
# Decentralized Application-Level Adaptive Scheduling for Multi-Instance DNNs on Open Mobile Devices

Hsin-Hsuan Sung and Jou-An Chen, *Department of Computer Science, North Carolina State University;* Wei Niu, Jiexiong Guan, and Bin Ren, *Department of Computer Science, William & Mary;* Xipeng Shen, *Department of Computer Science, North Carolina State University*

https://www.usenix.org/conference/atc23/presentation/sung

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

# Decentralized Application-Level Adaptive Scheduling for Multi-Instance DNNs on Open Mobile Devices

Hsin-Hsuan Sung[1], Jou-An Chen[1], Wei Niu[2], Jiexiong Guan[2], Bin Ren[2], and Xipeng Shen[1]

[1]Department of Computer Science, North Carolina State University
[2]Department of Computer Science, William & Mary

## Abstract

As more apps embrace AI, it is becoming increasingly common that multiple Deep Neural Networks (DNN)-powered apps may run at the same time on a mobile device. This paper explores scheduling in such multi-instance DNN scenarios, on general *open* mobile systems (e.g., common smartphones and tablets). Unlike *closed* systems (e.g., autonomous driving systems) where the set of co-run apps are known beforehand, the user of an *open* mobile system may install or uninstall arbitrary apps at any time, and a centralized solution is subject to adoption barriers. This work proposes the first-known decentralized application-level scheduling mechanism to address the problem. By leveraging the adaptivity of Deep Reinforcement Learning, the solution is shown to make the scheduling of co-run apps converge to a Nash equilibrium point, yielding a good balance of gains among the apps. The solution moreover automatically adapts to the running environment and the underlying OS and hardware. Experiments show that the solution consistently produces significant speedups and energy savings across DNN workloads, hardware configurations, and running scenarios.

## 1 Introduction

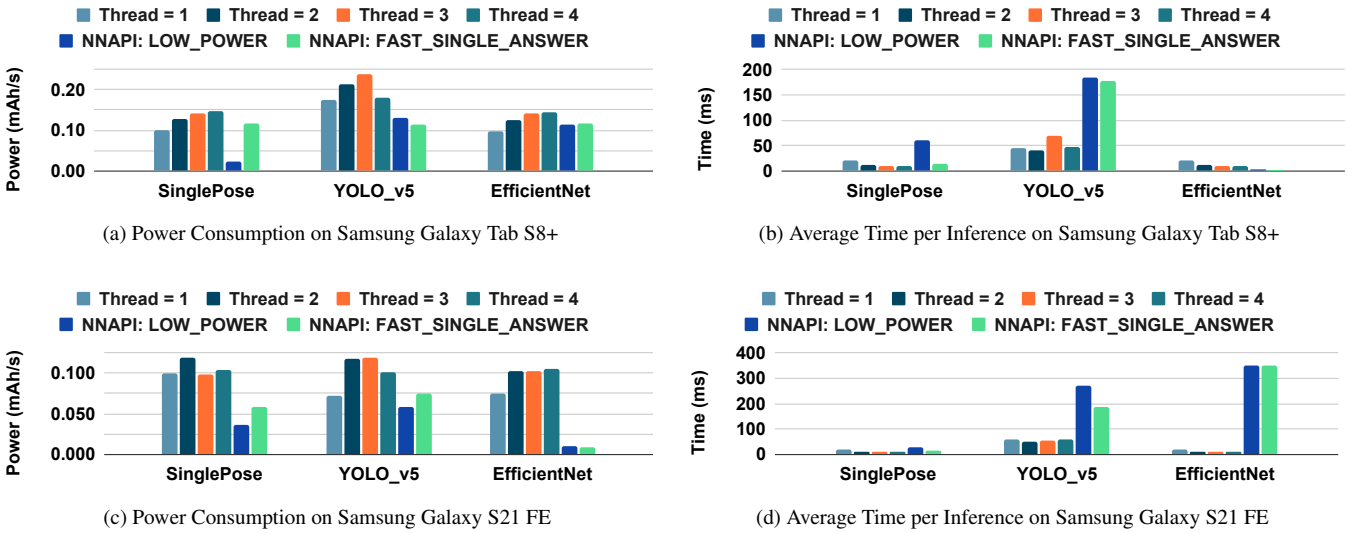
Deep Neural Networks (DNN) have attained remarkable success in various tasks. Recent years have witnessed increasing adoption of DNNs in mobile devices, thanks to the advancement in DNN compression [11, 18, 19], the increasing concerns on privacy, and the demands for real-time responses.

As more apps start to make use of AI, multiple DNN-equipped apps may run on a mobile device at the same time. For example, while a user is using her smartphone to examine some surveillance videos through a DNN-based object detection module, she may be speaking to the DNN-powered personal assistance app on her phone to take notes, while her social media app may be running some DNN-based recommendation algorithm in the background. We call such a co-run scenario *multi-instance DNN* executions.

Scheduling is important for multi-instance DNN executions, especially on resource-constrained systems. This paper particularly focuses on the *spatial* aspect of scheduling, which determines the placement of a DNN-based app on heterogeneous hardware units during each inference. It is critical for the computing efficiency of DNNs. On one hand, DNNs are computationally demanding, and their performance is heavily influenced by the type, configuration, and availability of the underlying computing resources. On the other hand, modern mobile devices (e.g., smartphones, tablets) are commonly equipped with heterogeneous computing units. On a Samsung Galaxy S21, for instance, there is one big "primary" CPU core (ARM Cortex-X1), three medium-sized "performance" CPU cores (Cortex-A78), four small "efficiency" CPU cores (Cortex-A55), one Adreno 660 GPU, and other accelerators. As a result, the speed and power consumption of a DNN running on the different computing units in a mobile device may differ as much as several times as illustrated in Figure 1. Multi-instance DNN executions further complicate the scheduling of DNNs to the best computing units, due to the contentions for computing resources by other co-running DNNs.

The objective of this work is to address such spatial scheduling problems on *open* mobile devices. Here, *open* mobile devices refer to mobile devices on which users can install or uninstall arbitrary apps anytime. In contrast, some devices (e.g., an autonomous driving system) are *closed*, where the applications to install and run are predetermined. The problem of multi-instance DNN scheduling also exists on *closed* devices, and has been explored in some previous studies [4, 9, 14, 28]. But those studies assume that the set of co-running apps are known beforehand and their schedules are fully controllable by a central agent (e.g., OS), which is not the case for *open* devices. Their solutions hence cannot apply to the *open* devices. To the best of our knowledge, no prior solutions have been proposed for multi-instance DNNs scheduling on open mobile systems.

This work proposes the first-known *decentralized* application-level adaptive spatial scheduler for multi-instance DNNs on open mobile devices. Being decentralized means




(a) Power Consumption on Samsung Galaxy Tab S8+

(b) Average Time per Inference on Samsung Galaxy Tab S8+

(c) Power Consumption on Samsung Galaxy S21 FE

(d) Average Time per Inference on Samsung Galaxy S21 FE

Figure 1: **Standalone Profiling of Three DNN Networks.** We profile the average power and inference time for different DNNs. They have their own best option for delegation. Thread = *i* means the model is executed on the CPU with *i* threads. NNAPI *low power* and *fast single answer* are two options that consider using GPU and accelerators.

that the spatial scheduling decisions are made by each application rather than by a centralized agent (e.g., OS). This distinctive design brings several benefits over centralized schemes: It is easy to adopt without the need for OS modifications; it requires no OS admin privileges; it preserves privacy and avoids inter-app communications or app-OS special communications and the associated overhead.

Specifically, the spatial scheduling considered in this work includes the decisions on running a DNN on GPU or on CPU and if on CPU how many CPU threads to use. We intentionally leave the temporal aspect of scheduling (i.e., at what time an app runs or gets evicted) and priority management as they are, because these tasks are what the OS is already taking care of. We meanwhile ensure that the spatial scheduling method can automatically adapt the scheduling decisions to the temporal scheduling by the underlying OS. This design makes the solution easy to adopt (as no OS modifications are necessary), applicable across systems, and workable regardless of what the other co-running apps are and what scheduling policies they follow. It also retains the fairness guarantees and starvation-avoidance provided by the underlying OS.

Our solution achieves these properties by leveraging the adaptivity of Deep Reinforcement Learning (DRL) [17,20,25]. We develop a DRL-based scheduling library. It is decentralized, working at the application level. Any app may call the library to dynamically determine, for the next DNN inference, whether CPU or accelerators is to be used, what modes (e.g., performance or power-efficiency modes) to use, and if CPU, what is the best number of threads to launch. It requires no direct knowledge about other apps. It gives recommendations based on the current state of the executing environment and the estimated rewards this application is expected to obtain for each of the possible schedules—produced by a model learned through the self semi-supervised approach of DRL.

By drawing on previous theoretical results on the Nash equilibrium of RL, we provide discussions on the convergence of the scheduling algorithm and the empirical evidences.

In a set of co-run scenarios formed by the subsets of nine DNNs, the proposed solution improves the average latency by as much as 4×, and saves the average energy consumption by as much as 3× compared to Android NNAPI, the official Android tool that automatically selects the computing units to use for running a DNN. Experiments on a smartphone (Samsung Galaxy S21) and a tablet (Samsung Tab S8+) show that the benefits are consistently significant. Further experiments show that the benefits remain even if those DNNs co-run with uncontrolled apps (i.e., apps that do not employ the proposed scheduling algorithm). The scheduling algorithm converges quickly (within seconds) and adapts to the running environments automatically, making it an immediately adoptable solution across mobile systems.

Overall this work makes the following main contributions:

- To our best knowledge, the solution in this work is the first decentralized application-level adaptive scheduling for multi-instance DNNs on open mobile devices.
- This work uncovers a set of novel insights: (i) It is possible for a scheduler to work effectively without direct knowledge of other apps in multi-instance DNN scheduling; (ii) the DRL-based scheduling is effective in adapting to the factors in the execution environment (OS, other apps, priorities, etc.); (iii) the decentralized scheduling

algorithm is quick in converging to a balance point.

- This work empirically evaluates the efficacy of the proposed solution, showing that it consistently gives significant speedups and energy savings across DNN workloads, hardware configurations, and running scenarios (with or without uncontrolled apps, various background/foreground combinations).

## 2 Background

***Reinforcement Learning and Deep-Q-Network*** Reinforcement learning (RL) [20] is a machine learning approach in which an agent learns from environmental feedback and a series of decisions to maximize the total cumulative rewards to stimulate better decision-making. Q-learning [27] is a type of RL algorithm that seeks an offline policy to maximize the expected total rewards across all steps. "Q" refers to the policy function $Q(S,A)$ that inputs a state and action pair and outputs a Q-value. Two typical Q-learning schemes are Q-table and Deep-Q-network (DQN) [17]. DQN improves the limited representation of the Q-table.

Q-table stores the state and action pairs with the estimated Q-value. The Q-value of each step can be obtained by table lookup. However, this approach only works when states and actions are discrete values and the sets are small. In contrast, DQN replaces the Q-table with a neural network (NN) as a function approximator. Now, the sets of states and actions can be large, and the state space can be continuous. In the training phase, the loss function minimizes the squared error between the target Q-value and the predicted Q-value given by the NN. So the loss function of NN can thus be formulated as follows: $L = (Q(S,A) - (R + \gamma * \max(Q(S',A'))))^2$, where $R$ is the immediate reward for taking action $A$ in state $S$, $\gamma$ is the discount factor (a value between 0 and 1 that determines the importance of future rewards), and $S'$ and $A'$ are the next state and the possible actions in that state, respectively.

***Nash Equilibrium*** The Nash equilibrium is a concept in game theory that describes the optimal behavior of players in a game. It is a stable, self-enforcing, and Pareto optimal solution, where each player has chosen an optimal action, given the other players' actions. There are several methods for finding the Nash equilibrium of a game, including using best response functions and mixed strategies. The Nash equilibrium is a helpful tool for analyzing strategic interactions and predicting players' behavior in a game.

One way to find the Nash equilibrium is to use the best response function, which maps each player's action to the action that maximizes their reward, given the actions of the other players. The Nash equilibrium is then the set of actions where each player's action is the best response to the actions of the other players. Another approach is to use a mixed strategy, where players randomly choose their action with a certain probability. The Nash equilibrium is then the set of possibilities where each player's mixed strategy is the best response to the combined strategies of the other players.

## 3 Problem Statement and Research Questions

This section first provides a definition of the focused scheduling problem, and then lists the important research questions.

### 3.1 Problem Definition

**Given:** A set of apps $A$ that may execute on a device $V$, and some of them may run at the same time. $A = A_c \cup A_u$, where, each app in $A_c$ contains some DNNs and employs a policy $P$ to decide the execution configuration of each of its DNNs, while the apps in $A_u$ do not follow policy $P$. For each DNN, there are K possible configurations which affect the usage of CPUs and GPUs of the DNN differently.

**Objective:** Finding $P$ such that the following is minimized:

$$\sum_{i \in A_c(DNN)} \sum_{j} L_{i,j} * W_{i,j}$$

where, $L_{i,j}$ and $W_{i,j}$ are respectively the latency and power consumption of the $j^{th}$ inference of $DNN_i$, $A_c(DNN)$ is the DNNs in the apps in $A_c$. We use the product of latency and power to capture the common interest in both speed and energy usage on mobile devices.

**Constraints:** (1) The scheduling policy $P$ in an app cannot access the information of another app (due to the isolation enforced by mobile systems); (2) each app's priority and temporal scheduling are controlled by the underlying OS.

### 3.2 Design Considerations and Principles

The main aspect of scheduling focused in this work is the execution configuration that affects the usage of computing units—which is essential for the performance and power consumption of DNN. The relevant factors include some that are controllable at the application level, and some at the OS level. For a DNN written in TFLite [3] (a popular development framework for mobile AI), for instance, the application can explicitly specify whether the DNN should run on CPU or accelerators (e.g., GPU), and the number of CPU threads to use. It may also call APIs in NNAPI [8] (the Android official library for running DNNs) with either a performance mode or an efficiency mode; the APIs will automatically determine the CPUs or accelerators to be used for the DNN. We uniformly regard such configurations as application-level controllable configurations, which specifically include the explicit specification of computing units and CPU thread number and the calls to other relevant libraries.

When a DNN runs with a certain configuration, the OS may exert further influence on the usage of the computing units.

For instance, if the application-level control sets the DNN to run on a CPU with 2 CPU threads, the OS will ultimately determine which two CPU cores the threads will run on. If they are small cores, the speed may be much lower than on medium cores.

We design our solution with the following principles:

(1) The scheduling should be decentralized, functioning inside each application. This is because most of the relevant factors are inside the app, and application-level solutions are easier to adopt as they do not need changes to the underlying OS.

(2) The scheduling should be able to adapt to the influence of the underlying OS and other environmental factors (e.g., priorities of apps, background/foreground differences).

(3) The scheduler should work efficiently and adapt to changes in the system agilely.

### 3.3 Research Questions

Creating a decentralized application-level scheduler on open mobile systems has many differences from centralized scheduling on closed systems and hence raises many new questions. We summarize them into the following six open research questions (RQ). They are gradually addressed in the rest of this paper.

**RQ1: How can the scheduler work effectively without direct knowledge of other apps?**

The apps on a smartphone may be developed by many different authors. For security and privacy, open mobile systems typically impose strong isolations among apps. One app cannot access the direct info of another app. How can the scheduler work well under such a constraint?

**RQ2: How can the solution deal with the effects of the scheduler in the underlying OS?**

OS influences the execution of the Apps: Ultimately, it is the OS that allocates computing units and other resources to each App and determines when an app runs and its priority level. The OS schedulers differ from one version of OS to another. We avoid demanding changes to the underlying OS for easy adoption of our solution. The user-level scheduling hence must be made adaptive to OS.

**RQ3: Can decentralized scheduling converge to a good result?**

On an open mobile system, apps come and go, and the workloads on the system may vary continuously. Without the knowledge of other co-running apps, can decentralized scheduling converge to good results?

**RQ4: How fast can the decentralized scheduling learn and adapt?**

Machine learning-based decision models usually take time to learn, but the dynamic nature of mobile systems demands fast responses. Can decentralized scheduling meet the speed needs?

**RQ5: Can the solution work if there are uncontrollable apps?**

In real mobile usage scenarios, not all Apps will adopt the same scheduling policy. The workload from uncontrollable apps can be unpredictable. Can the proposed solution still function well in the presence of such uncontrollable apps?

## 4 Decentralized DQN Scheduler

**Algorithm 1:** DQN Algorithm for each Apps

**Input:** Environment $E$; Replay Memory $M$; Exploration Ratio $\varepsilon$; The parameters of Policy Network $\theta$ and Target Network $\theta^-$; Discount Factor $\gamma$; Batch Size $B$; Update Steps $C$; Huber loss function $L$

**Output:** $Q(s,a;\theta)$

**Initialize:** Take observation from $E$ and generate current state $s$

```
while Inference start do
    if rand() < ε then
        Select action a randomly
    else
        Select action a ← argmax_a Q(s,a; θ)
    Run inference on a target defined by action a
    When Inference ends, Calculate reward r
    Observe from E and generate next state s'
    Store transition (s, a, s', r) to M
    if M.size() > B then
        Sample a mini-batch N from M
        for each transition (s_j, a_j, s'_j, r_j) in N do
            y_j = r_j + γ max_a' Q(s_j, a' : θ^-)
            Calculate Loss l_j = L(y_j, Q(s_j, a_j : θ))
        Batch Update θ using SGD algorithm by loss vector l
    s ← s'
    i ← i + 1
    if i mod C == 0 then
        θ ← θ^-
```

In this section, we introduce the design of the decentralized scheduler, which also answers RQ1 and RQ2.

The design is based on deep reinforcement learning (DQN). As Section 2 mentions, DQN is a deep reinforcement-learning method. As a semi-supervised method, it requires no manual labels, but actively explores the environment, learns the relations between actions and rewards automatically, and uses the learned model to predict the next suitable action. This nature makes it a good fit for the dynamic environment in our problem. In contrast, a DNN-based approach would require offline labels of many training cases and be slow in adapting to dynamic changes.

The DQN-based RL agent is also light-weighted and converges fast. For the storage overhead, the total extra mem-

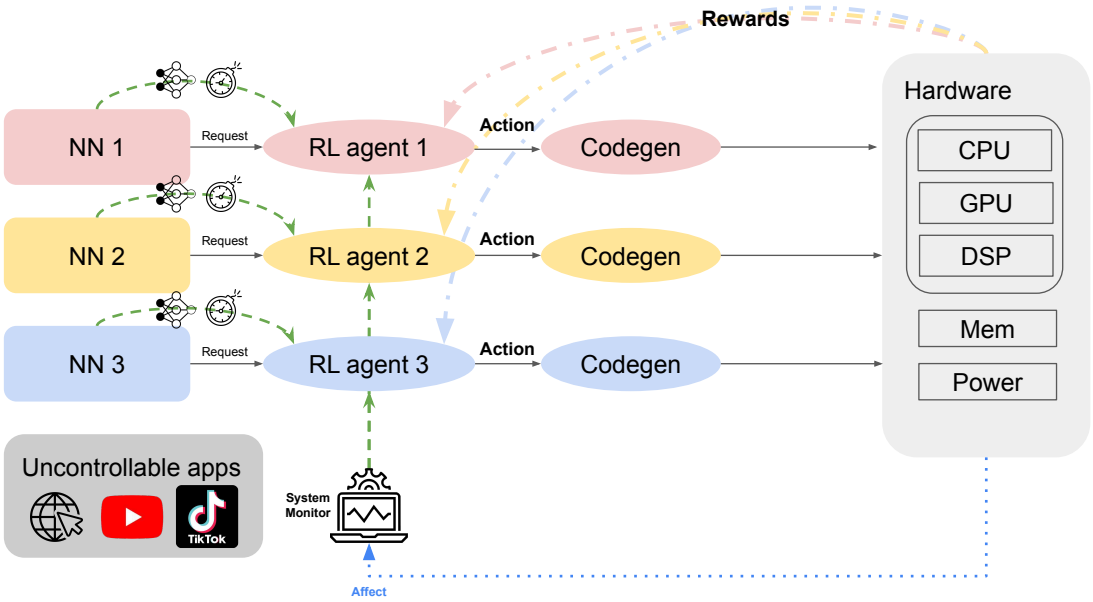



Figure 2: **The Structure of Decentralized DQN Scheduler.** The figure shows three controllable DNN applications that have their RL agent that selects actions (different code generation of the model). All of them collect system status as their RL agent input state. Also, we include the co-running uncontrollable apps that do not contain RL agents inside.

ory footprint needed less than 250KB. For the computation overhead, the inference time of the network only takes less than 1ms and one training process only takes about 1ms to 2ms when three DQN agents co-running. In general real-time constraints are 50ms for each inference, and those extra computations overhead in each model can be neglected.

Figure 2 illustrates the role of the scheduler in a multi-instance scenario. Other than the uncontrolled apps, each DNN-based app contains a DQN agent for configurations. They do not have access to other apps but can obtain the state of the resource utilization of the whole system.

We define the learning procedure of our DQN-based agents with *States*, *Rewards*, and *Actions* in the multi-DNN execution environment as follows:

**States** We use the following variable to capture the static and variance environment information. For static features, we use the number of convolution layers, the number of fully-connected layers, and the MAC operations of DNN models. For environment variance information, we use the CPU utilization, GPU utilization, and memory usage of co-run apps.

**Rewards** The reward function $R$ is composed of three important metrics: latency, power consumption, and deadline. It is defined as follows:

$$R(L_i^{s,a}, W_i^a, d_i) = \begin{cases} -1000 & \text{, if } \quad L_i^{s,a} > d_i \\ -L_i^{s,a} \times Power_i^a & \text{, otherwise} \end{cases}$$

where, $L_i^{s,a}$ represents the latency of the inference of $DNN_i$ with action $a$ on state $s$, $W_i^a$ is the average power of the inference of $DNN_i$ with action $a$, and $d_i$ is the deadline for the inference of $DNN_i$.

Because ($latency \times power$) can be regarded as the energy consumption estimation, we call it **Energy Factor** in our paper. Also, it is used as one of the metrics in our evaluation (Section 6). The multiplication of power and latency gives a linear relationship, with no skew towards either factor. For instance, if the latency doubles, the entire reward function doubles, and the same applies to power consumption. The simple production form of the reward function avoids additional hyper-parameters. A large negative reward is used when a deadline is passed to discourage missing deadlines.

**Actions** The action space involves the configurations that can affect the usage of computing resources on the hardware. In our study, we include six configurations as detailed in Section 6.1.

**Neural Networks** DQN includes a *policy neural network* and a *target neural network* inside, which learn about the relations between states, actions and rewards. We want to make the networks as simple as possible to control the runtime overhead. So we adopt a model that only maintains two fully-connected (FC) layers as our policy and target network. The first FC layer input channel is 8, and the output channel is 100. The second FC layer input channel is 100, and the output channel is 6. For the input, it is the vector of the state. Its dimension is 8, which contains four CPU and GPU usage pairs. The output is a vector that represents the q-values of six actions.

The DQN agents go through an exploration and learning

stage until they reach convergence. As listed in Algorithm 1, at the start of each episode, each agent selects the action with the maximum q-value generated by the policy network with parameters θ and current state $s$ but has some exploration rate ε to select action randomly. Here, one episode is one inference of the DNN model. After making the inference with the selected action, the power consumption and latency are collected to compute the reward $r$. Then, the agent observes the environment and generates the updated state $s'$. One transition $(s, a, s', r)$ is then pushed into the replay memory $M$. The training process starts when the replay memory $M$ has enough data. It goes as follows. It first samples a mini-batch $N$ from $M$. Then, it calls the *target network* to generate the expected q-value $y$ for each transition. After that, it uses the Huber Loss function [10] to calculate the loss value between the expected q-value and the current q-value output by the *policy network*. The final step of the training process is to update the parameters θ based on the loss value. In every $C$ episode, the parameters θ of the *policy network* are copied to the *target network* as its new parameters. In the targeted co-run scenario, each controllable App is equipped with a DQN agent that is trained for scheduling its DNN inferences.

## 5 Convergence Discussion

This section discusses the convergence of the DQN-based scheduling algorithm (RQ3). Prior works [6] generalize the multi-agent Q-learning method as a general-sum stochastic game and prove all reward functions in each agent are guaranteed to converge. Specifically, Hu and Wellman [13] present an algorithm to solve the general-sum stochastic games, and Bowling [6] strengthens the proof with further assumptions.

Their convergence theorem has four necessary assumptions, two of which are about exploration and the decay of the learning rate, and are similar to those used in the Deep-Q-Learning algorithm. It is assumed that they have been met. The remaining two assumptions [6, 13] are as follows:

**Assumption .1** *A Nash equilibrium* $(\pi^1_*(s), \pi^2_*(s))$ *for any stochastic game* $(Q^1_n, Q^2_n)$ *satisfies one of the following properties:*

*1. The equilibrium is a global optimal.*

*2. The equilibrium receives a higher payoff if the other agent deviates from the equilibrium strategy.*

**Assumption .2** *The Nash equilibrium of all stochastic games* $Q_n(s)$*, as well as* $Q_*(s)$ *must satisfy property 1 in Assumption 1* ***or*** *the Nash equilibrium of all stochastic games,* $Q_n(s)$*, as well as* $Q_*(s)$ *must satisfy property 2 of Assumption 1*

Assumption 1 includes a property that states that there exists a set of strategies for the agents, where each agent individually obtains the highest possible payoff. This also guarantees that this set of strategies forms an equilibrium since no agents would gain from deviating from their chosen strategy. Assumption 2 includes another property in which the game's Nash equilibrium is a "saddle point." This implies that if an agent deviates from the equilibrium, the agent would not gain, but other agents would, which makes no agent want to deviate from the equilibrium.

Finding a globally optimal solution for the multi-agent problem is known to be NP-hard [7]. However, by reaching Nash equilibrium during convergence, RL can ensure that each agent adheres to a strategy that gives a good payoff to both itself and the other agents. Reflected in our scheduling context, it means that all controllable Apps may adhere to a strategy that helps meet their deadlines while minimizing energy consumption. The achievement of Nash equilibrium eliminates fairness concerns among controllable Apps, as they all reach a stable state where each maximizes its benefits within the given constraints.

Based on prior studies [6, 13], it has been demonstrated that a zero-sum stochastic game converges. Our scheduling problem for multi-DNN applications in this paper bears a resemblance to a zero-sum stochastic game. In our case, each agent corresponds to our DQN agent for each controllable DNN application. All DQN agents involved compete for limited resources, such as CPU and GPU, with a maximum utilization boundary. While our problem may not strictly adhere to all the definitions of a stochastic game, we empirically demonstrate its convergence under our circumstances in Section 6.

## 6 Evaluation

This section evaluates our decentralized application-level adaptive scheduler (called DQN for short in this section) by comparing it with three baseline scheduling methods that are designed for single DNN execution: two static scheduling settings used by Android Neural Networks API (NNAPI) [8] (`NNAPI LOWER_POWER` that minimizes the power consumption for each DNN and `NNAPI FAST_SINGLE_ANSWER` that minimizes the inference latency for each DNN) and an offline profiling-based scheduling method (`Best Standalone` that based on offline profiling selects the best setting for each individual DNN among all delegate settings introduced in Section 6.1). This evaluation has three objectives as follows: 1) demonstrating that as the *first* decentralized DNN co-run scheduling method, DQN outperforms all baseline scheduling approaches that are designed for single DNN execution in multiple representative DNN co-run scenarios (Section 6.2); 2) verifying DQN's convergence and fast converging speed, and studying the underlying reason why DQN outperforms baseline scheduling methods by a reward convergence analysis (Section 6.3); 3) proving DQN's benefits remain even if the DNNs co-run with uncontrolled apps with both predictable and unpredictable workloads (Section 6.4).

## 6.1 Evaluation Methodology

**Benchmarks.** Table 1 characterizes the nine DNN models from various domains used in the evaluation[1]. All models are in TensorFlow Lite [3] format. These DNN models form three groups: `Image`, `Audio & Image`, and `Video & Image` with three models in each group. Our evaluation runs models in each group simultaneously to simulate the real-world DNN co-run scenario. For example, Group 1 (G1) simulates an intelligent camera running varied AI capabilities simultaneously, pose detection (`SinglePose`), object detection (`YOLO-v5`), and image classification (`EfficientNet`). Other groups simulate more complex scenarios with audio and image or video and image co-processing. In addition, these DNNs have varied model sizes, standalone latency, and power consumption, representing three different cases: relatively balanced workloads, mild imbalanced workloads, and severe imbalanced workloads, respectively. Therefore, they show different behaviors in the evaluation. Please find more discussions in Section 6.2.

Table 1: **Nine DNN Models Used in Our Evaluation.** They form three groups. DNN models in each group are executed simultaneously.

| Group | Models | Sizes (KB) |
|---|---|---|
| G1: Image | SinglePose [1] (SP) | 9,154 |
| | YOLO-v5 [5] | 7,428 |
| | EfficientNet [23] (ENet) | 6,265 |
| G2: Audio&Image | YamNet [24] | 4,031 |
| | MobileNetv1 [12] (MNv1) | 4,188 |
| | WDSR [29] | 1,252 |
| G3: Video&Image | Movenet [16] | 24,440 |
| | Esrgan [26] | 4,877 |
| | MobileNetv2 [21] (MNv2) | 13,666 |

Table 2: **Absolute Latency of DQN.** This table reports the absolute latency of DQN in Figure 3, Figure 6, and Figure 7.

| | Figure 3 | | Figure 6 | | Figure 7 | |
|---|---|---|---|---|---|---|
| (Uint: ms) | Tablet | Phone | Tablet | Phone | Tablet | Phone |
| G1: SP | 8.5 | 10.7 | 35.0 | 32.5 | 32.1 | 34.4 |
| G1: YOLOv5 | 331.1 | 397.1 | 591.9 | 318.7 | 542.4 | 446.6 |
| G1: ENet | 5.0 | 47.2 | 9.7 | 337.8 | 7.0 | 341.3 |
| G2: YamNet | 4.3 | 3.8 | 23.4 | 23.0 | 20.4 | 17.1 |
| G2: MNv1 | 6.2 | 34.3 | 10.5 | 30.8 | 8.2 | 32.1 |
| G2: WDSR | 6.8 | 116.9 | 6.3 | 114.2 | 5.0 | 108.2 |
| G3: Movenet | 34.2 | 33.0 | 76.2 | 71.3 | 78.5 | 70.1 |
| G3: Esrgan | 62.3 | 61.5 | 64.6 | 75.2 | 94.4 | 69.8 |
| G3: MNv2 | 40.2 | 25.2 | 80.2 | 42.5 | 17.1 | 35.4 |

**Software settings.**

Our evaluation considers two co-run scenarios: *controllable DNN tasks co-run* and *uncontrollable tasks co-run. Controllable DNN tasks* refer to Apps that incorporate our RL agents, while *uncontrollable tasks* refer to Apps that do not involve our RL agent. *Uncontrollable tasks* may or may not use DNNs.

[1]These DNNs are collected from Tensorflow Hub [2] and GitHubs [22].

Table 3: **Absolute Energy Factor of DQN.** This table reports the absolute energy of DQN in Figure 3, Figure 6, Figure 7.

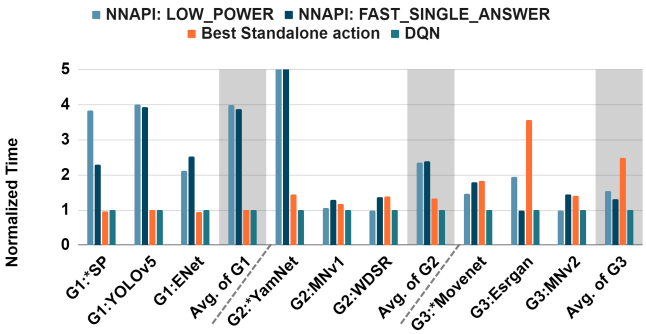
| | Figure 3 | | Figure 6 | | Figure 7 | |
|---|---|---|---|---|---|---|
| (Uint: Joule) | Tablet | Phone | Tablet | Phone | Tablet | Phone |
| G1: SP | 18.9 | 15.9 | 13.48 | 18.1 | 25.2 | 21.8 |
| G1: YOLOv5 | 950.3 | 446.3 | 1.6k | 365.1 | 1.5k | 509.8 |
| G1: ENet | 9.8 | 122.3 | 16.8 | 67.9 | 13.8 | 69.6 |
| G2: YamNet | 7.7 | 7.0 | 17.4 | 39.9 | 52.5 | 33.5 |
| G2: MNv1 | 11.3 | 58.9 | 12.5 | 57.4 | 19.8 | 64.9 |
| G2: WDSR | 11.0 | 41.7 | 10.7 | 58.2 | 11.2 | 35.3 |
| G3: Movenet | 48.0 | 19.3 | 73.3 | 26.6 | 59.3 | 29.5 |
| G3: Esrgan | 33.2 | 14.5 | 36.0 | 20.4 | 23.4 | 16.4 |
| G3: MNv2 | 27.7 | 32.8 | 11.8 | 38.2 | 60.8 | 29.6 |

*Controllable DNN Tasks Co-Run.* We build an Android demo app with Java that can run each DNN individually. Users can control this demo app to start and stop DNN inference. This demo app relies on TensorFlow Lite (TFLite) [3] to run DNNs. Our evaluation employs multiple delegate settings in TFLite to run DNNs: using 1, 2, 3, or 4 CPU threads[2], respectively, and using two NNAPI [8] modes, LOWER_POWER or FAST_SINGLE_ANSWER, respectively. Particularly, NNAPI is designed for accelerating TensorFlow Lite DNN execution on mobile devices with supported hardware accelerators including GPU, DSP, and NPU. It automatically partitions a DNN model, maps each partition to a processor, and calls corresponding kernel codes for that processor. Thus, we can treat it as *static offline scheduling* for each individual DNN. Our evaluation particularly employs two NNAPI modes as the baseline, `LOWER_POWER` which minimizes the power usage, and `FAST_SINGLE_ANSWER` which minimizes the inference latency. Besides them, our evaluation also employs an offline profiling-based scheduling method as a baseline: `Best Standalone` that selects the best setting (i.e., with the best energy factor defined as $power\_consumption \times latency$) for each individual DNN among all delegate settings (including using 1 to 4 CPU threads and two NNAPI modes) based on offline profiling results[3].

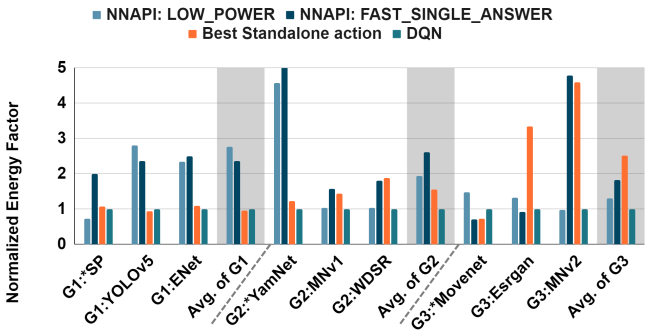
*Uncontrollable Tasks Co-Run.* We select two widely used real-world applications `TikTok` and `Web Browser` to experiment with two popular user behaviors, watching social media video and browsing web pages. Here we select the default web browser in Android, Google Chrome as our target.

[2]The evaluated mobile chip has 8 CPU cores, but the Android OS only allows background Apps to access the 4 small CPU cores. Thus, we make it consistent throughout our evaluations: the foregrounds Apps access the 4 cores (prioritize to access the big core, medium core, then small cores), and the background Apps access the 4 small cores.

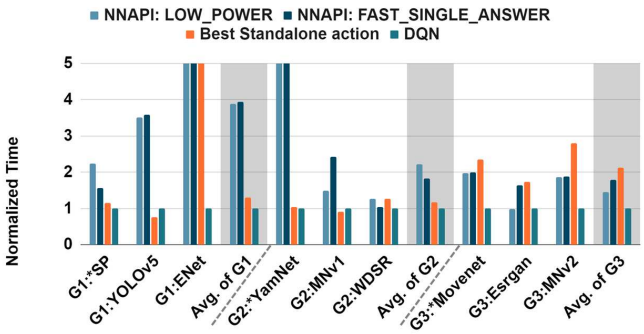
[3]we profile the power values through the Android Developer API "dumpsys batterystats".



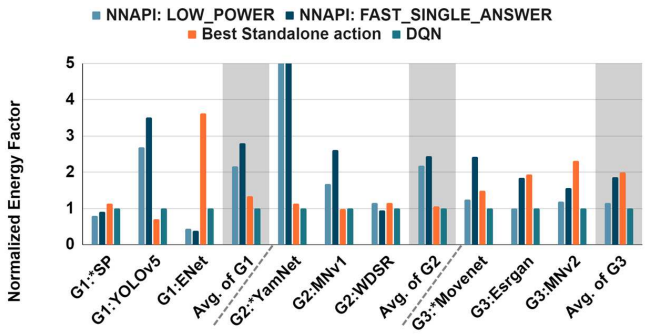

(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+




(b) Normalized Energy Factor on Samsung Galaxy Tab S8+




(c) Normalized Average Inference Time on Samsung Galaxy S21 FE




(d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 3: **Overall Comparison between DQN and Three Baselines on Three DNN Groups.** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each group co-running three DNN apps (* denotes this DNN runs in the foreground and the others run in the background).

**Evaluation platforms.** DQN is evaluated on two edge devices: (1) `Samsung Galaxy S21 FE 5G` mobile phone, equipped with Android 12 OS, and Qualcomm SM8350 Snapdragon 888 5G SoC with Octa-core CPU (1x2.84 GHz Cortex-X1 & 3x2.42 GHz Cortex-A78 & 4x1.80 GHz Cortex-A55), Adreno 660 GPU (Version 1), and Hexagon 780 DSP. Its storage capacity is 128GB with 6GB RAM and its voltage is 4.3V. (2) `Samsung Tab S8+ tablet`, equipped with Android 12 OS as well, and Qualcomm SM8450 Snapdragon 8 Gen 1 SoC with Octa-core (1x3.00 GHz Cortex-X2 & 3x2.50 GHz Cortex-A710 & 4x1.80 GHz Cortex-A510), Adreno 730 GPU, and Hexagon DSP. Its storage capacity is 128GB with 8GB RAM and its voltage is 4.1V.

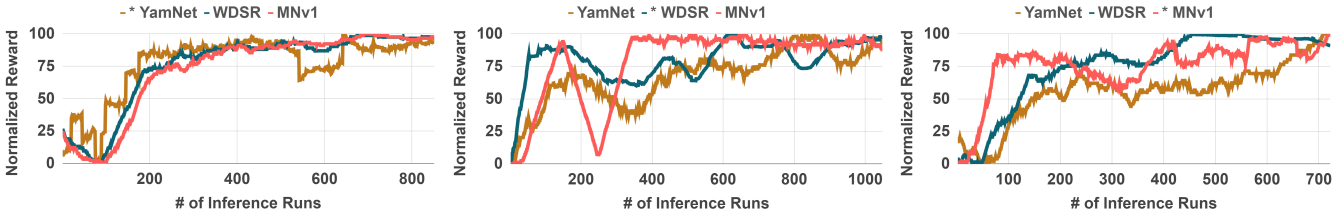
## 6.2 Overall DQN Scheduling Performance

This section evaluates DQN on the three groups of co-run DNNs in Table 1 by comparing it with the three scheduling baselines aforementioned: `NNAPI LOWER_POWER`, `NNAPI FAST_SINGLE_ANSWER`, and `Best Standalone`. Figure 3 shows the comparison results, in which the x-axis shows the three DNNs in each group and the average performance of each group. It is worth noting that the *star (*)* before the DNN name indicates that this DNN model is executed in the foreground (and two other DNNs in the same group are executed in the background) for this co-run[4]. We intentionally use this setting to simulate the real-world Apps co-run on the Android system (and bring the OS impact on the user-level scheduling into account).

Figure 3 employs two metrics to compare our decentralized DQN system with three baselines: `average inference latency` (as shown in Figure 3a and Figure 3c) and `energy factor` ( as shown in Figure 3b and Figure 3d). The energy factor is defined as $power\_consumption \times latency$ for each inference, which is also used as our reward function in each DQN agent, the lower the better. To improve the readability, we normalize the results in Figure 3 by setting DQN performance as 1. Table 2 and 3 summarizes the absolute values for reference.

Figure 3 shows that for the average inference latency, our decentralized DQN-based scheduler achieves up to 4× speedup over two baselines of NNAPI (`NNAPI LOWER_POWER` and `NNAPI FAST_SINGLE_ANSWER`), and 2.7× speedup over

[4]Android OS grants foreground and background Apps different priorities/limitations, e.g., normally, background Apps have lower priority than foreground ones, and background Apps cannot access the big core of CPUs.




(a) YamNet at Foreground (b) WDSR at Foreground (c) MNv1 at Foreground

Figure 4: **Reward Convergence Trend for the Three Apps Co-Run in G2.** It reports DQN's normalized reward trend to prove DQN converges in different situations. It uses DNNs in G2 and places each DNN in the foreground.

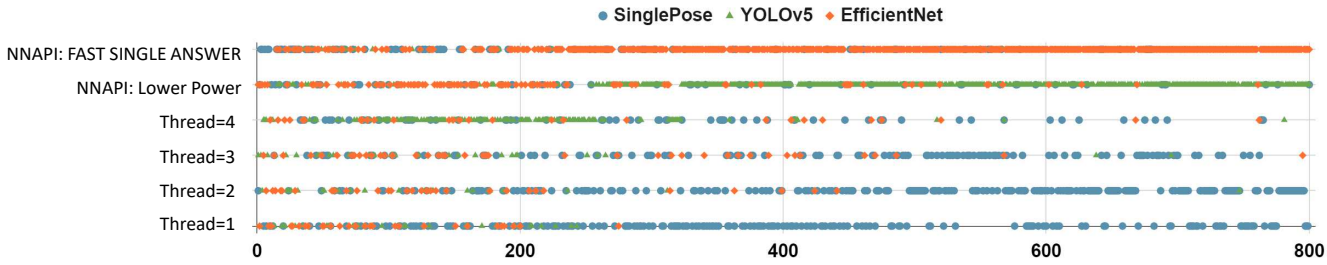



Figure 5: **Selected Action Convergence Trend.** This figure shows the action selection through runs for co-running results in G1 (all of them are in the background). The action selection will be converged into one or two options.

Table 4: **Selection Rates of Actions of Last 100 Runs of Figure 5.**

| | Thread = 1 | Thread = 2 | Thread = 3 | Thread = 4; | NNAPI: LOW_POWER | NNAPI: FAST_SINGLE_ANSWER |
|---|---|---|---|---|---|---|
| SinglePose | 14% | 75% | 6% | 1% | 1% | 3% |
| YOLO_v5 | 0% | 1% | 1% | 0% | 98% | 0% |
| EfficientNet | 0% | 3% | 0% | 3% | 0% | 94% |

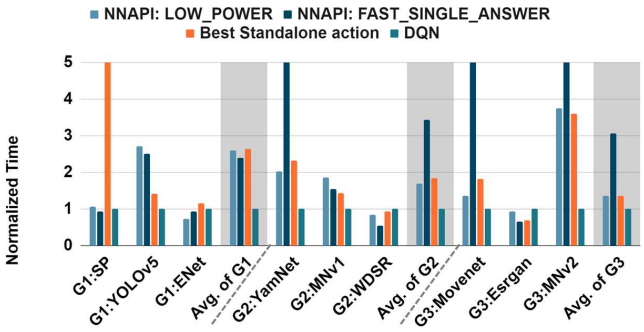
`Standalone Best` action selection, respectively, for average results of three co-running DNN groups (gray area). For the energy factor, our decentralized DQN-based scheduler achieves up to 3× energy saving over `NNAPI LOWER_POWER` and `NNAPI FAST_SINGLE_ANSWER`, and 2.6× energy saving over `Standalone Best` action selection, respectively, for average results of three co-run DNN groups. Comparing the DQN performance across three groups, we can see that as the workload imbalance increases (from G1 to G3), the benefit of DQN over `Best Standalone` grows while its benefit over both NNAPI schedulers drops. This is mainly because `Best Standalone` partitions each DNN workload into more processing units than NNAPI schedulers, so it reduces the resource competition caused by the workload imbalance. DQN has a similar effect. Moreover, although the average inference latency and energy factor vary for different groups (and each DNN model) under various settings, our decentralized DQN-based scheduler always performs better than baseline methods. These results prove that DQN is robust enough to deliver high-quality scheduling results for various DNN applications and environment settings (e.g., varied foreground/background DNN settings, DNN structures/target domains, and executing devices). The following sections further verify this claim.

## 6.3 Reward Convergence Analysis

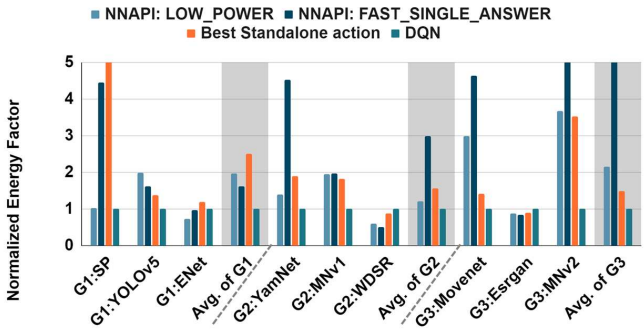
This reward convergence analysis has three objectives: 1) to verify the rewards of multiple agents converging empirically, 2) to measure the DQN convergence speed[5], and 3) to study why DQN outperforms baseline scheduling by analyzing its trend of action selection.

We verify the convergence and measure the convergence speed by taking Group 2 (in Table 1) as an example due to the space limitation and other groups show similar trends. Figure 4 shows the convergence trends of DQN training on co-running three DNNs in Group 1 under three different set-

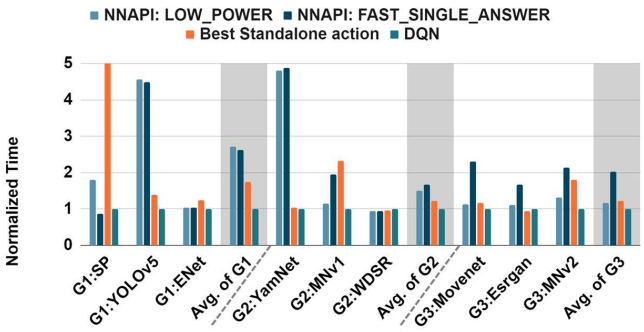
[5]Theoretical convergence proof only proves all DQN converges eventually without showing the convergence speed.



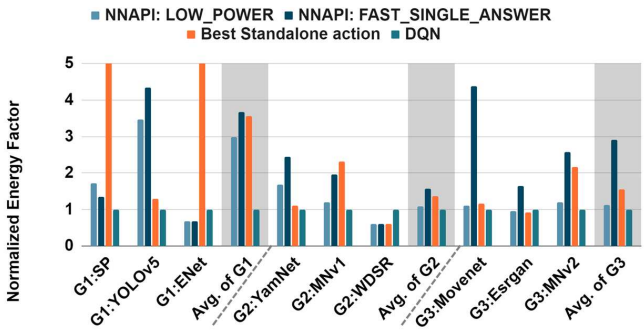

(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+




(b) Normalized Energy Factor on Samsung Galaxy Tab S8+




(c) Normalized Average Inference Time on Samsung Galaxy S21 FE




(d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 6: **Co-Run Three DNN Groups with Uncontrollable App TikTok.** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with uncontrollable app TikTok that plays video posts on the foreground.

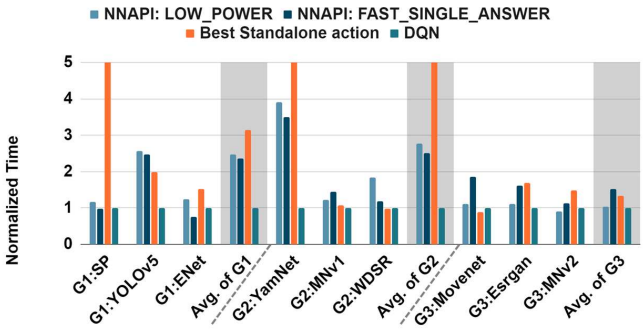
tings that place `YamNet`, `WDSR`, or `MNv1` at the foreground, respectively. The x-axis is the number of DQN inference runs (which is equivalent to the number of DNN inference runs), and the y-axis is the average action selection rewards of 100 times. Regardless of the environment settings, the DQN agents can reach convergence within 800-1000 inference runs. The scheduling time overhead for a single execution of the DNN App includes the forward- and backward-propagation phases of the DQN agent, which typically take 1.2 ms on average and only contribute to 0.5-5% of the overall DNN App execution time (which ranges 20-176ms). Additionally, the energy overhead amounts to approximately 0.5-2% of the DNN App execution. This convergence time is trivial compared to the time required to profile and configure each model manually.

We next study why the DQN agent can outperform the single DNN scheduling baselines by analyzing its trend of action selections. We take Group 1 (in Table 1) with `SinglePose`, `YOLOv5`, and `EfficientNet` as an example this time. All models run in the background, and the evaluation results are shown in Figure 5. In addition, Table 4 shows the percentage of action selection in the last one hundred runs to give an insight into action selection after the model is converged.

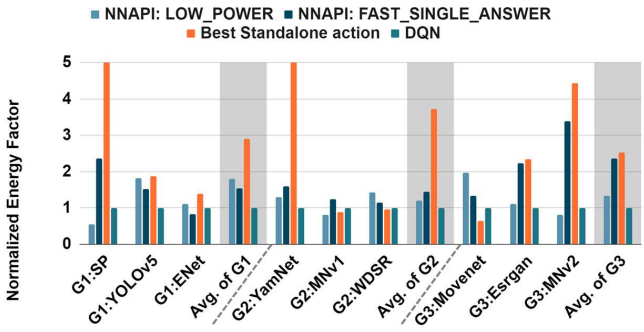
Figure 5 and Table 4 offer us two key insights in DQN: first, even though each DNN starts with multiple selections, their decision is converged into one or two actions, and second, a precise boundary exists between the actions only using CPU and those cooperating with GPU. The regular pattern of the action selection in the first insight implies that the DQN model converges at the end, empirically proving that the multi-agent DQN game converges (in another setting). The second insight tells us that DQN can effectively avoid computing resource contention. For example, when YOLOv5 and EfficientNet run on GPUs, DQN is able to schedule SinglePose on CPUs, thus preventing competition for the limited resource of accelerators; while other baseline scheduling methods fail to do this, resulting in GPU contention.

## 6.4 DQN Performance w/ Uncontrollable Apps

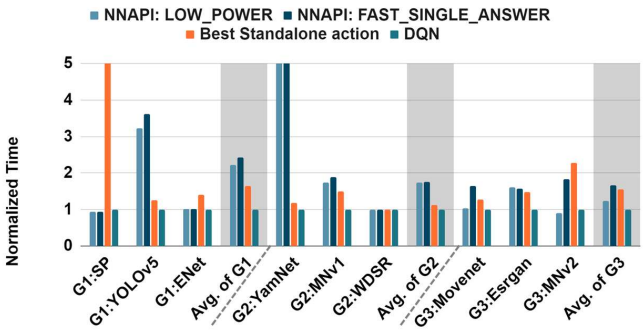
This section evaluates the performance of our decentralized DQN scheduler under uncontrollable application co-running situations. More specifically, we aim to 1) compare the performance between our decentralized DQN scheduler and those



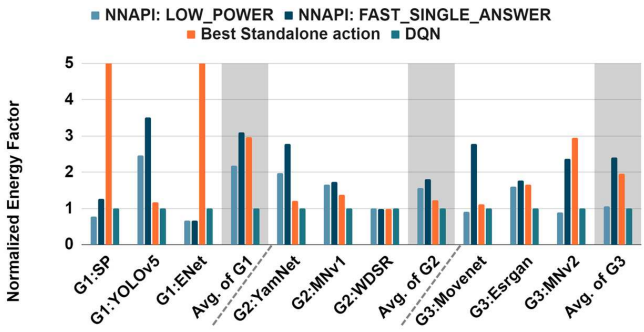

(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+




(b) Normalized Energy Factor on Samsung Galaxy Tab S8+




(c) Normalized Average Inference Time on Samsung Galaxy S21 FE




(d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 7: **Co-Run Three DNN Groups with Uncontrollable App Web Browser (Google Chrome).** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with the uncontrollable app Google Chrome randomly browsing pages on the foreground.

baselines under the popular real-world Apps co-running circumstance, and 2) illustrate our DQN solution has strong adaptivity when co-running with Apps with dynamically changing workloads.

Specifically, we simulate the real-world environment by simultaneously running DNNs and other apps with predictable workloads (i.e., repeatedly playing a TikTok video post), and unpredictable workloads (i.e., randomly browsing webpages in a web browser, Google Chrome). Both TikTok and Google Chrome run in the foreground, while DNNs (and their demo applications) run in the background. TikTok and Google Chrome require CPU and accelerators (e.g., GPUs), thus careful workload scheduling of DNNs is desired if we would like to achieve optimized system performance. To verify the predictability of TikTok and the unpredictability of Google Chrome, respectively, we use `GPUWatch` to monitor both applications' execution and find that the major task, video processing in Tiktok requires a stable amount of GPU resources, while Google Chrome only consumes GPU resources when users touch or swipe across the screen, resulting in irregular GPU usage.

Figure 6 and Figure 7 compare DQN with all three baselines aforementioned, `NNAPI LOWER_POWER`, `NNAPI FAST_SINGLE_ANSWER`, and `Best Standalone` on two platforms under two uncontrollable cases (more predictable TickTok and more unpredictable Google Chrome), respectively. Our decentralized DQN-based approach outperforms all baselines for both cases in terms of both latency and energy for all three groups of DNNs. For the Tiktok case, DQN shows better performance because the DQN agent has more convincing history data that can predict more accurate action for the next step. The Google Chrome case empirically proves that our decentralized DQN-based approach is robust enough to handle DNNs co-run with an app that has unpredictable workloads.

**Compare with a Heuristic-Based Adaptive Method.** To confirm that simple heuristic-based adaptation is insufficient for the scheduling problem, we implement a heuristic method called "trial and set" (T&S). In T&S, each action performs $X$ (50 in our experiments) inference runs and selects the action with the minimum observed online energy factor in subsequent inferences. We compare it with our DQN results on G2 in each of three settings: three DNNs in G2 co-run (see

Table 5: **Comparisons with simple Heuristic-based adaptation.** This table reports the average latency and energy factor for G2 in three co-run settings described in Figures 3, 6, and 7

| Co-run Setting → | | No other Apps | | With Web Browser | | With TikTok | |
|---|---|---|---|---|---|---|---|
| DNN Apps ↓ | Metrics ↓ | T&S (X=50) | DQN | T&S (X=50) | DQN | T&S (X=50) | DQN |
| YamNet | Time (ms) | 3.690 | 4.250 | 127.120 | 20.482 | 53.400 | 23.050 |
| | Energy Factor | 0.461 | 0.502 | 15.890 | 3.392 | 4.429 | 2.707 |
| SSD_MNv1 | Time (ms) | 8.260 | 6.170 | 191.740 | 8.248 | 60.670 | 30.860 |
| | Energy Factor | 1.161 | 0.734 | 24.351 | 1.284 | 8.037 | 3.890 |
| WDSR | Time (ms) | 126.290 | 6.750 | 56.320 | 5.042 | 142.000 | 114.220 |
| | Energy Factor | 15.701 | 0.714 | 8.599 | 0.729 | 18.034 | 3.948 |
| Avg. Energy Factor | | 5.774 | 0.65 | 48.840 | 5.405 | 10.166 | 3.514 |

Figure 3), G2 co-run with a predictable App (see Figure 6), and G2 co-run with an unpredictable App (see Figure 7). Each execution of the DNN-based app conducts 250 inferences in total. Table 5 shows the results. The result shows that the simple adaptation by T&S is insufficient for fitting the continuously changing execution environments. The schedules it picks cause $3-10\times$ larger energy factors as well as frequently substantial slowdowns compared to the results by DQN.

## 7 Related Work

DNN workload under the stochastic runtime variance has been addressed in Autoscale [15]. The authors propose a lightweight scaling engine for DNN inference on a cloud-edge environment. It applies an offline-trained Q-table that observes DNN characteristics and runtime variance as states and selects execution targets as action. It is, however, only for single DNN execution.

Multi-tenancy DNNs [28] have been an active research topic in recent years. NestDNN [9] proposes an efficient scheduler that works with different model pruning ratios. The scheduling decision is guided by the proposed minimum total cost and minmax cost. The solution enhances the multi-DNN inference accuracy and video frame processing rate while reducing energy consumption. NeuOS [4] proposes a layer-by-layer multi-DNN scheduler. At each layer boundary, the system will determine the power configuration for each DNN based on their deadlines. Band [14] presents a model analyzer and scheduler to organize a multi-DNN workload on a heterogeneous platform. The model analyzer partitions multiple DNN models into several subgraphs and dynamically designates them with an eligible execution target. The scheduling decision is based on subgraph execution latency estimation using their tensor size and FLOPS.

Those solutions are however all centralized approaches, assuming the set of DNNs is fixed and there is a central runtime scheduler managing all the instances, making them inapplicable to the multi-instance DNN scheduling on open mobile systems.

## 8 Conclusion

This paper proposes the first-known decentralized application-level adaptive scheduler for multi-instance DNNs on open mobile devices. It builds on DQN, a reinforcement learning algorithm that actively explores and learns the relations between the states, actions, and rewards in a dynamic environment. The exploration uncovers a set of insights. It shows that it is possible for a decentralized scheduler to work effectively without direct knowledge of other apps in multi-instance DNN scheduling. The DQN-based scheduling works well regardless of the differences among underlying systems, hardware, and execution settings. The algorithm is shown to converge quickly and effective in improving co-run efficiency. As an application-level solution, it is ready to be immediately adopted across various mobile systems.

## Acknowledgement

We thank Dr. Saurabh Bagchi for shepherding the preparation of the final version of this paper. This material is based upon work supported by the National Institute of Health (NIH) under Grant No. 1R01HD108473-01 and the National Science Foundation (NSF) under Grant No. CCF-2047516 (CAREER). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NIH or NSF.

## References

[1] movenet/singlepose/thunder/4. Accessed on : insert access date.

[2] Tensorflow hub, 2022. Platform for hosting and serving machine learning models.

[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. https://tensorflow.org/, 2015.

[4] BATENI, S., AND LIU, C. Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 371–385.

[5] BOCHKOVSKIY, A., WANG, C.-Y., AND LIAO, H.-Y. M. Yolov5. https://github.com/ultralytics/yolov5, 2021.

[6] BOWLING, M. Convergence problems of general-sum multiagent reinforcement learning. In *ICML* (2000), pp. 89–94.

[7] CHEN, X., AND DENG, X. Settling the complexity of two-player nash equilibrium. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)* (2006), IEEE Computer Society, pp. 261–272.

[8] DEVELOPERS, A. Android neural networks api. https://developer.android.com/ndk/guides/neuralnetworks/index.html. Accessed: January 10, 2023.

[9] FANG, B., ZENG, X., AND ZHANG, M. 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* (2018), pp. 115–127.

[10] FRIEDMAN, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[11] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[12] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[13] HU, J., WELLMAN, M. P., ET AL. Multiagent reinforcement learning: theoretical framework and an algorithm. In *ICML* (1998), vol. 98, pp. 242–250.

[14] JEONG, J. S., LEE, J., KIM, D., JEON, C., JEONG, C., LEE, Y., AND CHUN, B.-G. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (2022), pp. 235–247.

[15] KIM, Y. G., AND WU, C.-J. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), IEEE, pp. 1082–1096.

[16] KOLOTOUROS, N., TZOUVARAS, P., WANG, X., ALAHARI, K., AND ZITNICK, C. L. Movenet: A video-based human motion estimation network. *arXiv preprint arXiv:2003.04831* (2020).

[17] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., ET AL. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[18] NIU, W., MA, X., LIN, S., WANG, S., QIAN, X., LIN, X., WANG, Y., AND REN, B. 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* (2020), pp. 907–922.

[19] NIU, W., SUN, M., LI, Z., CHEN, J.-A., GUAN, J., SHEN, X., WANG, Y., LIU, S., LIN, X., AND REN, B. RT3D: Achieving real-time execution of 3d convolutional neural networks on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35:10, pp. 9179–9187.

[20] PAGANI, S., MANOJ, P. S., JANTSCH, A., AND HENKEL, J. Machine learning for power, energy, and thermal management on multicore processors: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 1 (2018), 101–116.

[21] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *arXiv preprint arXiv:1801.04381* (2018).

[22] TAN, F. Some super resolution tflite models, 2022. GitHub repository.

[23] TAN, M., AND LE, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946* (2019).

[24] TEAM, G. A. Yamnet: An audio event recognition model. *arXiv preprint arXiv:1909.12916* (2019).

[25] VAN HASSELT, H., GUEZ, A., AND SILVER, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence* (2016), vol. 30.

[26] WANG, X., YU, K., WU, S., GU, J., LIU, Y., AND DONG, C. Esrgan: Enhanced super-resolution generative adversarial networks. *arXiv preprint arXiv:1809.00219* (2018).

[27] WATKINS, C. J., AND DAYAN, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.

[28] YU, F., WANG, D., SHANGGUAN, L., ZHANG, M., LIU, C., AND CHEN, X. A survey of multi-tenant deep learning inference on gpu. *arXiv preprint arXiv:2203.09040* (2022).

[29] YUN, S., SEO, S., KIM, J., AND LEE, K. M. Wide activation for efficient and accurate image super-resolution. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 557–572.