannot perfectly predict the delay or energy. However, these mis-predictions made by $CoINF$ should not cause heavy overhead as those incorrect partition points selected are usually suboptimal (all 3 selection in Table[V]), and the running delay & energy consumption of these incorrect selections are actually very close to the optimal partition. For example, $CoINF$ incorrectly selects the AvgPool_0a layer as the partition point when $PropT$ equals to 0 under the LG Urbane along with the CPU-powersave status, while the optimal choice would be input. But partitioning at AvgPool_0a leads to an end-to-end latency of 4.341s, only 0.3% increasing against the partitioning at input (4.328s).

B. Latency & Energy Improvements

To demonstrate how $CoINF$ can help improve the the end-to-end latency and the overall energy consumption, we test it under two extreme cases: optimizing for latency ($PropT = 0$) and optimizing for energy ($PropT = +\infty$). We compare the performance of $CoINF$ with other two naive strategies: handheld-only (offloading all tasks to the handheld) and wearable-only (executing the entire model on the wearable without performing offloading).

Execution speedup. Figure[10] shows $CoINF$’s execution speedup (normalized) over naive strategies across 8 DL models and varied device specifications & status. Bars in different colors represent different hardware configurations. LG and S2 are abbreviated for Urban LG and Galaxy S2. CPU-it, CPU-ps, and GPU mean utilizing Nexus 6 under CPU-interactive, CPU-powersave, and GPU at handheld-side, respectively. The black bar represents the data of handheld- or wearable-only approaches, and is always normalized to 1. The red bar is the average speedup of the 6 hardware configurations for each model.

Figure[10(a)] shows that compared to the handheld-only strategy, $CoINF$ can help reduce the end-to-end latency for 7 out of 8 models, with an average improvement (red bar) ranging from 1.01X (WaveNet) to 9.14X (TextCNN). Similarly, Figure[10(b)]
shows that compared to wear-only strategy, CoINF is able to help reduce the end-to-end latency of running 4 out of 8 models at specific scenarios, with an average improvement ranging from 1.07X (MNIST) to 8.86X (WaveNet). For cases such as running WaveNet on LG Urbane with Nexus GPU 6 available, CoINF can even speed up the processing for more than 20 times (23.0X) compared to the wearable-only strategy.

Another observation from these two figures is that different models exhibit very diverse results. We find that the execution speedup achieved by CoINF depends on two factors related with the model structure: computation workloads and data size. A model graph with small computation workloads (TextCNN, TextRNN) or with large input data size (image-processing applications such as Inception and MobileNet) usually cannot benefit from offloading since the bottleneck resides in the data transmission rather than the local processing. Hence, in these cases, CoINF can have significant advantages over the handheld-only approach, but less improvements over wear-only approach. In contrast, when running DL models that requires lots of computations on relatively small size of data, CoINF exhibits more improvements compared to the wearable-only approach rather than the handheld-only approach.

**Energy saving.** Figure [11] shows how much energy CoINF consumes compared to the handheld-only and the wearable-only strategies across those 8 DL models and varied device specifications & status. Figure [11](a) shows that compared to the handheld-only strategy, CoINF can help reduce energy consumption of running 6 out of 8 models, with an average improvement (red bar) ranging from 3.3% (Inception) to 81.3% (TextCNN). Similarly, Figure [11](b) shows that compared to the wearable-only strategy, CoINF can help reduce energy consumption for 5 out of 8 models, with an average improvement (red bar) ranging from 3.8% (TextRNN) to 85.5% (WaveNet).

In addition, we have a similar observation made previously: the energy saving heavily depends on DL model structures. CoINF can save more energy compared to the handheld-only approach for models that require only few computation workloads but large data size during execution. Because for those models, energy consumed in Bluetooth transfer contributes the bottleneck of the overall energy consumption, making offloading no advantage over local execution.
C. Awareness of Applications’ Latency Requirements

We also evaluate how the developer-specified latency ($PropT$) affects $CoINF$’s decision on offloading. We select three deep-learning models and three practical $PropT$ values for each of the models, and compare the performance of $CoINF$ with that of the handheld-only and the wearable-only strategies. The results are shown in Table VI.

Overall, for 7 out of 9 configurations, $CoINF$ can satisfy the latency requirement, while the handheld-only and the wearable-only have only 4 and 6, respectively. Two special cases of $CoINF$’s incapability to deliver a proper latency, i.e., $PropT = 2.0s$ for Inception and $PropT = 1.0s$ for LSTM, are unavoidable since even the lowest possible latency is higher than $PropT$. In those cases, $CoINF$ chooses to minimize the end-to-end latency and ignore the energy consumption. In summary, for all cases, $CoINF$ yields satisfactory results.

Another key observation from Table VI is that $CoINF$ can dynamically adjust its decisions based on applications’ requirements – a desirable feature in practice. Taking TextRNN as an example. When $PropT$ is low, $CoINF$ keeps all workloads in local
wearable device to satisfy (58.9ms) the latency requirement (200ms). This is exactly the same as what the wearable-only strategy does but the handheld-only strategy fails to achieve. When \( PropT \) becomes higher (300ms), \( CoINF \) chooses different partition points in order to consume a lower energy than the wearable-only strategy, while keeping a relatively low end-to-end latency. The wearable-only strategy instead consumes 21.6% more energy than \( CoINF \) in this case.

### D. Adaptiveness to Environmental Dynamics

In this section, we evaluate \( CoINF \)'s adaptiveness to diverse factors that may vary in real-world environments: the device battery level (\( W_w \)), the Bluetooth bandwidth (\( B \)), and the processor load level (\( S_p \)). Our experimental results show that \( CoINF \) can effectively adapt to the dynamics caused by these external factors.

**Battery level.** As mentioned in Section III-D, \( CoINF \)'s offloading decision should consider the battery level of both the wearable and the handheld, in order to better balance their battery life. This is achieved by tuning the parameters \( W_w \) and \( W_p \). We exemplify a possible policy as follows. When the handheld is being charged, we focus on saving the energy for wearable (i.e., \( W_w = 1, W_p = 0 \)), whereas when the handheld’s battery is running out, we should more aggressively use the wearable’s battery (e.g., by setting \( W_w = 0.2 \) and \( W_p = 0.8 \)).

We test \( CoINF \)'s robustness against the varying values of \( W_w \) and \( W_p \) (set to \( 1 - W_w \)). As shown in Figure 12, the partition decision of \( CoINF \) keeps changing according to the configuration of energy weight. As a result, \( CoINF \) always consumes no more energy than either the wearable-only or the handheld-only strategy. Take TextRNN as an example as shown in Figure 12(a). When \( W_w \) is low (0 \( \sim \) 0.3), \( CoINF \) chooses to run the model locally as the wearable energy is relatively “cheap”. When \( W_w \) becomes higher (0.3 \( \sim \) 0.8), the model is partitioned and executed on both sides. During this stage, \( CoINF \) outperforms both wearable-only and handheld-only strategies. When \( W_w \) is high, \( CoINF \) offloads all workloads to the handheld to save the energy of wearable. The results of MobileNet, another example shown in Figure 12(b), are similar to TextRNN, except that for MobileNet there is no partial offloading stage. Such a difference stems from the different internal structure of MobileNet.

**Bluetooth bandwidth.** The Bluetooth bandwidth for data transmission between the wearable and handheld can change dynamically, e.g., when other applications are using Bluetooth or the distance between wearable and handheld devices varies. \( CoINF \) profiles this bandwidth online and takes it into consideration for partition decision. Figure 13 shows how \( CoINF \) reacts...
Fig. 14: End-to-end latency of CoINF across different handheld processor load level ($S$) compared to handheld-only and wear-only strategies. We use Urbane LG and Nexus 6 GPU & CPU to carry out this experiment.

Fig. 15: Optimized throughput of CoINF with pipeline. Results are normalized by CoINF without this technique. We use Urbane LG and Nexus 6 CPU-interactive to carry out this experiment.

to the changing bandwidth in consideration of end-to-end latency. As observed from both Figure 13(a) (the MNIST model) and Figure 13(b) (the Inception model), CoINF tends to execute the whole deep learning model locally when the bandwidth is low, since it indicates that heavy overhead will be introduced by offloading. When the bandwidth is high, CoINF will perform offloading more aggressively. If necessary, CoINF also chooses to partially offload the workload. For example, when running MNIST with bandwidth around 100KB $\sim$ 140KB, leading to better performance than both the wearable-only and the handheld-only strategies.

Handheld processor load level. We then evaluate CoINF’s robustness against varying load level of the handheld processors (CPU and GPU). We use a script [4] to generate specified CPU workloads, and an application [46] that generates GPU workloads by rendering graphics in the background. As shown in Figure 14, when the processor load is low, CoINF always offloads the deep-learning tasks to handheld to make use of the under-utilized processing power. In this stage, the performance of CoINF is similar to the handheld-only strategy, and has significant end-to-end latency reduction compared to the wearable-only strategy (e.g., more than 1 second for LSTM model shown in Figure 14(b)). When the handheld processor’s load increases, CoINF chooses to execute workloads locally on the wearable device, thus outperforms the handheld-only approach. For example, when running MNIST with the handheld GPU load of 80%, CoINF can reduce almost 50% of the end-to-end latency compared to the handheld-only strategy (188.2ms vs. 99.8ms).

E. Handling Streaming Data

We also evaluate how our techniques described in Section III-E can help improve the throughput for streaming data. As shown in Figure 15, applying pipelining in CoINF can help improve the overall throughput by 43.75% averaged over the 8 models. For some models such as MNIST, the throughput improvement as high as 84% can be achieved through these two techniques. We also observe that the improved throughputs for different models depend on the performance divergence running on the wearable and the handheld. For models that exhibit large divergence when running at local wearable or offloaded to handheld, the proposed technique can often have fewer improvements. For example, running WaveNet at local, Urbane LG leads to 7.7s end-to-end latency, almost 13 time longer than offloading on Nexus CPU (0.54s). Thus, performing pipelining
for streaming input to WaveNet has only 5% throughput improvement. The reason of this observation is quite straightforward. Utilizing the processors from both wearable and handheld concurrently cannot benefit a lot compared to using only the optimal one of them, if the non-optimal choice has trivial contribution to the overall throughput. In contrast, for models that exhibit similar performance at the wearable and the handheld, pipelining leads to a higher throughput improvement (84% for MNIST model).

F. System Overhead

As evaluated previously, CoINF can dynamically select the best partition point at least no worse than the wearable-only and the handheld-only approaches at most times. But CoINF may have higher latency or energy consumption compared to the optimal one between the wearable-only and the handheld-only, since CoINF incurs computation overhead to execute our partition algorithm. However, this overhead is a bit trivial compared to the original processing time & energy. As we have measured for all 8 popular DL models under different combination of contexts, the incurred overhead contributes to the overall performance of CoINF ranging from 0.49% (Inception) to 4.21% (TextRNN), which is not too significant. The reasons of such low overhead is multifolded. First, our heuristic-based algorithm, as presented in Section II-D can reduce the computation complexity to almost $O(n)$, where $n$ is the number of DL model nodes. Second, the original DL computation is already heavy-load, making the overhead relatively trivial.

Another energy overhead comes from our System Profiler, which needs to periodically profile the processor status and Bluetooth bandwidth. We measure this overhead for 1 hour with a default triggered frequency of every 5 minutes, and the result turns out to be small compared to the energy saving achieved by CoINF (5.2% for MobileNet). This overhead can be further improved by triggering this profiler only when CoINF-integrated apps are opened and DL models are loaded.

VI. DISCUSSION

In this section, we discuss the limitations of our proposed framework and possible future work.

The current version of CoINF focuses only on the inference stage in deep learning, which requires a pre-trained model integrated in applications or downloaded in advance. Although performing inference may be sufficient for most applications such as augmented reality and speech recognition, we notice that in recent years there have emerged requirements to train (consume) the data immediately when it is produced on wearable devices. To this end, we plan to support the model training in CoINF, of which the most challenging and interesting part is designing new latency & energy prediction models for training procedure, typically based on the backpropagation algorithm.

CoINF makes partition decision based on two key metrics of user experience: the end-to-end latency and the energy consumption. Besides, memory usage (both average and peak) is another important metric that should be taken into account [41]. We plan to consider memory usage as a developer-specified policy similar to the latency (PropT). This extension can be integrated into CoINF via a runtime predicator of memory usage for different partitions and a new set* API for developers.

When implementing CoINF, we notice that there is no mature deep-learning framework that can run on GPU of commodity devices. Thus, we leverage a research prototype called RSTensorFlow [23]. Although RSTensorFlow works adequately in most cases, it also has some significant flaws. First, RSTensorFlow accelerates only the processing of convolution and matmul operations. Second, RSTensorFlow has some deployment issues which have been reported [15]. Fortunately, there are many on-going efforts that plan to push the GPU support to commodity mobile devices. We plan to integrate these more adequate GPU frameworks into CoINF when they are released.

We have tested only current version of CoINF on three devices (LG Urbane, Galaxy S2, and Nexus 6) and 8 popular deep-learning models. These models are quite popular and widely used. To make CoINF more generalized, we plan to test its effectiveness and efficiency with more devices, which needs to generate prediction models offline and integrate the results into CoINF. We also plan to evaluate CoINF with more deep learning models such as VGG [64].

We now use TensorFlow as the algorithm driver in CoINF to implement the inference procedure. TensorFlow has rich features and a good developer community. But some flaws could possibly affect our implementation. One problem is that TensorFlow is mainly used in PC environment and has not much optimization for mobile devices. In the future, we plan to test more mobile-optimized deep-learning frameworks such as Caffe2 [8] and ncnn [16], which may be more friendly to users in consideration of memory usage, latency overhead, and energy consumption.

Currently, CoINF is designed and implemented as a user-space library. Another alternative is integrating the handheld-side part of CoINF into Android OS as a system-level component (service). Although this approach can lead to more deployment difficulty as it requires help from device vendors, there are also several advantages of positioning CoINF as a system-level component. First, it relieves developers from importing handheld-side library and other related efforts. Second, it saves system resources such as memory when multiple apps co-run DL tasks and share the same service on the handheld. Third, the shared system service can make better scheduling decision since they have the access and communicate to all DL apps on the wearable.
VII. Related Work

In this section, we discuss existing literature studies that relate to our work in this paper.

Optimizing Deep learning for wearables. Building deep-learning applications and optimizing its execution has drawn a dramatic attention in both industry and academia. For example, Google combines the deep neural networks and graph-based machine learning to build an entirely “on-device” ML technology for powering smart messaging on Android Wear 2.0, enabling technologies like Smart Reply to be used for any application [6]. In academia, there are several directions to scale down the deep-learning workloads for wearables and mobile devices. Some efforts [53, 20, 68, 42] have proposed deep models that are much smaller than normal without sacrificing too much accuracy, so they can run directly on mobile CPU or even DSP. Some other efforts such as [27, 77, 28, 40] aimed at building customized hardware architectural support for machine/deep learning algorithms. Model compression [51, 21, 52, 33, 49, 41] is another popular approach of accelerating deep learning and reducing the energy consumption. For example, DeepX [51] dramatically lowers the resource overhead by leveraging two inference-time resource control algorithms: Runtime Layer Compression (RLC) and Deep Architecture Decomposition (DAD). Those approaches essentially made trade-off among the accuracy and resource consumption. Among various compression methodologies, weight pruning [73, 60, 54, 72] is a widely explored approach to optimizing executing CNN models. The key of this approach is selecting the “proper” weights to prune or compress. A recent effort [73] proposed to preferentially prune the weights of nodes that are predicted to be energy-hungry.

Differently from the preceding work, CoINF explores how to efficiently utilize the processing capacity from nearby paired handhelds. All the existing techniques can be leveraged atop CoINF to optimize the deep learning processing collaboratively.

Workloads offloading. Some previous research efforts [61, 57, 29, 51, 70, 80, 79, 44] have focused on offloading computation from the mobile to cloud. However, all these frameworks share a key reason that makes them not applicable enough for the computation partition in CoINF. These frameworks are control-centric, as they make decisions about regions of code (or functions), whereas CoINF leverages the domain knowledge from deep-learning technology and makes partition decisions based on the model topology. Code-level offloading cannot work make good partition decisions because layers of a given type (even if mapped to the same function) within one deep-learning model can have significantly different compute and data characteristics.

Neurosurgeon [48] explores a similar offloading idea with CoINF in some aspects, as it can automatically partition DNN computation between mobile devices and data-centers at the granularity of neural network layers. However, CoINF has several important and unique aspects. First, CoINF focuses on the collaboration between wearable devices and their paired handheld devices rather than remote cloud. Such an architectural style is quite similar to the concept of edge computing. The offloading in CoINF exposes more challenges, e.g., balancing the energy consumption on both devices. Second, Neurosurgeon can schedule only for optimal latency or energy, while CoINF first makes attempts to satisfy the latency requirement specified by app developers and then make proper trade-off between latency and energy. Our experiments show that the latter strategy can provide substantial benefits. Third, Neurosurgeon targets at only linear DNN models, while CoINF utilizes a novel approach to handling more complex deep-learning models as shown in Figure 8.

DeepX [51] also tries to partition deep learning models. However, DeepX only distributes partitioned sub-models on different local processors (CPU, GPU, and DSP) while CoINF offloads the workloads to a remote device processor. The key difference is that CoINF needs to take the data transition overhead into consideration, which can have ultimate impacts on the decision made as we have already shown.

VIII. Conclusion

Wearables are playing an increasingly important role in the mobile computing ecosystem. Their on-body and ubiquitous nature makes them an ideal data source for numerous smart applications that can be powered by deep-learning tasks. To this end, we have developed CoINF, a practical, adaptive, and flexible DL framework designed for wearables. Our work demonstrates that the wearable-side deep-learning applications can achieve satisfactory performance and low energy consumption through strategically selective offloading to a paired handheld device. In our future work, we plan to leverage the CoINF to develop real DL applications for COTS wearable devices, and deploy them on real users through an IRB-approved user trial. We also plan to open-source our CoINF prototype.

REFERENCES

[1] Smartwatches now more popular than Swiss watches. https://www.cnet.com/news/smartwatches-now-more-popular-than-swiss-watches-thanks-largely-to-apple/.
[2] Swiss watchmaking in March 2016 – Steep decline. http://www.fhs.swiss/scripts/getstat.php?file=comm_160303_a.pdf.
[3] The Swiss and world watchmaking industry in 2015. http://www.fhs.swiss/file/59/Watchmaking_2015.pdf.
[4] A Programmable CPU Load Generator. https://github.com/ptitiano/cpuloadgen] 2012.
[5] How Google Translate squeezes deep learning onto a phone. https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html 2015.
[6] On-Device Machine Intelligence. https://research.googleblog.com/2017/02/on-device-machine-intelligence.html 2016.
[7] Android Message API. https://developer.android.com/reference/com/google/android/gms/wearable/MessageApi.html 2017.
[8] Caffe2 deep learning framework. https://github.com/caffe2/caffe2 2017.
[9] Cut in graph theory. https://en.wikipedia.org/wiki/Cut_%28graph_theory%29 2017.
