Flame: A Self-Adaptive Auto-Labeling System for Heterogeneous Mobile Processors

JIE LIU, University of California, Merced, USA
JIAWEN LIU, University of California, Merced, USA
ZHEN XIE, University of California, Merced, USA
XIA NING, The Ohio State University, USA
DONG LI, University of California, Merced, USA

How to accurately and efficiently label data on a mobile device is critical for the success of training machine learning models on mobile devices. Auto-labeling data on mobile devices is challenging, because data is incrementally generated and there is a possibility of having unknown labels among new coming data. Furthermore, the rich hardware heterogeneity on mobile devices creates challenges on efficiently executing the auto-labeling workload. In this paper, we introduce Flame, an auto-labeling system that can label dynamically generated data with unknown labels. Flame includes an execution engine that efficiently schedules and executes auto-labeling workloads on heterogeneous mobile processors. Evaluating Flame with six datasets on two mobile devices, we demonstrate that the labeling accuracy of Flame is 11.8%, 16.1%, 18.5%, and 25.2% higher than a state-of-the-art labeling method, transfer learning, semi-supervised learning, and boosting methods respectively. Flame is also energy efficient, it consumes only 328.65mJ and 414.84mJ when labeling 500 data instances on Samsung S9 and Google Pixel2 respectively. Furthermore, running Flame on mobile devices only brings about 0.75 ms additional frame latency which is imperceivable by the users.

CCS Concepts: • **Heterogeneous Hardware** → **Mobile Processors**; • **Data Labeling** → Incremental Learning.

Additional Key Words and Phrases: Semi-supervised learning, dynamic data labeling, mobile devices

ACM Reference Format:
Jie Liu, Jiawen Liu, Zhen Xie, Xia Ning, and Dong Li. 2021. Flame: A Self-Adaptive Auto-Labeling System for Heterogeneous Mobile Processors. *ACM Trans. Graph.* 37, 4, Article 111 (August 2021), 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Machine learning (ML) has been increasingly utilized in mobile devices (e.g., face and voice recognition and smart keyboard). However, many ML applications running on mobile devices (such as smartphone) mainly focus on ML inference not ML training. Recently, training machine learning (ML) models on mobile devices instead of on clouds attracts more and more attention, because of the concerns on data privacy, security and network bandwidth of using clouds for training [Eom et al. 2015; Konečnỳ et al. 2016]. For example, Google trains a RNN model on mobile devices for keyboard-based input prediction with federated learning [Konečnỳ et al. 2016]. However, data generated on mobile devices usually do not have labels, causing difficulty of training ML models (especially supervised ML models). How to accurately and efficiently label data on mobile devices is critical for the success of training ML models on mobile devices. Using automatic labeling is a solution to address the above problem. Studies of automatic labeling in the past decade have been focused on data stored on servers [Haas et al. 2015; Ratner et al. 2017; Varma and Ré 2018; Yang et al. 2018]. Those approaches do not consider hardware resource constraint and only work well on static datasets (i.e., the number of data instances and labels are pre-determined and fixed).

Compared with data labeling on servers, labeling data generated on mobile devices faces some unique challenges.

Hardware resource constraint. The existing methods [Ratner et al. 2017; Varma and Ré 2018] build labeling functions based on supervised machine learning models to label data. Each labeling function works well for only a portion of data. Hence a large number of labeling functions (usually more than 100) are needed to have high data coverage. However, using those functions requires abundant computational resources and memory space (e.g., tens of GBs), while mobile devices typically do not have. The labeling method on mobile devices must be lightweight to make data labeling feasible.

Labeling of dynamic data. Data on a mobile device are dynamically generated in a streaming manner; Those dynamically generated data can belong to a new label unseen in the existing set of labels. For example, in real-world mobile applications, such as image classification, the number of the labels is not fixed, and new unseen labels may occur at any time during the usage. When the data instances belong to new unseen labels (e.g., new types of flowers), those mobile applications can not recognize the emergence of the new unseen labels.

Using heterogeneous hardware in mobile processors. Mobile processors are often equipped with rich heterogeneity for high energy efficiency and performance [Liu et al. 2019b; Wu et al. 2019]. For example, Samsung S9 has eight CPU cores, a GPU, and a Hexagon Digital Signal Processor (DSP). The rich hardware

Authors' addresses: Jie Liu, University of California, Merced, Merced, CA, USA, jliu279@ucmerced.edu; Jiawen Liu, University of California, Merced, Merced, CA, USA, 78229, jliu265@ucmerced.edu; Zhen Xie, University of California, Merced, Merced, CA, USA, 78229, zxie10@ucmerced.edu; Xia Ning, The Ohio State University, Columbus, Ohio, USA, 78229, ning.104@osu.edu; Dong Li, University of California, Merced, Merced, CA, USA, 78229, dli35@ucmerced.edu.

0730-0301/2021/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

heterogeneity makes the workload scheduling complicated, because we must comprehensively analyze the characteristics of the workloads and choose proper computing units for high energy efficiency and performance.

To address the above challenges, we introduce Flame, an auto-labeling system for mobile processors. Flame is featured with mobile hardware-aware algorithms and system designs.

To overcome the hardware resource constraint, Flame includes a new lightweight method, named *clustering with minimal impurity*, to build a number of labeling functions. After the clusters are built based on a limited number of labeled data instances, Flame replaces the data instances within the same cluster by the cluster's *prototypes* to reduce the computation overhead. The decision boundary of each labeling function is determined by its prototypes. Any data instance falling outside the decision boundaries of all the labeling functions is identified as a data instance potentially with a new label. Flame interprets the presence of a sufficiently large number of such data instances with strong cohesion among themselves as the emergence of a new label. Furthermore, because of the dynamic characteristics of the data to be labeled on mobile devices, the labeling functions must be updated from time to time to capture the accurate distribution of data. In Flame, because each labeling function consists of a number of prototypes, and updating the labeling functions is just a matter of updating its prototypes.

To guarantee the labeling accuracy, Flame uses two estimators, *Association* and *Purity*, to measure the labeling confidence of each labeling function. We theoretically show that the use of these estimators can guarantee the labeling accuracy. Finally, Flame uses an ensemble method to gather the labeling confidence of labeling functions to determine final labels.

To utilize the heterogeneous hardware, Flame is featured with a hardware heterogeneity-aware execution engine to run the auto-labeling algorithm (§4). The execution engine determines which part of the computation should be placed on a particular computing unit (CPU, GPU or DSP) based on the characteristics of workload and hardware. Some computation of Flame is placed on GPU to shorten execution time, because the computation can offer high thread-level parallelism and efficiently leverage fast shared-memory on GPU. Some computation is placed on DSP (the most power-efficient computing unit), when the energy consumption of the computation is high. The execution engine also coordinates the interaction between CPU and DSP to avoid wakeup latency suffered by CPU for energy saving. We summarize major contributions as follows.

- We propose a fast, accurate, and lightweight auto-labeling system, Flame, for mobile devices. It is the first system to label data dynamically generated on mobile devices. We implement it on two realistic mobile phones (Samsung S9 and Google Pixel2).
- The labeling accuracy of Flame is 11.8%, 16.1%, 18.5%, and 25.2% higher than that of Snuba [Varma and Ré 2018] (a state-of-the-art auto-labeling system), transfer learning, semi-supervised learning algorithms and boosting methods respectively. Also, Flame can detect unseen labels while all other systems cannot.

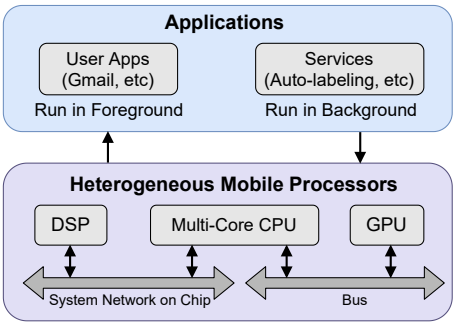


Fig. 1. An example of heterogeneous mobile processors.

- Flame has high energy efficiency. It consumes only 328.65mJ and 414.84 mJ when labeling 500 data instances on Samsung S9 and Google Pixel2 respectively, while the full energy of their battery is $3''88 \times 10^4$ and $3''49 \times 10^4$J respectively. This makes Flame a highly feasible system for mobile phones.
- Flame running on a mobile phone has minimum impacts on the user experience of using another application. Flame brings only 0.75 ms additional frame latency to the user application.

2 BACKGROUND ON MOBILE PROCESSORS

We introduce background information in this section.

Heterogeneous mobile processors. The modern mobile processors in mobile devices are characterized with hardware heterogeneity. Figure 1 shows such an example commonly found in many mobile devices (e.g., Samsung S9, Google pixel2, Huawei P8 and Xiaomi Mi 10). In our study, we use mobile devices, each of which has eight-core CPU (four slow cores and four fast cores), mobile GPU, and DSP. DSP is typically the most power-efficient computing unit in a mobile device [Codrescu et al. 2014]. However, DSP is not good at handling some operations (e.g., square root and division).

Mobile applications. The applications running on mobile devices can be classified as foreground and background applications. The mobile system usually sets a higher priority to run the foreground applications to enable smooth interaction between the user and mobile devices. The background applications have low priority to be scheduled and run by the mobile system. The background applications are not expected to introduce significant latency to the foreground applications. Flame is a background application.

3 MODEL DESIGN

Figure 2 overviews our auto-labeling model. Flame includes three components, *Labeling Functions Generation*: a labeling function generator to generate a number of labeling functions for assigning labels. *Labeling Functions Self-adaption*: a self-adaptive strategy to update the existing labeling functions, detect the emergence of new unseen labels in a dynamic setting, and build new labeling functions for the data instances belong to new unseen labels. *Labeling Results Guarantees* calculates a labeling confidence value for each labeling function during data instance labeling, and then

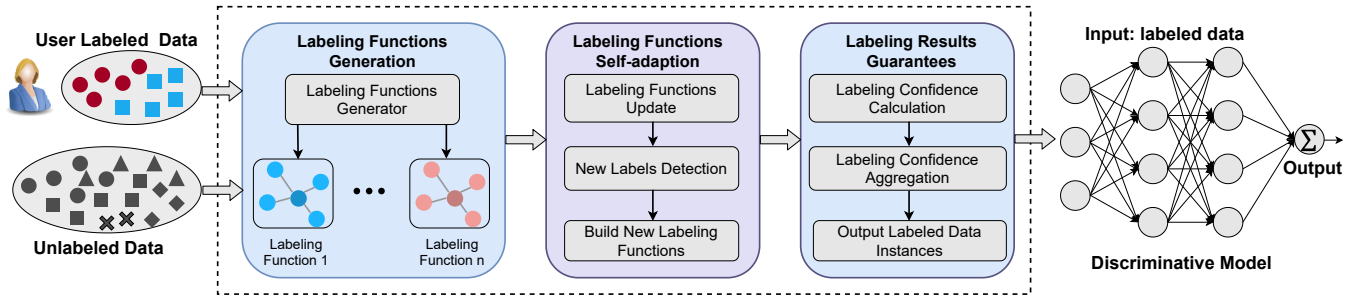


Fig. 2. An overview of the model design in Flame. (1) Labeling Functions Generation generates a number of labeling functions (LFs) based on the user labeled data instances, each labeling function includes several prototypes. (2) Labeling Functions Self-adaption applies the LFs on the dynamic unlabeled data, these LFs can be updated by the Flame. Furthermore, Flame can detect the emergence of new labels and building new labeling functions for the data instances with new labels. (3) Labeling Results Guarantees calculates a labeling confidence value for each labeling function, then it aggregates all the labeling confidence values, finally, it assigns labels for unlabeled data instances. (4) The labeled data instances can be used to train discriminative classification models, such as a deep neural network.

aggregates and verifies the labeling results of Flame for an unlabeled data instance. The input/output of Flame is discussed as follows.

Input data. The input data of Flame is a small number of labeled data and a large number of unlabeled data. The labeled data is represented as $!$ = $\{G_8 \cdot \sim_8\}_{8=1}^{\#!}$, where $G_8 \in$ '3 is the 3-dimension features of the data 8 and $\sim_8 \in$. = {1·2·"", } is the associated label (different known labels in total). The non-stationary unlabeled data is represented as $*$ = $\{G_C\}_{C=1}^{\#*}$ ($G_C \in$ '3), where $\#* \in [0 \cdot \infty)$ is the number of unlabeled data. In our setting, $\#*$ can be large, as the new data is continuously generated.

Output data. The output of Flame is the confidence of a label $\sim_8 \in .'$ = {1·2·"", ·"", '} for data G_8 in the unlabeled dataset $*$, where .' is the set of result labels including known labels and new unseen labels (' ≥). Here, ' ≥ , which indicates that some unseen labels that are not in . may appear in .', as new data is incrementally generated. The final labeling confidence value is calculated through an ensemble method in Flame (§3.3).

3.1 Labeling Functions Generation

We design a lightweight method to generate labeling functions. Existing studies [Ratner et al. 2017; Yang et al. 2018] use supervised machine learning models (e.g., Decision Tree, K-Nearest Neighbor) to build labeling functions. However, these methods cannot work well on mobile devices because of two reasons. First, the data generated on mobile devices are seldom labeled. Therefore, using the labeling functions built based on supervised learning models causes low labeling accuracy. Second, a large number of labeling functions (more than 100) are needed to have high data coverage, which requires abundant computational resources and memory space.

We design an impurity-based clustering method to determine the boundary of each labeling function. A cluster is completely *pure* if the data instances within this cluster belong to the same label (along with some unlabeled data). Given a limited amount of labeled data, the goal of impurity-based clustering is to create a number of clusters by minimizing the intra-cluster dispersion, and at the same time by minimizing the impurity of each cluster, we refer it as *Clustering with Minimal Impurity*. In order to determine the boundary of each labeling function in fine granularity, each created cluster is further divided into a number of cliques. A clique of a cluster is a group of data instances with cohesion larger than 0.5 in the corresponding cluster, the cohesion is calculated by a commonly used method called q-NSC [Haque et al. 2016].

Then, we use the *prototype* of a clique to replace the data instances in that clique. A prototype indicates the best exemplar of the data instances within a clique. The corresponding prototypes of all the cliques of a cluster could provide a concise representation for the entire raw data instances within a cluster. In this paper, a prototype of a clique is defined as below,

Definition 1. ***Prototype****: the prototype of a clique is a tuple denoted by* ? = Ÿ `·A·3·=·5 ¡ *, where* ` *is the centroid of the clique,* A *represents the radius of the clique,* 3 *denotes the sum of squared Euclidean distance from data instances in the clique to* `, = *is the total number of data instances in the clique, and* $\bar{5}$ *is a vector recording the number of data instances belonging to different labels in the clique.*

The $\bar{5}$ is referred as *frequencies* in the rest of the paper. Here is an example of $\bar{5}$, $\bar{5}$ = (5_1·5_2·"",5_C), where each element 5_8 in $\bar{5}$ is the frequency of the corresponding label $\sim_8$ assigned to the data instances. Finally, each cluster is represented by one or more prototypes depending on the data distribution in the cluster. Each prototype is denoted by $?_{89}$, where the subscript 8 indicates the index of the 8^C built cluster, 9 ¡ 0 is the prototype index. We denote the set of prototypes for cluster 8 by $\mathbb{P}_8$, i.e., $?_{89} \in \mathbb{P}_8$. Figure 3 depicts an example of the relationship among cluster, cliques, and prototypes.

The boundary of each labeling function is determined by a collection of one or multiple prototypes. Flame maintains a labeling function pool which is represented as ! , assuming ! contains labeling functions, that is, ! = {;$_{58}$·"",;5 }, where ;$_{58}$ ∈ ! is an individual labeling function. Each labeling function ;$_{58}$ could

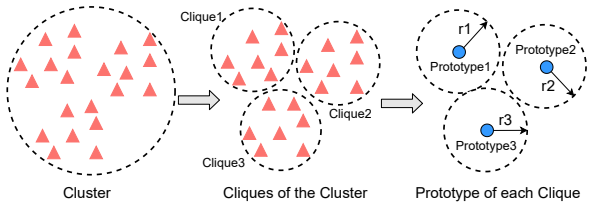


Fig. 3. Illustration of the cluster, cliques, and prototypes. In this example, the cluster consists of three non-overlapping cliques, each of which is represented by its prototype. The cliques of a cluster can also be overlapped to cover all the data instances.

Algorithm 1: PrototypeInitialization ($_{89}$)

Input: $_{89}$: the data instances in clique $_{89}$ of cluster 8
Output: $?_{89}$: the initialized prototype for clique $_{89}$

$\grave{}_{89} = \frac{1}{|\ _{89}|} \sum_{G \in\ _{89}} G$; // $|\ _{89}|$ is the size of $_{89}$
$A_{89} \leftarrow \max_{G \in\ _{89}} \|G - \grave{}_{89}\|_2^2$;
$3_{89} \leftarrow \sum_{G \in\ _{89}} \|G - \grave{}_{89}\|_2^2$;
$=_{89} \leftarrow |\ _{89}|$;
$5 \leftarrow (5_1, \ldots, 5_8 + 1, \ldots, 5\ )$;
return $?_{89}$

consisted by $"$ prototypes, therefore, $;5_8$ in $!$ is represented as $;5_8 = \{?_{81}, \ldots, ?_{8"}\}$.

Prototype initialization. The initialization for the prototypes is based on the limited number of labeled dataset $_! = \{G_8, \sim_8\}_{8=1}^{\#_!}$. Here, the number of labels in $_!$ may be small when compared to the eventual labels that may occur over time. Data instances associated with label $\sim_8 \in .$ are denoted by $_8$. $_8$ consists of $"$ cliques ($_{81}, \ldots, _{8"}$) and the data instances in clique $_{89}$ are represented as $_{89}$. We create a prototype for each clique by selecting a data instance from $_{89}$, uniformly at random. Algorithm 1 details the prototype initialization process for a given label $\sim_8 \in .$.

Objective function. When building the labeling functions using the *Clustering with Minimal Impurity* method, the objective is to minimize the dispersion and impurity of clusters. We formulate the objective function as follows,

$$\$1\mathfrak{g}(G) = \sum_{8=1}^{} \sum_{9=1}^{"} \sum_{G \in\ _{89}} \|G - \grave{}_{89}\|^2 + \_ \sum_{8=1}^{} \sum_{9=1}^{"} {}_{89} \times =\mathbb{CA}{>}?{\sim}_{89} \quad (1)$$

In Equation 1, the first term is used to minimize the dispersion of data instances within the scope of each prototype; is the total number of labeling functions in $!$; $_{89}$ is the set of data instances within the scope of the prototype $?_{89}$; and $\grave{}_{89}$ is the centroid of the prototype $?_{89}$. The second term in Equation 1 is used to minimize the impurity of data instances in each clique, and $\_$ is a hyper-parameter controlling the importance of the second term. The impurity is constructed based on labeling diversity and the entropy value of data instances in the scope of a prototype, and it is calculated as $_{89} \times =\mathbb{CA}{>}?{\sim}_{89}$, where $_{89}$ is the *Aggregated Dissimilarity Count* (ADC) of the prototype $?_{89}$ and $=\mathbb{CA}{>}\mathbb{C}{\sim}_{89}$ is the entropy of the prototype $?_{89}$. $_{89}$ is calculated as follows,

$$_{89} = \sum_{G \in\ _{89}} \sum_{9=1}^{} {}_{89}(G,)\ \quad (2)$$

where $_{89}(G,)$ denotes the *Dissimilarity Count* (DC) of a data instance G in the prototype $?_{89}$ having the label , it is calculated as the total number of labeled instances in that prototype belonging to labels other than .That is,

$${}_{89}(G,) = |!_{89}| - |!_{89}( )|, \quad (3)$$

where $|!_{89}|$ is the total number of labeled data instances within the scope of prototype $?_{89}$, and $|!_{89}( )|$ is the number of instances in the prototype $?_{89}$ belonging to label . The entropy in Equation 1 is calculated as follows,

$$=\mathbb{CA}{>}?{\sim}_{89} = \sum_{=1}^{} \left(-\frac{|!_{89}( )|}{|!_{89}|} \times \log\left(\frac{|!_{89}( )|}{|!_{89}|}\right)\right), \quad (4)$$

where $\frac{|!_{89}( )|}{|!_{89}|}$ is the prior probability of the label , $'$ is the number of labels.

Algorithm 2: UpdatePrototypes ($G_{=4F}, \mathbb{P}_8,)_8$)

Input: $G_{=4F}$: new coming data instance in cluster 8,
$\mathbb{P}_i$: current prototypes set of cluster 8,
$)_8$: threshold value for cluster 8,
Output: Updated Prototypes: $\mathbb{P}_8^D$

$: \leftarrow |\mathbb{P}_8|$; // Current number of prototypes in $\mathbb{P}_8$
$_{89}^{=4F} = \arg\min_{9=1, \ldots, :} \|G_{=4F} - \grave{}_{89}\|_2^2$;
if $_{89}^{=4F}\ \ddot{Y}\ )_8$ **then**
  // Update the prototypes $\mathbb{P}_8$
  $\grave{}_{89} \leftarrow \frac{1}{=_{89}+1}(G_{=4F} + =_{89} * \grave{}_{89})$;
  $3_{89} \leftarrow 3_{89} + {}_{89}^{=4F}$;
  $=_{89} \leftarrow =_{89} + 1$;
else
  // Create a new prototype $?_{8,:+1}$
  $?_{8,:+1} \leftarrow \%A{>}\mathbb{C}{>}\mathbb{C}{\sim}?4\ =8\mathbb{C}8\mathbb{O};(G_{=4F})$; // Algorithm 1
  $\mathbb{P}_8^D = \mathbb{P}_8 \oplus ?_{8,:+1}$;
$)_8^D \leftarrow *?3\mathbb{O}\mathbb{C}4()_8, \mathbb{P}_8^D)$; // Update the threshold
for $?_{89}, ?_{8:} \in \mathbb{P}_8^D$ **do**
  // Merge close prototypes
  if $\|\grave{}_{89} - \grave{}_{8:}\|_2^2\ \ddot{Y}\ )_8^D$ **then**
    $\grave{}_{89} = \frac{{}_{89} + {}_{8:}}{2}$;
    $3_{89} \leftarrow 3_{89} + 3_{8:}$;
    $=_{89} \leftarrow =_{89} + =_{8:}$;
    $\mathbb{P}_8^D = '4{<>}E4(?_{8:}, \mathbb{P}_8^D)$
return $\mathbb{P}_8^D$

3.2 Labeling Functions Self-Adaption

In this section, we introduce how the built labeling functions adapt to the dynamic datasets. Flame can update the prototypes of

the labeling functions and it can also build new labeling functions for those new coming labels.

Labeling functions update. Since each labeling function is consisted by a number of prototypes, the labeling functions can be updated by updating its prototypes. We use a threshold to determine whether a new coming data instance $G_{=4F}$ can be associated to any of the existing prototypes in $\mathbb{P}_8$ of cluster 8. If $G_{=4F}$ is close to a prototype $?_{89} \in \mathbb{P}_8$, then we update $?_{89}$. If not, we create a new prototype using Algorithm 1 and add it to $\mathbb{P}_8$. Algorithm 2 describes the prototype update process. We first compute a threshold $)_8$ associated with the cluster 8 using all its existing prototypes, i.e., $)_8 = \text{mean}(3_{89}) + 0''5 * \text{std}(3_{89})$ for all $?_{89} \in \mathbb{P}_8$. Here, *mean* and *std* are the mean and standard deviation of sum of squared distances in each prototype of cluster 8. We then compute the closest prototype for the new coming data instance $G_{=4F}$ by $\arg\min_{9=1.''''.:} {}^{=4F}_{89}$, where ${}^{=4F}_{89} = \|G_{=4F} - `_{89}\|_2^2$, $:$ is the current number of prototypes in $\mathbb{P}_8$ (line 2). If ${}^{=4F}_{89}\ \ddot{Y}\ )_8$, then the prototype $?_{89}$ is updated (line 3-6). If not, then a new prototype is created using $G_{=4F}$ (line 7-9). The prototype update process may generate a large number of prototypes. Too many prototypes can cause overfitting, furthermore, storing large number of prototypes consumes too much memory space. To avoid this scenario, we determine whether any two given prototypes in $\mathbb{P}_8$ can be merged using the the updated threshold $)_8$ (line 11-16). At last, the updated prototype is returned (line 17).

New labels detection. Before building new labeling functions for the new unseen labels, Flame needs to detect the appearance of unseen labels first. Similarly to some existing methods [Mitchell et al. 2018], we compute the new unseen labels detection threshold $){}^{8}_{=4F}$ for each cluster 8 to reject data instances belong to new unseen labels and assign labels for data instances with existing labels. We refer it as *Nearest Mean Clustering* (NMC).

Due to the dynamic characteristics of the data to be labeled, Flame requires to continuously update prototypes. An optimal threshold value for $){}^{8}_{=4F}$ should be determined based on the current data patterns. Here, we assume that data of a same label follow a Gaussian distribution. Applying the average inner-cluster distance with a small range of float for each cluster, we obtain the statistic for confidence threshold. A cluster C_8 is consisted by cliques, the centroids of these cliques are $`_{81}.''''.`_8$. For the cluster C_8, we have,

$$38BC_8 = \frac{1}{\|C_8\|} \tilde{O}_{G \in C_8} \min_{9=1''''} \|G - `_{89}\|_2^2 \tag{5}$$

where $\|C_8\|$ is the size of the cluster C_8, $\min_{9=1.''''.} \|G - `_{89}\|_2^2$ means the distance of data instance $G$ to it's nearest prototype $?_{89} \in \mathbb{P}_8$. The desired threshold value $){}^{8}_{=4F}$ for cluster $C_8$ is calculated by $){}^{8}_{=4F} = 38BC_8 + I * BC3_8$, where $BC3_8$ is the standard deviation of $\|G - `_{89}\|_2^2$.

The boundary of a labeling function $;5_8 \in !$ is determined by it prototypes. Each prototype corresponds to a "hypersphere" in the feature space with a centroid and radius. The coverage of a labeling function $;5_8$ is the union of the hyperspheres encompassed by all prototypes in $;5_8$. To assign a label for a new instance $G_{=4F}$ by the labeling function $;5_8$, we compute the distance set ${}^{8}_{=4F}$ from $G_{=4F}$ to the nearest prototype by $\min_{9=1.''''.} \|G - `_{89}\|_2^2$. Finally, we select the minimum distance from set ${}^{8}_{=4F}$, i.e., $\min {}^{8}_{=4F}$, If $\min {}^{8}_{=4F}$ is less than the threshold $){}^{8}_{=4F}$, the label of $G_{=4F}$ is from one of the existing labels. Else, $G_{=4F}$ is regarded as a data instance with an unknown label.

When the data instance $G_{=4F}$ is from one of the existing labels, we assign the labels for $G_{=4F}$ as follows. Assume the value of $\min {}^{8}_{=4F}$'s corresponding prototype is $?_{89}$. In the prototype $?_{89}$, $5_{<OG}$ is the highest frequency value in the frequency vector 5, then $5_{<OG}$'s corresponding label ~ will be assigned to data instance $G_{=4F}$. Each $;5_8 \in !$ maintains an assigned label for the data instance $G_{=4F}$ and a labeling confidence value (§3.3) for its labeling result. The label of the data instance $G_{=4F}$ is determined by taking the majority vote among all labeling functions.

Build new labeling functions. Flame builds new labeling functions for the data instances belong to new unseen labels. When the data instance $G_{=4F}$ is potentially with a new label, it will be stored into a buffer . Flame periodically checks the buffer and applies our clustering with minimal impurity (§3.1) method on the data instances in buffer to build the new labeling functions. Therefore, Flame can incrementally incorporate new label information from data instances in . At last, those data instances that belong to the new label are removed from , and the released space is used to collect the subsequent data instances potentially belong to other new labels.

3.3 Labeling Results Guarantees

Labeling confidence calculation. In Flame, each labeling function $;5_8 \in !$ uses a metric ($>=5_8$) to quantify its labeling confidence for a data instance G. Flame employs two heuristics, i.e., *association* and *purity* to estimate the $>=5_8(G)$. After each individual labeling function's confidence value $>=5_8(G)(8 = 1.''.$) acquired, Flame combines these values together to get the entire labeling functions $!$'s labeling confidence value $>=5(G)$.

The closest prototype from G in labeling function $;5_8$ is $?_{89}$. The $?_{89}$ is the 9^C prototype of $;5_8$, $_{<OG}$ is the label having highest frequency in $?_{89}$. The *Association* and *Purity* of the prototype $?_{89}$ are calculated as follows:

- *Association* is calculated by $'_{89} - {}_{89}(G)$, where $'_{89}$ is the radius of $?_{89}$ and ${}_{89}(G)$ is the distance between G and $?_{89}$. If ${}_{89}(G)$ is small, it means G is close to the prototype $?_{8:}$, then $('_{89} - {}_{89}(G))$ is large which leads to a high the association and confidence of labeling.
- *Purity* is calculated by $\frac{|!_{89}(\ _{<OG})|}{|!_{89}|}$, where $|!_{89}|$ is the sum of all frequencies in $?_{89}$, and $|!_{89}(\ _{<OG})|$ is the frequency of $_{<OG}$ in $?_{89}$. A large $!_{89}(\ _{<OG})$ means the high purity of the prototype $?_{89}$, which also leads to a high confidence of labeling.

We denote the *Association* and *Purity* of the labeling function $;5_8$ as ${}_8$ and $\%_8$ respectively. Given the ${}_8$ and $\%_8$ of the labeling function $;5_8$, its confidence value $>=5_8(G)$ is calculated as below,

$$>=5_8(G) = {}_8(G) \times \%_8(G). \tag{6}$$

Labeling confidence aggregation. Flame calculates the confidence value $>=5_8(G)$ for each $;5_8$ in $!$ for a given data instance G. These confidence values are normalized between 0 and 1, and

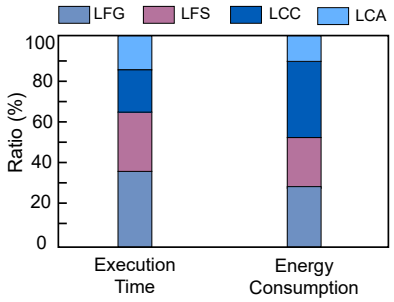


Fig. 4. Execution time and energy consumption breakdown of different componments in Flame. Here, "LFG" is short for Labeling Functions Generation, "LFS" is short for Labeling Functions Self-adaptation, "LCC" is short for Labeling Confidence Calculation, and "LCA" is short for Labeling Confidence Aggregation.

then aggregated together to calculate the overall labeling confidence of all labeling functions as follows,

$$>=5(G) = \max_{\in .'} \{ \tilde{O}_{8=1} \mathbb{1}(;5_8(G) =) \times >=5_8(G) \} \quad (7)$$

where $>=5(G)$ is the aggregated labeling confidence for data instance G, $\mathbb{1}(;5_8(G) =)$ is an indicator function returns 1 if $;5_8(G) =$ and returns 0 otherwise. After $>=5(G)$ is calculated by Equation 7, we have a threshold g to decide if $>=5(G)$ is high enough. If it is higher than g, the label is assigned; Otherwise, the data instance is added into the buffer for further new unseen labels detection (§3.2). The value of g is specified by the programmer. We empirically determine $0''6 \ddot{Y} g \ddot{Y} 0''8$ based on the sensitivity study using datasets listed in Table 1.

4 SYSTEM DESIGN

Flame has four components, *Labeling Functions Generation* (§3.1), *Labeling Functions Self-adaption* (§3.2), *Labeling Confidence Calculation* and *Labeling Confidence Aggregation* (§3.3). Figure 1 shows the execution time and energy consumption breakdown for the four components in Flame. The results are received by labeling 3000 data instances in Samsung S9 by Flame. We can see *Labeling Functions Generation* and *Labeling Functions Self-adaption* together consume more than 60% of time, and *Labeling Confidence Calculation* component consumes about 40% of energy. Given a mobile device with three heterogeneous processing units (CPU, GPU and DSP), we map the four components to the three processing units based on the workload characteristics of the four components and hardware. Furthermore, we use a performance model to decide the optimal way to utilize fast shared-memory in GPU when running the workload of *Labeling Functions Generation*. We also use a performance model to coordinate the usage of CPU and DSP while reducing the wake-up rate of CPU for high performance. We discuss our system design in detail as follows.

4.1 Leverage GPU

Flame uses GPU to run *Labeling Functions Generation* and *Labeling Functions Self-adaption*, because these two components spend more than 60% of time and the clustering with minimal impurity method (§3.1) used by these two components is suitable for parallelism. Therefore, we run these two components on mobile GPU to speedup the labeling task. We do not offload the two components to DSP, because they involve heavy computation (such as square root), which cannot be efficiently processed on DSP [Georgiev et al. 2014]. To reduce execution of *Labeling Functions Generation*, we make the best use of fast shared-memory on GPU to store data instances and centroids, which brings a challenge. In particular, the fast memory has a rather small capacity. For example, in our platform (a Samsung S9 mobile phone), shared memory is only 64KB. We store some data instances and centroids in shared memory, such that they do not have to be repeatedly fetched from slow global-memory to build labeling functions. To host as many data instances in fast memory as possible, we apply a sampling method to approximate data instances without impacting labeling accuracy. In particular, given a data instance (an image), we use spatial sampling which selects every $=$-th row for sampling where $=$ is determined based on the image size [Maier et al. 2019].

There is a non-trivial tradeoff between placing centroids in shared memory and placing data instances in shared memory. To enable high-performance memory accesses to data instances, Flame fetches a batch of data instances ($=_3$) into shared memory and then processes them one by one in shared memory. Leveraging the spatial locality, fetching $=_3$ data instances together causes less global memory accesses. To get the nearest centroid for each data instance in shared memory, Flame must access all centroids. To enable high-performance memory accesses to centroids, Flame also fetches centroids to shared memory batch by batch (the batch size is $=_2$). Placing too many data instances (or too many centroids) in shared memory can cause frequent data movement to fetch centroids (or data instances).

We formulate the above discussion to decide the optimal numbers of data instances ($=_3$) and centroids ($=_2$) to be placed on shared memory as follows. Assume that the total number of data instances to be processed on GPU is $\#_3$, the total number of centroids is $\#_2$, the execution time of processing $=_3$ data instances and $=_2$ centroids on shared memory is $\mathbb{C}$, and the time to transfer $=_3$ data instances and $=_2$ centroids from GPU global memory to shared memory is $\mathbb{C}_3$ and $\mathbb{C}_2$ respectively. The total execution time $)$ to process $\#_3$ data instances and $\#_2$ centroids is modeled as follows. We want to minimize $)$ under the constraint of shared memory capacity ($''4<_{B\ OA43}$).

$$) = \min_{=_3 \geq 0 \bullet =_2 \geq 0} \lceil \frac{\#_3}{=_3} \rceil \cdot \mathbb{C}_3 + \lceil \frac{\#_2}{=_2} \rceil \cdot \mathbb{C}_2 + \lceil \frac{\#_3}{=_3} \rceil \cdot \lceil \frac{\#_2}{=_2} \rceil \cdot \mathbb{C} \bullet \quad (8)$$

$$\text{subject to } =_3 + =_2 \leq \frac{''4<_{B\ OA43}}{3\mathbb{C}\mathbb{C}_{B8I4}} \bullet =_3 \geq 0 \bullet =_2 \geq 0'' \quad (9)$$

where $3\mathbb{C}\mathbb{C}_{B8I4}$ is the size of a data instance. In Equation 8, $\lceil \frac{\#_3}{=_3} \rceil \cdot \mathbb{C}_3$ denotes the time to transfer $\#_3$ data instances from global memory to share memory, $\lceil \frac{\#_2}{=_2} \rceil \cdot \mathbb{C}_2$ denotes the time to transfer $\#_2$ centroids from global memory to share memory, and $\lceil \frac{\#_3}{=_3} \rceil \cdot \lceil \frac{\#_2}{=_2} \rceil \cdot \mathbb{C}$ denotes the execution time to process $\#_3$ data instances and $\#_2$

centroids on GPU. C_3, C_2, and C are measured offline. Flame solves the above programming problem using the ALGLIB [solver [n.d.]] (a cross-platform numerical analysis and data processing library).

4.2 Leverage DSP and CPU

Flame runs *Labeling Confidence Calculation* on DSP (not on CPU or GPU), because *Labeling Confidence Calculation* is the most energy-consuming component: It takes 41.6% of energy consumption of the whole auto-labeling workflow. Compared with GPU and CPU, DSP consumes the least power [Codrescu et al. 2014]. Furthermore, Flame runs *Labeling Confidence Aggregation* on CPU, not on DSP or GPU. Because this component involves a large amount of division and square root computation, which are not supported effectively on DSP [Codrescu et al. 2014]; This component has low thread-level parallelism, making it less efficient to run on GPU either.

The workload execution on CPU and DSP introduces inevitable interaction between CPU and DSP. In particular, the execution of *Labeling Confidence Aggregation* has dependency on *Labeling Confidence Calculation*. Only after *Labeling Confidence Calculation* is done on DSP, *Labeling Confidence Aggregation* can be executed on CPU. We use a common mechanism in mobile phones, the remote procedure call (particularly named *FastRPC* in our evaluation platforms, Samsung S9 and Google Pixel2 mobile phones) for interaction between CPU and DSP.

Wake up CPU by DSP. CPU must be woken up by DSP if the execution on DSP takes too long time [Snapdragon [n.d.]] (e.g., a few seconds) and such wake-up operation on CPU takes 60 mJ, which is very energy-consuming. In Flame, once DSP finishes *Labeling Confidence Calculation* for a batch of data instances, which typically takes 23.6 seconds, the intermediate results generated by DSP must be transferred to CPU cores for *Labeling Confidence Aggregation*. At this moment, DSP must wake up CPU. Compared to the power consumption to wake up CPU, the energy consumption for data transfer between DSP and CPU is relatively small. This is because data transfer between CPU and DSP consists of simply passing data instances to be labeled, remote invocation parameters, and labeling confidence values calculated by DSP, which is typically 74 KB per transfer and consumes only 3.3 mJ. In general, the rate of waking up CPU is critical to the energy consumption of Flame. We must reduce the wake-up frequency to maintain low-energy consumption.

To minimize the wake-up frequency, we define the CPU wake-up interval and formulate it as follows. The CPU wake-up interval is defined as the time duration from the point where a batch of intermediate results is transferred from DSP to CPU to the point where the next batch is transferred. The CPU wake-up interval heavily depends on memory capacity in DSP and data instance size. Larger memory capacity or smaller data instance size causes longer CPU wake-up interval, and vice versa. Furthermore, DSP processes data instances batch by batch, and processes data instances within the same batch in parallel. Given DSP memory capacity ($"4<_{3B?}$), memory consumed by DSP invocation parameters ($"4<_{?OAO}$), the size of each data instance ($3OCO_{B814}$), and the number of threads used by DSP ($=_C$), the number of batches

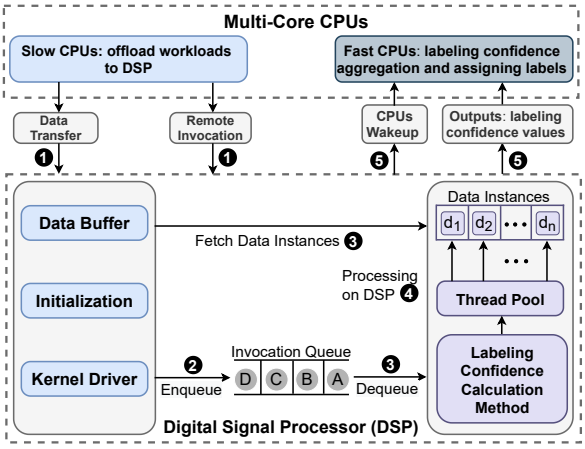


Fig. 5. The interaction between CPU and DSP.

of data instances is , where $= \lceil \frac{"4<_{3B?} - "4<_{?OAO}}{3OCO_{B814} \times =_C} \rceil$. The CPU wake-up interval $\triangle)$ is formulated as follows.

$$\triangle) = \mathbb{W} + \sum_{8=0}^{\tilde{O}1} \{ \max_{9 \in [0 \bullet =_C - 1]} C_{89} \} + d \tag{10}$$

where $\mathbb{W}$ is the time to transfer data instances and remote invocation parameters from CPU to DSP, and d is the time to transfer calculated labeling confidence values for each data instance from DSP to CPU. The CPU wake-up interval includes the time to process a batch of data instances in DSP, which is formulated as $\max_{9 \in [0 \bullet =_C - 1]} C_{89}$ in Equation10, where C_{89} is the execution time of 9^C thread for 8^C batch of data instances, where $8 \in [0 \bullet - 1]$ and $9 \in [0 \bullet =_C - 1]$, $=_C$ is the total number of threads in DSP. $\mathbb{W}$ and d are measured offline. Flame determines the maximum CPU wake-up interval by using ALGLIB on Equation 10.

Data transfer for DSP. DSP needs to load data to be labeled from CPU main memory to DSP local memory; DSP also needs to transfer labeling confidence values for each data instance between CPU and DSP. To load data from CPU main memory, we use slow CPU cores, because we find that using fast CPU cores often causes a crash on DSP because of a run-out-of-memory error, and using slow CPU cores does not have this problem. Such an error happens because DSP cannot timely process the data in the local memory before the new data comes in. We use fast CPU cores to transfer data between CPU and DSP.

Overall workflow. Figure 5 generally depicts the interaction between CPU and DSP. It includes five stages. At Stage one, the slow CPU cores transfer data instances to be labeled and initiates DSP remote invocation. At Stage two, the *FastRPC* kernel driver receives the remote invocations and enqueues them up to wait for the response from DSP. A buffer on DSP is used to store the data instances to be labeled. At Stage three, once DSP is ready for labeling the data instance, it dequeues invocations from the invocation queue and dispatches them for processing. At Stage four, DSP computes the labeling confidence values of data instances in parallel. At last, the labeling confidence values (i.e., the intermediate

Table 1. Characteristics of datasets.

Datasets	Application	# features	# labels	# instances
MNIST	Classification	28×28	10	70,000
EMNIST	Classification	28×28	62	814,255
ImageNet	Classification	224×224	100	60,000
Cifar100	Classification	$3 \times 28 \times 28$	100	60,000
UCF50	Recognition	320×240	50	15,000
UCF101	Recognition	320×240	101	50,500

results) are transferred to fast CPU cores for labeling confidence aggregation and labeling.

4.3 Implementation

We implement Flame using C++ with Native Development Kit (NDK) on Android 9.0 and Android 8.0. The system is evaluated on two mobile phones (Samsung S9 with Snapdragon 845 SoC and Google Pixel2 with Snapdragon 835 SoC). Our implementation includes about 8,000 lines of code in total. In our mobile platforms, we have three types of mobile processors, which are GPU, fast CPU and slow CPU. Each mobile platform has mobile GPU, fast CPU, slow CPU, and DSP.

5 EVALUATION

We compare Flame with other baseline methods, in terms of labeling quality and performance. Our evaluation aims to achieve the following four goals.

- Is the labeling quality of Flame better than that of the baseline methods? (§5.1)
- Does Flame effectively utilize hardware heterogeneity of mobile processors? (§5.2 and §5.3)
- Does each component of the Flame effectively boost the overall labeling quality? (§5.4)
- Does Flame have imperceivable influences on user experiences on mobile devices? (§5.5)

Experimental Setup. We use two mobile systems.

- Samsung S9: It uses Qualcomm Snapdragon 845 SoC and Android 9.0 Pie operating system (OS). The 845 SoC includes a 4-core fast CPU, a 4-core slow CPU, an Adreno 630 mobile GPU, and a Hexagon 685 DSP. The fast and slow CPU cores are different in terms of frequency, cache hierarchy, and instruction scheduling. The CPU architecture has 4x Kryo 385 cores (Cortex-A75) at up to 2.8 GHz (max) for performance and 4x Kryo 385 at 1.8 GHz (max) for efficiency.
- Google Pixel2: It uses Qualcomm Snapdragon 835 SoC and Android 8.0 Oreo OS. The 835 SoC includes a 4-core fast CPU, a 4-core slow CPU, an Adreno 540 mobile GPU, and a Hexagon 682 DSP. The CPU architecture has 4x Kryo 280 at 2.45 GHz (max) for performance and 4x Kryo 280 at 1.9 GHz (max) for efficiency.

Datasets. Table 1 describes the six datasets used to evaluate Flame. Those datasets are commonly used for classification or recognition, which are common applications in mobile devices. We test Flame on both static and dynamic datasets. (1) Static Datasets. All the labels are known and the data instances are not incrementally generated. (2) Dynamic Datasets. We rearrange instances in each dataset to emulate a dynamic environment where data instances are incrementally generated with previously unseen labels. For MNIST [Lecun [n.d.]], we randomly choose two labels as known, and the rest eight labels as unknown and needed to be detected by Flame. For EMNIST [Cohen et al. 2017], we randomly choose six labels as known, and the rest 56 labels as unknown and needed to be detected. For Cifar100 [toronto [n.d.]] and mini-ImageNet [Vinyals et al. 2016], we randomly choose ten labels as known, and the rest 90 labels as unknown. UCF50 [Reddy and Shah 2013] and UCF101 [Soomro et al. 2012] datasets contain 59, and 101 types of human activities respectively, and they also contain 13,421 short videos created for activity recognition. We use the ffmpeg [github. [n.d.]] tool to extract raw images from the above video datasets and feed the unstructured images to Flame sequentially. We select six and ten types of activities as known for UCF50 and UCF101 respectively, and the rest types of activities as unknown and needed to be detected.

For those data instances with known labels, we sample a part of them to form a subset $_!$. $_!$ is used to build labeling functions $!$ for Flame at the beginning of auto-labeling. $_!$ takes up to 5% of all the data instances in a dataset. The rest of the dataset ($_*$) is used to simulate the scenario where new data is incrementally generated for auto-labeling.

Baselines. We compare Flame with four baselines: (1) Boosting (AdaBoost), which uses the labeled data to generate one complex decision tree or multiple, simple decision trees to label unlabelled data; (2) Semi-supervised learning, which uses both the labeled and unlabeled data to assign labels; (3) Transfer learning, which uses MobileNet_V3 [Howard et al. 2017] pre-trained on ImageNet for labeling; (4) Snuba [Ratner et al. 2017], a state-of-the-art work for auto-labeling on servers. Snuba cannot run on mobile devices because of the lack of a set of system libraries. So we just report its labeling accuracy.

Evaluation metrics. For the dynamic datasets, we use the following metrics to evaluate the labeling quality. (1) ***Accuracy(%)*** $= \frac{\#_{=4F}+\#_{4G8BC}}{\#}$, where $\#_{=4F}$ is the total number of data instances with unknown labels correctly labeled, $\#_{4G8BC}$ is the total number of data instances with known labels correctly labeled, and $\#$ is total number of data labeled by the system. Let $\%$ represent the the number of data instances that should be assigned with known labels but is mislabeled with unknown labels (i.e., previously unknown labels); Let $\#$ represent the number of data instances that should be assigned with unknown labels but is mislabeled with known labels; Let $\#_{;}$ represent the number of data instances assigned with unknown labels. We use the following two metrics based on $\%$, $\#$, and $\#_{;}$. (2) $\boldsymbol{M_{new}}$, the percentage of data instances that should be assigned with unknown labels but is mislabeled with known labels, ($"_{=4F} = \frac{\# \times 100}{\#_{;}}$); and (3) $\boldsymbol{F_{new}}$: the percentage of data instances that should be assigned with known labels but is mislabeled with unknown labels, ($_{=4F} = \frac{\% \times 100}{\# - \#_{;}}$).

Furthermore, we measure the performance of Flame in terms of execution time and energy consumption. We measure the execution time using Flame to label 5000 data instances from each

Table 2. Comparison of labeling accuracy on dynamic datasets between various baselines and different versions of Flame. "ACC" is the accuracy of the labeling results; The notation "-" denotes failure of detecting unknown labels. $"_{=4F}$ is the percentage of data that should be assigned with unknown labels but is mislabeled with known labels; $_{=4F}$ is the percentage of data that should be assigned with known labels. For $"_{=4F}$ and $_{=4F}$, a lower value indicates a better result.

Methods	MNIST			EMNIST			ImageNet			Cifar100			UCF50			UCF101		
	Acc	S_{new}	L_{new}	Acc	S_{new}	L_{new}	Acc	S_{new}	L_{new}	Acc	S_{new}	L_{new}	Acc	S_{new}	L_{new}	Acc	S_{new}	L_{new}
Boosting	62.8	78.9	-	56.2	82.3	-	57.2	67.8	-	64.6	72.3	-	64.8	82.6	-	63.4	67.3	-
Transfer	76.8	62.4	-	72.6	64.8	-	65.7	58.3	-	70.6	69.4	-	70.2	64.4	-	66.7	61.9	-
Semi-s	71.5	48.3	9.8	70.7	39.7	11.3	62.4	40.4	9.4	71.7	38.4	8.9	67.2	38.3	8.8	62.4	41.7	12.9
Snuba	81.2	42.5	8.3	78.2	31.2	8.5	70.0	34.7	8.7	75.3	31.8	6.4	73.3	34.8	8.2	71.2	37.1	9.8
Flame_C	**87.3**	23.6	**3.7**	85.4	**22.8**	**5.9**	86.2	31.1	7.9	85.7	23.9	5.7	86.3	34.4	**6.4**	83.7	28.4	8.7
Flme_CG	86.2	**22.7**	4.5	**86.1**	24.7	6.2	85.8	**29.9**	**7.6**	**87.5**	25.0	**5.2**	85.7	**32.3**	6.6	82.4	27.7	9.7
Flame_Full	86.7	22.9	5.7	85.3	23.9	6.9	**87.2**	30.3	8.1	85.8	**23.7**	6.1	**87.1**	33.7	6.9	**84.8**	**26.9**	**8.3**

Table 3. Comparison the accuracy of the discriminative model when training on the labeled datasets by baselines and Flame. The column "Highest Accuracy of Baseline" means the highest accuracy achieved by the baselines. The accuracy lift over the six datasets shows that labeling results of Flame improves the model accuracy.

Datasets	Highest Accuracy of Baseline (%)	Flame (%)	Lift
MNIST	81.1	88.2	+7.1
EMNIST	76.6	84.1	+7.7
ImageNet	75.2	84.6	+9.4
Cifar100	72.3	79.6	+6.3
UCF50	78.3	85.2	+6.9
UCF101	76.5	83.7	+7.2

dataset in Table 1. We also measure the impact of Flame on user interactions when running Flame on mobile systems.

5.1 Labeling Quality

Labeling quality on dynamic datasets. Table 2 shows the results. In general, Flame performs best. We conclude the following. (a) Flame outperforms the boosting and semi-supervised methods. The average labeling accuracy of Flame is higher than that of the two methods by 25.2% and 18.5% respectively. The reason is that Flame has the ability to immediately detect new coming unknown labels. This advantage is obvious when the number of unknown labels in the dataset is large (e.g., ImageNet, Cifar100, UCF101); (b) Flame outperforms the transfer learning method up to by 16.1%. This is because the pre-trained model is directly used for labeling without learning a representation of the data from scratch; (c) Flame outperforms Snuba by 11.8%. Snuba's labeling quality is based on trained labeling functions and it fails to assign unknown labels to data instances, because Snuba's labeling functions can not adapt to dynamic datasets; (d) The labeling quality of different versions of Flame (i.e., CPU Only, CPU and GPU, and the full featured version which uses CPU, GPU and DSP) does not vary significantly. The slight variance in the labeling quality comes from randomness in execution order of parallel threads in the heterogeneous computing environment.

End-to-end impact. We evaluate the effect of using the auto-labeled data to train DNN models. In our evaluation, we use a DNN model with three fully-connected layers and each layer contains 128 neurons, which is commonly deployed in mobile phones [Sahin 2021]. We stop the training process until the DNN model's loss value converges. Table 3 shows the accuracy lift after using auto-labeled data from Flame. In general, the model accuracy is improved by up to 9.4% over six datasets.

5.2 Analysis on Execution Time

We compare the execution time of different labeling methods. We also show how our system design (§4) can reduce the execution time of Flame.

Comparison with different versions of Flame. Figure 6 presents execution time of labeling 5,000 data instances on Samsung S9 and Google Pixel2. We use three execution strategies to evaluate the effectiveness of Flame: (1) using CPU Only; (2) using CPU and GPU; and (3) using CPU, GPU and DSP (i.e., the full-featured Flame). Figure 6 shows that compared with using CPU Only, using CPU and GPU leads to an average of 2"1× speedup, because of using GPU. Using the full-featured Flame, there is average 6"8× performance improvement, because of using GPU and DSP. Flame fully taps the capability of hardware heterogeneity in mobile processors. Such a large reduction in execution time is not paid by using larger energy consumption, discussed as follows.

Comparison with the baselines. Figure 7 presents the execution time of labeling 5,000 data instances on Samsung S9 and Google Pixel2 using different methods. Compared with the baselines, Flame's execution time is the shortest. Figure 7 shows that Flame reduces the execution time by average 5"6×, 4"2×, and 3"8× respectively, compared with the transfer learning, semi-supervised learning, and the boosting methods. Such a reduction in execution time is because of two reasons: (1) Flame uses the lightweight clustering with minimal impurity method (§3.1) to build the labeling functions; and (2) Flame uses Algorithm 2 to merge the close prototypes to reduce the number of prototypes contained in each labeling function, which significantly reduces computation overhead in Flame.

5.3 Analysis on Energy Consumption

We analysis the energy consumption of different labeling methods over the six datasets. We also analyze how full-featured Flame saves energy, compared with the other two versions of Flame.

Comparison with different versions of Flame. Figure 8 shows the energy consumption of the three strategies (CPU Only, CPU and GPU, and the full-featured Flame) on Samsung S9 and Google

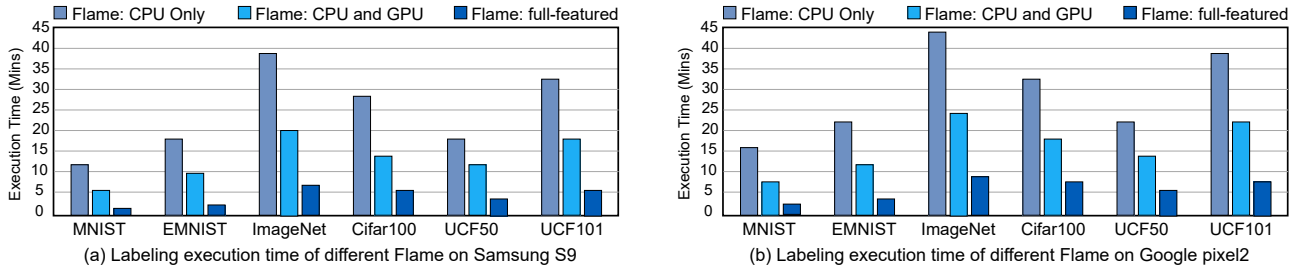


Fig. 6. Comparison between the three versions of Flame in terms of execution time.

Table 4. Ablation study results of different components in Flame. We measure how the evaluation metrics change when involving different components in Flame. $"_{=4F}$ is the percentage of data that should be assigned with unknown labels but is mislabeled with known labels; $_{=4F}$ is the percentage of data that should be assigned with known labels but mislabeled with unknown labels.

Datasets	Labeling Functions Generation			Labeling Functions Self-adaption			Labeling Results Guarantees		
	Accuracy	$"_{=4F}$	$_{=4F}$	Accuracy	$"_{=4F}$	$_{=4F}$	Accuracy	$"_{=4F}$	$_{=4F}$
MNIST	+14.4	-5.1	-8.3	+7.8	-7.2	-5.1	+6.5	-4.2	-3.5
EMNIST	+18.5	-8.8	-12.2	+12.5	-18.3	-17.1	+9.1	-7.7	-6.4
ImageNet	+21.3	-7.8	-13.7	+14.2	-25.4	-14.4	+9.1	-8.2	-8.4
Cifar100	+24.1	-10.4	-13.1	+14.6	-27.3	-20.6	+12.9	-8.8	-7.7
UCF50	+17.7	-9.2	-11.0	+8.7	-13.5	-11.5	+11.1	-7.2	-6.1
UCF101	+19.5	-7.4	-12.8	+9.7	-23.3	-18.6	+12.7	-7.4	-9.8

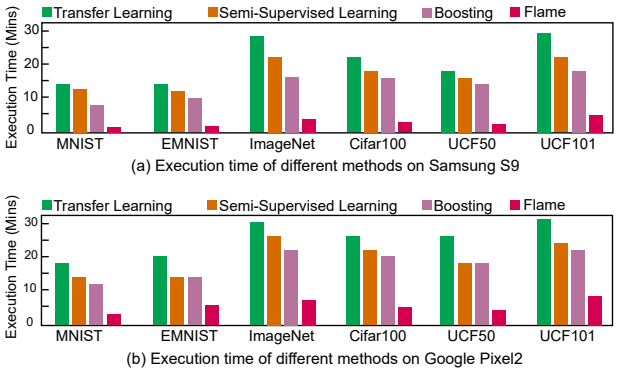


Fig. 7. Comparison between different labeling methods on two mobile platforms in terms of execution time.

Pixel2 . The energy consumption in Figure 8 is normalized by that of running Flame on Google Pixel2 with only CPU. Figure 8 shows that the average energy consumption of using the strategies of "CPU and GPU" and full-featured Flame is 73.4%, and 46.2% of that of Flame using only CPU, respectively. Full-featured Flame uses the least energy, which shows the advantage of using DSP.

Comparison with the baselines. Figure 9 shows energy consumption of the semi-supervised learning, transfer learning, boosting, and Flame. In Figure 9, energy results are normalized by the energy consumption of using the transfer learning. In general, Flame consumes the least energy in all datasets. This is largely because we offload the most energy-consuming component of Flame (*Labeling Confidence Calculation*) to DSP (§4.2).

5.4 Micro-Benchmarking Results

We evaluate the components of Flame (i.e., *Labeling Functions Generation*, *Labeling Functions Self-adaption*, and *Labeling Results Guarantees*) and show how each component can affect the auto-labeling quality. Table 4 shows the results.

Labeling functions generation. We quantify the contribution of *Labeling Functions Generation* (§3.1) to the labeling quality. Table 4 shows that (1) compared with the other two components, *Labeling Functions Generation* contributes more to the labeling accuracy, and (2) including *Labeling Functions Generation* improves the accuracy by up to 24.1% (Cifar100). We conclude *Labeling Functions Generation* component is useful for improving the labeling accuracy, especially for those datasets with a large numbers of unknown labels (e.g., Cifar100 and ImageNet). Additionally, $"_{=4F}$ and $_{=4F}$ are decreased by up to 10.4% and 13.1% respectively after involving *Labeling Functions Generation*. Therefore, this component also boosts the capability of recognizing unknown labels and known labels.

Labeling functions self-adaption. We quantify the contribution of *Labeling Functions Self-adaption* component (§3.2) to the labeling quality. Table 4 shows that (1) *Labeling Functions Self-adaption* leads to more reduction in $"_{=4F}$ and $_{=4F}$ than the other two components, indicating that it is more efficient to recognize unknown labels and known labels than the other two components. (2) *Labeling Functions Self-adaption* is more helpful to those datasets with a larger number of unknown labels. For example, Cifar100 has 100 labels and MNIST has 10 labels. The reduction of $"_{=4F}$ and $_{=4F}$ on Cifar100 dataset is 27.3% and 20.6%

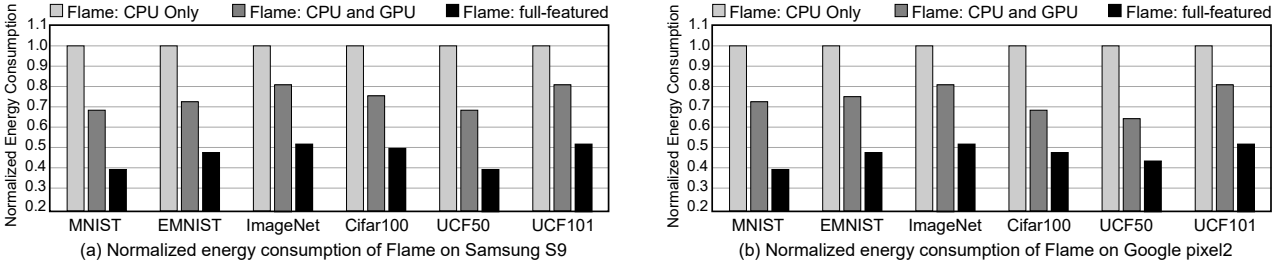


Fig. 8. Comparison between the three versions of Flame in terms of energy consumption. The energy results are normalized by the energy consumption of using full-featured Flame on each dataset.

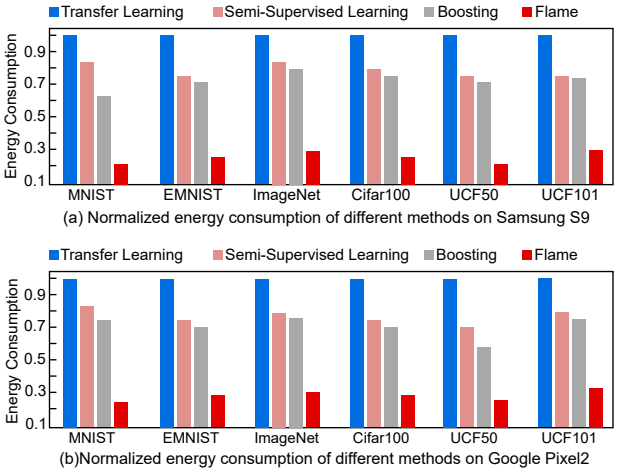


Fig. 9. Comparison between different labeling methods on two mobile platforms in terms of energy consumption. The energy results are normalized by energy consumption of using the transfer learning method on each dataset.

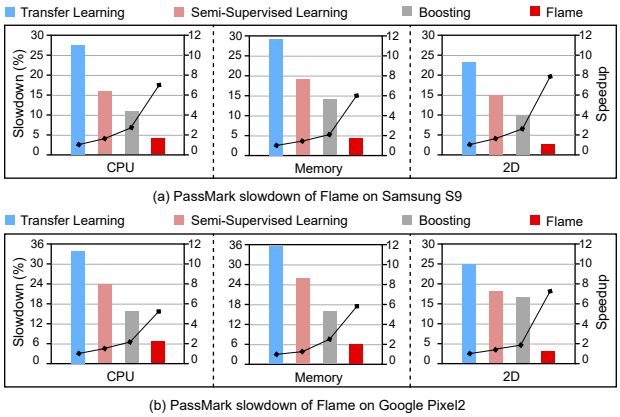


Fig. 10. PassMark slowdown with auto-labeling running in background. We compare Flame with other labeling methods in terms of PassMark slowdown.

respectively, while the reduction on Labelme dataset is only 7.2% and 5.1% respectively.

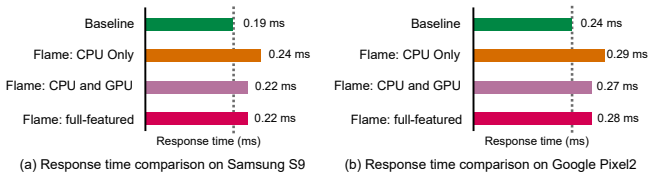
Labeling Results Guarantees. We quantify the contribution of *Labeling Confidence Aggregation* (§3.3) to the labeling quality. Table 4 shows that (1) *Labeling Confidence Aggregation* improves the accuracy by up to 12.9%, and (2) $"_{=4F}$ and $_{=4F}$ are decreased by up to 8.8% and 9.8% respectively after involving *Labeling Confidence Aggregation*.

5.5 Evaluation on User Experience

We run Flame in background as a service in mobile phones, in order to avoid the impact of using Flame on the user applications running in foreground. In this section, we evaluate the impact of using Flame on the user experience. Android always sets higher priority to the foreground applications compared to the background applications to provide prompt response to user input, more resources are allocated to the foreground applications. Since labeling is an intensive task, running it in background can reduce its impact on user experience of other applications on the devices. Therefore, We run Flame in background.

Impact on other applications. We evaluate how using different labeling methods impacts the performance of another application running in foreground. We use a benchmark PassMark [PassMark.2015. [n.d.]] as the user application. We use PassMark, because it involves CPU tests, memory tests, and graphics tests, representing workloads with various characterization. Figure 10 shows the slowdown of PassMark while running various labeling methods in background. In general, Flame has the least impact on PassMark. (a) For the CPU tests, the full-featured Flame leads to up to 4.6% and 6.1% slowdown in PassMark on Samsung S9 and Google Pixel2 respectively. The CPU slowdown of the full-featured Flame is up to 7"1× better than the baselines. (b) For the memory tests, the full-featured Flame leads to up to 4.7% and 5.9% slowdown in PassMark on Samsung S9 and Google Pixel2 respectively. Compared to the baselines, the full-featured Flame achieves up to 5"7× and 5"9× slowdown improvement on Samsung S9 and Google Pixel2 respectively. (c) For 2D graphic experience, the full-featured Flame leads to up to 3.1% and 4.3% slowdown in PassMark on Samsung S9 and Google Pixel2 respectively. Compared to the baselines, the slowdown improvement for 2D graphic experience is up to 7"9× and 7"4× on Samsung S9 and Google Pixel2 respectively.

Impact on the interaction between the user and mobile devices. We aim to find out whether running Flame affects a user's



(a) Response time comparison on Samsung S9 (b) Response time comparison on Google Pixel2

Fig. 11. Impacts of using Flame on user experience. The response time in the figure shows how fast the application reacts to user input events.

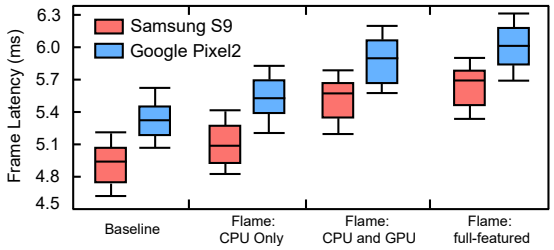


Fig. 12. The frame latency distribution of using different versions of Flame on user experience. Baseline is the frame latency without running Flame.

interactive experience with the mobile device. We perform our tests using an application that models the user interaction with a device by taking a user's input from the touch screen and rendering a response on the screen. By analyzing the rendering latency in response to the user input, we can quantitatively understand the interactivity. We use Android dumpsys tool to measure the latency. We use two metrics to quantify the latency: (1) response time, i.e., the time for the mobile device to process the user input; and (2) frame latency, i.e., the time to render a new frame based on the user input. Figure 11 lists the 95th percentile response time and Figure 12 lists the frame latency distribution under four different configurations. These configurations are (1) without running Flame (baseline), (2) running Flame on CPU (Flame: CPU Only), (3) running Flame on CPU and GPU (Flame: CPU and GPU), and (4) running Flame on CPU, GPU and DSP (Flame: full-featured).

Figure 11 shows that the response time increases by only 0.03 ms and 0.04 ms when running full-featured Flame on Samsung S9 and Google Pixel2. Meanwhile, as Figure 12 shows, the median frame latency only increases by 0.75 ms and 0.64 ms when running full-featured Flame on Samsung S9 and Google Pixel2 respectively. Such increase is very small, compared with the minimum threshold of user-perceivable latency, 100 ms [Chen et al. 2019]. Figure 12 also shows that the additional frame latency caused by Flame: the full-featured Flame causes negligible latency, compared with using CPU and GPU to run Flame. This means that involving DSP into Flame does not affect the user's graphic experience. For the above testing results, we conclude that the impact of the labeling tasks is not perceivable by users, because the minimum threshold of perceivable latency is 100 ms [Chen et al. 2019].

6 RELATED WORK

Automatic labeling. We provide an overview of automatic labeling methods, which label data automatically based on generated labeling functions using both labeled and unlabeled data. The main challenge of auto-labeling is to build proper labeling functions that can cover the most data instances in the dataset [Bach et al. 2017; Blum and Mitchell 1998; Varma et al. 2017a,b; Varma and Ré 2018; Wang and Rudin 2015; Wang et al. 2015; **?**; **?**]. Labeling functions with high quality are difficult to be acquired. In [Varma and Ré 2018], Varma et. al propose a method that uses machine learning models to build labeling functions under weak supervision. Other work [Hastie et al. 2009; Ratner et al. 2017; Wang et al. 2014; Weiss et al. 2016] uses distant supervision [Hastie et al. 2009; Joglekar et al. 2015; Weiss et al. 2016], in which the training sets are generated with the help of external resources, such as knowledge bases. Some recently proposed approaches [Ratner et al. 2017; Sheng et al. 2008] demonstrate the use of proper strategies to boost the labeling quality by ensemble technique [Bach et al. 2017; Lao and Cohen 2010]. The existing approaches focus on static datasets with fixed size and pre-determined number of labels. Our work focuses on the dynamically increased datasets on mobile devices. Our work has the capability to identify new labels that are never seen before. To solve the hardware resource constraint problem on mobile devices, we leverage processor heterogeneity to efficiently run the auto-labeling workload. Our work not only labels dataset with high quality, but also is highly feasible to be deployed on mobile devices.

Optimization of machine learning on mobile devices. Recently, there are many existing efforts that optimize the performance of machine learning models on both server side and edge side, including dynamic resource scheduling [Di et al. 2021; Guo et al. 2016; He et al. 2021; Lane et al. 2016; LiKamWa and Zhong 2015; Liu et al. 2021a, 2019a, 2021b, 2020; Ogden and Guo 2018], computation pruning [Gordon et al. 2018; Li et al. 2018a, 2019; Ma et al. 2020; Niu et al. 2020], model partitioning [Jeong et al. 2018; Kang et al. 2017; Lane et al. 2016; Li et al. 2018b], model compression [Fang et al. 2018; He et al. 2018; Liu et al. 2018], coordination with cloud servers [Georgiev et al. 2016; Kang et al. 2017] and memory management [Fang et al. 2018; LiKamWa and Zhong 2015]. Flame is different from them, because it focuses on data labeling task on heterogeneous mobile processors. In particular, DeepX [Lane et al. 2016] proposes a number of resource scheduling algorithms to decompose DNNs into different subtasks on mobile devices. LEO [Georgiev et al. 2016] introduces a power-priority resource scheduler to maximize energy efficiency. NestDNN [Fang et al. 2018] compresses and prunes models based on the available hardware resource on mobile devices.

7 CONCLUSION

Auto-labeling on mobile devices is critical to enable ML training on mobile devices. However, it is challenging to enable auto-labeling on mobile devices, because of unique data characteristics on mobile devices and heterogeneity of mobile processors. In this paper, we introduce the first auto-labeling system for mobile devices, named Flame, to address the above problem. Flame includes auto-labeling algorithms to detect unknown labels from dynamic data; It also includes an execution engine that executes labeling workloads on heterogeneous mobile processors.

REFERENCES

Stephen H Bach, Bryan He, Alexander Ratner, and Christopher Ré. 2017. Learning the structure of generative models without labeled data. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 273–282.

Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*. 92–100.

Yitao Chen, Saman Biookaghazadeh, and Ming Zhao. 2019. Exploring the capabilities of mobile devices in supporting deep learning. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 127–138.

Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. 2014. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro* 34, 2 (2014), 34–43.

Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2921–2926.

Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 503–516.

Heungsik Eom, Renato Figueiredo, Huaqian Cai, Ying Zhang, and Gang Huang. 2015. Malmos: Machine learning-based mobile offloading scheduler with online training. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 51–60.

Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 115–127.

Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. 2014. Dsp. ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. 295–309.

Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. 2016. 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)*. ACM, 320–333.

github. [n.d.]. Ffmpeg software development kit. https://github.com/tanersener/mobile-ffmpeg.

Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. 2018. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1586–1595.

Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. 2016. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

Daniel Haas, Jiannan Wang, Eugene Wu, and Michael J Franklin. 2015. Clamshell: Speeding up crowds for low-latency data labeling. *Proceedings of the VLDB Endowment* 9, 4 (2015), 372–383.

Ahsanul Haque, Latifur Khan, Michael Baron, Bhavani Thuraisingham, and Charu Aggarwal. 2016. Efficient handling of concept drift and concept evolution over stream data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 481–492.

Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. 2009. Multi-class adaboost. *Statistics and its Interface* 2, 3 (2009), 349–360.

Xin He, Jiawen Liu, Zhen Xie, Hao Chen, Guoyang Chen, Weifeng Zhang, and Dong Li. 2021. Enabling energy-efficient DNN training on hybrid GPU-FPGA accelerators. In *Proceedings of the ACM International Conference on Supercomputing*. 227–241.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 784–800.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. 2018. IONN: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*. 401–411.

Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. 2015. Comprehensive and reliable crowd assessment algorithms. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 195–206.

Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 615–629.

Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE Press, 23.

Ni Lao and William W Cohen. 2010. Relational retrieval using a combination of path-constrained random walks. *Machine learning* 81, 1 (2010), 53–67.

Yann Lecun. [n.d.]. The Mnist Batabase. http://yann.lecun.com/exdb/mnist/.

Dawei Li, Xiaolong Wang, and Deguang Kong. 2018a. Deeprebirth: Accelerating deep neural network execution on mobile devices. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

En Li, Zhi Zhou, and Xu Chen. 2018b. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*. 31–36.

Hongjia Li, Ning Liu, Xiaolong Ma, Sheng Lin, Shaokai Ye, Tianyun Zhang, Xue Lin, Wenyao Xu, and Yanzhi Wang. 2019. ADMM-based weight pruning for real-time deep learning acceleration on mobile devices. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. 501–506.

Robert LiKamWa and Lin Zhong. 2015. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 213–226.

Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. 2021a. Athena: high-performance sparse tensor contraction sequence on heterogeneous memory. In *Proceedings of the ACM International Conference on Supercomputing*. 190–202.

Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. 2019a. Runtime Concurrency Control and Operation Scheduling for High Performance Neural Network Training. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 188–199.

Jie Liu, Jiawen Liu, Wan Du, and Dong Li. 2019b. Performance Analysis and Characterization of Training Deep Learning Models on Mobile Devices. *CoRR* (2019). arXiv:1906.04278 http://arxiv.org/abs/1906.04278

Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021b. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 318–333.

Jiawen Liu, Zhen Xie, Dimitrios Nikolopoulos, and Dong Li. 2020. {RIANN}: Real-time incremental learning with approximate nearest neighbor on mobile devices. In *2020 {USENIX} Conference on Operational Machine Learning (OpML 20)*.

Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. 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)*. ACM, 389–400.

Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. 2020. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5117–5124.

Daniel Maier, Nadjib Mammeri, Biagio Cosenza, and Ben Juurlink. 2019. Approximating Memory-bound Applications on Mobile GPUs. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 329–335.

Tom Mitchell, William Cohen, Estevam Hruschka, Partha Talukdar, Bishan Yang, Justin Betteridge, Andrew Carlson, Bhanava Dalvi, Matt Gardner, Bryan Kisiel, et al. 2018. Never-ending learning. *Commun. ACM* 61, 5 (2018), 103–115.

Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 907–922.

Samuel S Ogden and Tian Guo. 2018. {MODI}: Mobile Deep Inference Made Efficient by Edge Computing. In *Workshop on Hot Topics in Edge Computing (HotEdge)*.

PassMark.2015. [n.d.]. PassMark Software - PerformanceTest System Benchmarks. http://www.passmark.com/baselines/index.php.

Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment* 11, 3 (2017), 269–282.

Kishore K Reddy and Mubarak Shah. 2013. Recognizing 50 human action categories of web videos. *Machine vision and applications* 24, 5 (2013), 971–981.

Sahin. 2021. Introduction to Apple ML Tools. In *Develop Intelligent iOS Apps with Swift*. Springer, 17–39.

Victor S Sheng, Foster Provost, and Panagiotis G Ipeirotis. 2008. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 614–622.

Snapdragon. [n.d.]. qualcomm cpu sleep benchmarking. https://developer.qualcomm.com/docs/snpe/benchmarking.html.

Qp solver. [n.d.]. Quadratic programming solving kit. https://github.com/hjkuijf/ALGLIB.

Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. 2012. UCF101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402* (2012).

toronto. [n.d.]. The Cifar100 Batabase. https://www.cs.toronto.edu/~kriz/cifar.html.

Paroma Varma, Bryan He, Payal Bajaj, Imon Banerjee, Nishith Khandwala, Daniel L Rubin, and Christopher Ré. 2017a. Inferring generative model structure with static analysis. *Advances in neural information processing systems* 30 (2017), 239.

Paroma Varma, Bryan D. He, Payal Bajaj, Imon Banerjee, Nishith Khandwala, Daniel L. Rubin, and Christopher Ré. 2017b. Inferring Generative Model Structure with Static Analysis. *Advances in neural information processing systems* 30 (2017), 239–249.

Paroma Varma and Christopher Ré. 2018. Snuba: Automating Weak Supervision to Label Training Data. *PVLDB* 12 (2018), 223–236.

Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. 2016. Matching networks for one shot learning. *arXiv preprint arXiv:1606.04080* (2016).

Fulton Wang and Cynthia Rudin. 2015. Falling rule lists. In *Artificial Intelligence and Statistics*. 1013–1022.

Tong Wang, Cynthia Rudin, Finale Doshi-Velez, Yimin Liu, Erica Klampfl, and Perry MacNeille. 2015. Or's of and's for interpretable classification, with application to context-aware recommender systems. *arXiv preprint arXiv:1504.07614* (2015).

William Yang Wang, Kathryn Mazaitis, and William W Cohen. 2014. Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. 1199–1208.

Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* 3, 1 (2016), 9.

Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. 2019. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344.

Jingru Yang, Ju Fan, Zhewei Wei, Guoliang Li, Tongyu Liu, and Xiaoyong Du. 2018. Cost-effective data annotation using game-based crowdsourcing. *Proceedings of the VLDB Endowment* 12, 1 (2018), 57–70.