Antler: Exploiting Task Affinity for Efficient Multitask Learning on Low-Resource Systems

Yubo Luo
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
yubo@cs.unc.edu

Le Zhang
University of California San Diego
San Diego, CA, USA
le-zhang@ucsd.edu

Zhenyu Wang
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
zywang@cs.unc.edu

Shahriar Nirjon
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
nirjon@cs.unc.edu

ABSTRACT

We present Antler, which exploits the affinity between all pairs of tasks in a multitask inference system to construct a compact graph representation of the task set and finds an optimal order of execution of the tasks such that the end-to-end time and energy cost of inference is reduced while the accuracy remains similar to the state-of-the-art. We implement two systems: a 16-bit TI MSP430FR5994-based custom-designed ultra-low-power system and a 32-bit ARM Cortex M4/M7-based off-the-shelf STM32H747 board. We conduct both dataset-driven experiments as well as real-world deployments with these systems. We observe that Antler's execution time and energy consumption are the lowest compared to all baselines, and by leveraging the similarity of tasks and by reusing the intermediate results from the previous task, Antler reduces the inference time by 2.3X–4.6X and saves 56%–78% energy, compared to the state-of-the-art. The source code is available here[1].

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Multitask learning, task affinity, task ordering, context switch.

1 INTRODUCTION

In recent years, we see an increased number of low-resource systems that are running deep neural networks under extreme CPU, memory, time, and energy constraints [16, 24, 45]. Nowadays, it is becoming common to see multiple neural networks co-residing on the same portable, wearable, and mobile edge device in order to offer a wide variety of intelligent applications and services to the user [15, 18]. Many IoT devices have built-in voice assistants that authenticate the speaker, understand what they say, and recognize gestures, facial expressions, and emotions. Mobile vision technology [10, 27–29, 35, 38, 44] is built into many mobile and social robots that perform on-device object recognition, obstacle detection, scene classification, localization, and navigation. In order to increase the accuracy and robustness of these classifiers, numerous *multitask learning* (MTL) techniques have been proposed in the mainstream machine learning literature [37, 40]. Some of these techniques have been adopted by the embedded systems community to scale up the number of classifiers that co-exist on an embedded system [22, 25].

Unfortunately, multitask learning on low-resource embedded systems still remains a challenge. Slow CPU, scarce memory (RAM), and high overhead of external storage (flash) make the response time and the energy cost of multitask inference on these systems extremely high. To deal with these challenges, recent works [22, 25, 26] have proposed bold measures such as squeezing all [22, 25] or most [26] of the neural networks into the main memory (RAM) — in order to avoid the high overhead of storage and to rely on fast, in-memory computation for the most part. However, speedup gained in such extreme ways inevitably comes at the cost of lower accuracy and/or hidden time and energy cost that overshadows the benefit of in-memory execution. In general, state-of-the-art multitask inference techniques for low-resource systems lack two major aspects that could significantly reduce the inference time and energy consumption:

Firstly, inference tasks that run on the same system generally show affinity. For instance, a speaker identification task and a speech recognition task running on a voice assistant device share common latent subtasks such as noise compensation and phoneme identification. These overlapping subtasks should be factored out and executed only once to reduce the time and energy waste due to repeating them for both tasks. Existing works [22, 25, 26] pack multiple tasks in the main memory by sharing task constructs at the granularity of weights. They do not exploit the higher-level affinity between tasks and thus fail to recognize that even though in-memory operations are faster, repeatedly executing subtasks adds up to significantly higher overhead, especially when it involves multiple convolutional layers.

Secondly, inference tasks that run on the same system generally manifest inter-task and intra-task dependencies – requiring the system to execute tasks and subtasks in a certain order. This also creates opportunities to skip a dependent task or a subtask. For instance, voice classification tasks are routinely preceded by a lightweight voice activity detector to reduce computational overhead. Likewise, subtasks such as noise compensation and phoneme detection often precede the rest of the audio processing pipeline in typical speech classification tasks. Existing works [22, 25, 26] that merge or load tasks based on the byte-values of the weights are not capable of exploiting higher-level inter-task and intra-task

[1]Source Code: https://github.com/YuboLuo/Antler.git

dependencies, and thus they waste time and energy in executing tasks and subtasks that are unnecessary.

In this paper, we introduce Antler – which exploits the affinity between tasks in a multitask inference system to construct a compact graph representation of the task set. Unlike existing task grouping techniques that are primarily concerned with inference accuracy only, we construct task graphs considering both the accuracy as well as the time and/or energy waste from repeated execution of subtasks. Furthermore, we observe that different pairs of tasks exhibit different degrees of affinity and the cost of switching from one task to another is nonidentical. We formally prove that ordering tasks in a multitask learning scenario is NP-Complete and provide an integer linear programming formulation to solve it. We extend the formulation to include dependency constraints between tasks and subtasks. We describe a genetic algorithm to solve the optimization problem for both constrained and unconstrained cases.

In order to evaluate Antler, we develop two systems: 1) a 16-bit TI MSP430FR5994-based custom-designed ultra-low-power system, and 2) a 32-bit ARM Cortex M4/M7-based off-the-shelf STM32H747 board. We conduct dataset-driven experiments as well as real-world deployments with these systems. In the dataset-driven experiments, we compare the performance of Antler against four baseline solutions, including three state-of-the-art multitask inference systems for low-resource systems: YONO [22], NWV [25] and NWS [26] over nine datasets that are used in the literature. We observe that Antler's execution time and energy consumption is the lowest compared to all baseline systems. By leveraging the similarity of tasks and by reusing the intermediate results from previous task, Antler reduces the inference time by 2.3X – 4.6X and saves 56% – 78% energy, when compared to the baselines. In the real-world deployments, we implement two multitask inference systems having five audio and four image inference tasks. Results show that Antler reduces the time and energy cost by 2.7X – 3.1X while its inference accuracy is similar to running individually-trained classifiers within an average deviation of ±1%.

2 OVERVIEW OF ANTLER

Antler is a tool for developing efficient multitask deep learning models for low-resource systems that have extreme CPU, memory, and energy constraints. We provide an overview of Antler, deferring its technical details to later sections.

2.1 Input and Preprocessing

Tasks and Subtasks. Antler takes a set of inference tasks as the input. For a given set of tasks, $\tau = \{\tau_1(X, y_1), \tau_2(X, y_2), \cdots, \tau_n(X, y_n)\}$ defined over input domain X having *m* examples, each task τ_i maps an example, $x_j \in X$ to a class label, $y_{(j,i)}$, where $y_i = [y_{(1,i)}, \cdots y_{(m,i)}]$. A portion of a task is referred to as a *subtask*.

Preprocessing. Antler instantiates a neural network τ_i for each task using a common network architecture, which is either trained on the dataset (X, y_i), or obtained via knowledge distillation when pre-trained models are provided. A common network architecture is necessary since Antler's ultimate goal is to form a multitask neural network that consists of two or more networks sharing one or more of their network layers. A common network architecture is methodologically obtained by running a network architecture search [8] that empirically optimizes the accuracy of all networks. To speed up the search, we start with a library of popular architectures [22, 25, 26] and run a hyper-parameter search to obtain an architecture that maximizes the minimum accuracy of all tasks. Figure 1(a) shows the network instances having identical architecture but different weights.

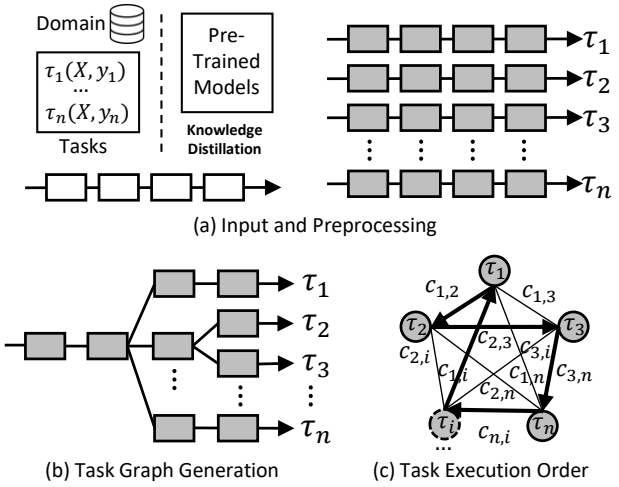


Figure 1: Overview of Antler: (a) A set of tasks defined over a domain or pre-trained models are taken as the input. A common network architecture is individually trained to produce network instances (one for each task). (b) A task graph is formed considering both accuracy and task execution cost. The task graph is retrained following standard multitask learning training practices. (c) An optimal task execution order minimizes the task execution cost.

2.2 Task Graph Generation

Task Graph, Block, and Path. Tasks in a multitask learning scenario typically share their first few layers since layers closer to the input tend to encode simpler basic patterns which are the building blocks for similar inference tasks. For example, early layers of an audio classifier for human voice encodes phonemes and morphemes that are building blocks to downstream tasks such as keyword spotting, speech recognition, and sentiment analysis.

Different pairs of tasks may share different number of layers depending on how similar the tasks are. In Antler, this is represented by a tree-like structure, which we call a *task graph*, as shown in Figure 1(b). Each rectangular box in the figure represents a *block* which consists of one or more layers. A *path* from the root (i.e., the leftmost block) to a leaf (i.e., one of the rightmost blocks) corresponds to one neural network inference task. Notice that a block may be shared by two or more neural networks.

Task Graph Generation. Antler analyzes the *affinity* between the network instances to form a task graph that has optimal sharing of blocks between tasks. Compact task graphs are generally desirable since they require less storage, save time and energy by avoiding repeated computations, and take advantage of multitask learning such as reduced overfitting and knowledge transfer which is facilitated by the shared network structures. Compact task graphs, however, generally have less network capacity due to less number

of parameters, which limits their ability to accurately classify large and complex data. Antler finds an optimal task graph that balances these two opposing forces. All tasks are re-trained using [42].

2.3 Task Execution Order

Task Graph Execution Process. In memory-constrained systems, network weights and parameters corresponding to one or more layers are brought into the main memory from the non-volatile storage (e.g., flash) prior to the execution of those layers. Hence, the cost of executing the tasks in a task graph depends not only on the number of blocks the task graph contains but also on how often each block is brought into the main memory for execution.

In Antler, memory is statically allocated in the RAM having the size of the common network architecture. Prior to executing a task τ_i, its weights and parameters are loaded into the main memory to initialize the common network architecture. Since tasks in Antler share blocks, Antler skips loading a block that is already in memory in order to reduce the read/write overhead. Additionally, intermediate results after executing each block are cached in memory buffers (one buffer after each block) to avoid repeated computation.

Optimal Task Execution Order. Since in-memory blocks are not reloaded or re-executed if the next task needs them, and since different pairs of tasks generally share different number of blocks, the order in which tasks are executed affects the total cost of executing all tasks. In Figure 1(c), the overhead of switching from one task to another is represented by a weighted complete graph whose nodes represent tasks, and weights $c_{i,j}$ on each edge represent the cost of switching between tasks. Finding the least cost ordering of the tasks is therefore equivalent to finding a least-cost Hamiltonian cycle (shown with arrows) on this graph [34].

Furthermore, tasks may have precedence constraints and conditional dependencies between them. These add additional constraints on their execution order. Antler finds an optimal ordering of tasks for a given task graph where tasks may have ordering constraints.

3 TASK GRAPH GENERATION

We describe how task graphs are generated from the network instances obtained after the preprocessing step.

3.1 Quality of a Task Graph

Task Affinity. Task affinity refers to the degree at which two tasks are similar in their data representation [7, 20, 42]. For a pair of tasks in Antler, we choose D layers, which are referred to as the *branch points*, and measure the similarity of outputs of the two networks at these branch points over a subset of K random samples from the dataset. Computing task affinity is a two-step process:

Step 1 – Each task is profiled using K data samples. At each branch point, for all pairs of samples, the dissimilarity of their representations is computed using inverse Pearson's correlation coefficient (1 - Pearson correlation) [7, 20, 42] to obtain a $D \times K \times K$ dimensional tensor. This tensor is called Representation Dissimilarity Matrix (RDM) and is flattened into a vector that encodes the data representation profile of a task. Each task must use the same number of samples to ensure that the RDMs have the same dimensions. The process is repeated for each task.

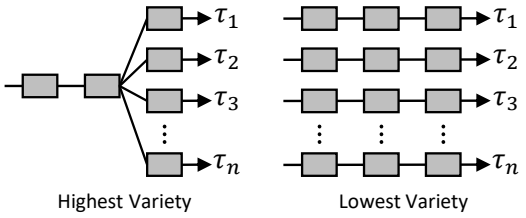


Figure 2: Variety score illustration: The left task graph puts all tasks in one group, and thus the inter-task dissimilarity within the group is the highest. The right task graph puts each task in its own group, and thus the inter-task dissimilarity within each group is the lowest (zero).

Step 2 – Affinity score for each pair of tasks is computed. At each branch point, for all pairs of tasks, the similarity of their data representation profile tensors is computed using Spearman correlation coefficient [7, 20, 42] to obtain a $D \times n \times n$ dimensional matrix where n is the number of tasks. Spearman correlation captures the nonlinear relationship between data representations. This matrix encodes the similarity between each pair of tasks at each branch point. This information is used later when tasks are grouped to form affinity-aware task graphs.

"Variety" Score of Task Graphs. We extend the definition of task affinity to task graphs. The subtree rooted at each branch point of a task graph contains a subset of tasks that share one or more blocks. In other words, all the blocks from the root of the graph to the root of the subtree are shared by all tasks that are on the leaf of the subtree. At the root of the subtree, tasks diverge and follow different paths. We quantify this using *variety* score.

We define *variety* score for the tasks under each branch point as the average maximum dissimilarity score over all pair of tasks under that branch point. The variety score quantifies the dissimilarity or misfit within tasks under each branch point. The overall variety score of a task graph is the sum of variety scores at all branch points. Computing the variety score of a task graph is a two-step process:

Step 1 – Variety score at each branch point is computed using Equation 1, where $S_{\rho,i,j}$ denotes the affinity between tasks τ_i and τ_j at branch point ρ, c_k denotes the k-th child branch, and m denotes the total number of child branches.

$$v_\rho = \frac{1}{m}\left[\sum_{k=1}^{m} \max_{i,j \in c_k} \left(1 - S_{\rho,i,j}\right)\right] \tag{1}$$

Step 2 – Variety score, v_ρ from all branch points are summed to obtain the variety score V_g for the task graph:

$$V_g = \sum_{\rho} v_\rho \tag{2}$$

Although we use *affinity*, a similarity metric, to quantify the similarity between two tasks, we use *variety* score, a measure of dissimilarity, to quantify a task graph's quality. This may seem counter-intuitive, but this is similar to intra-cluster distance in clustering algorithms that measure a cluster's impurity.

3.2 Tradeoff Analysis

Task graphs with low variety scores are desired as variety score tends to correlate with inference accuracy inversely [7]. However,

the lower the variety score of a task graph is, the higher its time, energy, and storage overhead is.

For example, the task graph in Figure 2 (left) has the highest variety score — all tasks are in one group. This is the most compact representation for any task set and has several benefits such as the least storage requirement and the least time and energy overhead when switching tasks. However, since these tasks share almost all layers, the likelihood of individual task performing accurate inferences is low.

On the other hand, the task graph in Figure 2 (right) has the lowest variety score — each task forms its own group, has the maximum time, energy, and storage overhead (as no blocks are shared), but since each task retains its weights, the inference accuracy is likely to be relatively higher.

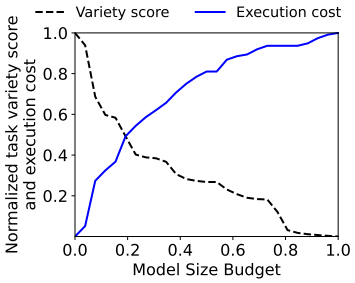


Figure 3: Tradeoff between variety score and execution cost.

Empirical Tradeoff Curve. Figure 3 shows this tradeoff using empirical data obtained from one of our experiments. We define five image classification tasks on the dataset [23] and use a five-layer CNN having 2 convolutional and 3 fully-connected layers as the common network architecture. We generate all possible task graphs, compute their variety scores, estimate their execution costs, and note their sizes.

To draw the tradeoff curve, we vary the maximum model size budget, and for each budget, we pick the task graph having the lowest variety score and whose size is within the budget. The variety score and the execution cost of that task graph are normalized and plotted on the Y-axis. Thus, we get trend lines for variety score and execution cost.

We observe that although an increased model size budget allows us to have a task graph with a lower variety score, it comes at the cost of increased execution overhead. To balance these two opposing goals, Antler uses the following formula to decide the optimal model size budget:

$$\alpha \times V_g + (1 - \alpha) \times C_g \tag{3}$$

where V_g and C_g denote normalized variety score and execution cost, and α is a hyperparameter that controls their relative strengths. In our experiments, we set $\alpha = 0.5$.

3.3 Task Graph Generation Algorithm

Given n individually trained neural networks having the same architecture, generating the task graph is a four-step offline process:

Step 1 – For each pair of tasks, their affinity score is computed at D branch points to obtain a $D \times n \times n$ matrix.

Step 2 – The set of all task graphs containing n tasks, $G_T(n)$ is generated through a recursive process. For every task graph $g \in G_T(n-1)$, where g contains $n-1$ tasks, we generate $\Lambda(g)$ new task graphs, each containing n tasks. $\Lambda(g)$ denotes the number of non-leaf internal node of g. This is because the n-th task can only branch out from one of the non-leaf internal nodes of g.

Step 3 – For each task graph, $g \in G_T(n)$, its variety score, model size, and execution cost are estimated. The variety score is obtained using Equation 2. The execution cost is estimated from empirical measurements of the cost of executing each block of the common network architecture. Execution cost estimation also requires the optimal execution order of the tasks, which is obtained using the algorithm described in the next section.

Step 4 – The variety score vs. execution cost tradeoff curve is computed. The task graph $\hat{g}$ satisfying Equation 3 is selected and retrained using [42].

4 OPTIMAL TASK EXECUTION ORDER

We describe how Antler achieves optimal task ordering for a given task graph which is an NP-complete problem.

4.1 NP Completeness

Since a task can switch to any task, a tour that contains each task exactly once can be constructed. The total cost of the tour is the sum of switching cost, $c_{i,j}$ corresponding to the edge (τ_i, τ_j). Finally, we determine if the cost is minimum. This is completed in polynomial time. Therefore, the task ordering problem is in NP.

To prove NP-hardness, we take an instance of Hamiltonian cycle [19], G(V, E) and construct an instance of task ordering problem. We construct a complete graph $G'(V, E')$, where we define E' as $E' = \{(u, v) | u, v \in V, i \neq j\}$. Note that G' is not a task graph but rather a complete graph whose nodes are the tasks from the given task graph and edges are the cost of switching between tasks. We define a cost function as:

$$\gamma(u, v) = \begin{cases} 0, & \text{if } (u, v) \in E \\ 1, & \text{otherwise} \end{cases} \tag{4}$$

Using the cost function above we can argue that if a Hamiltonian cycle exists in G, that cycle will have a cost of 0 in G' by construction. In other words, if G has a Hamiltonian cycle, we have an ordering of tasks of 0 overhead.

Conversely, we assume that G' has a tour (i.e., an ordering of tasks) of cost at most 0. Since edges in E' are 0 or 1, each edge on the tour (i.e., each task switching overhead on the chosen ordering of tasks) must be of cost 0 as the cost of the tour is 0. Therefore, the tour contains only edges in E.

This proves that G has a Hamiltonian cycle if and only if G' has an ordering of tasks of at most 0 overhead.

4.2 Task Execution Order

Significance. Task graphs provide a compact representation of tasks but do not explicitly impose any order for executing the tasks. We observe that not all n! execution orders of n tasks cost the same as different pairs of tasks in a task graph generally share different number of blocks. Figure 4 shows an example task graph and the cost of task switching is shown on the weighted complete graph on its right. For simplicity, we assume the cost of loading and executing

each block is 1 unit. We observe that executing the tasks in the order: $\tau_2 \rightarrow \tau_1 \rightarrow \tau_3 \rightarrow \tau_5 \rightarrow \tau_4$ incurs significantly higher overhead when compared to the optimal order: $\tau_1 \rightarrow \tau_5 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$.

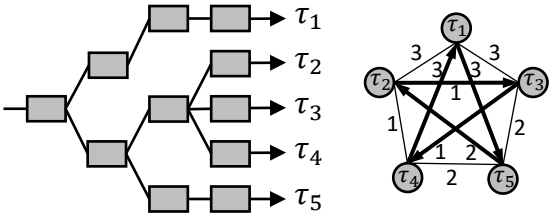


Figure 4: Switching cost is different for different task pairs.

Cost Matrix. The cost matrix $C_{n,n}$ is an $n \times n$ matrix, in which, each entry $c_{i,j}$ denotes the additional cost of loading and executing task τ_j, given that the last executed task was τ_i. These values are obtained empirically by measuring the time or energy overhead of switching between all pairs of tasks. The cost matrix explains why task execution order matters. If all entries of the cost matrix were the same, the execution order of the tasks would not matter. This may only happen in extreme cases when tasks are too similar (i.e., they share all intermediate layers) or too different (i.e., they share nothing). The cost matrix is used to find the optimal execution order of the tasks.

$$C_{n,n} = \begin{pmatrix} 0 & c_{1,2} & c_{1,3} & \cdots & c_{1,n} \\ - & 0 & c_{2,3} & \cdots & c_{2,n} \\ - & - & 0 & \cdots & c_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ - & - & - & \cdots & 0 \end{pmatrix} \tag{5}$$

4.3 Optimal Task Execution Order

Given a set of n tasks, $\tau = \{\tau_1, ..., \tau_n\}$ and cost matrix, $C_{n,n}$, our goal is to find an optimal ordering of the tasks so that the total cost of executing the task set is minimized.

Mathematical Formulation. We define a binary variable x_{ij} to denote whether a task switching happens from τ_i to τ_j:

$$x_{i,j} = \begin{cases} 1, & \text{if a task switch happens from } \tau_i \text{ to } \tau_j \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

The task ordering problem is formulated as the following integer linear programming problem:

$$\text{minimize} \quad \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} c_{i,j} x_{i,j}$$

$$\text{subject to} \quad \sum_{i=1, i\neq j}^{n} x_{i,j} = 1 \qquad j = 1, \cdots, n$$

$$\sum_{j=1, j\neq i}^{n} x_{i,j} = 1 \qquad i = 1, \cdots, n$$

$$\sum_{i\in Z} \sum_{j\neq i, j\in Z} x_{i,j} \leq |Z| - 1 \qquad \forall Z \subsetneq \{1, \ldots, n\}, |Z| \leq 2$$

where the objective function minimizes the overall task switching overhead for all tasks. The first two constraints ensure that tasks are executed only once. The last constraint ensures that there is no subset that can form a sub-tour and thus the final solution is a single execution order and not a union of smaller sub-orders [34].

4.4 Inter-Task Dependencies

We have thus far discussed the scenario where tasks are executed in an orderly manner. In many real-world systems, however, there are additional constraints that affect task execution decision: *precedence* and *conditional* constraints.

Precedence Constraints. These constraints dictate that certain tasks (prerequisites) must be executed prior to some other tasks (dependents). These constraints are static. They are determined at the design time of the classifiers. We express these constraints using directed edges on a graph as shown in Figure 5(a) where each node, τ_i denotes a task and each edge, (τ_i, τ_j) denotes a precedence constraint such that τ_i must finish before τ_j starts.

(a) Precedence Constraint **(b) Conditional Constraint**

Figure 5: Precedence and conditional constraints are expressed by directed edges. Unlike precedence constraints, conditional constraints affect the task switching cost.

To account for the precedence constraints, we augment the optimization problem described in Section 4.3 with additional constraints. We assume a given set of precedence constraints, P of tuples of tasks, $(\tau_i, \tau_j) \in P \subseteq |\tau| \times |\tau|$, for which, the task, τ_j must start after τ_i finishes. For each task-pair, (τ_i, τ_j), we define the remaining execution time to finish τ_j, given that τ_i has already finished, as $d_{i,j}$. To formally incorporate precedence constraints, we define a binary variable:

$$s_{i,t} = \begin{cases} 1, & \text{if } \tau_i \text{ starts by time t} \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

The following constraint ensures the inclusion of all precedence constraints:

$$\sum_{t' \leq t} s_{i,t'} \quad \geq \quad \sum_{t' \leq t + d_{i,j}} s_{j,t'} \tag{8}$$

Conditional Constraints. These are a special type of precedence constraints where the decision to execute a dependent task depends on the outcome of a prerequisite task. These constraints manifest dynamically at runtime when a prerequisite task finishes and its inference result is available.

Conditional constraints are also represented by directed edges. However, these constraints being dynamic, to accommodate their effect on the task switching cost (which in turn affects the task ordering), we utilize their probability of execution. We estimate this probability offline over a dataset by counting the fraction of the time a dependent task is executed after its prerequisite task finishes. We assume a given set of conditional constraints, R of triplets $(\tau_i, \tau_j, p_{i,j}) \in R \subseteq |\tau| \times |\tau| \times [0, 1]$, where $(\tau_i, \tau_j) \in P$ and $p_{i,j}$ is the probability of executing τ_j after τ_i finishes. This probability

is used to determine the expected cost of switching to a dependent task as shown in Figure 5.

Since conditional constraints are a special type of precedence constraints, we include the same linear constraints as in Equation 8 to account for them.

4.5 Solving the Optimization Problem

We describe two alternative solvers which run offline on a high-end machine.

Brute-force Solver. In extremely resource-constraint systems, we expect fewer inference tasks. In such cases, a brute-force solver would suffice that generates all possible permutations of the tasks, discards the permutations that violate precedence constraints, and selects the best ordering that maximizes a fitness score. We define *fitness score* for each permutation that does not violate the precedence constraints as the sum of task-switching overheads:

$$f(\pi_1, \cdots, \pi_n) = \sum_{1 \le i < n} c_{\pi_i, \pi_{i+1}} \tag{9}$$

where, π_i refers to the task that executes at position i.

For conditional constraints, we adjust the fitness score to account for the probabilistic execution of dependent tasks by multiplying the probability to the switching cost:

$$f(\pi_1, \cdots, \pi_n) = \sum_{1 \le i < n} p_{\pi_i \pi_{i+1}} \; c_{\pi_i, \pi_{i+1}} \tag{10}$$

The complexity of the brute-force solver is $O(n!)$.

Genetic Algorithm Solver. Although a brute-force solver is reasonably fast for a small number of tasks, the solver is repeatedly invoked during the task graph generation step – once for each task graph as they are enumerated – which as a whole takes significant time. Furthermore, in the future, the number of inference tasks running on low-resource systems could increase significantly as technology advances. Hence, we propose an efficient, scalable, genetic algorithm-based solver [1, 2, 33, 41, 46] to solve the task ordering problem. The advantage of adopting genetic algorithm is that the same solution framework is customized to solve all cases of the optimal task ordering problem, i.e., with and without precedence and conditional constraints.

The algorithm begins with a set of individuals or candidate solutions which is called a population. We define the j-th individual as $\pi^j = (\pi_1^j, \pi_2^j, \ldots, \pi_n^j)$, where π_i^j is the task that executes at the i-th position. We define the fitness of each individual using Equation 9 (or, Equation 10 for conditional constraints). At each round of the algorithm, we select the best K pairs of individuals based on their fitness scores; and for each pair, we randomly choose a crossover point, $k \in \{1, 2, \cdots, n\}$ and swap the first k elements of the pair to generate their offspring; and for each offspring, we perform mutation by swapping the values at two randomly chosen indices, $\{m_1, m_2\}, m_1, m_2 \in \{1, 2, \ldots, n\}$; and finally, we discard all individuals that are not a valid ordering. This whole process is repeated until we reach a point when the fitness score of the best solution does not improve anymore. The complexity of the genetic algorithm solver is $O(nkp)$, where n is the number of tasks, k is the number of generations, p is the population size.

5 EVALUATION

This section describes the evaluation of the proposed algorithms as well as the end-to-end system performance.

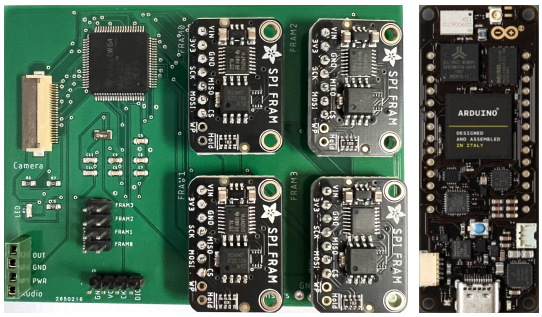
5.1 Experimental Setup

Dataset and Network Architecture. We use the datasets and network architectures used in recent multitask inference literature for low-resource systems [22, 25, 26]. Table 1 provides a summary. The network architecture shown on the table (rightmost column) is used as the common network architecture in Antler and each task on a dataset corresponds to recognizing one class. All datasets have 10 tasks, except for HHAR which has six. We use 80% of the data for training and 20% for testing.

Modality	Dataset	Architecture
Image	MNIST	LeNet-5
	F-MNIST	LeNet-5
	CIFAR-10	DeepIoT
	SVHN	Neuro.Zero
	GTSRB	LeNet-4
Audio	GSC-v2	KWS
	ESC	Mixup-CNN
	US8K	TSCNN-DS
IMU	HAAR	DeepSense

Table 1: Datasets and Network Architectures.

Baselines for Comparison. We use four baselines for comparison: YONO [22], NWV [25], NWS [26] and Vanilla. The first three are the state-of-the-art. Vanilla refers to independently trained classifiers running sequentially on the system. We use NWV and NWS in both 16-bit and 32-bit experiments. Since their source codes are not available, we use our own implementation and cross-check with their reported results to ensure that they are consistent with ours. We use YONO only in 32-bit experiments. Since our 32-bit hardware platform is identical to YONO's, we use the reported results from their work.



(a) 16-bit Custom **(b) 32-bit Portenta**

Figure 6: Hardware platforms.

Evaluation Platforms. We use the two platforms listed on Table 2: a 16-bit custom made MSP430FR5994-based system and a 32-bit STM32H747. Data samples are pre-loaded into the non-volatile memory from where they are read and executed. We measure the time and energy consumption by connecting a 100Ω resistor in

Platform	Custom PCB	STM32H747
CPU	MSP430FR5994	ARM Cortex-M4/M7
	16-bit, ≤16MHz	32-bit, ≤480MHz
Memory	8KB SRAM	1MB SRAM
	512KB+2MB FRAM	2MB eFlash
Power	1.8V - 3.6V	3.3V
	118 uA/MHz (active)	100 mA

Table 2: Hardware specification.

series with the board and by measuring the voltage across the resistor with Analog Discovery 2. We run all off-line experiments on a server having 12 Intel i7-7800X CPUs and 32GB RAM.

5.2 Algorithm Scalability Analysis

This section evaluates the task graph generation and task ordering algorithms.

Effect of Network Structure. We evaluate how different network structures might affect the task graph generation algorithm. Ideally, if two tasks are correlated, they should have high task affinity irrespective of the network structure used by Antler. We empirically test this hypothesis by comparing inter-task similarity matrices corresponding to three different common network architectures. For each network, we first calculate RDMs at each layer and use the averaged RDM as the final representation for each task. Then, the affinity score is computed for each pair of tasks using their corresponding RDM. We use CIFAR-10 [21] dataset and vary the size and number of layers of the common network architecture. The results are shown in Figure 7.

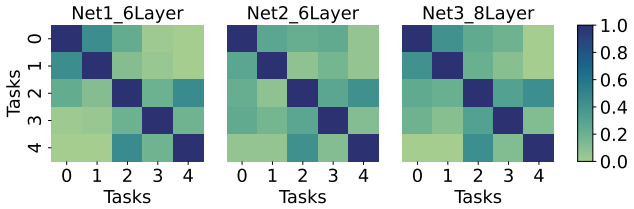


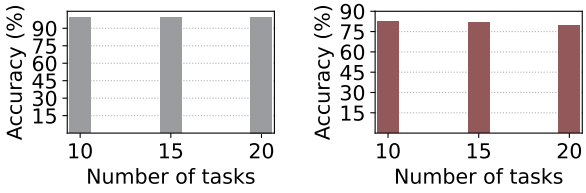
Figure 7: Inter-task affinity for different networks.

We observe that the relative inter-task similarity is consistent irrespective of the choice of networks. This demonstrates that the task graph generation algorithm is not sensitive to the choice of common network architecture.

Effect of Number of Tasks. We evaluate how the number of tasks affects the inference accuracy. Since we target low-end microcontrollers, we explore the effect of the number of tasks for up to 20 tasks. We use both image and audio datasets (GTSRB [39] and GSC-v2 [43]). The results are shown in Figure 8.

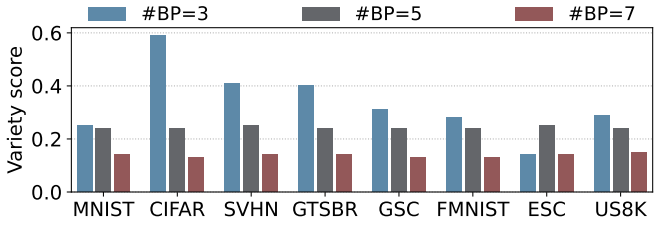
We observe that the image classification task has almost no accuracy degradation as the number of tasks increases. The audio classification task incurs a very small accuracy decrease of -2.7%.

Effect of Branch Points. We evaluate the effect of branch points by performing a sensitivity analysis of variety and execution cost. We use execution time as the cost. We vary the number of branch points, BP = {3, 5, 7}. Results are shown in Figure 9(a) and 9(b).

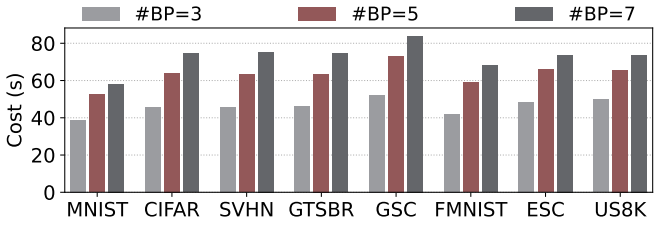


(a) Image (GTSRB) (b) Audio (GSC)

Figure 8: Effect of number of tasks.



(a) Variety Score



(b) Execution Cost on 16-bit MSP

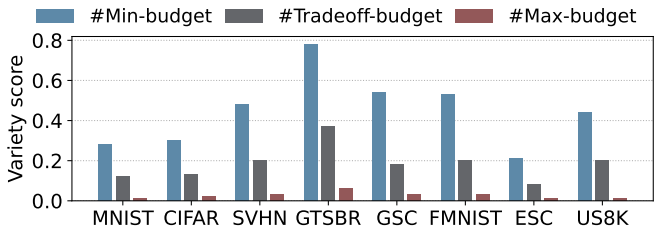
Figure 9: Effect of number of branch points.

We observe that more branch points improve the variety score (lower is better) but worsen the overhead. This is because more branch points decompose and group tasks at a finer granularity which causes more tasks to branch out at deeper layers and thus decreases task-switching efficiency.

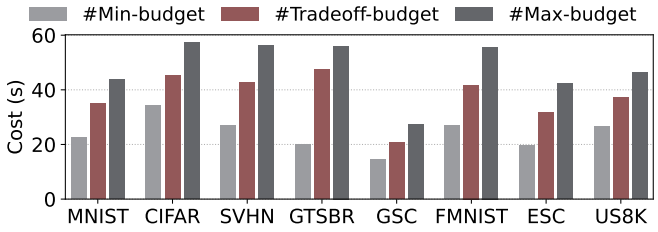
Variety Score vs. Execution Cost Trade-off. Figure 10 shows the trade-off between variety score and execution cost for eight datasets. We compare three network size budgets: two extreme cases of minimum and maximum budget and a trade-off budget where variety and cost trend lines intersect. We observe that a low budget favors execution cost, a high budget favors variety, and the trade-off budget balances the variety score and execution cost.

Effect of New Data Points. In real-world machine learning scenarios, after the system is deployed, it may encounter new data that have a different distribution than the training data. The new data points may change one or more tasks in such a way that it affects the inter-task affinity. As a consequence, the task graph constructed during the offline phase may no longer be optimal. We conduct an experiment where we create this scenario by randomly choosing a subset of the training data to construct the task graph and then comparing it with a task graph that is constructed using the entire training dataset. Figure 11 shows the result.

We observe that as long as at least 40% of the training data is used to construct the task graph, it remains pretty stable. The relative affinity between different pairs of tasks beyond this point does not change so much that it can alter the task graph. This experiment



(a) Variety Score



(b) Execution Cost on 16-bit MSP

Figure 10: Variety score vs. execution cost tradeoff.

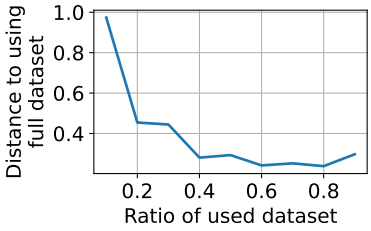


Figure 11: Effect of new data points.

also reveals that when an inadequate amount of data is used, the constructed task graph is not optimal. In this case, Antler's time and energy efficiency will also be suboptimal.

Performance of Genetic Algorithm. We evaluate the performance of the genetic algorithm for task ordering. We use a popular public dataset for Traveling Salesperson Problems (TSP) called the TSPLIB [36] and repurpose it for task ordering problems. This dataset already contains test cases that have precedence constraints. To include conditional constraints, we add weights to the graph's edges. Table 3 compares our results with the ground truth for all three cases of task ordering problems. Our result is identical to the ground truth for all cases except for a few conditional constraint cases with a 5% deviation.

5.3 Comparison with Baseline Solutions

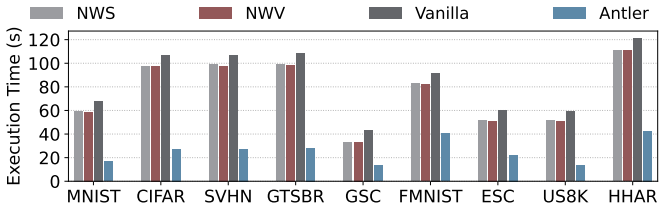
Execution Time and Energy Cost. We compare the execution time and energy consumption of Antler against the baselines in Figure 12 and Figure 13, respectively. The Y-axis shows the total execution time (or energy) to execute all tasks for an input. We report results for both 16-bit and 32-bit systems. The energy consumption is estimated by connecting a resistor in series and then measuring the voltage across the resistor and the system separately. Since

Variant	Dataset	Node/ Pre/Cnd	Optimal	Antler
Regular	FIVE	5/0/0	19	19
	P01	15/0/0	291	291
	GR17	17/0/0	2085	2085
Precedence	ESC07	9/6/0	2125	2125
	ESC11	13/3/0	2075	2075
	br17.12	17/12/0	55	55
Conditional	ESC07	9/6/3	982	982
	ESC11	13/3/3	1901	2000
	ESC12	14/7/3	1398	1423

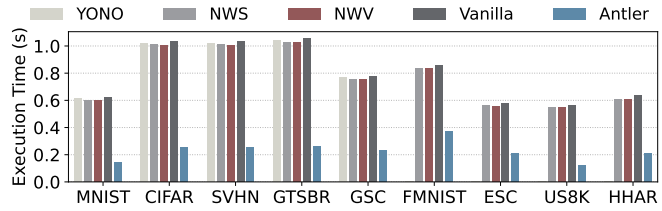
Table 3: Evaluation of genetic algorithm for task ordering. Node/Pre/Cnd represent the number of nodes, precedence, and conditional constraints, respectively. The last two columns represent the cost of the optimal result and that of Antler. The lower the better.

inference time and energy on microcontrollers is pretty stable, the error bars are practically zero in these figures.

We observe that while the general trend remains the same in both systems, the execution time on STM32H747 is 100X faster. On both systems, Antler's execution time is the lowest. This is because Antler leverages the similarity of tasks and reuses the intermediate results to reduce the execution time by 2.3X – 4.6X which baseline solutions do not. Even though NWV and YONO perform complete in-memory inferences and have zero switching overhead, they fall short of Antler as the cost of repeatedly executing shared subtasks is higher, especially when it involves convolution layers. We observe similar pattern in energy consumption. Overall, Antler saves 56%–78% energy compared to the baselines.



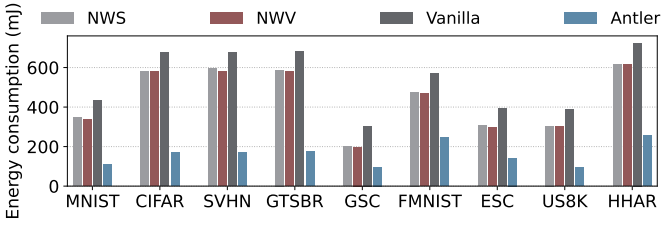
(a) 16-bit MSP430FR5994



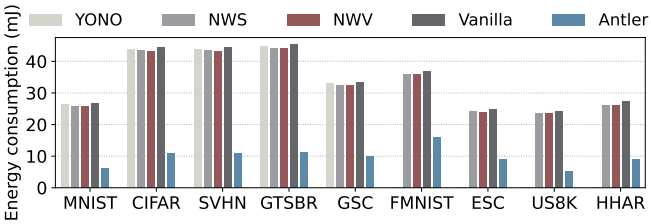
(b) 32-bit STM32H747

Figure 12: Comparison of execution time.

Breakdown of Time and Energy Overhead. We breakdown the total time and energy cost into two parts: inference-only cost that corresponds to in-memory execution of the networks and switching overhead that corresponds to loading weights from external memory. We compare Antler with Vanilla and NWS since the other two (NWV and YONO) do not use external memory and thus have no switching cost. Results are shown in Figure 14.

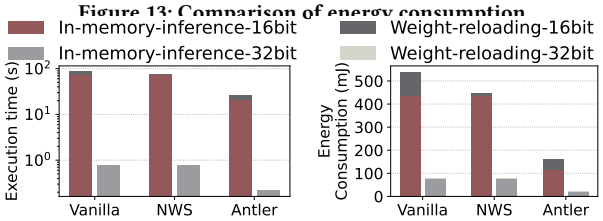


(a) 16-bit MSP430FR5994



(b) 32-bit STM32H747

Figure 13: Comparison of energy consumption



(a) time breakdown (b) energy breakdown

Figure 14: Time and energy cost breakdown.

The Y-values in Figures 14(a) and 14(b) are averaged over all datasets. We observe that 32-bit STM32H747 has very little weight reloading overhead (the striped area on top of each bar is almost invisible) for both time and energy breakdown. The time and energy cost related to weight reloading in NWS is also negligible as it only has around 7% of the total weights stored in external memory. Antler's time and energy cost related to weight-reloading is 54%-56% less than Vanilla.

Inference Accuracy. We compare the inference accuracy of all systems in Figure 15. Examples of task graphs are shown in Figure 16. The accuracy is averaged over all tasks. Antler's inference accuracy is similar to YONO, NWS, and Vanilla within a margin of ±3% deviation. Recall that Antler's target is to reduce the time and memory cost of inference while achieving a high accuracy. In this case, all classifiers show reasonably high accuracy of over 90%, except for NWV whose accuracy does not scale with the number of tasks. YONO does not use the later five datasets and thus its accuracy could not be included in Figure 15 for those datasets.

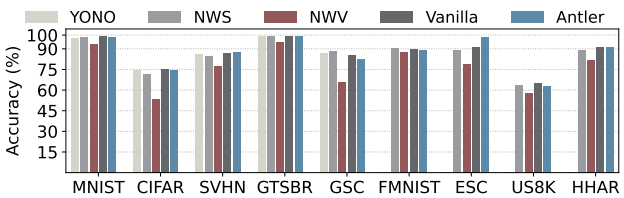


Figure 15: Comparison of inference accuracy.

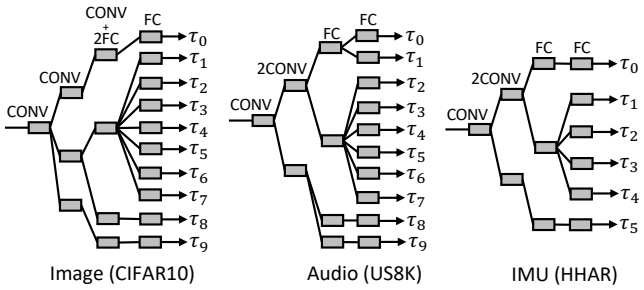


Figure 16: Example task graphs for different modalities.

Memory Efficiency. We measure the total memory consumption of all tasks for each baseline and summarize in Table 4. We observe that Antler consumes more memory than NWS, NWV and YONO. This is because NWV and YONO perform complete in-memory execution and thus they are limited by the size of the RAM. Unlike them, Antler and NWS are able to utilize external memory and put no hard restrictions on the total size of the tasks. Antler consumes significantly less memory than Vanilla since Antler reduces memory consumption by exploiting the task affinity.

System	Vanilla	Antler	NWS	NWV	YONO
Memory (KB)	1328	587	213	140	114

Table 4: Comparison of memory consumption.

Knowledge Distillation. We investigate a special scenario where the original training dataset is not available, and only pre-trained models are provided. We employ knowledge distillation to deal with this case. The pre-trained model serves as the teacher model, and our binary classification tasks are the student model (denoted by Stu-Distilled). We also train the same student model on the original dataset (denoted by Stu-Labeled) to show the difference between learning from the teacher model and from the original dataset. For a comprehensive investigation, we choose three popular network architectures, i.e., VGG, MobileNet, and ResNet, and CIFAR-10 dataset. The results are shown in Figure 17. We observe that with knowledge distillation, we achieve similar accuracy as in training from the original training dataset.

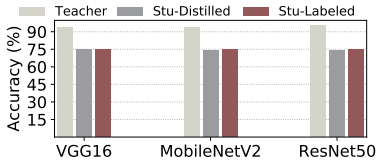


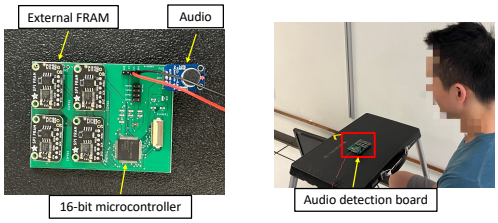
Figure 17: Effect of knowledge distillation on accuracy.

6 REAL-WORLD DEPLOYMENT

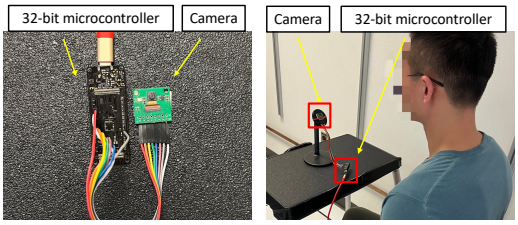
We deploy Antler in two real-world multitask learning scenarios that involve audio and image classification tasks.

6.1 Multitask Audio Inference System

Inference Tasks. We implement five audio-based tasks: 1) a speaker presence detection task (τ_0) which detects if there is human voice in the audio, 2) a command detection task (τ_1) which detects eleven commands {yes, no, up, down, go, stop, left, right, on, off, Alexa}, 3)



(a) Multitask Audio Inference System

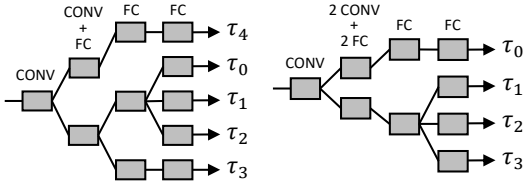


(b) Multitask Image Inference System

Figure 18: Deployment Setup.

a speaker identification task (τ_2) which identifies who is speaking (five speakers), 4) an emotion classification task (τ_3) which classifies the audio into three emotions {positive, negative, neutral}, and 5) a distance classification task (τ_4) which tells whether the speaker is close to or far from the device.

System Setup. We use the 16-bit custom MSP430FR5994 [14] to conduct the audio-based experiments as shown in Figure 18(a). Audio signal is sampled at 2KHz and converted to a feature map having a window-length of 128ms and a stride of 64ms after performing the FFT.



(a) Audio Inference Graph (b) Image Inference Graph

Figure 19: Multitask inference graphs.

Data Collection and Network Training. Five volunteers (four male and one female) participate in this experiment. We have followed Institutional Review Board (IRB) approved protocol to conduct this study. We collect 15 samples for each task from each volunteer. We use 80% data for training and 20% for testing from each participant. We design a 5-layer CNN having 2 convolutional and 3 dense layers and pre-train it on [43] prior to training on our own dataset. We use 3 branch points and $\alpha = 0.5$.

6.2 Multitask Image Inference System

Inference Tasks. We implement four image classification tasks: a human presence detection task (τ_0) which detects human faces in an image, a mask detection task (τ_1) which detects if the person is wearing a mask, a person identification task (τ_2) which recognizes a person's face (5 volunteers), and an emotion recognizer (τ_3) which classifies three emotions as in the audio inference system.

System Setup. We use off-the-shelf 32-bit STM32H747 H7 as shown in Figure 18(b). Images are taken with a HM01B0 camera module and has the dimensions of 64×64 pixels.

Data Collection and Network Training. Data collection and network training processes are identical to the audio inference system except for the neural network which is a 7-layer CNN having 3 convolutional and 4 dense layers and is pre-trained on [9].

6.3 Evaluation Results

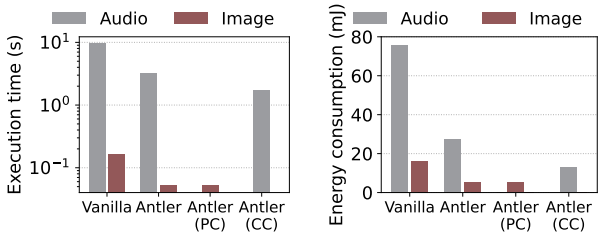
Task Decomposition and Grouping. Figure 19 shows task graphs for both applications. There are 4 blocks in each task graph (for 3 branch points). One of the blocks (second block) contains multiple layers. This is unlike task graphs observed earlier in Section 5, where deeper layers are lumped into the same block. Overall, having more layers lumped in earlier blocks decreases execution cost but may decrease accuracy as well. Antler finds a trade-off point between the two to optimize both accuracy and cost.

We observe that in the audio inference graph, Figure 19(a), τ_4 is isolated from all other tasks, which is logical since τ_4 is a distance classification task which relies on very different audio features compared to human voice classification tasks. Likewise, in the image inference graph, Figure 19(b), τ_0 is isolated since it represents human presence detection task which requires very different facial features compared to other tasks.

Task Dependency and Ordering. We include a precedence constraint in the image inference system that the presence detection task (τ_0) must be executed before any other task. Additionally, in the audio inference system, we make presence detection a conditional constraint such that other tasks are executed at 80% probability.

The ordering of tasks in audio and image inference systems are: $\tau_0 \rightarrow \tau_3 \rightarrow \tau_4 \rightarrow \tau_2 \rightarrow \tau_1$ and $\tau_0 \rightarrow \tau_3 \rightarrow \tau_1 \rightarrow \tau_2$, respectively, and they are just one of several orderings that yield the best performance for these tasks.

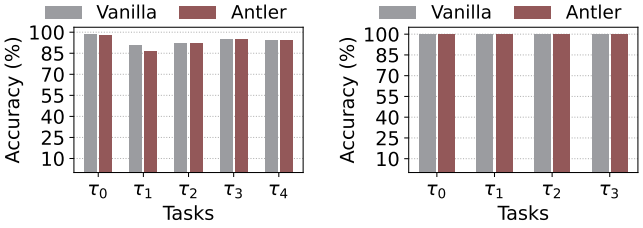
Inference Time and Energy Consumption. We evaluate Antler's execution time and energy consumption for three cases: Antler having no constraints, Antler-PC having precedence constraints, and Antler-CC having conditional constraints, and compare their performance with the Vanilla system. Figure 20 shows that Antler yields 2.7X – 3.1X reduction in time and energy costs and the results are consistent across both systems. Antler-PC has the same overhead as Antler because there are only four tasks and the execution order with precedence constraint is already in the optimal ordering. For Antler-CC, overhead decreases as tasks are skipped occasionally based on the conditional probability.



(a) Execution Time (b) Energy Consumption

Figure 20: Execution time and energy consumption.

Inference Accuracy and Memory Usage. Figure 21 shows the average accuracy of all tasks for both systems. We observe that in the audio inference system, except for the command detection task, all tasks have over 90% accuracy. This is because the command detection task is the hardest of these tasks with eleven class labels. The accuracy of both Antler and Vanilla are very similar within an average deviation of ±1%.



(a) Audio Inference System **(b) Image Inference System**

Figure 21: Inference accuracy of audio and image classifiers.

The memory usage of both systems are shown in Table 5. We observe that the memory usage of Antler is approximately half of Vanilla's, which is consistent with earlier results from the dataset-driven experiments in Section 5.

System		Vanilla	Antler
Memory (KB)	Audio	397	202
	Image	445	222

Table 5: Memory usage.

7 DISCUSSION

Fine-Grained Task Decomposition. Although Antler uses only three branch points to in our experiments, it can be easily extended to have more fine-grained decomposition to form more compact task graphs. Such fine-grained decomposition of tasks might be necessary when the number of tasks is above 20.

Optimization Alternatives. Antler's task graph formation and task ordering are formed as two independent optimization problems and solved independently. We choose this design to make task ordering flexible so that dynamic constraints can be handled flexibly. An alternative approach such as joint optimization of task graph formation and execution order determination is possible but the downside is that if task dependencies change, the optimization problem has to be solved again.

Generalization to Other Systems. Although Antler is motivated by the constraints of a low-resource system, some of the techniques such as affinity-aware task graphs that execute under constraints and optimal ordering of tasks should apply to server-grade larger multitask inference systems as well. Cloud-based inference systems that execute more complex and larger number of inference tasks, exploiting task affinity could help reduce a server's response time.

Generalization to Other Workloads. Although Antler's scope is limited to neural networks, the concept of tasks, task affinity, and task graphs are generalizable to any workload where a task can be factored into subtasks, compared, and merged. Antler is readily applicable to many non-neural classifiers such as decision tree and random forest that process data in stages.

Improvement Over Antler. Antler's advantage over the state-of-the-art in-memory multitask learning systems [22, 25] is its ability to reduce time and energy cost while offering the same level of accuracy. Antler, however, consumes more memory than these systems. We envision that Antler will inspire new techniques that will not only reduce time and energy overhead of multitask inference on embedded systems but also improve inference accuracy and perform complete in-memory execution of tasks.

These devices will be self-contained, long lasting, and feature-packed with several on-device classifiers.

8 RELATED WORK

Single Network Compression. This class of algorithmic techniques refer to approaches that take one DNN at a time and compress it down to a desired size by employing a wide variety of methods such as knowledge distillation, low-rank factorization, pruning, quantization, compression with structured matrices, network binarization, and hashing [3, 4, 11, 12]. The disadvantages of this technique are: first, there is no cross-DNN knowledge sharing or joint compression that trains multiple DNNs together; second, since each DNN is compressed individually using different compression methods, this technique is not scalable and they do not have the advantages of multitask learning; third, a significantly compressed DNN does not run nearly as significantly faster since most parameters are pruned in the dense layers while convolutional layers consume most computation time [12].

Multiple Networks Compression. This class of algorithmic techniques compress multiple DNNs together; e.g., PackNet [31] compresses multiple DNNs to a single DNN with iterative pruning of redundant parameters in DNNs to remove weights that can be used by other tasks. The number of DNNs that can participate in the process, however, is limited when free weights fall short as the number of DNNs increase while a single network is maintained. [5] proposes a technique that merges DNNs by integrating convolutional layers. However, their technique works for two networks only and it requires layer alignment for merging. Learn-them-all [17] trains a single network to deal with many tasks simultaneously. However, choice of a suitable architecture is generally hard for learning all the tasks apriori. Besides, this requires a large training data from different types and sources which is a tedious task.

There exists multiple studies on sharing weights among a set of DNNs, e.g., MultiTask Zipping [13] combines DNNs for cross-model compression with a layer-wise neuron sharing; Sub-Network Routing [30] modularizes the shared layers into multiple layers of sub-networks; Cross-stitch Networks [32] apply weight sharing [6] after pooling and fully-connected layers of two DNNs. The scope and methods of weight sharing in these works are limited by the choice of network architecture and task type.

Most Relevant Works. The three most relevant state-of-the-art systems to Antler are NWV [25], NWS [26], and YONO [22]. NWV and YONO propose complete in-memory execution of DNNs on memory-constrained systems. NWS extends NWV by allowing some of the high-significance weights into the flash memory to increase the accuracy of NWV. NWS essentially points out that complete in-memory packing and execution of DNNs in MCUs is not accurate. All three approaches fail to leverage the affinity

among tasks and their dependencies, and thus repeatedly execute overlapping common subtasks that significant increases the time and energy cost of multitask inference which is avoided by Antler.

9 CONCLUSION

We envision a future where a wide variety of ultra-low-power sensing and inference systems will sense and classify every aspect of our personal and physical world. To realize this vision, we need to significantly lower the time and energy cost of running multiple neural networks on low-resource systems while ensuring that their application-level performance does not degrade. To achieve this goal, we propose Antler, which is the first system that exploits the similarity among a set of machine learning tasks to identify overlapping substructures in them that is combined and executed in an optimal order to reduce the execution time by 2.3X – 4.6X and the energy overhead 56% – 78% when compared to the state-of-the-art multitask learners for low-resource systems.

10 ACKNOWLEDGEMENT

This work was supported, in part, by grants NSF CAREER-2047461 and NIH 1R01LM013329-01.

REFERENCES

[1] M. Ab Rashid, M. Jusop, N. Mohamed, and F. Romlay. Optimization of travelling salesman problem with precedence constraint using modified ga encoding. *Advanced Science Letters*, 24(2):1484–1487, 2018.

[2] Z. H. Ahmed and S. N. Pandit. The travelling salesman problem with precedence constraints. *Opsearch*, 38(3):299–318, 2001.

[3] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker. Learning efficient object detection models with knowledge distillation. In *Advances in Neural Information Processing Systems*, pages 742–751, 2017.

[4] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[5] Y.-M. Chou, Y.-M. Chan, J.-H. Lee, C.-Y. Chiu, and C.-S. Chen. Merging deep neural networks for mobile devices. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1686–1694, 2018.

[6] L. Duong, T. Cohn, S. Bird, and P. Cook. Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 845–850, 2015.

[7] K. Dwivedi and G. Roig. Representation similarity analysis for efficient task taxonomy & transfer learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12387–12396, 2019.

[8] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1), 2019.

[9] T. B. E. L.-M. G.B. Huang, M. Ramesh. Lfwcrop face dataset, 2022.

[10] A. Giusti, J. Guzzi, D. C. Cireşan, F.-L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, et al. A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, 2015.

[11] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.

[12] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[13] X. He, Z. Zhou, and L. Thiele. Multi-task zipping via layer-wise neuron sharing. In *Advances in Neural Information Processing Systems*, pages 6016–6026, 2018.

[14] T. Instruments. Msp430fr5994, 2022.

[15] B. Islam, Y. Luo, S. Lee, and S. Nirjon. On-device training from sensor data on batteryless platforms. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, pages 325–326, 2019.

[16] B. Islam, Y. Luo, and S. Nirjon. Zygarde: Time-sensitive on-device deep intelligence on intermittently-powered systems. 2019.

[17] L. Kaiser, A. N. Gomez, N. Shazeer, A. Vaswani, N. Parmar, L. Jones, and J. Uszkoreit. One model to learn them all. *arXiv preprint arXiv:1706.05137*, 2017.

[18] F. Kawsar, C. Min, A. Mathur, A. Montanari, U. G. Acer, and M. Van den Broeck. esense: Open earable platform for human sensing. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 371–372. ACM, 2018.

[19] B. H. Korte, J. Vygen, B. Korte, and J. Vygen. *Combinatorial optimization*, volume 1. Springer, 2011.

[20] N. Kriegeskorte, M. Mur, and P. A. Bandettini. Representational similarity analysis-connecting the branches of systems neuroscience. *Frontiers in systems neuroscience*, page 4, 2008.

[21] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[22] Y. D. Kwon, J. Chauhan, and C. Mascolo. Yono: Modeling multiple heterogeneous neural networks on microcontrollers. *arXiv preprint arXiv:2203.03794*, 2022.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] S. Lee, B. Islam, Y. Luo, and S. Nirjon. Intermittent learning: On-device machine learning on intermittently powered system. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4):1–30, 2019.

[25] S. Lee and S. Nirjon. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 175–190, 2020.

[26] S. Lee and S. Nirjon. Weight separation for memory-efficient and accurate deep multitask learning. In *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2022.

[27] Y. Liao, S. Kodagoda, Y. Wang, L. Shi, and Y. Liu. Understand scene categories by objects: A semantic regularized scene classifier using convolutional neural networks. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 2318–2325. IEEE, 2016.

[28] Y. Luo and S. Nirjon. Spoton: Just-in-time active event detection on energy autonomous sensing systems. *Brief Presentations Proceedings (RTAS 2019)*, 9, 2019.

[29] Y. Luo and S. Nirjon. Smarton: Just-in-time active event detection on energy harvesting systems. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 35–44, Los Alamitos, CA, USA, jul 2021. IEEE Computer Society.

[30] J. Ma, Z. Z. J. C. A. Li, and L. Hong. Snr: Sub-network routing for flexible parameter sharing in multi-task learning. 2019.

[31] A. Mallya and S. Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

[32] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003, 2016.

[33] C. Moon, J. Kim, G. Choi, and Y. Seo. An efficient genetic algorithm for the traveling salesman problem with precedence constraints. *European Journal of Operational Research*, 140(3):606–617, 2002.

[34] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[35] J. Redmon and A. Angelova. Real-time grasp detection using convolutional neural networks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1316–1322. IEEE, 2015.

[36] G. Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[37] S. Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[38] D. Sarikaya, J. J. Corso, and K. A. Guru. Detection and localization of robotic tools in robot-assisted surgery videos using deep neural networks for region proposal and detection. *IEEE transactions on medical imaging*, 36(7):1542–1549, 2017.

[39] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *The 2011 international joint conference on neural networks*, pages 1453–1460. IEEE, 2011.

[40] X. Sun, R. Panda, R. Feris, and K. Saenko. Adashare: Learning what to share for efficient deep multi-task learning. *Advances in Neural Information Processing Systems*, 33:8728–8740, 2020.

[41] J. Sung and B. Jeong. An adaptive evolutionary algorithm for traveling salesman problem with precedence constraints. *The Scientific World Journal*, 2014, 2014.

[42] S. Vandenhende, S. Georgoulis, B. De Brabandere, and L. Van Gool. Branched multi-task networks. *arXiv preprint arXiv:1904.02920*, 2019.

[43] P. Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[44] L. Xie, S. Wang, A. Markham, and N. Trigoni. Towards monocular vision based obstacle avoidance through deep reinforcement learning. *arXiv preprint arXiv:1706.09829*, 2017.

[45] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, L. Su, and T. Abdelzaher. Deep learning for the internet of things. *Computer*, 51(5):32–41, 2018.

[46] Y. Yun and C. Moon. Genetic algorithm approach for precedence-constrained sequencing problems. *Journal of Intelligent Manufacturing*, 22(3):379–388, 2011.