DeepWear: Adaptive Local Offloading for On-Wearable Deep Learning

Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, Xuanzhe Liu *Member, IEEE*

Abstract—Due to their on-body and ubiquitous nature, wearables can generate a wide range of unique sensor data creating countless opportunities for deep learning tasks. We propose DeepWear, a deep learning (DL) framework for wearable devices to improve the performance and reduce the energy footprint. DeepWear strategically offloads DL tasks from a wearable device to its paired handheld device through local network connectivity such as Bluetooth. Compared to the remote-cloud-based offloading, DeepWear requires no Internet connectivity, consumes less energy, and is robust to privacy breach. DeepWear provides various novel techniques such as context-aware offloading, strategic model partition, and pipelining support to efficiently utilize the processing capacity from nearby paired handhelds. Deployed as a user-space library, DeepWear offers developer-friendly APIs that are as simple as those in traditional DL libraries such as TensorFlow. We have implemented DeepWear on the Android OS and evaluated it on COTS smartphones and smartwatches with real DL models. DeepWear brings up to **5.08X** and **23.0X** execution speedup, as well as **53.5%** and **85.5%** energy saving compared to wearable-only and handheld-only strategies, respectively.

Index Terms—Wearables; Deep Learning; Offloading

1 Introduction

Making deep learning (DL for short in the rest of this paper) tasks run on mobile devices has raised huge interests in both the academia [1], [2], [3], [4], [5], [6], [7], [8] and the industry [9], [10], [11]. In this paper, we focus on 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. According to a recent market research report, the estimated global market value of smartwatch is $10.2 billion in 2017, and is expected to witness an annual growth rate of 22.3% from 2018 to 2023 [12]. Second, DL on wearable devices enables new applications. 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 data creates countless applications for DL. Third, despite a plethora of work on DL on smartphones, so far very few studies focus specifically on the interplay between DL and the wearable ecosystem.

In practice, supporting DL on wearable devices is quite challenging, due to the heavy computation requirements of DL and constrained processing capacity on today's COTS (commercial off-the-shelf) wearable devices. Intuitively, running DL tasks locally is not a good option for most wearables. Then an instinct idea is to perform *offloading* [13], [14]. Instead of offloading computations to the remote cloud, we instantiate the idea of Edge Computing [15] by *offloading DL tasks to a nearby mobile device* (e.g., typically a smartphone or a tablet) that has local connectivity with the wearable. Such a "local" offloading is indeed feasible for three reasons. (1) As to be demonstrated in our study, today's handheld devices such as smartphones are sufficiently powerful with multi-core CPU, fast GPU, and GBs of memory. (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 recent user study [16] reports that a smartwatch is paired with a smartphone during 84% of the day time. (3) Prior efforts have been invested in reducing the computation overhead of DL tasks through various optimizations such as model compression [3], [17], [18], [19]. In our work, we strategically integrate and instantiate some of their concepts into our practical system to make DL tasks wearable-friendly.

We envision that such a local (edge) offloading approach has three key advantages. First, offloading to a handheld does not require the not-always-reliable Internet connectivity that can lead to high energy and monetary cost (e.g., over cellular networks). 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, users routinely carry *both* wearables and the paired handheld devices, making offloading ubiquitously feasible. Third, offloading to paired handhelds minimizes risks of privacy leak because the potentially sensitive data (e.g., medical sensor data) generated from wearables is never leaked to the network.

Motivated by the preceding analysis, we design, implement, and evaluate DeepWear, a holistic DL framework that supports local offloading for wearable DL applications.

- *Mengwei Xu, Mengze Zhu, Feifan Huang, and Xuanzhe Liu are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871. Email: {xumengwei, zhumengze, huangfeifan, liuxuanzhe}@pku.edu.cn*
- *Feng Qian is with the Computer Science and Engineering Department at University of Minnesota – Twin Cities, 200 Union Street SE, Minneapolis MN 55455. Email: fengqian@umn.edu*
- *Saumay Pushp is with Korea Advanced Institute of Science and Technology, 291 Daehak-ro, Eoeun-dong, Yuseong-gu, Daejeon. Email: saumay@nclab.kaist.ac.kr*

DeepWear has several salient features as described below.

• **Context-aware offloading scheduling**. We make a first in-depth measurement study to demystify the performance of wearable-side DL tasks and reveal the potential improvements that can be gained through offloading. Making an appropriate offloading decision involves scrutinizing a wide range of factors including the DL model structure, the application's latency requirement, and the network connectivity condition, etc. In addition, our offloading target (the handheld) introduces additional complexities: despite being more powerful than a wearable, a handheld still has limited processing power (compared to the cloud) and battery life; as a personal computing device, a handheld also runs other apps that consume system resources by incurring bursty workload. Therefore, DeepWear further takes into account the status of handheld. We incorporate the preceding considerations into a *lightweight online scheduling algorithm* that judiciously determines which, when, and how to offload.

• **Partial offloading**. Instead of making a binary decision of offloading the whole DL model versus executing the entire model locally, DeepWear supports the *partial offloading*. Specifically, DeepWear splits a model into two submodels that are separately executed first on the wearable and then on the handheld. We found that in some scenarios, partial offloading outperforms the binary decision, because an internal layer inside the model may yield a smaller intermediate output compared to the original input size, thus reducing the data transfer delay. To support the partial offloading, we develop a heuristic-based algorithm that efficiently identifies a set of candidate partition points whose exhaustive search takes exponential time. The optimal splitting point can then be quickly determined by examining the small candidate set. Our partial offloading approach can work with any DL model with arbitrary topology.

• **Optimized data streaming**. DeepWear introduces the additional optimization for streaming input such as video frames and audio snippets continuously fed into the same model. Specifically, DeepWear employs *pipelined processing* on wearable and handheld, which helps fully utilize the computation resources on both devices and thus effectively improves the overall throughput.

• **Application transparency and good usability**. We propose a modular design of the DeepWear system whose most logic is transparent to the application. Developers can use the same APIs as those of traditional DL libraries (e.g., TensorFlow [20]) to perform DL inference. In addition, DeepWear provides simple interfaces for developers or users to flexibly specify policies such as the latency requirement and energy consumption preferences. Overall, DeepWear is readily deployable to provide immediate benefits for wearable applications.

We have implemented the DeepWear prototype on Android OS (for handheld) and Android Wear OS (for wearable). We evaluated our prototype on COTS smartphones and smartwatches using the state-of-the-art DL models. DeepWear can effectively identify the optimal partition for offloading under various combinations of device hardware, system configurations, and usage contexts, with the accuracy being up to **97.9%**. DeepWear brings on average **1.95X** and **2.62X** (up to **5.08X** and **23.0X**) DL inference speedup compared to the handheld-only and wearable-only execution strategies, respectively. In addition, it brings on average **18.0%** and **32.7%** (up to **53.5%** and **85.5%**) energy saving compared to the two strategies respectively. Meanwhile, DeepWear can adapt its offloading strategies to diverse contexts such as the battery level on either wearable or handheld, the Bluetooth bandwidth, the handheld processor load level, and the user-specified latency requirement. In addition, our pipelining technique helps improve the processing throughput by up to **84%** for streaming data compared to the non-pipelining approach. Finally, DeepWear incurs negligible runtime and energy overhead.

It should be noted that, there have been various code offloading efforts, including MAUI [13], CloneCloud [21], COMET [22], DPartner [23], and so on. These systems focus on optimizing the general-purpose computation-intensive tasks instead of deep learning applications, and the offloading decisions are often manually defined at the design time (e.g., profiling [21] or manually labeled [23]). However, DeepWear intuitively differs from these systems as it relies on the domain knowledge of deep learning models, i.e., the *data topology* between layers of a DL model rather than the code-level characteristics, and the offloading decision is dynamically made at runtime rather than manually predefined. Additionally, compared to recent efforts on mobile DL offloading such as [24], our work specifically focuses on wearable devices, with additional effective mechanisms such as streamed data processing. To summarize, we make the following major technical contributions in this paper.

- We conduct to the best of our knowledge the most comprehensive characterization of wearable DL offloading, by applying 8 representatively popular DL models on COTS wearables/smartphones under various settings and quantifying several key tradeoffs. We demonstrate that whether and how much users can benefit from the wearable-to-handheld offloading depends on multiple factors such as hardware specifications, model structures, etc. We reveal that in some cases partitioning the DL models into two parts and running them separately on the wearable and the handheld would have better performance and quality of user experience.
- We design and implement DeepWear, a DL framework for wearable devices. It intelligently, transparently, and adaptively offloads DL tasks from a wearable to a paired handheld. With the help from local offloading, DeepWear better preserves users' privacy and thus realizes a more ubiquitous offloading without requiring the Internet connectivity. DeepWear introduces various innovative and effective techniques such as context-aware offloading, strategic model partition, and pipelining support, to better utilize the processing capacity from nearby handhelds while judiciously managing both the devices' resource and energy utilization.
- We comprehensively evaluate the DeepWear approach over COTS wearable and handheld devices. The results demonstrate that DeepWear can accurately identify the optimal partition strategy, and strike a much better tradeoff among the end-to-end latency and the energy consumption on both

the handheld and the wearable, compared to the wearable-only and the handheld-only strategies.

The remainder of the paper is organized as follows. We survey the related work in Section 2. We present our measurements about wearables DL in Section 3. We describe the design and implementation of DeepWear in Section 4 and Section 5, respectively. We comprehensively evaluate DeepWear in Section 6. We discuss the limitations and possible future work in Section 7 and conclude the paper in Section 8.

2 Related Work

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

2.1 Ubiquitous Deep Learning

In the past few years, DL is the state-of-the-art AI technique that has been widely applied in numerous domains, such as computer vision, pattern recognition, natural language processing, and so on [25], [26], [27]. A DL model is essentially a directed graph where each node represents a processing unit that applies certain operations to its input and generates output. Accordingly, developers need to first construct a specific model graph, and then use data to train the model (known as the training stage). Once trained, the model can be applied for prediction (known as the inference stage).

Tremendous efforts have been made towards reducing the computation overhead of DL tasks, making it feasible on resource-constrained devices such as smartphones. For example, some recent efforts [6], [28], [29], [30] have proposed lightweight DL models that can run directly on low-end mobile processors. Some other efforts such as [31], [32], [33], [34] aimed at building customized hardware accelerators for DL or other machine learning tasks. Besides, various model compression techniques [35], [3], [17], [18], [19], [36] have been proposed for accelerating the DL task and reducing its energy consumption.

In contrast, DeepWear specifically focuses on wearable devices that have specific features and application contexts compared to smartphones. DeepWear proposes novel techniques such as strategic model partition and pipelining to efficiently utilize the processing capacity from a nearby handheld.

2.2 Offloading

Many prior efforts, such as MAUI [13], CloneCloud [21] COMET [22], and DPartner [23], have already studied the offloading problem from mobile devices to the remote server or cloud. In addition, DeepWear also learns lessons from the recent work on "edge cloud" or "cloudlet" offloading [37], [38], [39]. All these frameworks are control-centric, as they make decisions at the level of code or function. For example, COMET [22] offloads a thread when the execution time exceeds a pre-defined threshold, ignoring any other information, e.g., considerable data volume (of distributed shared memory) to transfer, wireless network available, etc. CloneCloud [21] makes similar offloading decisions for all invocations of the same function. MAUI [13] designs an enhanced offloading decision mechanism that makes predictions for every single function invocation separately and considers the entire application when choosing which function to offload. DPartner [23] requires offline profiling to identify the computation-intensive functions and the programmers' manual efforts to annotate whether these functions are "*offloadable*".

However, these *general-purpose* offloading efforts are not sufficiently adequate to the partition decisions of DL model, which essentially depends on the data topology. As a result, layers of a given type within the DL model can have significantly different computational and data characteristics [24], and can vary a lot even when executing the same code or functions. In contrast, DeepWear differs from these approaches in that its offloading leverages the domain knowledge of DL to make the partition decision. Although DeepWear still requires the offline profiling, it does not introduce any additional manual efforts (such as annotation) to programmers, and the partition is performed dynamically based on the runtime DL topology. Another important difference of DeepWear is the careful considerations of practical deployability. That is, existing function-level code offloading approaches are too complex and heavyweight, e.g., the complicated program state synchronization for DSM in COMET [22], and the high-volume data transfer of DPartner [23]. Such overhead can deter the deployability on wearables. DeepWear designs a very lightweight framework that is easy to be deployed on wearables. Additionally, DeepWear achieves the satisfactory performance and accuracy when making offloading decisions, which has not been well studied in existing offloading solutions.

2.3 DL Model Partitioning

The recently proposed DeepX [3] also partitions DL models for low-power DL inference. However, DeepX only distributes partitioned submodels onto different *local* processors while DeepWear performs collaborative DL inference on two devices and thus needs to take into consideration the data transfer overhead and many other external factors that play important roles in making offloading decisions. Also, DeepX targets at only linear DNN models, while DeepWear can handle complex DL models with non-linear structures. Some other work [24], [40] also split DL computation between client devices and remote clouds. DeepWear instead focuses on the collaboration between wearables and their paired handheld devices in order to preserve the privacy and realize ubiquitous DL without requiring Internet connectivity. Several unique challenges thus stem from the architecture we have chosen, such as balancing the resource consumption on both mobile devices. Furthermore, DeepWear introduces optimizations for streaming data processing, a missing feature in prior work.

3 A Measurement Study of Wearable DL

In this section, we begin with some empirical studies to demystify the performance and limitations of running DL tasks on wearables.

In this work, we study 8 state-of-the-art DL models that have been widely adopted in various applications, as shown in Table 1. For the *LSTM-HAR* model [43], we use a popular

Model	App	Input	FLOPs
MNIST [41]	digit recognition	grayscale image	15M
MobileNet [30]	image classification	rgb image	580M
GoogLeNet [42]	image classification	rgb image	2G
LSTM-HAR [43]	activity recognition	mobile sensor	180M
DeepSense [44]	activity recognition	mobile sensor	550M
TextRNN [45]	document classification	word vectors	11M
DeepEar [6]	emotion recognition	raw sound	9M
WaveNet [46]	speech recognition	mfcc features	3.8G

TABLE 1: 8 deep learning models used in this work.

Device	CPU	Memory	GPU
Nexus 6	Quad-core Krait 450	3 GB RAM	Adreno 420
LG Urbane	Quad-core Cortex-A7	512MB RAM	Adreno 305
Galaxy S2	Dual-core Cortex-A9	1GB RAM	Mali-400MP4

TABLE 2: Hardware specifications for wearables and smartphones used in this work.

configuration as 2-layer stacked, 1024 hidden state size to carry out our experiments. For other models, we use the default configurations as described in the original literature or open-sourced repositories. These models range from natural language processing, audio processing, to computer vision tasks and mobile sensor intelligence, all of which are well suited to ubiquitous and wearable scenarios. The rightmost column in Table 1 also lists the number of FLOPs (floating point operations) for conducting a single inference for each model. It is worth mentioning that DL 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 problems such as machine translation [47], question answering [48], and handwriting generation [49]. In particular, DeepWear does not assume any specific DL model structure, and can work with all of them.

We envision that DL will become an essential part in the wearable ecosystem due to wearables' unique sensing capabilities. However, running computation-intensive DL tasks on wearable devices is quite challenging due to wearables' relatively limited processing capabilities. A possible approach is thus to offload the workload from a wearable to its paired handheld. We choose the handheld over the cloud because offloading to the handheld does not require the Internet connectivity that can incur high energy and monetary cost. Doing so also minimizes risks of privacy breach because the potentially sensitive data is never leaked to the Internet. Note that there are quite a lot prior work [50], [51], [52] targeting at wearable offloading for better performance (see Section 2). However, none of them studies DL tasks, thus leaving an important question unanswered: *whether and how much offloading to a handheld can benefit DL on wearables?* To answer this question, 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 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 more promising 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 (real) wearable device. We also use an old phone, Galaxy S2 released in 2011, to emulate head-mount devices such as Vuzix M1000 [53] that shares hardware similar to Galaxy S2. Table 2 elaborates the hardware specifications of these three devices used in this study. We use TensorFlow [20] and an open-source library RSTensorFlow [54] to support running DL tasks on mobile CPU and GPU. We use Bluetooth for the data transfer between wearable and handheld due to Bluetooth's wide availability on wearable and its energy efficiency. [1] For energy measurement, we build the power model for the smartphone by using the Monsoon Power Meter [56] (following a high-level approach of component-based power modeling [57]), or obtain the model from the literature [16] for smartwatch. All experiments are carried out by fixing the distance between the wearable and handheld (0.5m) unless otherwise stated.

Factors of determining offloading decisions. Figure 1 and 2 indicate the latency breakdown of four popular DL 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). The percentage indicates the proportion of computation time (as opposed to network delay) within the overall latency. 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 the* GoogLeNet *model, the end-to-end latency is often not reduced due to the high data transfer latency over Bluetooth*. The results indicate that making a judicious offloading decision can have significant impacts on the user experience. For example, running the *DeepEar* model locally on LG Urbane can reduce up to 74% of latency compared to running it on handheld CPU, while for the *DeepSense* model, running it locally leads to more delay compared to offloading to a handheld.

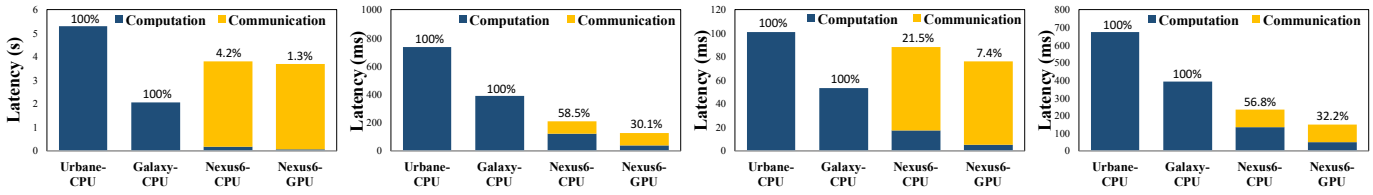
Overall, the optimal decision depends on various factors described below.

(1) **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 latency for the *GoogLeNet* model, LG Urbane should offload the task to Nexus 6 while Galaxy S2 does not need to do so according to Figure 1(a).

(2) **Model structure.** Different DL models can vary a lot in terms of computational overhead and input size. Models with high a computational overhead and a small input size such as *DeepSense* and *WaveNet* are more adequate for being offloaded to handhelds, while other models may not benefit from offloading such as *DeepEar*.

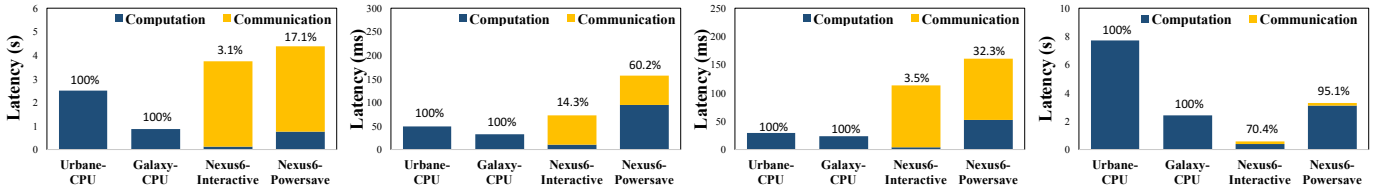
(3) **Processor status.** In real-world application scenarios, handheld CPUs often run under different governors adapting to different device environments, e.g., switching from the default *interactive* governor (high frequency) to the *powersave* governor (low frequency) when the screen is turned off or the battery level is low. Observed from Figure 2, CPU

1. Also Google has recommended it as the proper way of performing data communication on wearable devices [55].



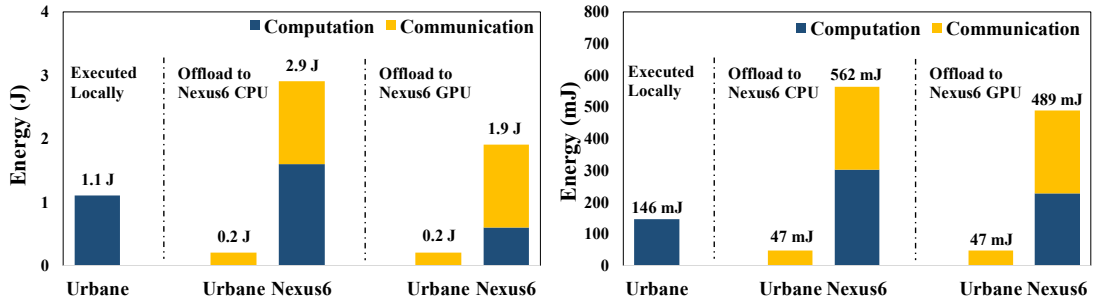
(a) *GoogLeNet* model. (b) *LSTM-HAR* model. (c) *MNIST* model. (d) *DeepSense* model.

Fig. 1: End-to-end latency breakdown for different models and offloading scenarios. 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.



(a) *MobileNet* model. (b) *TextRNN* model. (c) *DeepEar* model. (d) *WaveNet* model.

Fig. 2: End-to-end latency breakdown under different handheld CPU governor. 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.



(a) *GoogLeNet* model. (b) *LSTM-HAR* model.

Fig. 3: Energy breakdown on both wearable and handheld devices for different running strategy. Offloading to the handheld can sometimes consume more energy than wearable execution due to the high energy overhead consumed by Bluetooth module.

status can have substantial impacts on the latency as well as the offloading strategy. Take *WaveNet* as an example. It takes almost 7X more time under the *powersave* governor than the *interactive* governor, with the former rendering offloading no longer beneficial. While enforcing the handheld to switch to a high-power governor is sometimes possible, there are other scenarios where the handheld CPU/GPU is inherently overloaded (e.g., by other computationally intensive apps that are running concurrently).

(4) **Latency vs. energy preference.** Besides the end-to-end latency, the energy consumption is another key metric to consider as wearable devices have small battery capacities [16]. As shown in Figure 3, although offloading can help save wearable battery, it will also cause the non-trivial energy consumption for the handheld (around 2.9 J for Nexus 6 CPU for *GoogLeNet*).

Overall, the above results indicate the challenge of *balancing the tradeoff among three factors when making judicious offloading decisions: end-to-end latency, energy consumption of the wearable, and energy consumption of the handheld*. In real-world scenarios, a static policy may not always satisfy users' requirements. For instance, when a user's handheld (wearable) is low on battery, DeepWear needs to focus on saving the energy for the handheld (wearable). Therefore it is necessarily beneficial to adjust the offloading decisions dynamically based on external factors such as battery life, network condition, and CPU/GPU workload.

Partial offloading. The preceding pilot experiments consider only 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 DL model into two sub-models and executing them separately

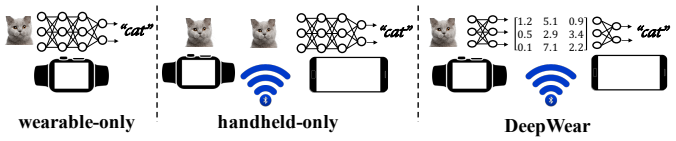


Fig. 4: Different wearable DL execution approaches. wearable-only (offload nothing), handheld-only (offload everything), and DeepWear (partial offloading). Offloading nothing means executing all DL task on wearable. Offloading everything means offloading all DL task to handheld. Partial offloading, which is adopted in DeepWear, means partitioning computation among wearable and handheld.

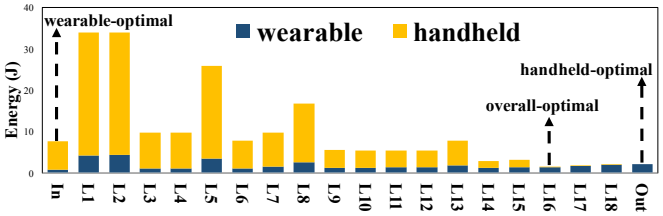


Fig. 5: Energy consumption of running *GoogLeNet* on LG Urbane and Nexus 6 with different partition points. We only select 20 partition points to present the figure. X-axis presents the layers that we select as partition point, after which output data is sent to handheld for continuous processing. The left-most bar represents handheld-only processing and the right-most bar represents wearable-only processing.

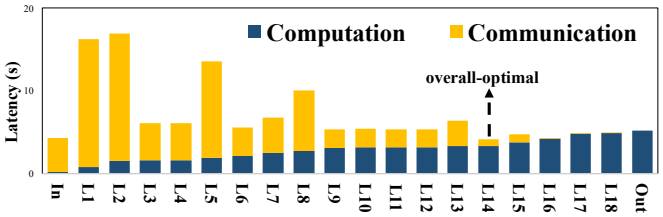


Fig. 6: End-to-end latency of running *GoogLeNet* on LG Urbane and Nexus 6 with different partition points. The experimental setup is the same as that in Figure 5.

on the wearable and the handheld as shown in Figure 4, can sometimes achieve even better results.

We confirm the benefit of partial offloading through controlled experiments. Figure 5 plots the energy consumption with different partition points for the *GoogLeNet* model. The X-axis presents the layer that we select as partition point, after which the output data is sent to handheld for further processing. The left-most and right-most bars correspond to handheld-only and wearable-only processing, respectively. Note that the energy consumption of the handheld in Figure 5 (and all energy results thereafter) is calibrated as $E = original\_E/Handheld\_capacity{*}Wearable\_capacity$. $original\_E$ is the absolute energy consumed by the handheld; $Handheld\_capacity$ and $Wearable\_capacity$ are the battery capacity of the handheld (3220 mAh for Nexus 6) and the wearable (410 mAh for LG Urbane), respectively. Since the phone and watch have different battery capacities, the above adjustment essentially calibrates the phone and wearable's energy consumption with respect to their heterogeneous actual battery capacities.

As shown in Figure 5, 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 amount of energy for the wearable. However, users often care about the battery life of both devices, therefore we need to find an optimal partition to achieve the least total energy consumption. In this case, the overall optimal partition point resides in an internal layer (L16). Doing such a partial offloading helps save around 84% and 29% of energy compared to the wearable-only and handheld-only strategies, respectively.

Using the same setup as that in Figure 5, Figure 6 plots the end-to-end latency with different partition points for the *GoogLeNet* model. As shown, performing partial offloading may also help minimize the overall latency (L14 in Figure 6). This is because an internal layer may yield a small intermediate output compared to the original input size, thus reducing the network transmission delay. Therefore, a key design decision we make for DeepWear is to support partial offloading.

4 The DeepWear Design

Our measurements in Section 3 indicate that it is challenging to develop an offloading framework for wearables with various factors being considered. 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 DeepWear, which can help applications optimally determine *whether or not, how, and what to offload*. We now describe the design details of DeepWear whose design goals include the following.

- **Latency-aware.** Different DL apps have diverse latency requirements, ranging from dozens of milliseconds (augmented reality) to several minutes (activity tracking). As a result, DeepWear 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.** DeepWear should not require developers' additional efforts to retrain the deep learning models. This is important as most app developers today utilize only off-the-shelf models in an "as-it-is" style.
- **No accuracy loss.** DeepWear should not compromise the accuracy when running DL models under diverse settings. In other words, DeepWear should maintain consistently adequate accuracy results regardless of the offloading decision.
- **Trade-off flexible.** DeepWear 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.
- **Developer-friendly.** DeepWear 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, DeepWear should abstract wearable and handheld devices as

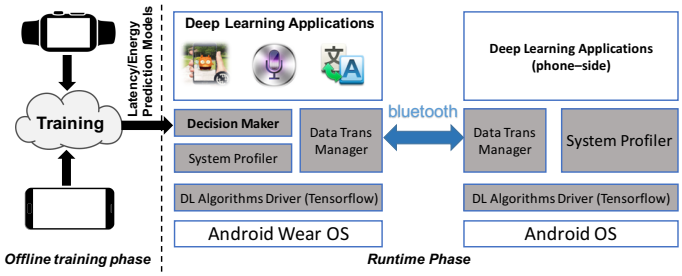


Fig. 7: Overview of DeepWear. Grey parts constitute a library provided for deep learning application developers.

one entity by shielding low-level details such as offloading decisions. For example, developers should be freed from programming on the data transferring via the Android APIs, which is extremely tedious and error-prone.

4.1 Architecture Overview

The overall architecture of DeepWear is shown in Figure 7. To use DeepWear, there are two main steps involved.

(1) **The offline training phase** involves a one-time effort of constructing the latency and energy prediction models[2], i.e., given a DL model structure, what is the end-to-end latency and energy consumption to run this model on a given device (see Section 4.2). It should be mentioned that, the latency and energy prediction models are device-specific and heavily depend on the underlying hardware architecture. However, building such a model is a one-time effort and does not incur much engineering overhead based on our experience. In practice, the device or DL software vendors can perform such profiling and let developers download the model.

(2) **The runtime phase**, where DL applications rely on DeepWear to perform adaptive offloading for DL tasks. There are following major components.

- **Decision Maker** is the core part of DeepWear. Given a DL model to run, it identifies the optimal model partition point based on the latency/energy prediction models and both devices' running status. The decision dictates which part of the model should be executed locally and which part should be offloaded to the paired handheld, including two special cases of offloading none or the entire task. A key logic of the Decision Maker is to balance the tradeoff between the latency and the energy consumption (see Section 4.3).
- **System Profiler** periodically profiles the system status such as the pairing state, processor status, and the Bluetooth bandwidth, which will be used by the Decision Maker to balance key tradeoffs.
- **DL Algorithms Driver** is the library that implements the DL algorithms. Currently, DeepWear directly employs TensorFlow [20] as the driver.
- **Data Transmission Manager** deals with the data transmission between the wearable and its paired handheld. It is realized using the standard Data Layer API [55] in Android.
- **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 4.5.

2. The latency and energy prediction models should be distinguished from the DL models themselves.

4.2 Deriving Prediction Models

Now we consider how to construct the prediction model of the latency and energy for a given (partial) DL model. A straightforward way is modeling each layer individually and then combining the prediction models across all layers into the final prediction model. To demonstrate the feasibility of this approach, we carried out controlled experiments via running DL models and logging the latency/energy in total as well as for each layer. [3] Through this controlled experiment, we find 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. In fact, summing up the latency/energy across all layers yields no more than 1.82% of deviation compared to the direct measurement, for the eight models shown in Table 1.

Nevertheless, we still need to deal with a practical challenge: there exist 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 incur 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: convolutional (conv), fully-connected (fc), pooling, and activation layers. As shown in Table 3, these four layer types constitute up to more than 90% of the inference latency of popular DL models. Although current DeepWear considers only these layers, other layer types can be easily incorporated into our framework. It is quite important to note that for RNN models, a recurrent layer is composed of fully-connected layer and activation layer. Therefore, by modeling the aforementioned layers, i.e., convolutional, fully-connected, pooling, and activation layers, we are able to accommodate the RNN model as well. 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 might be a large latency variation across different layer parameters (e.g., the kernel sizes of 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 4, 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 subtypes based on metrics such as the kernel size[4] and the activation function. We then apply a linear-regression model to each of those sub-types to get the final predicted results. As shown in Table 4, 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

3. Built-in TensorFlow functionality to log individual layer performance: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/stat_summarizer.h

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

Model	Conv	Fc	Pooling	Activation	Total
MNIST	39.0%	54.2%	1.1%	3.1%	97.4%
MobileNet	45.4%	N.A.	N.A.	51.1%	96.5%
GoogLeNet	80.2%	0.1%	7.5%	8.1%	95.7%
LSTM-HAR	N.A.	8.4%	N.A.	87.8%	96.2%
DeepSense	51.6%	21.1%	N.A.	25.3%	98.0%
TextRNN	N.A.	16.0%	N.A.	79.1%	95.1%
DeepEar	N.A.	92.6%	N.A.	7.2%	99.8%
WaveNet	82.6%	N.A.	N.A.	11.6%	94.1%

TABLE 3: The major latency composition.

Layer Type	Prediction Model	Latency Acc.	Energy Acc.
Conv	decision tree input: *filter_size*, linear regression input: *batch* · *input_width* · *input_height* · *channel* · *kernel_number* ÷ $stride^2$	0.993	0.973
Pooling	decision tree input: *filter_size*, linear regression input: *batch* · *input_width* · *input_height* · *channel* · *kernel_number* ÷ $stride^2$	0.784	0.772
Fully-connected	linear regression input: *a_width* · *a_height* · *b_width*, *a_width* · *a_height*, *b_width* · *b_height*	0.945	0.922
Activation	decision tree input: *activation function type*, linear regression input: *input_size*	0.998	0.970

TABLE 4: Our latency & energy prediction models for different kinds of DL 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).

of Determination (R^2) [58] to measure the accuracy. R^2 is a commonly used metric for evaluating regression models. It assesses how well a model predicts future outcomes. R^2 is calculated as $1 - \frac{SSE}{SST}$ where SSE and SST are the sum of squared errors of the regression model and the sum of squared errors of the baseline model (always using the mean as the prediction), respectively.

Energy Prediction. We use a similar approach to predicting the energy consumption of a layer. In our study, we typically build power models for the smartphone by using the Monsoon Power Meter [56] (following a high-level approach of component-based power modeling [57]) or obtain them from the literature [16] for smartwatch. All experiments are done with device screen off, and the energy data we used is subtracted by the baseline power in the idle state.

As shown in Table 4, our energy prediction model also has a satisfactory accuracy ($>$ 92%) for 3 out of the 4 layer types. The Pooling layer has a lower accuracy (0.772). Nevertheless, as shown in Table 4 this layer contributes little to the overall latency and energy compared to other layers.

4.3 Making Offloading Decision

Utilizing the prediction models described above, DeepWear dynamically selects the optimal partition point. The decision making procedure involves two steps: finding a set of possible partitions for a given graph, and identifying the optimal one among them.

Dynamic Partition. A DL 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 valid partition equals to a *cut* [59] of the DAG and

Input: G: pre-trained graph to be executed
$p(G)$: binary-partition function, returns a list of partitions $\langle G_w, G_h, dt \rangle$, where *dt* is the size of data to be transferred
f(G, $\mathcal{S}$), *g*(G, $\mathcal{S}$): pre-trained models for predicting the latency/energy of executing G under device status s
$\mathcal{S}_w$, $\mathcal{S}_h$: current device running status for wearable and handheld, including CPU frequency, CPU loads, etc
$\mathcal{B}$: current Bluetooth uplink bandwidth
PR, *PT*: rx/tx power consumption over Bluetooth
PropT: proper latency that the app is supposed to run on
$\mathcal{W}_w$, $\mathcal{W}_p$: weights of battery for wearable and handheld
Output: Optimal partition choice

$partitions \leftarrow p(G), L = E = \emptyset$
foreach $\langle G_w, G_h, dt \rangle \in partitions$ **do**
  if *streaming_opt_on* **then**
    $l \leftarrow \max(f(G_w, \mathcal{S}_w), f(G_h, \mathcal{S}_p) + dt/\mathcal{B})$
  else
    $l \leftarrow f(G_w, \mathcal{S}_w) + f(G_h, \mathcal{S}_p) + dt/\mathcal{B}$
  $E_w \leftarrow g(G_w, \mathcal{S}_w) + dt * PT$
  $E_p \leftarrow g(G_h, \mathcal{S}_p) + dt * PR$
  $L.append(l), E.append(\mathcal{W}_w * E_w + \mathcal{W}_p * E_p)$
end
if *PropT* == *0 or* $\min(L) > PropT$ **then**
  $opt\_index \leftarrow \arg\min_{i \in \{1 \ldots N\}} (L[i])$
else if *PropT* == $+\infty$ **then**
  $opt\_index \leftarrow \arg\min_{i \in \{1 \ldots N\}} (E[i])$
else
  $\mathcal{R} \leftarrow$ list of index i that satisfies L[i] $\leqslant PropT$
  $opt\_index \leftarrow \arg\min_{i \in \mathcal{R}} (E[i])$
return $partitions[opt\_index]$;

Algorithm 1: DeepWear Partition Algorithm.

requires the source and the sink to be placed in different subsets. Finding all *cuts* of a given graph shall need the $\mathcal{O}(2^n)$ complexity where n is the number of nodes. For a large DL model, e.g., the *GoogLeNet* model with 1,096 nodes, such a complexity is prohibitive. As pointed out previously by Kang *et al.* [24], existing DL-partition approaches simply assume these graphs are linear. Hence, each single edge represents a valid partition point. However, we observe that such an assumption is not always true for many popular DL models (e.g., *GoogLeNet*), as there can be branches and intersections in the graph. This 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.

Figure 8 illustrates how our algorithm works. First, DeepWear prunes all computationally light nodes, only keeping the computationally heavy nodes such as those shown in Table 4. After identifying these light nodes, DeepWear removes them and connects their input nodes and output nodes. Second, we observe that a DL (e.g., CNN and RNN) model often has repeated subgraph structures, which we call "frequent subgraphs", that frequently appear in the DAG. DeepWear thus bundles each frequent subgraph into one virtual node without further splitting. To mine the frequent subgraphs, i.e., detecting the frequent patterns in a model and the nodes associated with those patterns, the most straightforward way is to utilize the node *namespace*: nodes under the same namespace are often in the same subgraph. However, the idea of *namespace* is not supported in all DL frameworks; more importantly, setting the names-

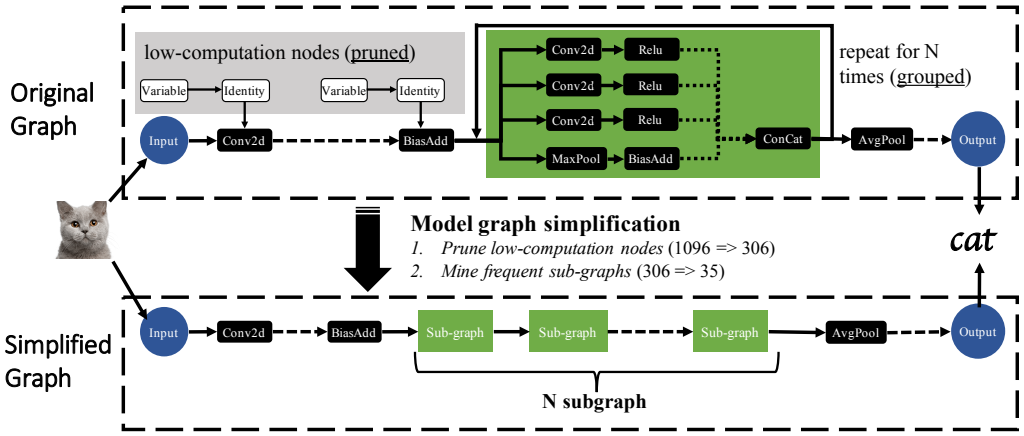


Fig. 8: Example of how DeepWear simplifies *GoogLeNet*. Each node presents a layer, while each edge presents the data flow among those layers. Dash lines indicate many more nodes are hidden to save space. DeepWear first prunes the model graph by keeping only the computation-intensive nodes (as listed in Table 3), and then grouping the repeated subgraphs together. After these two steps, a complex directed acyclic graph often becomes a linear and much simpler graph.

paces is rather subjective and optional, and requires developers' additional support. We utilize GRAMI [60], a fast and versatile algorithm that automatically mines frequent subgraphs. After the graph is simplified, there will be much fewer nodes (e.g., 1,096 to 35 for *GoogLeNet*). Additionally, the graph exhibits a mostly linear structure. This allows us to apply a brute-force approach to identifying all *cuts*. In addition, this simplification results can be cached for every single DL model and reused. We empirically observed that our heuristic-based partition identification approach is effective and robust.

Optimal Partition Selection. The algorithm for determining an optimal partition is demonstrated in Algorithm 1. Taking as input possible partitions generated in the previous step, DeepWear analyzes the partitioned subgraphs on the wearable and the handheld, and uses the prediction models (Section 4.2) to estimate the corresponding latency and energy consumption (line 2∼10). Note that the overall energy consumption metric is a weighted mean from the energy consumed on both the wearable and handheld. Our algorithm provides a general framework for diverse usage scenarios. If the DL app integrated with DeepWear is latency-sensitive specified by developers, we select the partition with the smallest latency (line 11∼12). In contrast, if the app is latency-insensitive, then we select the partition with the lowest energy consumption (line 13∼14). In a more general case, the developer is able to quantitatively specify the latency requirement. We then select the most energy-efficient partition satisfying this requirement (line 15∼17).

The models and parameters in Algorithm 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 4.1), including the handheld status $(\mathcal{S})$ and Bluetooth bandwidth *(B)*, (3) application-specified parameters, including the expected end-to-end latency of DL inference *(PropT)*, (4) configurable trade-off parameters, including energy consumption weights on wearable and handheld $(\mathcal{W}_w, \mathcal{W}_p)$.

4.4 Optimizing Streaming Data Processing

DL 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 continuously fed into the same model. Here we use "frame" to denote an input unit for a DL model, e.g., an image or an audio snippet. Compared to non-streaming data, streaming data cares more about the overall throughput, i.e., how many frames can be processed per time unit, rather than the latency for every single frame. For the 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 the 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, however, DeepWear employs *pipelined processing* on wearable and handheld. Specifically, when the n-th frame finishes 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 helps fully utilize the computation resources on both devices and thus effectively improves the overall throughput. To 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.

4.5 Provided Developer APIs

DeepWear exposes a set of easy-to-use APIs for developers for running the 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 DeepWear provides is a hint function for specifying the latency requirement (Line 3 in List 1), which helps DeepWear make offloading decisions.

Listing 1: A code sample of using DeepWear

```
1 DeepWearInference infer =
2   new
      DeepWearInference("/path/to/model");
3 infer.set_expected_latency(100); // 100ms
4 infer.feed(input_name, input_data);
5 infer.run();
6 float[] result =
    infer.fetch(output_name);
```

As exemplified in the code snippet 1, using the APIs provided by DeepWear is quite similar to using the standard Java APIs [61] provided by TensorFlow. It typically consists of four steps: loading pre-trained model, feeding the input, executing the graph, and finally fetching the output. Unlike traditional general-purpose offloading frameworks, DeepWear doesn't require any manual annotation to specify what can be offloaded. In contrast, DeepWear hides the offloading details from the perspective of developers.

5 Implementation of DeepWear

We have implemented DeepWear on commodity smartphone and smartwatches running off-the-shelf Android and Android Wear OS respectively. Our prototyping efforts consist of around 3,200 lines of code written in Java, excluding the scripts for constructing and analyzing prediction models. Developers can easily integrate DeepWear into their apps by importing the DeepWear library on both the wearable side and the handheld side. In the one-time initialization phase when the app is being installed, DeepWear will also locate other necessary components such as the DL models (stored at both the wearable and the handheld) and the latency/energy prediction models (stored at the wearable). The handheld-side library also provides a console allowing users to configure offloading policies as described in Section 4.3).

Currently, DeepWear employs the popular TensorFlow [20] as our DL algorithms driver (Figure 7). Other popular frameworks such as Caffe2 [62] and PyTorch [63] can also be easily integrated into DeepWear with very small adaptation. To realize the System Profiler, DeepWear obtains the processor status via the *sysfs* interface. 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 Android Wearable Data Layer API [55]. Specifically, we use the *Message API* [64] 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 profiling is performed either passively (by measuring the offloaded data transfer) or actively (by sending lightweight probing packets). The active probing is triggered periodically (every 1 minute by default) as well as by Bluetooth signal strength changes, in the absence of offloading transfers. We are currently working on adding Direct WiFi support for offloading.

6 Evaluation

We now comprehensively evaluate DeepWear using the aforementioned 8 popular DL models under different device configurations. The experimental setup is the same as that used in Section 3. Each experiment is repeated for 20 times to make the results statistically meaningful.

6.1 Partition Selection Accuracy

Table 5 shows the partition points selected by DeepWear under different devices and DL models. Each cell represents the DL layer name at which DeepWear performs the partition, indicating that the output data of this layer shall be offloaded to the handheld. The red block indicates that DeepWear fails to make the optimal partition choice. 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 in our case. We obtain the optimal partition choice by exhaustively testing each possible partition point. In summary, DeepWear is able to select the best partition point for 47 out of 48 cases we tested (97.9%). The mis-predictions occur because of two reasons. First, our prediction models used in DeepWear consider only a subset of layer types as explained in Section 4.2. Second, those prediction models themselves cannot perfectly predict the delay or energy. Also note that for all 3 suboptimal partition points in Table 5, their delay and energy consumption are actually very close to those of the optimal partitions.

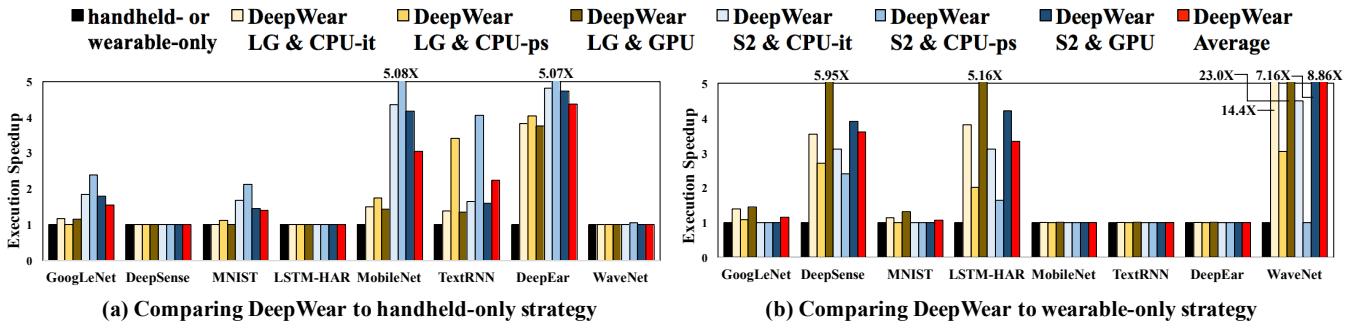
6.2 Latency and Energy Improvements

To demonstrate how DeepWear can help improve the end-to-end latency and energy consumption, we test it under two extreme cases: optimizing for latency ($PropT = 0$) and optimizing for energy ($PropT = +\infty$). We present the results under 6 running scenarios about wearables (LG Urbane and Galaxy S2) and handhelds (CPU-interactive, CPU-powersave, and GPU). We compare the performance of DeepWear with two baseline strategies: handheld-only (offloading all tasks to the handheld) and wearable-only (executing the entire model on the wearable without performing offloading).

Speedup. Figure 9 shows DeepWear's execution speedup (normalized) over the baseline strategies across 8 DL models and varied device specifications & status. Bars in different colors represent different hardware configurations. LG and S2 are abbreviated for Urbane LG and Galaxy S2. CPU-it, CPU-ps, and GPU refer to utilizing Nexus 6 under CPU-interactive, CPU-powersave, and GPU on the handheld side, respectively. The black bar represents the latency of handheld- or wearable-only approaches, and is used as a baseline to normalize other approaches (normalized to 1 itself). The red bar is the average speedup for each model.

Wearable	Handheld	Models							
		MNIST	*GoogLeNet*	*MobileNet*	*WaveNet*	*LSTM-HAR*	*DeepSense*	*TextRNN*	*DeepEar*
LG Urbane	**CPU-interactive**	input	input	Squeeze	input	input	input	BiasAdd	output
	CPU-powersave	add_3	AvgPool_0a	Squeeze	input	input	input	BiasAdd	output
	GPU	input	input	Squeeze	input	input	input	BiasAdd	output
Galaxy S2	**CPU-interactive**	add_3	AvgPool_0a	Squeeze	input	input	input	BiasAdd	output
	CPU-powersave	add_3	Squeeze	Squeeze	logit/out	input	input	BiasAdd	output
	GPU	add_3	Squeeze	Squeeze	input	input	input	BiasAdd	output

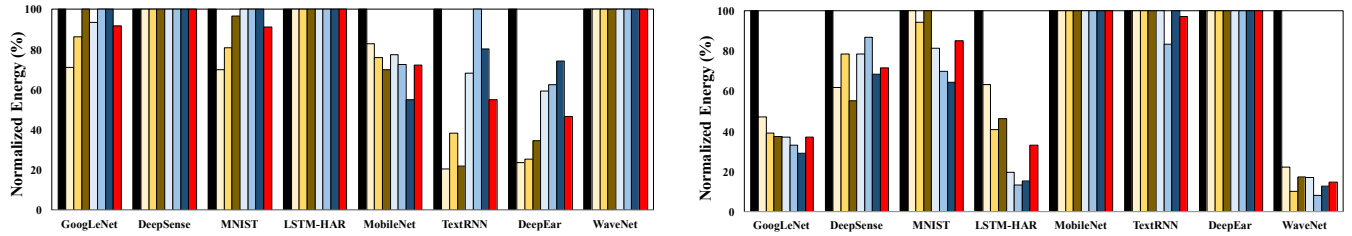
TABLE 5: DeepWear partition point selections under different devices and models ($PropT = 0$). Red blocks indicate DeepWear fails to make the optimal partition choice and white block means the optimal partition point is picked.



(a) Comparing DeepWear to handheld-only strategy

(b) Comparing DeepWear to wearable-only strategy

Fig. 9: Normalized execution speedup of DeepWear to two naive strategies: handheld-only and wearable-only. We present the results under 6 configurations for wearables (LG Urbane and Galaxy S2) and handhelds (CPU-interactive, CPU-powersave, and GPU). Note that numbers shown represent the relative speedup, with the handheld/wearable-only being the comparison baseline. We use configuration $PropT = 0$, so that DeepWear will chase for the smallest end-to-end latency. Results show that DeepWear can improve the latency by 1.95X and 2.62X on average compared to wearable-only and handheld-only, respectively. Additionally, the improvement can be up to 5.08X and 23.0X in some cases.



(a) Comparing DeepWear to handheld-only strategy

(b) Comparing DeepWear to wearable-only strategy

Fig. 10: Normalized energy consumption of DeepWear to two naive strategies: handheld-only and wearable-only. We present the results under 6 running scenarios about wearables (LG Urbane and Galaxy S2) and handhelds (CPU-interactive, CPU-powersave, and GPU). We use configuration $PropT = +\infty, \mathcal{W}_w = \mathcal{W}_p = 0.5$, so that DeepWear will chase for the smallest energy consumption. Results show that DeepWear can save energy by 18.0% and 32.7% on average compared to the handheld-only and the wearable-only, respectively. Additionally, the improvement can be up to 53.5% and 85.5% in some cases. Note that the handheld energy consumption is calibrated using the method used for Figure 5, in order to take into consideration the phone and wearable's heterogeneous battery capacities.

Figure 9(a) shows that compared to the handheld-only strategy, DeepWear can help reduce the latency for 6 out of 8 models, with an average improvement ranging from 1.01X (*WaveNet*) to 4.37X (*DeepEar*). Similarly, Figure 9(b) shows that compared to the wearable-only strategy, DeepWear reduces the latency of running 5 out of 8 models 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, DeepWear can even speed up the processing for more than 20 times (23.0X) compared to the wearable-only strategy. Overall, DeepWear can improve the latency by 2.62X and 1.95X on average compared to wearable-only and handheld-only, respectively, across all 8 models.

Our another observation is that different models can exhibit quite diverse results. We find that the execution speedup achieved by DeepWear depends on two factors related to the model structure: computation workloads and data size. A model graph with small computation workloads (*DeepEar*, *TextRNN*) or with a large input data size (image-processing applications such as *GoogLeNet* and *MobileNet*) are unlikely to benefit from the offloading since the performance bottleneck often resides in the data transmission rather than the local processing. Hence, in these cases, DeepWear can have significant improvements over the handheld-only approach, but less improvement over

the wearable-only approach. In contrast, when running DL models that require lots of computations on a relatively small size of data, DeepWear exhibits more improvements compared to the wearable-only approach rather than the handheld-only approach.

Energy saving. Similarly, Figure 10(a) shows that compared to handheld-only, DeepWear can help reduce the energy consumption for 5 out of 8 models, with an average improvement (the red bar) ranging from 8.3% (*GoogLeNet*) to 53.5% (*DeepEar*). Similarly, Figure 10(b) illustrates that compared to wearable-only, DeepWear lowers the energy consumption for 6 out of 8 models, with an average improvement ranging from 3.8% (*TextRNN*) to 85.5% (*WaveNet*). Overall, DeepWear can on average save the energy by 18.0% and 32.7% compared to the handheld-only and the wearable-only approach, respectively.

6.3 Local Offloading vs. Cloud Offloading

We also compare DeepWear's local offloading approach to offloading to the remote cloud. We use a server equipped with Tesla K80 GPU, 2.3GHz Intel Xeon CPU, and 60GB memory to play as the remote cloud. We carry out the experiments under two WiFi conditions: poor ($\sim$ 100kbps) and good ($\sim$ 5mbps). [5] The results are all normalized by the wearable-only performance (no offloading). Note that for cloud offloading, we ignore the cloud server's energy consumption.

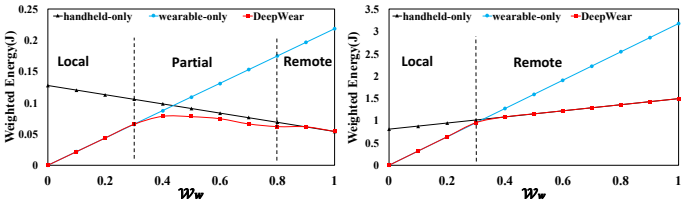
For latency improvements, as shown in Figure 14(a), cloud offloading outperforms both local execution and DeepWear under good network condition. However, when the network condition becomes poor, cloud offloading is comparable and sometimes performance-wise worse than DeepWear. Regarding the energy consumption, as shown in Figure 14(b), DeepWear can even outperform cloud offloading under good network condition (*WaveNet*). The reason is that the Internet access over WiFi is more energy-consuming than accessing the handheld over local radio. Note that under cellular network (e.g., LTE) the energy consumption can be even more than WiFi, thus DeepWear is expected to exhibit more improvements. Finally, recall that compared to cloud offloading, DeepWear offers other benefits such as ubiquitousness and better privacy as described in Section 1.

It's worth mentioning that even though the cloud offloading may perform better than DeepWear under many circumstances, offloading user data to cloud still suffers from privacy concerns, since these data such as images, sensor output, and audio used for these DL models often contain sensitive personal information. Since DeepWear instead performs local offloading, it reduces such privacy concerns to the minimum.

6.4 Adaptive to Environment Dynamics

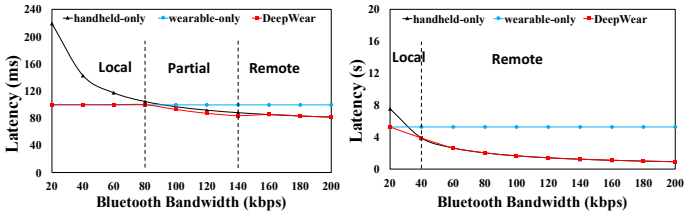
In this section, we evaluate DeepWear's adaptiveness to diverse factors that may vary in real-world environments: the device battery level ($\mathcal{W}_w, \mathcal{W}_p$), the Bluetooth bandwidth (B), and the processor load level ($\mathcal{S}_p$). Our experimental results show that DeepWear can effectively adapt to the dynamics caused by these external factors.

5. We notice wearable's WiFi connectivity is oftentimes slower than phone due to wearable's form factor (smaller antenna).



(a) *TextRNN* model (b) *MobileNet* model

Fig. 11: Weighted energy consumption for different $\mathcal{W}_w$ and $\mathcal{W}_p = 1 - \mathcal{W}_w$. $\mathcal{W}_w$ and $\mathcal{W}_p$ are the energy weight of the wearable and the handheld, respectively. The Y-axis represents the weighted sum of the energy consumption of both devices as $\mathcal{W}_w \cdot E_w + \mathcal{W}_p \cdot E_p$. We use Galaxy S2 and Nexus 6 CPU-powersave to carry out this experiment.



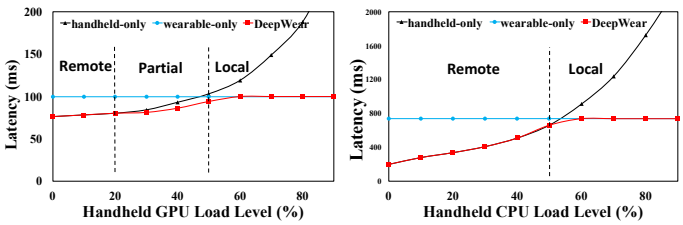
(a) *MNIST* model (b) *GoogLeNet* model

Fig. 12: End-to-end latency across different $\mathcal{B}$. We use Urbane LG and Nexus 6 CPU-interactive to carry out this experiment.

Battery level. As mentioned in Section 4.3, DeepWear'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 $\mathcal{W}_w$ and $\mathcal{W}_p$. We exemplify a possible policy as follows. When the handheld is being charged, we focus on saving the energy for wearable (i.e., $\mathcal{W}_w = 1, \mathcal{W}_p = 0$), whereas when the handheld's battery is running out, we should more aggressively use the wearable's battery (e.g., by setting $\mathcal{W}_w = 0.2$ and $\mathcal{W}_p = 0.8$).

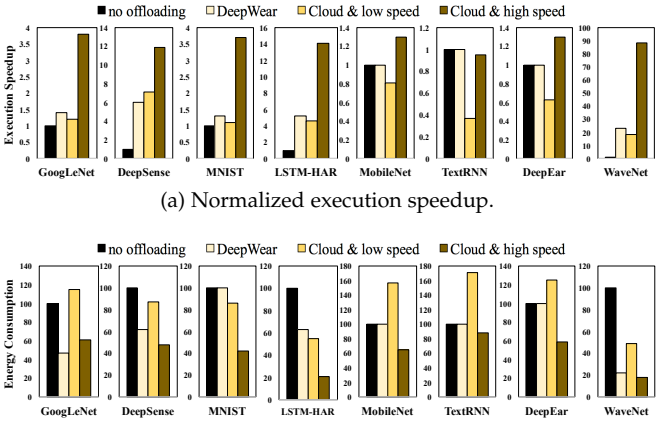
We test DeepWear's robustness against the varying values of $\mathcal{W}_w$ and $\mathcal{W}_p$ (set to $1 - \mathcal{W}_w$). As shown in Figure 11, the partition decision of DeepWear keeps changing according to the configuration of energy weight. As a result, DeepWear always consumes no more energy than either the wearable-only or the handheld-only strategy. Taking *TextRNN* as an example, when $\mathcal{W}_w$ is low ($0 \sim 0.3$), DeepWear chooses to run the model locally as the wearable energy is relatively "cheap". When $\mathcal{W}_w$ becomes higher ($0.3 \sim 0.8$), the model is partitioned and executed on both sides. During this stage, DeepWear outperforms both wearable-only and handheld-only strategies. When $\mathcal{W}_w$ is high, DeepWear offloads all workloads to the handheld to save the energy of wearable. The results of *MobileNet*, another example shown in Figure 11(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 between the wearable and the handheld can change dynam-



(a) *MNIST* model (b) *LSTM-HAR* model

Fig. 13: End-to-end latency across different $\mathcal{S}$. We use the Urbane LG and the Nexus 6 (GPU and CPU) to carry out this experiment.



(a) Normalized execution speedup.

(b) Normalized energy consumption. For DeepWear, we measure the total energy consumption of both wearable and handheld. For the cloud offloading, we measure only the energy consumption on the wearable.

Fig. 14: Compare DeepWear to cloud offloading

ically according to their distance. DeepWear profiles and takes into account this bandwidth online for the partition decision. Figure 12 shows how DeepWear reacts to the changing bandwidth in consideration of end-to-end latency. As observed from both Figure 12(a) (the *MNIST* model) and Figure 12(b) (the *GoogLeNet* model), DeepWear tends to execute the whole DL model locally when the bandwidth is low; when the bandwidth is high, DeepWear performs offloading more aggressively. Additionally, DeepWear also chooses to partially offload the workload. For example, when running *MNIST* with a bandwidth of 100kbps to 140kbps, partial offloading leads to better performance than both the wearable-only and the handheld-only strategies.

Handheld processor load level. We then evaluate DeepWear's robustness against varying load level of the handheld processors (CPU and GPU). We use a script [65] to generate CPU workloads, and use another application [66] to generate GPU workloads by introducing background graphics rendering. As shown in Figure 13, when the processor load is low, DeepWear always offloads the DL tasks to handheld to make use of the under-utilized processing power. In this stage, the performance of DeepWear is similar to handheld-only, and has significant latency reduction compared to wearable-only (e.g., more than half a second for *LSTM-HAR* model shown in Figure 13(b)). When the handheld processor's load increases, DeepWear chooses to execute workloads locally on the wearable device, as doing so outperforms the handheld-only approach. For example, when running *MNIST* with the handheld GPU load of 80%, DeepWear can reduce almost 50% of the latency compared to the handheld-only strategy (188.2ms vs. 99.8ms).

6.5 Latency Awareness

We also evaluate how the developer-specified latency (*PropT*) affects DeepWear's decision on offloading. The results are shown in Table 6. Overall, for 7 out of 9 configurations, DeepWear can satisfy the latency requirement, while the handheld-only and the wearable-only have only 4 and 6, respectively. The only case where DeepWear is unable to provide the desired latency improvement, i.e., *PropT* = 2.0s for *GoogLeNet*, is unavoidable since even the lowest possible latency is higher than *PropT*. In those cases, DeepWear chooses to minimize the end-to-end latency and ignore the energy consumption. In summary, in all cases, DeepWear yields satisfactory results.

Another key observation from Table 6 is that DeepWear can adaptively adjust its decisions based on applications' requirements – a desirable feature in practice. Taking *TextRNN* as an example. When *PropT* is low, DeepWear keeps all workloads on the local wearable device to satisfy (58.9ms) the latency requirement (200ms). This is the same as what the wearable-only strategy does but the handheld-only strategy fails to achieve. When *PropT* becomes higher (300ms), DeepWear chooses different partition points in order to consume less energy than the wearable-only strategy, while keeping a relatively low end-to-end latency. Instead, the wearable-only strategy consumes 21.6% more energy than DeepWear.

6.6 Handling Streaming Data

We also evaluate how the pipelining technique described in Section 4.4 can help improve the throughput for streaming data. As shown in Figure 15, applying pipelining in DeepWear can help improve the overall throughput by 43.75% averaged over the 8 models (the comparison baseline is the basic DeepWear that treats each DL instance separately). For some models such as *MNIST*, the throughput improvement as high as 84% can be achieved through pipelined processing. We observe that the throughput boost depends on the processing time difference between the wearable and the handheld. For models that exhibit large performance difference, applying pipelining achieves less improvement. For example, running *WaveNet* locally yields a latency of 7.7s on Urbane LG, almost 13 times higher than that achieved by offloading to Nexus CPU (0.54s). As a result, applying pipelining increases the throughput by only 5%. This is because when the processing capabilities of the wearable and handheld differ significantly, the little contribution of the weaker device (typically the wearable) makes pipelining fallback to the handheld-only strategy. In contrast, for models that exhibit similar performance on the wearable and the handheld, pipelining leads to a much higher throughput improvement (84% for *MNIST* model).

6.7 System Overhead

DeepWear incurs the computation overhead of executing the partition algorithm (Section 4.3). We have measured all 8 DL models under different configurations. The incurred

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2019.2893250, IEEE Transactions on Mobile Computing

Model	*PropT*	handheld-only			wearable-only			DeepWear		
		Selection	Latency(ms)	Energy(mJ)	Selection	Latency(ms)	Energy(mJ)	Selection	Latency(ms)	Energy(mJ)
TextRNN	200ms	input	239.60	181.79	BiasAdd	58.90	218.28	BiasAdd	58.90	218.28
	250ms							cell/mul	238.78	194.32
	300ms							Sigmoid	256.90	179.45
GoogLeNet	2s∼3s	input	7,306.21	9,361.00	Squeeze	2,058.50	7,616.64	Squeeze	2,058.50	7,616.64
LSTM-HAR	1s∼2s	input	207.02	317.27	output	733.53	1,207.26	input	207.02	317.27

TABLE 6: End-to-end latency and energy consumption (of both the wearable and handheld) of DeepWear across varied developers-specified latency (*PropT*). We use Galaxy S2 and Nexus 6 CPU-powersave to carry out this experiment. We set $\mathcal{W}_w$ and $\mathcal{W}_p$ as 0.5 equally.

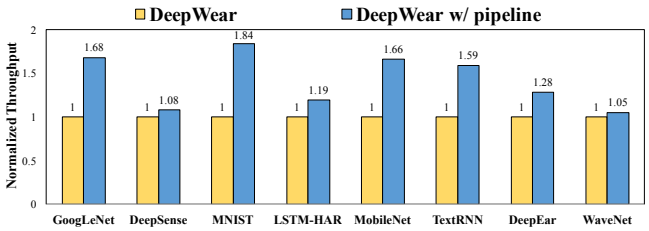


Fig. 15: Throughput of DeepWear with pipelined processing. Results are normalized by DeepWear without pipelining. We use the Urbane LG and the Nexus 6 CPU-interactive to carry out this experiment.

overhead in terms of the fraction of latency is low, ranging from 0.49% (*GoogLeNet*) to 4.21% (*TextRNN*). The reason for such low overhead is multifold. First, our heuristic-based algorithm, as presented in Section 4.3, can reduce the computation complexity to almost $\mathcal{O}(n)$, where n is the number of DL model nodes. Second, the original DL computation is already heavy-load, making the overhead relatively trivial.

Another source of overhead comes from the *System Profiler*. Our measurements indicate that such an overhead is non-trivial when the Bluetooth bandwidth is measured passively. DeepWear can optionally measure the Bluetooth bandwidth through active probing (Section 5). In that case the energy overhead is less than 5% for the wearable. The overhead can be further reduced by using Bluetooth Low Energy (BLE) as instead of classic Bluetooth.

7 Limitations

We discuss some limitations of DeepWear and highlight several future research directions.

• DeepWear currently focuses on the inference stage as opposed to the training stage. In deep learning, which requires a pre-trained model integrated into applications or downloaded in advance. Although performing inference may be sufficient for most applications, we also notice that in recent years there have emerged requirements to train (consume) the data immediately when it is produced on wearable devices. We plan to extend DeepWear for the model training phase. The challenging issues for supporting the model training in DeepWear are in two folds. (1) Designing new latency and energy prediction models for the training procedure (e.g., based on the backpropagation algorithm). (2) Designing new offloading decision algorithms. Since the training phase requires both the forward and backward data flow in our model graph, it is not immediately clear how much benefits partial offloading can reward, which shall be further explored in our future work.

• DeepWear makes partition decision based on two key metrics of user experience: the end-to-end latency and the energy consumption. Besides them, other metrics such as memory usage (both average and peak) is another important metric that should be taken into account [19]. We plan to consider memory as a developer-specified policy similar to the latency (*PropT*). This extension can be integrated into DeepWear via a runtime predicator of memory usage for different partitions and a new set of APIs for developers.

• We have tested DeepWear on only 3 devices (LG Urbane, Galaxy S2, and Nexus 6) and 8 widely used DL models. Though These models are representative and widely used, we plan to assess DeepWear more broadly on other hardware platforms and DL models.

• In many common scenarios, our offloading scheme in DeepWear exhibits unique advantages over the cloud-based offloading in terms of resource utilization, ubiquitous access, and privacy preservation. However, we should point out that due to the limited processing capacity on handheld devices, DeepWear might still suffer from poor performance. For example, there are other concurrent workload (typically as background services) running on a handheld, or the DL task is too heavyweight to be carried out on a handheld. Future work towards such problems includes adaptively offloading DL tasks to multiple personal mobile devices (e.g., a smartphone, a tablet, and a laptop), as well as using the cloud as an alternative offloading target when local resources are too insufficient while the privacy is not a critical concern.

8 Conclusion

Wearables provide an important data source for numerous applications that can be powered by DL tasks. To enable efficient DL on wearables, We have developed DeepWear, a practical DL framework designed for wearables. DeepWear can intelligently, transparently, and adaptively offload DL computations from a wearable to a paired handheld. It introduces various novel techniques such as context-aware offloading, strategic model partition, and pipelining supports to better utilize the processing capacity from the wearable's nearby handhelds. Our evaluation on COTS devices and popular DL models demonstrate DeepWear significantly outperforms both wearable-only and handheld-only approaches by striking a better balance among the latency and the energy consumption on both sides. We believe that the lessons learned from our DeepWear design and implementation will shed the light on developing future AI-powered systems on mobile, wearable, and Internet-of-things (IoT) applications. In our future work, in addition to performing the tasks proposed in §7, we plan to apply

DeepWear to develop real-world DL applications for off-the-shelf wearables. To help the research community reproduce our study, we will release the source code of DeepWear to be publicly available.

ACKNOWLEDGMENT

This work was supported by the National Key R&D Program under the grant number 2018YFB1004800 and the National Natural Science Foundation of China under grant number 61725201. Feng Qian's research was supported in part by the NSF grant CCF-1629347. Xuanzhe Liu acts as the corresponding author of this work.

REFERENCES

[1] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, 2017, pp. 68–81.

[2] V. Radu, N. D. Lane, S. Bhattacharya, C. Mascolo, M. K. Marina, and F. Kawsar, "Towards multimodal deep learning for activity recognition on mobile devices," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'16)*, 2016, pp. 185–188.

[3] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'16)*, 2016, pp. 23:1–23:12.

[4] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "LEO: scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom'16)*, 2016, pp. 320–333.

[5] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys'16)*, 2016, pp. 176–189.

[6] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'15)*, 2015, pp. 283–294.

[7] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?" in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications (HotMobile'15)*, 2015, pp. 117–122.

[8] M. Xu, F. Qian, Q. Mei, K. Huang, and X. Liu, "Deeptype: On-device deep learning for input personalization service with minimal privacy concern," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 4, p. 197, 2018.

[9] "On-Device Machine Intelligence," https://research.googleblog.com/2017/02/on-device-machine-intelligence.html, 2016.

[10] "How Google Translate squeezes deep learning onto a phone," https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html, 2015.

[11] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "When mobile apps going deep: An empirical study of mobile deep learning," *arXiv preprint arXiv:1812.05448*, 2018.

[12] "Smartwatch Market Size, Share, Growth, Industry Report, 2018–2023," https://www.psmarketresearch.com/market-analysis/smartwatch-market, 2018.

[13] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, 2010, pp. 49–62.

[14] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, 2015, pp. 137–150.

[15] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys'14)*, 2014, pp. 68–81.

[16] X. Liu, T. Chen, F. Qian, Z. Guo, F. X. Lin, X. Wang, and K. Chen, "Characterizing smartwatch usage in the wild," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, 2017, pp. 385–398.

[17] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, (CVPR'16)*, 2016, pp. 4820–4828.

[18] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'14)*, 2014, pp. 1269–1277.

[19] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: an approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*, 2016, pp. 123–136.

[20] "TensorFlow," https://www.tensorflow.org/, 2017.

[21] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the 6th European conference on Computer systems (EuroSys'11)*, 2011, pp. 301–314.

[22] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "COMET: code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'12)*, 2012, pp. 93–106.

[23] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring Android Java code for on-demand computation offloading," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, 2012, pp. 233–248.

[24] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. N. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017, pp. 615–629.

[25] J. Wang, J. Sun, H. Lin, H. Dong, and S. Zhang, "Convolutional neural networks for expert recommendation in community question answering," *SCIENCE CHINA Information Sciences*, vol. 60, no. 11, pp. 110 102:1–110 102:9, 2017.

[26] P. Li, M. Liu, X. Zhang, X. Hu, B. Pang, Z. Yao, and H. Chen, "Novel wavelet neural network algorithm for continuous and noninvasive dynamic estimation of blood pressure from photoplethysmography," *SCIENCE CHINA Information Sciences*, vol. 59, no. 4, pp. 042 405:1–042 405:10, 2016.

[27] W. Qu, D. Wang, S. Feng, Y. Zhang, and G. Yu, "A novel cross-modal hashing algorithm based on multimodal deep learning," *SCIENCE CHINA Information Sciences*, vol. 60, no. 9, pp. 092 104:1–092 104:14, 2017.

[28] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP'14)*, 2014, pp. 4087–4091.

[29] E. Variani, X. Lei, E. McDermott, I. Lopez-Moreno, and J. Gonzalez-Dominguez, "Deep neural networks for small footprint text-dependent speaker verification," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP'14)*, 2014, pp. 4052–4056.

[30] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[31] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014, pp. 269–284.

[32] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays (FPGA'15), 2015, pp. 161–170.

[33] Y. Chen, J. S. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'16)*, 2016, pp. 367–379.

[34] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'16)*, 2016, pp. 243–254.

[35] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'18)*, 2018, pp. 389–400.

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

[37] L. Yang, J. Cao, Z. Wang, and W. Wu, "Network aware multi-user computation partitioning in mobile edge clouds," in *Proceedings of the 46th International Conference on Parallel Processing ICPP'17*, 2017, pp. 302–311.

[38] L. Yang, B. Liu, J. Cao, Y. Sahni, and Z. Wang, "Joint computation partitioning and resource allocation for latency sensitive applications in mobile edge clouds," in *Proceedings of the 10th International Conference on Cloud Computing (CLOUD'17)*, 2017, pp. 246–253.

[39] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, "Quantifying the impact of edge computing on mobile applications," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'16)*, 2016, pp. 5:1–5:8.

[40] S. A. Ossia, A. S. Shamsabadi, A. Taheri, H. R. Rabiee, N. Lane, and H. Haddadi, "A hybrid deep learning architecture for privacy-preserving mobile analytics," *arXiv preprint arXiv:1703.02952*, 2017.

[41] "Deep MNIST tutorial," https://www.tensorflow.org/get_started/mnist/pros, 2017.

[42] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR'15)*, 2015, pp. 1–9.

[43] "LSTM for human activitiy recognition," https://github.com/guillaume-chevalier/LSTM-Human-Activity-Recognition, 2017.

[44] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. F. Abdelzaher, "Deepsense: A unified deep learning framework for time-series mobile sensing data processing," in *Proceedings of the 26th International Conference on World Wide Web, (WWW'17)*, 2017, pp. 351–360.

[45] "Text Classification Using Recurrent Neural Networks on Words," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/learn/text_classification.py, 2017.

[46] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[47] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'14)*, 2014, pp. 3104–3112.

[48] D. Wang and E. Nyberg, "A long short-term memory model for answer sentence selection in question answering," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, (ACL'15)*, 2015, pp. 707–712.

[49] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[50] D. Huang, L. Yang, and S. Zhang, "Dust: Real-time code offloading system for wearable computing," in *Proceedings of the IEEE Global Communications Conference (GLOBECOM'15)*, 2015, pp. 1–7.

[51] B. Shi, J. Yang, Z. Huang, and P. Hui, "Offloading guidelines for augmented reality applications on wearable devices," in *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference, MM '15, Brisbane, Australia, October 26 - 30, 2015*, 2015, pp. 1271–1274.

[52] J. Ko, J. Lee, and Y. Choi, "Poster: A novel computation offloading technique for reducing energy consumption of smart watch," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion, Singapore, Singapore, June 25-30, 2016*, 2016, p. 46.

[53] "Vuzix M100 Smart Glasses," https://www.vuzix.com/Products/M100-Smart-Glasses, 2017.

[54] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "Rstensorflow: Gpu enabled tensorflow for deep learning on commodity android devices," in *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, 2017, pp. 7–12.

[55] "Wearable Data Layer API," https://developer.android.com/training/wearables/data-layer/index.html, 2017.

[56] "Monsoon power meter," https://www.msoon.com/LabEquipment/PowerMonitor/, 2017.

[57] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2010, part of ESWeek '10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24-28, 2010*, 2010, pp. 105–114.

[58] "Coefficient of Determination (R-squared) Explained," https://towardsdatascience.com/coefficient-of-determination-r-squared-explained-db32700d924e, 2018.

[59] "Cut in graph theory," https://en.wikipedia.org/wiki/Cut_(graph_theory), 2017.

[60] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: frequent subgraph and pattern mining in a single large graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 517–528, 2014.

[61] "TensorFlow inference Java APIs," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/android/java/org/tensorflow/contrib/android/TensorFlowInferenceInterface.java, 2017.

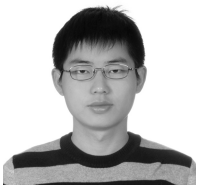
[62] "Caffe2 deep learning framework," https://github.com/caffe2/caffe2, 2017.

[63] "PyTorch," http://pytorch.org/, 2017.

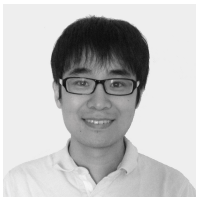
[64] "Android Message API," https://developer.android.com/reference/com/google/android/gms/wearable/MessageApi.html, 2017.

[65] "A Programmable CPU Load Generator," https://github.com/ptitiano/cpuloadgen, 2012.

[66] G. Huang, M. Xu, F. X. Lin, Y. Liu, Y. Ma, S. Pushp, and X. Liu, "Shuffledog: Characterizing and adapting user-perceived latency of Android apps," *IEEE Transactions on Mobile Computing*, vol. 16, no. 10, pp. 2913–2926, 2017.



Mengwei Xu is a Ph.D student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include mobile computing and operating system.



Feng Qian is an assistant professor in the Computer Science and Engineering Department at the University of Minnesota – Twin Cities. His research interests cover the broad areas of mobile systems, VR/AR, computer networking, and system security.

Mengze Zhu is an undergraduate student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include Mobile Systems and Machine Learning.

Feifan Huang is an undergraduate student in the School of Electronics Engineering and Computer Science of Peking University, Beijing, China. His research interests include Mobile Systems and Machine Learning.

Saumay Pushp is a Ph.D candidate in the School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, South Korea. His research interests include Mobile Systems and Networking.

Xuanzhe Liu is an associate professor in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His research interests are in the area of services computing, mobile computing, web-based systems, and big data analytics. He was selected as the CCF-IEEE Computer Society Young-Scientist in 2018. He is the corresponding author of this work.