AutoCITE: Automated tuning of Concurrent Inferencing and Training on Edge

Anonymous Author(s)
Submission Id: 613

Abstract

The proliferation of GPU accelerated edge devices like Nvidia Jetsons and the rise in privacy concerns are placing emphasis on DNN training on edge devices. Often, the same edge device that performs inferencing also accumulates local data to run model training, and can train alternative models on this data. So workloads with concurrent DNN training and inferencing are becoming common. Despite their similarities, inference and training are computationally different, and their Quality of Service (QoS) goals also diverge: inference tends to be latency-sensitive while training is throughput-sensitive. In the absence of native GPU sharing mechanisms like MPS or MIG, these processes are time-shared on the edge GPU, which can violate QoS goals. Further, edge devices in energy constrained or field deployments have power budgets. These require configuring the CPU core counts and CPU, GPU and memory frequency dimensions on Jetsons to meet the power, latency and throughput goals for concurrent workloads. With over 18k possible combinations of these power modes in a Jetson AGX Orin, profiling all of these is challenging, especially with changing DNNs. In this paper, we formulate this as a schedule optimization problem and propose an efficient traversal technique, GMD, over these power modes. GMD couples a multi-dimensional space search with a gradient descent formulation that takes into account the impact of each resource dimension on the workload's power, latency and throughput, while reducing the number of power modes profiled. GMD outperforms both simpler and more complex baselines, and is able to find solutions that satisfy latency and power budget more than 95% of the time, and are less than 4% away from optimal. These results are based on evaluating across 5500+ configurations per DNN and 5 DNNs each for inference, training and concurrent.

1 Introduction

Edge devices are usually deployed close to the data source and hence are preferred for latency-sensitive Deep Neural Network (DNN) inference tasks [25]. Inference tasks are typically lightweight and suited to resource-constrained edge devices like Raspberry Pi, Google Coral and Intel Movidius. Recently, edge devices like NVIDIA Jetsons have more powerful on-board GPU accelerators while still retaining a low power envelope [29]. For instance, the latest AGX Orin developer kit is equipped with a 12-core ARM CPU, an Ampere GPU with 2048 CUDA cores, and 32 GB of RAM, which is comparable to a GPU RTX 3080 Ti workstation. As a result, these edges can perform training over the local data to reduce data movement overheads to a remote server, e.g., for *continuous learning* [1, 33], where training is done recurrently over evolving data to adapt to varying real-world data and avoid degradation caused by data drift, or to enhance privacy by avoiding data sharing externally through *federated learning* [24].

Challenges. Given the presence of accelerated compute and data being generated and accumulated on the edge, it is becoming common to have DNN workloads that run training or inferencing in isolation on the edge devices [4], or run them concurrently [1]. Inference workloads are lightweight and latency-sensitive, and potentially run continuously or interactively based on user or environmental stimuli. Training workloads are computationally heavy and throughput sensitive, and run periodically on a schedule, when a data drift is detected, or as part of investigating new model architectures. The actual inferencing and training models that run can also change over time, as they increase in sophistication or as part of exploratory analytics. Such heterogeneous and evolving inferencing and training workloads are emerging.

Contemporary edge accelerators like Nvidia Jetsons do not support GPU sharing mechanisms such as CUDA Multi Process Service (MPS) [27] or Multi-Instance GPU (MIG) [28], which are present in server-grade GPUs. As a result, training and/or inferencing models in a workload are time-shared on the GPU. This time-sharing is not configurable and offers no control to the user, and can lead to sub-optimal Quality of Service (QoS), e.g., high inferencing latency or low training throughput. Further, edge devices are deployed in diverse settings ranging across forests [40], drones [23] and smart cities [18], which can impose additional constraints on power or energy for the edge device. For example, an edge device packaged in an IP-67 enclosure deployed in a forest to detect forest fires or wildlife may have a power constraint imposed due to high ambient temperatures that may damage the accelerator or an edge device onboard a drone may have an energy constraint due to battery capacity. Therefore, it is crucial to identify an optimal configuration for DNN workloads to meet the user's latency, throughput and power goals.

Nvidia Jetsons expose 1000s of *power modes*, which control the active CPU cores, and the CPU, GPU and memory frequencies at fine granularity. These offer a meaningful control knob to help meet the QoS goals. However, profiling

each power mode is time-consuming, taking 10s of seconds, and this needs to be done for each workload as it changes. Automatically selecting the best power mode to satisfy the application's needs is an open problem. Similarly, batch size in inferencing can affect the inferencing latency and, as we show later, also impact the training throughput for concurrent workloads [1].

Gaps. There has been much work on optimizing the power and performance of training workloads on GPU servers [38, 39, 41, 42]. Some of these use GPU partitioning mechanisms such as MPS and MIG, and some use knobs such as the GPU power limit, which are not supported on edge. None of these consider CPU, GPU and memory frequencies together, all of which play an important role in power and performance on the edge [29]. Therefore, optimization studies on the server are not directly applicable. There has also been work on characterizing, predicting and optimizing inference workloads on edge devices [2, 11, 36]. However, training on edge devices has been limited to characterization studies [29]. No prior work has examined optimizing training workloads on edge, either in isolation or concurrently with inference.

Contributions. In this paper, we propose **AutoCITE**: **Auto**mated Tuning of **C**oncurrent **I**nferencing and **T**raining on **E**dge to address this gap by optimizing the performance of workloads with training and inference tasks, performed both independently and concurrently. We make the following specific contributions:

1. We formulate this as a scheduling problem to meet user-defined power and inference latency constraints, and maximize the training throughput and optionally inferencing latency (§ 3).
2. We develop a gradient-based multi-dimensional search strategy (referred to as GMD) that identifies the best power mode and inference batch size configuration to meet these goals while reducing the number of power modes that are experimentally profiled. The strategy is customized for independent training and inferencing workloads, as well as for concurrent ones (§ 4). We complement this with an interleaving execution approach to run both training and inference (§ 4).
3. We perform a rigorous evaluation of GMD for 5000+ workload configurations, spanning ≈40 different power budgets, ≈10 different latency budgets, ≈10 different inference arrival rates and ≈25 workloads on a Jetson Orin AGX (§ 6). We compare our results with the "optimal" configuration identified by running 441 power modes per DNN workload; a Neural Network (NN) baseline approach that is trained on 250 power profiles per model configuration to predict the training/inferencing time and power usage; and a baseline using static profiling (RND). Our technique finds solutions over 95% of the time, always meeting latency and power budgets, and the time is within 4% of optimal for all DNNs, including training, inference, and concurrent tasks.

[1] While training is also done in minibatches, this is a user-specified hyperparameter that affects the model training accuracy, that we do not change.

2 Motivation

2.1 Edge accelerators

Nvidia Jetsons are the leading accelerated edge devices that are powerful enough to train DNN models. Among these, we use the latest generation Jetson AGX Orin devkit, which also leads in the MLPerf benchmark among edge devices [31]. A unique feature of the AGX Orin is its fine-grained control over CPU core counts, CPU, GPU and memory frequencies, the combination of which is known as the *power mode*. There are 12 possible core count settings, 29 CPU frequencies, 13 GPU frequencies and 4 memory frequencies, with the minimum and maximum frequencies spanning over an order of magnitude, resulting in $\approx 18,000$ possible power modes. These can help tune the performance–power tradeoff for various deployment scenarios, and our experiments show, for instance, ResNet training on MAXN (highest) power mode takes 3.1mins per epoch and 51.1W of power while a low power mode with 2 cores, CPU freq. of 268MHz, GPU freq. of 115MHz and memory freq. of 204MHz increases the time to 112mins per epoch and decreases power to 11.8W. The impact on power and performance also varies across training and inference DNN architectures. So, choosing a power mode to satisfy certain power and/or performance constraints is non-trivial, and an erroneous choice can cause high penalties.

2.2 Scenarios and Solution Approaches

It is not tractable to benchmark a substantial subset of power modes for every new DNN workload as profiling each power mode takes 2.4 – 87.2s, depending on the DNN and power mode, and it takes over 16hrs to profile ≈ 4300 (around 25%) power modes for training ResNet. This will not scale to different workloads submitted over time. Static/Random sampling of a few power modes and using them can be sub-optimal, as we show later in our results. Similarly, the inference batch size also has a considerable impact on power and performance for inferencing, and also training when they are interleaved for a concurrent workload. The default power mode is MAXN, which sets all frequencies to the highest possible value, and results in a very high power consumption. This may violate most power budgets, especially those on the lower side. Another alternate is a prediction-based approach, such as NN, where we train a Neural Network model to predict minibatch time and power. This starts with high prediction errors due to lesser data and gets better as the amount of data increases. However, prediction errors may still exist. Therefore, it is necessary to have a principled low-overhead

Table 1. Practitioner's Matrix: Scenarios, solution approaches and role of GMD.

Workload Type	Usecase	Occurrence, Pattern	Runtime	Solution Approach	Time to Solution
Inference only	Continuous inferencing using single model, e.g., demand prediction from smart power meters	Frequent, predictable	Few hrs to days	GMD first, switch to NN once enough samples	1-1.5hrs
Inference only	Various inference DNNs run on-demand, e.g., tracking a stolen vehicle using traffic cameras	Frequent, unpredictable	10 mins–1 hr mins	GMD	<10 min
Inference only	Outlier inference tasks, e.g., voice assistant	Occasional, unpredictable	Less than a min	MAXN power mode	0
Train only	Personalization/fine-tuning	One time, predictable	Few hrs	GMD	<10min
Inference + Train	Continual learning	Frequent, predictable	<1 hr	GMD first, switch to NN once enough samples	1-1.5 hr
Inference + Train	Inference + Federated learning as a background task	Unknown, unpredictable	Few mins to 1 hr	GMD	<10 min

approach that identifies the optimal configuration (power mode and batch size) given a power/latency constraint and generalizes to various DNNs/datasets and training/inference.

Table 1 summarizes common application and deployment scenarios with a suitable approach for each, with time estimates in the last column based on our experimental results.

For instance, there may be a steady inference DNN that arrives and runs on an edge device for a few hours before the DNN changes or exits (e.g., vision models for day v/s night). Here, the best solution is to start with GMD, which can provide a good solution in a few minutes for a given power latency budget. If the power or latency budget for the model changes, GMD tries out different search paths through the power modes, accumulating more profiling samples. Once there are around 250 samples, the NN approach becomes tractable as it can predict the optimal configuration with $\approx$ 90% accuracy. In other scenarios, there may be a change in the inference DNN due to the environment suddenly changing, e.g., a drone moving from a cityscape to a forest area. This workload may only run for a few minutes before switching back to normal, leading to a tighter time limit to find a solution. In this case, GMD would be the best approach as it can find a solution within 10mins. There may also be short-running inference DNNs for a few minutes. Here, we recommend running with the default MAXN high power mode as none of the solution approaches can give a fast optimal solution; else, if the model does not change, then we can profile all power modes and select from it.

Similarly, for training, there may be a one-time personalization or fine-tuning run where a pre-trained DNN is re-trained on small amounts of user data on an edge device, which lasts for a few hours. In this case, we recommend using GMD since this is a one-off workload. Continual learning is another common usecase where training is run for a few epochs periodically to update the model. Here again, we recommend starting with GMD and switching to NN once there are sufficient profiling samples. This NN can also be reused for future occurrences of the task. Finally, federated learning is another workload that may run as a background task alongside inference. The runtime of this workload is uncertain as the device may not be picked in further rounds and federated learning may run anywhere from a few minutes to an hour. Here, we recommend using GMD which gives a quick solution, since NN's reuse may be limited.

Table 2. Notations Used

Symbol	Description	Comments
$\mathbb{PM}$	Set of device power modes	
P_{budget}	Power budget	
m_{tr}	Training model	
t_{tr}	Train minibatch time	$t_{tr} = \lambda(m_{tr}, pm)$
θ_{tr}	Train thru'put (minibatches/s)	
m_{in}	Inf model	
b_{in}	Inf minibatch size	
a_{in}	Inf arrival rate (image/s)	
qt_{in}	Peak queueing time (s)	$qt_{in} = b_{in}/a_{in}$
l_{in}	Peak infer. latency (s/image)	$l_{in} = qt_{in} + t_{in} = b_{in}/a_{in} + t_{in}$
t_{in}	Inf minibatch time (s)	$t_{in} = \omega(m_{in}, b_{in}, pm)$
l_{budget}	Inf latency budget (s/image)	

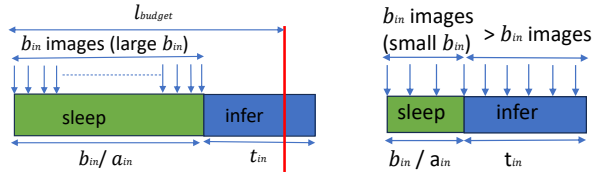
3 Problem Definition

We first formally define the scheduling problem we propose to solve. Based on the workload, there are 3 variants of the problem: *standalone training*, *standalone inference* and *concurrent training and inference*. We define the optimization goals and constraints for each variant below. The notation used is summarized in Table 2.

3.1 Standalone Training

Here, only a single training workload (m_{tr}) is running, and the user provides a power budget (P_{budget}). The goal is to find the power mode (pm) that maximizes the training throughput (θ_{tr}) while the observed power load during training (P_{tr}) stays within the given power budget.

$$\begin{aligned} &Given\ m_{tr}, \\ &\text{select } pm \in \mathbb{PM} \\ &\max\ \theta_{tr} \\ &\text{s.t.}\quad P_{tr} \leq P_{budget} \end{aligned}$$



(a) Large batch size - violates latency **(b)** Small batch size-unstable

Figure 1. Inference examples

3.2 Standalone Inference

In this case, inference is the only workload present, and the user specifies a power budget (P_{budget}) as well as a latency budget (l_{budget}). The input arrival rate (a_{in}) of data samples for inferencing is assumed to be fixed. The goal is to select a power mode (pm) and inference batch size (b_{in}) such that it minimizes the inference latency (l_{in}), stays within the latency budget and ensures that the observed power during inferencing (P_{in}) also remains within the power budget.

$$\begin{aligned} &Given\ a_{in}\ and\ m_{in}, \\ &\text{select}\ \ pm \in \mathbb{PM},\ b_{in} \\ &\min\ l_{in} \\ &\text{s.t.}\ l_{in} \leq l_{budget}\ \&\ P \leq P_{budget} \end{aligned}$$

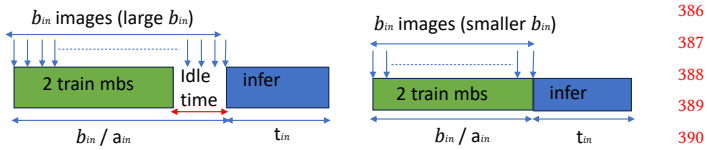
In Figure 1, we illustrate an inference workload and the conditions that need to be met. Initially, the inference process is idle (indicated by sleep), waiting for a batch size of b_{in} to arrive. The images arrive at a rate of a_{in}, leading to a queuing time of b_{in}/a_{in}. This minibatch takes time t_{in} to execute, during which images for the next minibatch continue to arrive. There are two counterpressures at play here. In Figure 1a, if the batch size is increased beyond a limit, the queuing and execution time increase, leading to a latency violation as shown. On the other hand, if the batch size is too small as in Figure 1b, the processing time can become too high, resulting in more images queuing up than those being processed. In other words, the inference rate will not be able to keep up with the arrival rate, leading to infinite queuing and instability. The goal is, therefore, to find the smallest stable inference batch size.

3.3 Concurrent Training and Inferencing

In this workload, both training and inference are provided. The user specifies the peak power budget (P_{budget}) and inferencing latency budget (l_{budget}). The goal is to select a power mode and inference batch size to maximize training throughput (θ_{tr}) while staying within the latency and power budgets. A secondary goal is to minimize the inference latency (l_{in}).

$$\begin{aligned} &Given\ m_{tr},\ m_{in} and\ a_{in}, \\ &\text{select}\ \ pm \in \mathbb{PM},\ b_{in} \\ &\max\ \ \theta_{tr} \\ &\text{s.t.}\ \ \ l_{in} \leq l_{budget}\ \&\ P \leq P_{budget} \\ &\min\ \ l_{in} \end{aligned}$$

Here, we propose to interleave the training and inference at the granularity of minibatches. As shown in Figure 2, we



(a) Largest batch size possible **(b)** Largest batch size that maximizes training minibatches

Figure 2. Interleaving examples

run x (returned by GMD) number of training minibatches after every inference minibatch. We choose this instead of running both simultaneously, which puts the execution under the control of the scheduler and does not guarantee QoS goals. Increasing batch size increases inference latency and allows for a larger number of training minibatches. However, this must be done such that inference conditions are not violated. Hence, we aim to pick the largest stable inference batch size that satisfies latency. Further, we aim to minimize inference latency as a secondary goal. As shown in Figure 2, if there are two inference batch sizes that can allow the same number of training minibatches, we pick the smaller batch size in order to minimize latency, which is a secondary goal.

3.4 Discussion

Non-goals: This work focuses on optimizing one training and one inference concurrently. We do not consider multiple training and multiple inferencing workloads concurrently. We also limit our study to a static inference rate, and do not consider dynamic inference rates. We do not tune any DNN training hyperparameters such as minibatch size, since these affect the time to accuracy as well.

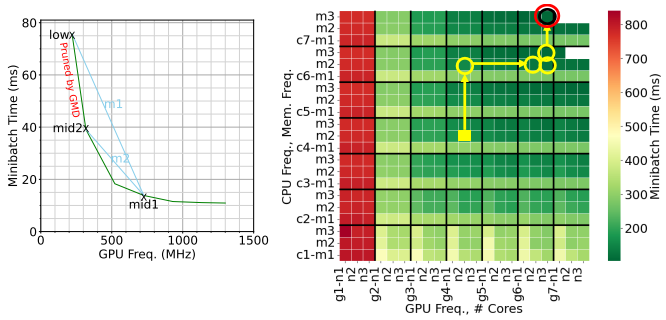
4 Proposed Approach

As shown in Figure 4, we propose 2 types of optimization techniques which we outline in detail here.

4.1 Domain-aware Search Strategies

Our broad solution approach is to start at some initial power mode for a given workload, perform profiling of the workload for that power mode, and use that knowledge to decide the next power mode to select and profile. We repeat this until we find a power mode that meets the problem constraints and maximizes/minimizes the optimization goal, with the intent of performing as few profile runs as possible to reduce the time to solution. That said, these profiling runs are themselves valid workload runs and their outputs can be used by the application, though the time and power performance may not meet the requirement. Also, once we run a benchmark for a power mode for a workload, it can be reused in the future. So, a key challenge here is where to start and which direction to search in the multi-dimensional space.

In Figure 5, we show the variation of training time per minibatch and power load during training as a function of GPU frequency for various CPU frequencies, keeping the



(a) GMD **(b)** Binary Search

Figure 3. Illustrations of Our Search Algorithms

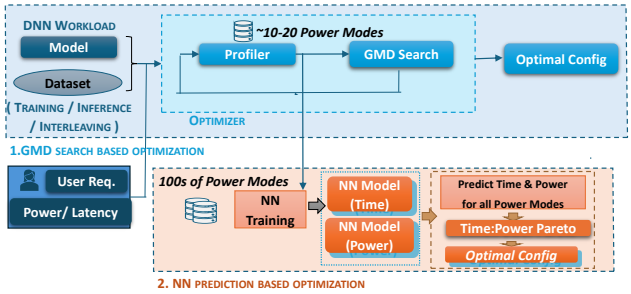


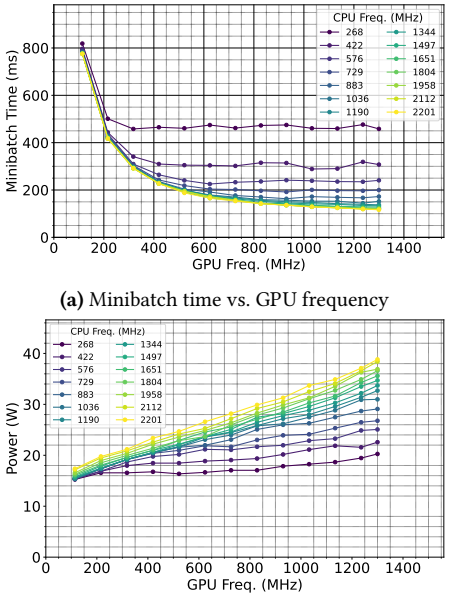
Figure 4. Summary of optimization techniques

number of cores and memory frequency constant. From Figure 5a, we see that the training time has a non-linear trend and initially drops sharply with the growth in GPU frequency and thereafter saturates. Power, on the other hand, steadily increases with GPU frequency (Figure 5b). We experimentally verify that this trend holds for diverse DNN model training workloads, and only vary in slopes. Based on this domain knowledge, we propose two power mode search strategies below. We also tried out alternatives such as Bayesian optimization, Reinforcement Learning, etc., but none of them could arrive at a solution with fewer than 50 profile runs, making them unsuitable due to their high overheads.

4.1.1 Multi-dimensional Binary Search (BS).

In this method, we extend the idea of traditional binary search to deal with the multiple dimensions of CPU, GPU, memory and cores that determine the power mode. This algorithm is described for training workloads in Algorithm 1, and the search path is pictorially depicted in Figure 3b.

We first initialize the power mode to one where all dimensions are set to the middle value of their range (Line 8). Then, we profile this power mode for DNN training and get the power and training time (Line 10). If the observed power is under or over the power budget, we decide which half of this dimension must be pruned out, similar to the traditional binary search (Lines 12-20). If the profiled power is under the budget, then this is a candidate solution and is compared with the current best solution and updated if better (Lines 15-19). Next, from the remaining search space, the middle



(a) Minibatch time vs. GPU frequency

(b) Power vs. GPU frequency

Figure 5. MobileNet training time and power CPU frequency (CPU Freq. MHz Cores: 12 and Mem Frequency:2133)

element of the next (remaining) dimension is chosen for profiling, with other dimension values retained from the earlier power mode (Lines 13 and 20) and this process is repeated. The dimension that is tested and pruned in each iteration changes in a round-robin fashion (Line 11). The specific order used is a hyperparameter. If a dimension is pruned small enough that there is only a single value left, that dimension is skipped from future changes, and the single value is retained.

During each iteration, the search space reduces by half, and this returns a solution in *(log N)* profiling tries, where N is $cores \times cpuf \times gpuf \times memf$. The search space is shown in Figure 3b as a heatmap across all 4 dimensions (X and Y axis have two nested dimensions) and coloured by minibatch time (darker is faster).The black circle indicates the optimal solution, the yellow lines and circles give the search trajectory, and the red box indicates the final solution found by binary search for this power budget.

Changes for Inference Workload We make one small modification to this algorithm for an inference workload. The inference batch size is considered as a fifth dimension, similar to cores, CPU, GPU and memory frequency (Line 2 in Algorithm 1) and is explored as part of the round-robin order (Line 11). As we show later, this strategy does well for training but not for inference, and therefore we do not explore it further for interleaved.

4.1.2 Gradient-based Multi-dimensional Search (GMD).

A shortcoming of the binary search approach is that it visits the dimensions in a fixed round-robin order, which may lead to candidate solutions being pruned out incorrectly. We address this limitation and refine it into another method we propose, Gradient-based Multi-dimensional Search (GMD).

Algorithm 1 Multi-dimensional Binary Search (BS) for Training Workload

```
1: procedure BinarySearch_TrainOnly(P_budget)
2:     dims = [CPU, GPU, Cores, MEM]
3:     dim_count ← 0
4:     best_t_tr ← ∞
5:     best_P ← 0
6:     best_pm ← NULL
7:     space = cores × cpuf × gpuf × memf
8:     cur_pm ← mid_pm                ▷ all dims set to mid
9:     while Size(space) > 1 do
10:        P_tr, t_tr ← profile_train(cur_pm)
11:        dim ← dims[dim_count%4]
12:        if P_tr > P_budget then
13:            space, cur_pm ← Prune(space, dim, high)
14:        else
15:            if (t_tr < best_t_tr) then          ▷ update best
16:                best_pm ← cur_pm
17:                best_t_tr ← t_tr
18:                best_P ← P_tr
19:            end if
20:            space, cur_pm ← Prune(space, dim, low)
21:        end if
22:        dim_count++
23:    end while
24:    return best_pm, best_t_tr, best_P
25: end procedure
```

Here, we develop a more intelligent logic to decide the next dimension to visit, rather than round-robin. This is described in Algorithm 2.

We again start by choosing the midpoint power mode along all dimensions to profile initially (Line 10). In addition, we sample 4 more power modes, one in each dimension, in a direction that is based on the power consumed by the initial power mode (under or over budget) (Lines 11-15). If the profiled power is over budget, the 4 power modes picked next are the lowest along each dimension; and the highest otherwise. Using these initial and four additional power modes, we calculate 4 sets of time and power slopes, with the initial power mode and each of the 4 additional power modes along the 4 dimensions forming a pair for the slope calculation. For each dimension, we get the ratios of the time slope to the power slope (Line 16).

We identify the dimension with the highest ratio to choose this dimension for the next search and prune step. This is based on the insight that all dimensions do not contribute equally to the performance and power of workloads, and this varies across workloads. The dimension with the highest time to power slope ratio gives the steepest decrease in time for the least increase in power. Therefore, exploring this dimension first can get us faster to the solution. We pick the middle value along this dimension to profile next, with other dimension values staying the same, and we prune the space along this dimension using the binary search process from before and the binary search repeats (Lines 17-32). In addition, we use a thresholding logic to detect cases where there is a negligible decrease in power that can artificially inflate the slope, and ignore such cases. Once all values in a dimension have been exhausted, we move to the dimension with the next highest slope and proceed, till we run out of the profiling budget, which is defaulted to 25 tries.

A pictorial representation of the algorithm is shown in Figure 3a. The actual time data is represented by the green line. We initially start by considering the points *low* and *mid*1 and calculating slope $m1$ for them. Based on profiling *mid*2 along this range, we eliminate the region *low* to *mid*2 (shown in red text) since the solution does not lie here. We update the slope to $m2$ which is between *mid*1 and *mid*2, which is a much closer approximation of the time data. This way, we keep getting better approximations of the slope in our area of interest.

Changes for Inference Workload. Unlike standard binary search, we do not consider batch size as another dimension for an inference-only workload. Instead, we perform the search with batch size set to 1 since we want to minimize inference latency. If no solution is found, then we perform *backtracking* as follows. Among the power modes visited in the first iteration, we identify those that satisfy the power budget, but violate the latency condition because the inference rate cannot keep up with the arrival rate. From our experiments, we notice that smaller batch sizes have a sub-linear increase in time, and increasing the batch size helps meet the arrival rate. Therefore, these power modes alone are our candidate set. We try them in order of increasing latency till we find a solution.

Changes for Concurrent Workload. The optimization goal for the concurrent workload can be achieved by maximizing the inference batch size that just satisfies the latency in order to increase the training throughput. So, we initialize the batch size to the largest candidate, 64.

Branch and Bound: Before we start to search, we first check if the fastest power mode, MAXN, can keep up with the arrival rate and meet the inference latency budget for this batch size. If not, we rule out this batch size, since every other slower power mode will only have a higher latency or not meet the arrival rate. We move to the next lower batch size and try this until we reach a batch size that satisfies the constraint. We limit our multi-dimensional search to this batch size.

The only difference during the search here is that we consider the slope ratios of the *dominant* workload. Based on profiling data that we get for a given power mode, we look at whether inference or training has a higher power, and identify this as the dominant workload at that step. We use the slope ratios of the dominant workload to decide which dimension to visit and prune. At every step, we re-evaluate the dominant workload.

Backtracking: If no solution is found in the first non-eliminated batch size, then we again try to identify candidate solutions that can work with smaller batch size. Contrary to inference,

Algorithm 2 Gradient-based Multi Dimensional Search (GMD) for Training Workload

procedure GradientBinarySearch_Train(P_{budget})
  $best\_t_{tr} \leftarrow \infty$
  $best\_P \leftarrow 0$
  $best\_pm \leftarrow NULL$
  $tries \leftarrow 0$
  $max\_tries \leftarrow 25$
  $dims = [CPU, GPU, Cores, MEM]$
  $space = cores \times cpuf \times gpuf \times memf$
  $pm1 \leftarrow mid\_pm$
  $P_{tr}, t_{tr} \leftarrow$ PROFILE_TRAIN(cur_pm)
  if $P_{tr} > P_{budget}$ **then**
    $pm[dim] \leftarrow$ Profile 1 low pm along each dim
  else
    $pm[dim] \leftarrow$ Profile 1 high pm along each dim
  end if
  $ratios[dim] \leftarrow$ Pairwise slope ratio along each dim
  while $tries < max\_tries$ **do**
    $dim \leftarrow$ highest slope ratio dim
    $pm1 \leftarrow$ mid along dim, rest as is
    $P_{tr}, t_{tr} \leftarrow$ PROFILE_TRAIN($pm1$)
    if $P_{tr} > P_{budget}$ **then**
      $space, pm1, pm2 \leftarrow$ Prune($space, dim, H$)
    else
      if $(t_{tr} < best\_t_{tr})$ **then** ▷ update best
        $best\_pm \leftarrow pm1$
        $best\_t_{tr} \leftarrow t_{tr}$
        $best\_P \leftarrow P_{tr}$
      end if
      $space, pm1, pm2 \leftarrow$ Prune($space, dim, L$)
    end if
    $ratios[dim] \leftarrow$ UPDATE_RATIO($pm1, pm2$)
    $tries++$
  end while
  return $best\_pm, best\_t_{tr}, best\_P$
end procedure

here, we eliminate those that cannot keep up with the rate as moving to a smaller batch size will not find a solution. Among those left, we move to the next lower batch size and try candidates in the order of increasing latency till a solution is found.

4.2 Neural Network Model-driven Baseline (NN)

Neural Networks [22] can model complex non-linear relationships in data. We propose a data-driven ML approach based on Neural Networks (NN), which uses prior profiling information from various power models to predict the training time and the power for a given configuration of DNN, power mode and inference batch size. Our NN consists of 4 dense layers with 256, 128, 64, *and*1 neurons, respectively. We use the ReLu as the activation function for the first 3 layers and linear for the last. The optimizer used is Adam, with a learning rate of 0.001 and the loss function as Mean Squared Error (MSE). We also add two dropout layers in between the dense layers to avoid overfitting. We arrived at this architecture after doing a hyperparameter search using the *RandomSearchCV* library. The input feature vector for training consists of the power mode configuration: CPU cores, CPU frequency, GPU frequency and Memory frequency. The input feature for inference consists of one additional parameter, the inference batch size. The output layer in both cases returns the predicted minibatch time or power. All input features are normalized using the *StandardScaler* library to ensure consistent scales across all inputs. We train the NN for 200 epochs in case of training and 1000 epochs in case of inference. We use an 80:20 train test split and model checkpointing so that the best weights from training are saved. The NN model is trained on different samples of power mode profiling conducted *a priori*, as configured later.

Table 3. Specifications of NVIDIA Jetson Orin AGX

Feature	Orin AGX
CPU Architecture	ARM A78AE
# CPU Cores	12
GPU Architecture	Ampere
# CUDA, Tensor Cores	2048,64
RAM (GB), Type	32, LPDDR5
Max Power (W)	60
Form factor (mm)	110 × 110 × 72
Price (USD)	$1999
Accelerators	DLA, PVA

5 Experimental Setup

5.1 Hardware and Configuration

We use the NVIDIA Jetson Orin AGX developer kit [26]. The specifications of the AGX Orin are shown in Table 3. As mentioned previously and described in Table 4, it supports over $\approx 18,000$ power modes that the user can choose from. We set the fan speed to max to avoid any thermal throttling effects. We also disable Dynamic Voltage and Frequency Scaling (DVFS) so that the frequencies configured remain static during our experiments. The Orin has frequency settings for the custom accelerators PVA and DLA, which we do not alter. We use the onboard power sensors INA3221 to get power readings for our experiments.

5.2 Training and inference workloads

We choose five popular DNN workloads each for DNN training and inference as shown in Table 5. Out of these five DNNS, two are used for image classification tasks, one for object detection, one for question answering and another for next word prediction. These are representative of typical usecases and workloads on edge device. [3, 16, 35]. We pick a variety of model architectures (CNN, RNN and transformer architectures) and tasks with sufficient diversity in dataset sizes (17.8MB–6.7GB), model sizes (3.2M–110M parameters) and computational requirements (225M–11.5T FLOPS). The specifications of the models and datasets are reported in detail in Table 5.

5.3 Software and Libraries

The Orin AGX runs Ubuntu 20.04 LTS with kernel version 5.10.65. We use NVIDIA JetPack version 5.0.1 which comes

with CUDA 11.4. We configure this with PyTorch v1.12 and torchvision v0.13. We configure PyTorch's Dataloader to run with 4 worker processes wherever possible [2]. We use a batch size of 16 for all training workloads, and run 5 batch sizes (1,4,16,32,64) for inference workloads [3].

5.4 Profiling Setup and Metrics

We sample the current power every 1s using *jtop* which is a wrapper around NVIDIA's *tegrastats* library. This gives just the Jetson module's power and eliminates peripheral (USB ports etc) power consumption, which has been found to be negligible [29]. We instrument the PyTorch code to measure the minibatch execution time using `torch.cuda.event` with the `synchronize` flag to accurately measure GPU time. Our profiling overhead has been found to be minimal and does not affect workload performance. When we profile a power mode, we run the DNN (training or inference) for $\approx$ 40 minibatches and record the minibatch time and sampled power. The very first minibatch takes a long time to execute, possibly owing to PyTorch's profiling to pick the best implementation, and hence discard this for our calculations. We notice that power initially fluctuates and takes $2-3$s to stabilize, so we use a sliding window detector to detect the stabilization point and use profiling data after this to ensure clean data for all power modes.

5.5 Data collection for ground-truth

We choose 441 power modes that are uniformly distributed through the $\approx 18,000$ power modes. This consists of 7 CPU frequencies, 7 GPU frequencies, 3 core counts and 3 memory frequencies. The lowest CPU and memory frequencies are excluded. We run workloads for all 441 power modes, and each is run for $\approx$ 40 minibatches as described earlier. During experiments, we found that the Jetson does not support changing from lower to higher frequencies for CPU and GPU without reboots. We overcame this limitation by finding an ordering of power modes that satisfies this requirement. We use this data to validate our results and understand how far away from the optimal results are. We also do a smaller set of runs for interleaving and verify that interleaved minibatch times match the sum of the minibatch times for training and inference and that interleaved power is equal to the maximum of the training and inference power.

6 Results

In this section, we present a detailed evaluation of our proposed methods BS and GMD across 5 standalone training and inference DNNs and 5 concurrent training and inference DNN pairs. We evaluate across a range of configurations such as power budgets, latency budgets and arrival rates.

[2]The model YOLO V8 has a bug that does not allow 4 workers. This is fixed in a later version of PyTorch, but not for this JetPack version (https://github.com/pytorch/pytorch/issues/48709)

[3]We did not run BERT inference for batch size 64 as it took more than 20s per minibatch for slower power modes

Table 4. Power Mode features of NVIDIA Jetson Orin AGX

Power Mode features	Orin AGX
CPU cores	1 to 12
# CPU freqs.	29
Max CPU Freq. (MHz)	2200
# GPU freqs.	13
Max GPU Freq. (MHz)	1300
# Mem freqs	4
Max Mem. Freq. (MHz)	3200
# Power modes	18,096

In each section, we present results across all configurations and also zoom into a small set of configurations to clearly highlight the trends.

6.1 Baselines

Random We use a static/random sampling (RND) baseline to pick 10 power modes out of the 441 for training. For inference, we choose 50 i.e. 10 power modes from each batch size. We use profiling data from these to construct a partial Pareto front of time and power. We then use a lookup to find the optimal configuration for a specific user requirement.
Optimal We also contrast our methods against the ground-truth optimal solution. This is found by constructing a Pareto front using profiling information from all power modes and batch sizes, and then performing a lookup based on the specified user requirements.
We also compare against the NN baseline described earlier.

6.2 Standalone training

We vary the power budget from $10-50$W ($10-60$W in case of BERT since power goes higher) in increments of 1W for all the training DNNs and compare the performance of BS, GMD and NN with RND. In Figure 6, we report the distance from optimal in the form of time and power violins. The time violin represents the percentage difference in time between the method and the optimal solution, and the power violin represents the absolute difference in power between the strategy and optimal. A positive value indicates that the time chosen by the method is higher than optimal, while a negative value indicates that the strategy picked a power mode that is faster, which leads to a power violation. An important distinction here is that since RND and GMD are observation-based methods, their final solution is always under the power budget. NN, on the other hand, is a prediction-based method and can wrongly estimate time or power (seen as positive values in the power violins).

GMD outperforms the RND baseline across all models. From Figure 6, we see that RND has very high excess % times of 42.76, 54.58, 95.80, 14.41 and 52.55 for ResNet, MobileNet, YOLO, BERT and LSTM respectively. In comparison, GMD's % excess times are below 5% in 4 out of 5 cases and below 10% in all cases. This is because RND has 10 uniformly distributed points to build the Pareto and pick the solution, while GMD can pick points along the dimension of highest impact. It is

Table 5. DNN workloads and Datasets used in Experiments. All models are trained with batch size of 16.

Type	Task	Model	# Layers	# Params	FLOPs†	Dataset	# Samples	Size
Training ,Inference	Image classification	**MobileNetv3 [14]**	20	5.5*M*	225.4*M*	**GLD23k [34]**	23*k*	2.8*GB*
Training	Image classification	**ResNet-18 [12]**	18	11.7*M*	1.8*G*	**ImageNet Val. [6]**	50*k*	6.7*GB*
Training, Inference	Object detection	**YOLO-v8n [17]**	53	3.2*M*	8.7*G*	**COCO minitrain [32]**	25*k*	3.9*GB*
Training	Question answering	**BERT base [8]**	12	110*M*	11.5*T*	**SQuAD V2.0 Train [30]**	130*k*	40.2*MB*
Training,Inference	Next word prediction	**LSTM [37]**	2	8.6*M*	3.9*G*	**Wikitext [15]**	36*k*	17.8*MB*
Inference	Image classification	**ResNet-50 [12]**	50	25.6*M*	3.8*G*	**ImageNet Val. [6]**	50*k*	6.7*GB*
Inference	Question answering	**BERT Large [7]**	24	340*M*	43.7*T*	**SQuAD V2.0 Dev [30]**	12*k*	4.2*MB*

† As per the typical practice, FLOPs reported correspond to a forward pass with minibatch size 1.

worth noting that for the same number of 10 power modes that are profiled, GMD is able to reach a much better solution.

BS performs better than RND, but GMD outperforms it in 3 out of 5 cases. As seen in Figure 6, for ResNet, BS has a median excess of 10.6% over optimal, while GMD does slightly better at 8.8%. Similarly, for BERT, BS has a median excess of 2.8% while GMD has an excess of just 0.5%. For LSTM, BS has an excess of 4.1% as compared to GMD's 2.6%. On average, GMD is able to outperform BS because it identifies a dominant dimension to prune, and this helps it find better solutions. In the case of BS, we notice that sometimes it prunes the wrong dimension due to the round-robin order, which cannot be corrected later, and it remains stuck in a sub-optimal solution.

NN250 performs comparably or slightly better than GMD in terms of time. However, NN250 may violate the power budget. The excess over optimal medians for NN250 are −15.8%, −11.3%, 3.4%, −25.2% and −18%. As the negative values indicate, this means that NN picks a power mode that is faster than the optimal, resulting in a power excess (seen in the positive medians of the power violins). In contrast, GMD never exceeds the power budget. As explained previously, prediction-based methods can exceed the power budget. From experiments, our NN time and power prediction models have 5 − 10% errors on average, which leads to these power violations. Interestingly, NN uses 250 tries to get to this accuracy, while GMD takes 9 tries on average and 10 at most.

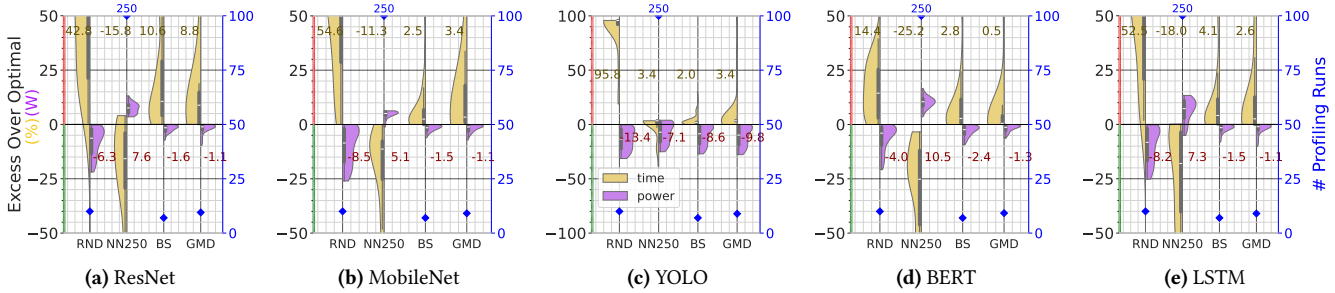
GMD performs close to optimal for 4 out of 5 DNNs. As seen from Figure 6, the time excess is very low for GMD (below 5% in 4 out of 5 cases and below 10% in all cases). It also never violates the power budget. This makes it quite close to the optimal configuration in terms of time and power.

6.2.1 Selected regions. Here in Figure 10a, we zoom in on one region for ResNet where the power budget is varied from 28−42W. We show the time taken by the 3 methods, the baseline and optimal as consecutive bars. On the right y-axis, we represent the power budget as a black marker, and NN250 power as a blue marker. We don't represent power for any other methods since they are always under the budget. Even though the power varies by 14*W* in this range, RND changes power modes much more coarsely. For instance, it picks the same power mode through 30 − 42W and even beyond till 50*W* (not shown in Figure). This is because it has only 10 power modes to choose from, and there was no power mode that went beyond 30W in this set. NN switches power modes frequently, but as mentioned previously, prediction errors lead to violation of power budgets in 5 cases out of 8. Out of these 5 cases, 2 are marginal violations that fall within a 1*W* threshold. BS does comparably to GMD in 4 cases out of 8, however, it does poorly in the other 4 cases. For instance, at 30W, it tries the mid CPU frequency which is under the power budget, prunes the lower half, and picks a high CPU frequency. After this, it tries to increase the GPU frequency, which overshoots the power budget by a lot, which it tries to compensate by lowering everything else, but still misses the optimal by a lot. This shows the significance of the order of visiting dimensions, which is non-trivial. GMD, on the other hand, closely follows optimal in all the cases.

6.3 Standalone inference

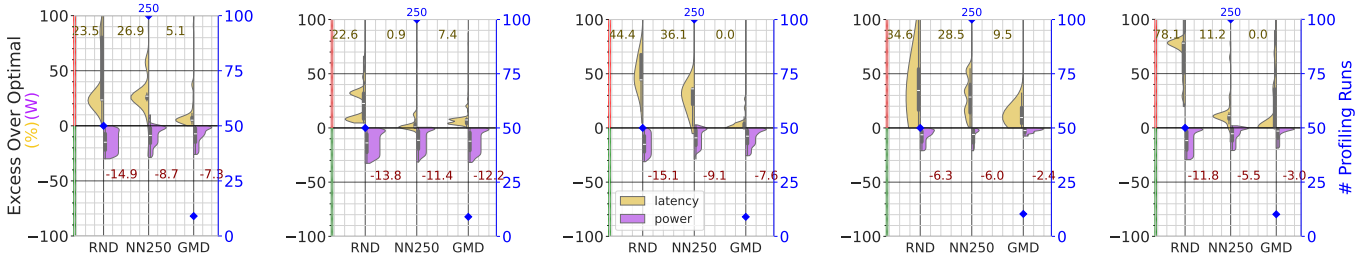
Here, we evaluate over a wider set of configurations. For ResNet, MobileNet, YOLO and LSTM, we vary the power budget from 10 − 50W in intervals of 1W, latency budget from 50 − 150ms in intervals of 10ms and arrival rate from 30 − 90 inference requests per second in intervals of 5. For BERT, which has a significantly higher execution time, we use a latency budget of 1 − 2.4s and an arrival rate of 1 − 5 requests per second in intervals of 1. This constitutes ≈ 5800 configurations per DNN for the other 4, and ≈ 3600 for BERT. We report summary results over all of these using the same time and power dual violins. The NN reported here is trained over 50 power modes on each of the 5 available batch sizes, a total of 250 configurations. Since inference has both latency and power budgets, in some cases, one or more of the methods do not find solutions even though optimal has a solution. We report these cases separately in Figure 9a. Also, we omit BS from the inference results as it performs quite poorly compared to GMD. From our analysis, we observed that batch size gets picked as the dominant dimension early on. This results in smaller batch sizes getting pruned early on and wrongly, resulting in a suboptimal solution.

GMD performs much better than RND and comparably or better than NN in 4 out of 5 cases. Looking at Figure 7, we see that ResNet GMD has a time excess of 5.1% which is much better than NN at 26.9% and RND at 23.5%. Similarly, for YOLO, GMD is at 0% while RND is at 44.38% and NN is at



(a) ResNet **(b)** MobileNet **(c)** YOLO **(d)** BERT **(e)** LSTM

Figure 6. Distance from Optimal % latency and absolute power (W) for various training DNN models.



(a) ResNet **(b)** MobileNet **(c)** YOLO **(d)** BERT **(e)** LSTM

Figure 7. Excess over Optimal % latency (ms) and absolute power (W) for GMD and baselines for various inference models.

36.06%. In case of MobileNet, NN does exceptionally well with an excess of just 0.88% while GMD has a modest excess of 7.37%.

GMD finds a solution in 10 tries on average and performs better or comparably to NN which takes 250 samples. Similar to training, GMD finds a solution in 10 tries on average for all DNNs. The maximum number of tries for GMD is limited to 11 across all DNNs. This is despite adding batch size as another dimension, and performing backtracking when we don't find solutions. In contrast, NN250 uses 250 tries, but still performs comparably.

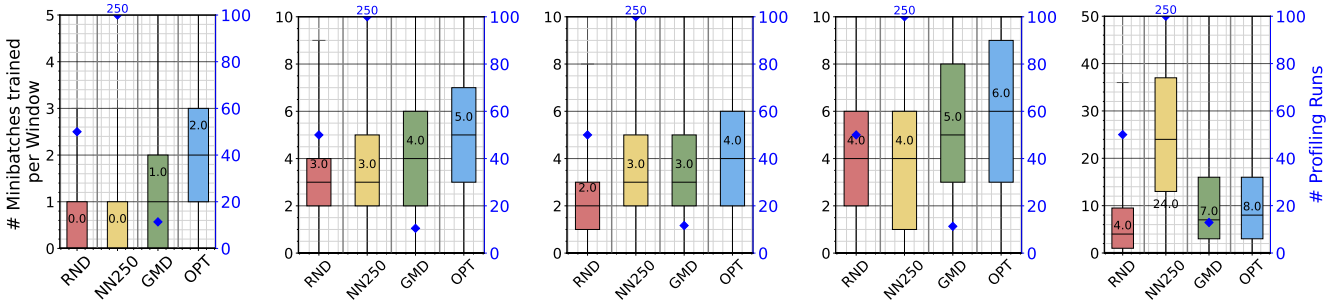
GMD finds the most number of solutions for all DNNs and is close to optimal. As seen in Figure 9a, out of the configurations we specify, even optimal is unable to find solutions for some cases, as seen by the different blue lines for each DNN. Out of the other methods, GMD is able to obtain the maximum number of solutions and is the closest to optimal. For instance, for ResNet, GMD finds solutions in 3570 out of 4041 cases (88.34%), while NN250 finds 3346 (82.80%) and RND 3091 (76.49%). Similarly, for YOLO, GMD finds solutions in 3277 out of 3427 cases (95.62%), while NN250 finds 1949 (56.87%) and RND 1365 (39.83%).

6.3.1 Selected regions. Here, we zoom in on one region for ResNet where the arrival rate is varied from 30 – 90 in increments of 10. We show the time taken by the 3 methods, the baseline RND and optimal as consecutive bars. We don't represent power/latency for any methods since all methods are under the budgets. As seen from Figure 10b, RND performs worse than optimal in many cases, again because it has only 50 power modes across 5 batch sizes to choose from. NN250 in this region does not have any power penalties, but it is far from optimal in 3 out of 7 cases. At an arrival rate of 55, NN250 switches to batch size 4 in order to meet the higher arrival rate. The optimal has a solution with batch size 1 at an arrival rate of 60, which NN250 misses because of its prediction errors, leading to an excess over optimal. GMD is able to find the batch size 1 solution at 60 and switches to batch size 4 only at 70 which is closer to the optimal.

6.4 Interleaving

We identify 5 pairs of DNN training and inference workloads to evaluate. These are picked such that all 5 DNN types we have are used either in training or in inference. Out of these, we have one combination where the train and inference workload is the same (MobileNet). We use the same configuration for power budget, but with a larger arrival rate range of $30 - 120$ requests per second and a larger latency budget of $500ms - 2s$ in steps of 100ms. Whenever BERT is used for inference, the latency is changed to $2 - 6$s, the power budget to $10 - 60$W and the arrival rate to $1 - 15$ requests per second. This results in $\approx$ 5500 configurations per workload pair and $\approx$ 6900 configurations where BERT is used. We show results over all these configurations as box plots of number of training minibatches (primary optimization goal).

GMD accommodates the maximum number of training minibatches within a given deadline and is very close to optimal. As seen from Figure 8, GMD is able to squeeze the most number of training minibatches per window without violating latency and power constraints. For YOLO training with ResNet inference, optimal is able to squeeze in a median of 2 minibatches per window, while GMD manages 1. In contrast, other strategies have a median of 0. Similarly, for MobileNet training with LSTM inference, optimal is able



(a) YOLO Train+ResNet Infer. **(b)** MNet Train+LSTM Infer. **(c)** MNet Train+MNet Infer. **(d)** ResNet Train+MNet Infer. **(e)** ResNet Train+BERT Infer.

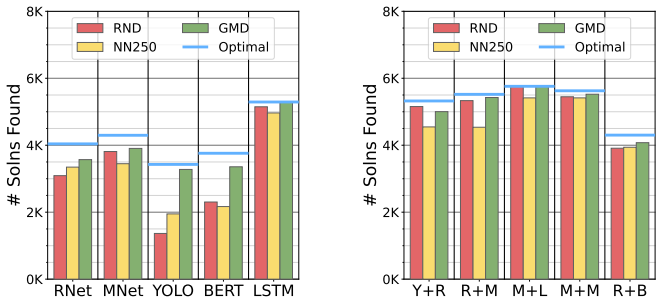
Figure 8. Throughput of training minibatches per window using GMD and baselines in interleaved DNN workloads.

to achieve 5 training minibatches per window (LSTM is a very small model, leaving lots of room for training) with GMD following closely at 4. For the other three workloads, see Figure 9b, GMD performs better and is 1 less than the optimal minibatches trained per window.

GMD finds the most number of solutions among all strategies. Similar to inference, even for interleaving, GMD finds the most number of solutions. As seen from Figure 9b, for YOLO training with ResNet inference, optimal finds a maximum of 5322 solutions out of which GMD finds 5003 (94%), NN250 finds 4547 (85.43%) and RND finds 5155 (96.86%). Similarly, for ResNet train with MobileNet inference, optimal finds a maximum of 5520 solutions out of which GMD finds 5425 (98.27%), NN250 finds 4537 (82.19%) and RND finds 5332 (96.59%). For MobileNet train with LSTM infer, MobileNet train with MobileNet infer and ResNet train with BERT infer GMD finds 100%, 98.16% and 94.74% of the solutions of the optimal, respectively.

NN performance is mixed, and it performs worse than GMD in 3 out 5 cases. From Figure 8, we see that NN achieves fewer number of minibatches as compared to GMD for 3 cases - YOLO training with ResNet infer, MobileNet training with LSTM inference and ResNet training with MobileNet inference. Out of the other 2, for ResNet training with BERT infer, it achieves a very high number of train minibatches, but violates power. For MobileNet train and MobileNet infer, it achieves the same minibatches as GMD, with a low excess latency over optimal. We notice that for interleaving, NN250 performance is unpredictable and varies a lot.

GMD prioritizes the primary goal of training minibatches over the secondary goal of inference latency minimization. We notice that for YOLO training and ResNet inference, GMD has a higher excess over optimal of 61.1% (Not shown in Figure) as compared to NN250 and RND, which have 25.2% and 9.9% respectively. However, as seen from Figure 8, GMD achieves a higher number of train minibatches compared to the others by choosing a larger batch size, without violating latency constraints. Only the minimization of latency, which is a secondary goal, is compensated in some cases.

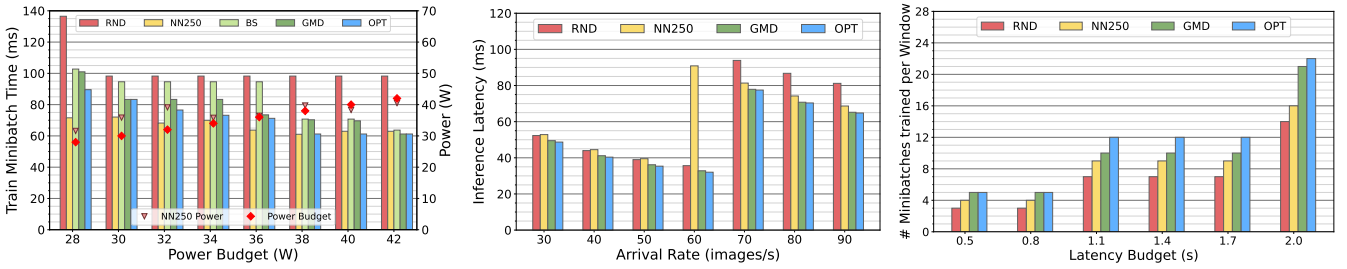


(a) Inference **(b)** Interleaving

Figure 9. Number of solutions found by each method

GMD finds a solution in under 12 tries in most cases and under 25 in all cases. From figure 8, As compared to training and inference, the average number of power modes is slightly higher at 12 for interleaved because of the branch and bound check at the start. Due to backtracking, the number of tries goes to 25 in a few cases. We profile both training and inference for each power mode identified.

6.4.1 Selected regions. We pick ResNet training and MobileNet inference as the workload pair with power budget set to 40W, arrival rate of 40 requests per second, and vary the latency budget from 0.5 to 2s. We plot the number of training minibatches achieved by all the methods in Figure 10c. Here, the limiting condition that causes a configuration change is the latency budget. As the latency budget increases, in general, there is more slack for training and the number of training minibatches per window increases across all strategies. As seen, GMD achieves the same number of training minibatches as optimal in 2 out of 6 cases, while being the closest to optimal, 1 lesser minibatch at a latency budget of 2s and 2 lesser minibatches in the other 4 cases. NN250 does better than RND in all cases, but is slightly lower than GMD. At a latency budget of 1.1s, NN250's predictions result in it picking a solution in the right batch size but having a higher execution time and therefore latency, causing the number of training minibatches to lower to 9 as compared to optimal's 12. Similarly, even for a latency budget of 2s, NN250 finds a solution with the same batch size as optimal but is



(a) RNet Training. **(b)** RN Inf: 40W, 100ms, varying arrival. **(c)** RN Train + MN Inf: 40im/s, 40W, varying latency.

Figure 10. Selected configurations for training, inference and interleaving

much slower, again leading to a lower number of training minibatches of 14 as compared to 22 of optimal.

7 Related Work

7.1 Multi inference on server and edge devices

Energy Time Fairness [21] designs a configurable scheduler based on energy-time tradeoffs for multiple inference workloads on edge devices. GSLICE [9] develops a dynamic GPU partitioning scheme for multiple inference workloads on GPU servers. Others [20] design analytical models to predict the performance of multiple inference workloads on Jetson Nanos and TPUs. MASA [5] ensures average response time for executing multiple DNN inference workloads on Raspberry Pis. However, none of these consider training and inference running on a single edge device, which is the primary focus of this work.

7.2 Training and inference on servers

Lyra [19] proposes a scheduling mechanism to loan idle inference servers to training jobs. Ekya [1] proposes a scheduler for continual learning (joint inference and retraining) on edge servers. These works are on GPU servers, which support GPU sharing mechanisms such as MPS and/or MIG. Jetson edge devices do not support these, and workloads are time-shared on the GPU by default.

7.3 Training only optimizations on server and edge

Gandiva [38] uses intra-job predictability to time slice GPUs efficiently across multiple DNN training jobs and migrates jobs dynamically to better fit GPUs and improve cluster efficiency. Antman [39] proposes a scheduling technique that uses spare resources to execute multiple jobs on a shared GPU while minimizing interference between jobs. MURI [42] packs Deep Learning training jobs along multiple resource types to achieve high utilization and reduce job completion time. Zeus [41] is an optimization framework that uses an exploration-exploitation based approach to find the optimal job and GPU level configurations to navigate the energy performance tradeoff of DNN training jobs. These works look at server GPUs, where power is usually not a limitation, unlike in edge devices. Additionally, none of them consider CPU, GPU and memory frequencies for optimization. Another recent work [29] characterized training of DNNs on Jetson edge devices and proposed a linear-regression based prediction model to predict training time and energy. This method is only evaluated on 10 power modes, and its errors are much worse than what we observe in this work.

7.4 Inference only optimizations on server and edge

Previous work [11] has looked at using roofline modeling on an older generation Jetson TK1 and TX1 to characterize the performance of matrix multiplication micro-benchmarks. MIRAGE [2] predicts runtime and power consumption of DNN inference workloads using gradient tree boosting. Other works [10, 13] have looked at profiling and characterizing the impact of frequencies on inference latency, power and energy. ALERT [36] selects a DNN and system resource configuration to meet latency and accuracy constraints of inference while minimizing energy on CPUs and desktop GPUs. None of these have looked at batch size as an additional parameter, considered CPU, GPU and memory frequencies for optimization or considered inference along with training.

8 Discussions and Conclusion

In this work, we propose an efficient traversal technique to optimize the performance of inference, training and concurrent workloads. GMD couples a multi-dimensional search with a gradient descent formulation that takes into account the impact of each resource dimension on the workload's power, latency and throughput, while reducing the number of power modes profiled. GMD outperforms both simpler and more complex baselines, and is able to find solutions that satisfy latency and power budget more than 95% of the time, and are less than 4% away from optimal. As part of future work, we plan to look at dynamic arrival rate for inference workloads and come up with methods that take into account the variability of inference workloads. We also plan to look into choosing separate power modes for inference and training, and dynamically switching between the power modes during interleaving. Training batch size is another variable we plan to include in our study.

References

[1] Romil Bhardwaj , Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Victor Bahl, and Ion Stoica. 2022. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *USENIX NSDI*.

[2] Hazem A. Abdelhafez, Hassan Halawa, Mohamed Osama Ahmed, Karthik Pattabiraman, and Matei Ripeanu. 2021. MIRAGE: Machine Learning-based Modeling of Identical Replicas of the Jetson AGX Embedded Platform. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*.

[3] Ahmed M. Abdelmoniem, Atal Narayan Sahu, Marco Canini, and Suhaib A. Fahmy. 2023. REFL: Resource-Efficient Federated Learning. In *Proceedings of the Eighteenth European Conference on Computer Systems*.

[4] Jiasi Chen and Xukan Ran. 2019. Deep Learning With Edge Computing: A Review. *Proc. IEEE* 107, 8 (2019).

[5] Bart Cox, Jeroen Galjaard, Amirmasoud Ghiassi, Robert Birke, and Lydia Y. Chen. 2021. Masa: Responsive Multi-DNN Inference on the Edge. In *2021 IEEE International Conference on Pervasive Computing and Communications (PerCom)*.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805, 1 (2018).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*.

[9] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*.

[10] Anurag Dutt, Sri Pramodh Rachuri, Ashley Lobo, Nazeer Shaik, Anshul Gandhi, and Zhenhua Liu. 2023. Evaluating the energy impact of device parameters for DNN inference on edge. In *International Green and Sustainable Computing*.

[11] Hassan Halawa, Hazem A. Abdelhafez, Andrew Boktor, and Matei Ripeanu. 2017. NVIDIA Jetson Platform Characterization. In *Euro-Par 2017: Parallel Processing*.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.

[13] Stephan Holly, Alexander Wendt, and Martin Lechner. 2020. Profiling Energy Consumption of Deep Neural Networks on NVIDIA Jetson Nano. In *2020 11th International Green and Sustainable Computing Workshops (IGSC)*.

[14] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

[15] Hugging Face. 2021. Datasets : wikitext. https://huggingface.co/datasets/wikitext.

[16] Deepthi Jallepalli, Navya Chennagiri Ravikumar, Poojitha Vurtur Badarinath, Shravya Uchil, and Mahima Agumbe Suresh. 2021. Federated Learning for Object Detection in Autonomous Vehicles. In *2021 IEEE Seventh International Conference on Big Data Computing Service and Applications (BigDataService)*.

[17] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. 2023. Ultralytics YOLOv8. https://github.com/ultralytics/ultralytics

[18] Latif U. Khan, Ibrar Yaqoob, Nguyen Tran, S.M. Kazmi, Tri Nguyen Dang, and Choong Seon Hong. 2020. Edge-Computing-Enabled Smart Cities: A Comprehensive Survey. *IEEE Internet of Things Journal* PP (2020).

[19] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*.

[20] Qianlin Liang, Walid A. Hanafy, Ahmed Ali-Eldin, and Prashant Shenoy. 2023. Model-driven Cluster Resource Management for AI Workloads in Edge Clouds. *ACM Trans. Auton. Adapt. Syst.* 18, 1 (2023).

[21] Q. Liang, W. A. Hanafy, N. Bashir, D. Irwin, and P. Shenoy. 2023. Energy Time Fairness: Balancing Fair Allocation of Energy and Time for GPU Workloads. In *2023 IEEE/ACM Symposium on Edge Computing (SEC)*.

[22] Warren S McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5 (1943), 115–133.

[23] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. 2022. A Survey on the Convergence of Edge Computing and AI for UAVs: Opportunities and Challenges. *IEEE Internet of Things Journal* 9, 17 (2022).

[24] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*.

[25] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. 2021. Machine Learning at the Network Edge: A Survey. *ACM Comput. Surv.* 54, 8 (2021).

[26] NVIDIA. 2022. Jetson AGX Orin Developer Kit. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[27] NVIDIA. 2022. Multi Process Service. https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[28] NVIDIA. 2024. NVIDIA Multi-Instance GPU (MIG). https://www.nvidia.com/en-in/technologies/multi-instance-gpu/.

[29] Prashanthi S.K, Sai Anuroop Kesanapalli, and Yogesh Simmhan. December 2022. Characterizing the Performance of Accelerated Jetson Edge Devices for Training Deep Learning Models. *POMACS* 6, 3 (December 2022).

[30] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[31] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*.

[32] Nermin Samet, Samet Hicsonmez, and Emre Akbas. 2020. HoughNet: Integrating near and long-range evidence for bottom-up object detection. In *European Conference on Computer Vision (ECCV)*.

[33] Konstantin Shmelkov, Cordelia Schmid, and Karteek Alahari. 2017. Incremental learning of object detectors without catastrophic forgetting. In *Proceedings of the IEEE international conference on computer vision*. 3400–3409.

[34] TensorFlow. 2022. TFF GLDv2. https://www.tensorflow.org/federated/api_docs/python/tff/simulation/datasets/gldv2/load_data.

[35] Yuanyishu Tian, Yao Wan, Lingjuan Lyu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. FedBERT: When Federated Learning Meets Pre-training. *ACM Trans. Intell. Syst. Technol.* 13, 4 (2022).

[36] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: accurate learning for energy and timeliness. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*.

[37] Essam Wisam. 2022. Language Modeling with LSTMs in PyTorch. https://towardsdatascience.com/language-modeling-with-lstms-in-pytorch-381a26badcbf/.

[38] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[39] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[40] Suwei Yang, Massimo Lupascu, and Kuldeep S. Meel. 2021. Predicting Forest Fire Using Remote Sensing Data And Machine Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 17 (2021).

[41] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. 2023. Zeus: Understanding and Optimizing {GPU} Energy Consumption of {DNN} Training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 119–139.

[42] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*.