APPROXCALIPER: A PROGRAMMABLE FRAMEWORK FOR APPLICATION-AWARE NEURAL NETWORK OPTIMIZATION

Yifan Zhao [1 *] **Hashim Sharif** [2 *] **Peter Pao-Huang** [1] **Vatsin Ninad Shah** [3] **Arun Narenthiran Sivakumar** [1] **Mateus Valverde Gasparino** [1] **Abdulrahman Mahmoud** [4] **Nathan Zhao** [5] **Sarita Adve** [1] **Girish Chowdhary** [1] **Sasa Misailovic** [1] **Vikram Adve** [1]

ABSTRACT

To deploy compute-intensive neural networks on resource-constrained edge systems, developers use model optimization techniques that reduce model size and computational cost. Existing optimization tools are *application-agnostic* – they optimize model parameters solely in view of the neural network accuracy – and can thus miss optimization opportunities. We propose **ApproxCaliper**, the *first programmable framework for* ***application-aware*** *neural network optimization*. By incorporating application-specific goals, ApproxCaliper facilitates more aggressive optimization of the neural networks compared to application-agnostic techniques. We perform experiments on five different neural networks used in two real-world robotics systems: a commercial agriculture robot and a simulation of an autonomous electric cart. ApproxCaliper achieves 5.3× higher speedup and 2.9× lower GPU resource utilization, and 36× and 6.1× additional model size reduction, respectively, than Learning Rate Rewinding (LRR), a state-of-the-art structured pruning tool used in an application-agnostic setting.

1 INTRODUCTION

Many emerging edge applications combine multiple neural network (NN) models (which extract actionable information from sensor data, such as RGB, LIDAR, audio, and GPS) together with other computations to achieve end-to-end goals. However, deep learning workloads are often compute- and power-intensive, which makes it challenging to deploy these models on resource-constrained edge compute devices with tight constraints on power, weight, size and production costs (Pedersen et al., 2006; Kim et al., 2010). An important opportunity, however, is that the applications exhibit a significant amount of **application-level error resilience**: reducing an individual component's accuracy is acceptable if it avoids any *observable impact on the end-to-end quality of the application*. Application-level error-resilience manifests in many different domains that use NNs, including autonomous navigation systems, augmented and virtual reality (AR/VR) stacks, and real-time data analytics. For instance, in an autonomously navigating robot, a reduction in the accuracy of the NN perception component may not adversely affect the navigation quality, due to the error resilience in the control components. Similarly, we found that for a real-time data analytics application (corn stem counting using object detection), the final stem counts are minimally affected by locationing errors in the object detection component.

Currently, application developers can reduce the costs of NN models with various NN-specific optimization techniques such as pruning, quantization, weight compression, and low-rank factorization (Sainath et al., 2013; Han et al., 2016; Anwar et al., 2017; Frankle & Carbin, 2019; Hubara et al., 2017; Swaminathan et al., 2020; Ruan et al., 2021; Xu et al., 2021). These techniques optimize a neural network with the goal to *maintain the same accuracy as the original model.* They have proven effective in many scenarios where the NN *is* the entire application.

However, using these techniques naïvely for composite edge applications misses the rich opportunity to take advantage of the application-level resilience. Instead, they take the conservative approach of *constraining the optimization levels to retain the accuracy* of the original unoptimized networks, even when this constraint is not required to satisfy the application's end-to-end goal.

We instead propose to incorporate the application-level error resilience in the optimization process, which allows the NN accuracy to be relaxed, yielding more aggressively optimized models and less costly execution. Application-aware optimization can provide several times more speedup and often more than an order of magnitude less memory consumption compared to application-agnostic techniques. These improvements make it possible to deploy compute-

[*]Equal contribution [1]University of Illinois at Urbana-Champaign [2]AMD Research [3]Google [4]Harvard University [5]Tesla. Correspondence to: Yifan Zhao <yifanz16@illinois.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022.

intensive NN models on resource-constrained edge systems, as we will demonstrate in this paper.

Our Work. We develop ApproxCaliper, the *first application-aware neural network optimization framework*. It is application-aware in that it uses a developer-specified application-level QoS (Quality of Service) goal for tuning. For instance, for an autonomous robot that uses neural networks for visual perception, a developer can specify an application-level QoS goal to autonomously navigate without collisions for a given distance. ApproxCaliper uses this QoS goal as constraints under which it relaxes NN accuracy with approximation techniques to gain higher performance benefits.

Tuning the accuracy-vs-performance tradeoffs for neural networks deployed in end-to-end applications is complicated and time-consuming. It is complicated because it requires tuning low-level system- and NN-specific parameters, which have complex interactions with each other, and their impacts vary across different applications. For instance, for a control system that uses predictions from multiple NN components, errors in one component may affect how much error can be tolerated from other NNs, before quality of control decisions becomes unacceptably low. Tuning is time-consuming because the search space is often huge, and empirically evaluating each configuration to measure application-level QoS can be expensive (e.g., running a robot in a field). Thus we need to guide the search toward the configurations that are likely to yield profitable tradeoffs.

To reduce the complexity and cost, our novel optimization algorithm starts from the observation that the feasibility of a configuration in the application-level tradeoff space depends on the levels of error in all NN components. This allows us to *reduce* the problem of searching the application-level space to searching the space of NN errors measured by N NN-specific error metrics. Our algorithm then navigates the NN error space with statistical error injection that varies the level of error in each NN component. We also assume that feasible configurations with the highest errors gives the highest flexibility for QoS-constrained optimization. Our optimization search strategy, therefore, is to direct the search to more actively explore such configurations.

We present a two-phase optimization approach with an *online* (on-device) error calibration phase and an *offline* model tuning phase. Our novel **error calibration algorithm** uses statistical error injection to efficiently separate valid (acceptable application QoS) vs. invalid regions in the error space. We show that this technique can guide the **model selection and tuning phase** (that makes use of one or more off-the-shelf application-agnostic NN optimization technique) towards configurations that maximize the given objective, while still satisfying the learnt error constraints.

ApproxCaliper automatically tunes multiple NN components jointly, selecting an optimized model variant for each component. This process is automated and only requires minimal developer inputs. It also simultaneously tunes for the choice of model variant and value of NN-specific performance metrics (such as model FPS) if given by the developer, to optimize *system-level* goals such as resource utilization. ApproxCaliper provides an easy-to-use programmable interface for all the necessary developer inputs.

1.1 Summary of Results

We evaluate two *real-world* autonomous cyber-physical systems, **CropFollow** and **Polaris-GEM** with five different NN architectures used for visual perception tasks (detailed in §2.2 and §2.3). For both systems, ApproxCaliper discovers significant room for relaxing NN accuracy while maintaining application-level QoS. For CropFollow, ApproxCaliper co-tunes two different NN models in the application, and achieves $5.3\times$ greater speedup and $36\times$ greater model size reduction compared to Learning Rate Rewinding (LRR), an application-agnostic state-of-the-art structured pruning algorithm. For Polaris-GEM, ApproxCaliper co-tunes for model FPS and pruned model variants, and achieves $2.9\times$ lower GPU resource utilization and $6.1\times$ greater model size reduction compared to LRR.

For CropFollow, these performance improvements allowed us to run the full system on a single $35 Raspberry Pi4. This compares favorably to the $99 Jetson Nano, the cheapest device to deliver necessary performance with application-agnostic pruning (LRR). Our work has led CropFollow's provider company (EarthSense) to consider lower cost alternatives for computing hardware.

1.2 Contributions

- We propose ApproxCaliper, the first programmable framework that optimizes neural networks in an *application-aware* manner. It captures how errors and performances of NN components in an application interact and affect the application-level QoS.
- We present a novel two-phase neural network model optimization strategy, with an error calibration phase and a model optimization and selection phase, which achieves higher end-to-end performance improvements compared to traditional empirical autotuning.
- Our evaluation on the end-to-end software stacks of two cyber-physical systems shows that ApproxCaliper provides significantly higher improvements compared to application-agnostic tuning approaches.

2 Background and Motivation

We assume an application includes one or more neural networks, and must achieve a design goal captured by a specified quality-of-service (QoS) metric. The goal of our work is to enable (and simplify) the use of aggressive optimizations that relax component-level semantics, such as NN inference accuracy, to the extent possible while ensuring the

QoS goal is met. We first define terminology used throughout the paper (§2.1), and then describe the two evaluated cyber-physical applications (§2.2 and §2.3).

2.1 Preliminaries and Terminology

A **Configuration** is an assignment of a *possibly optimized model variant* (e.g., a pruned neural network with specific pruning fractions) to each NN component in the application, and (optionally) NN performance parameters such as FPS (frames per second).

Application-level QoS is an application-specific metric defined by a real-valued executable function for a specified configuration (the "QoS Evaluator"), along with a lower-bound value for the desired application quality (the "QoS Target"). This function should quantify how well the application delivers on its desired goals. For example, for an autonomous mobile robot, a relevant QoS metric for navigation quality is the time the robot travels before requiring human intervention, e.g., due to a collision.

Valid and Invalid Configurations. A configuration is *valid* if it meets the QoS target (QoS $\geq$ QoSTarget), and is *invalid* otherwise (QoS $<$ QoSTarget).

Autotuning is a design space exploration technique that finds configurations that maximize an objective function, while meeting the QoS target. Since autotuning is a *heuristic search* performed over (usually) intractably large search spaces, it finds locally optimal configurations.

2.2 CropFollow Autonomous Navigation System

We evaluate a state-of-the-art *commercial* agriculture robot called *TerraSentia*, obtained from EarthSense (Earthsense, 2020), used by clients for high-throughput phenotyping and a variety of other agriculture tasks. The robot is equipped with an autonomous vision-guided navigation system named *CropFollow* (Sivakumar et al., 2021) used for *row-following* navigation through fields of crops.

CropFollow's goal is to navigate the robot through the center line of a (possibly curved) crop row. It contains a number of components that collaborate to keep the robot on its intended path, as shown in Figure 1:

- *Heading / distance prediction NNs.* 2 ResNet-18 NNs take 320×240 RGB images as input and estimate the heading angle θ and distance-to-edge d of the robot. θ is the angle of the robot's direction relative to adjacent crop rows. $d \in [0, 1]$ is the ratio between the robot's distance to the left crop row and the total row width.
- *Extended Kalman Filter (EKF).* EKF fuses (θ, d) predictions with measurements from an *Inertial Motion Unit* (IMU) to refine these predictions.
- *Model Predictive Controller (MPC).* MPC uses refined pose estimations from EKF to compute angular velocity commands to keep the robot on its intended path.

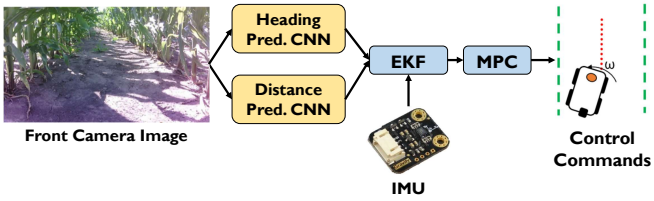

Figure 1: Cropfollow navigation system workflow.

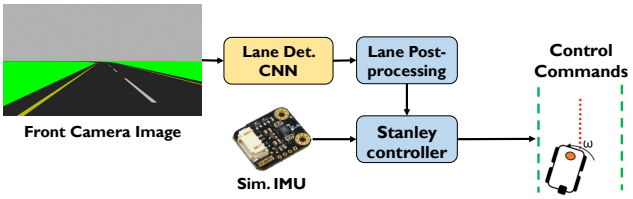

Figure 2: Polaris-GEM vehicle simulator workflow.

Figure 3: Workflow of the CropFollow and Polaris-GEM systems.

2.3 Polaris-GEM Vehicle Simulator

Polaris-GEM is a Gazebo-based simulation of an *autonomous lane-following*, commercially-sold electric cart, Polaris GEM e2 (Salfer-Hobbs & Jensen, 2020). The original simulator used GPS-based perception, but we modified it for vision-based perception using LaneNet (Neven et al., 2018), a state-of-the-art lane detection network. Figure 2 shows the components of Polaris-GEM, which include:

- *Lane Detection Network.* A LaneNet (Neven et al., 2018) with a VGG-16 backbone takes a 512×256 RGB image (from Gazebo-simulated front camera) and produces 2 outputs: a Boolean mask $B : 512 \times 256$ and an embedding tensor $E : 4 \times 512 \times 256$. B distinguishes lanes from the background; E distinguishes the lanes from each other.
- *Lane Post-processing.* The post-processing module combines the masks (B, E) with a clustering algorithm and finds each lane as a cluster of pixels. It then fits a *polynomial curve* through the pixels of each lane, and eventually converts these curve equations into heading and distance estimates (θ, d) (similar to CropFollow).
- *Stanley Controller.* A Stanley controller (Thrun et al., 2006) uses (θ, d) estimates and current velocity (from Gazebo-simulated IMU) to compute a steering angle and guide the vehicle back to the center.

3 ApproxCaliper Framework

Figure 4 shows the workflow of the ApproxCaliper framework. At a high-level ApproxCaliper takes as input (1) NN models to optimize, (2) NN-specific metrics that are used to quantify the output error of the NN models (used for error calibration analysis), (3) a developer-provided QoS evaluator and QoS target, and (4) a developer-provided application performance evaluator, and uses these to perform application-aware optimizations on the NN models. *To*

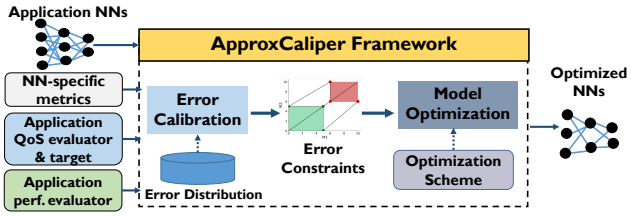

Figure 4: ApproxCaliper High-level Workflow

reduce the cost of tuning and improve the quality of configurations, ApproxCaliper performs optimization in two phases: (1) An *error calibration phase* (§3.2) that evaluates empirically how the end-to-end application QoS is affected by accuracy degradation of the NN components in the application pipeline, and (2) A *model tuning and selection phase* (§3.3) that optimizes the NNs based on the application's error constraints computed by the calibration phase.

First, we discuss the high-level intuition of our approach, and next, the algorithmic details of the error calibration and model tuning phases in ApproxCaliper. In §3.4, we discuss how developers can use the ApproxCaliper interface to optimize neural networks in an application-specific manner.

3.1 Intuition for our Two-phase Tuning Approach.

Motivation. Empirical autotuning is *computationally expensive* because (1) it usually requires many tuning iterations to achieve good results, and (2) each empirical evaluation can be expensive; for instance, for CropFollow, empirical evaluation can be expensive in terms of both time and human resources because the *Terrasentia* robot needs a human observer for potential manual interventions.

ApproxCaliper's two-phase tuning approach achieves higher speedups and model footprint improvements in the same tuning time budget as traditional empirical tuning. Systematically quantifying the error constraints before the tuning phase allows for filtering invalid configurations (unacceptably low QoS), and helps direct the search towards configurations that more likely has high speedups.

Goal of the Error Calibration Phase. The purpose of error calibration is to learn the extent to which the accuracy of NN components can be reduced without violating end-to-end application-level QoS targets. This is achieved by statistical error injection into NN outputs and empirical evaluation of the application QoS for a limited number of times. The injected errors are quantified using one or more traditional error metrics per NN, such as accuracy, precision, or any custom function. These error metrics $M_j, 1 \leq j \leq N$ define a bounded N-dimensional *error constraint space*. Figure 5 shows an example error constraint space with two error metrics $M1$ and $M2$. The error constraint space captures how *simultaneous* changes in the error metrics affects the end-to-end QoS. Error calibration partitions this space into three disjoint regions: **valid** (QoS $\geq$ QoSTarget, shown in green) **invalid** (QoS $<$ QoSTarget, shown in red) and **unvisited** (unknown QoS, shown in white).

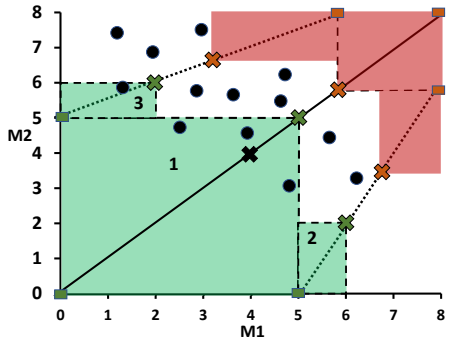

Figure 5: Example shows how the error calibration algorithm partitions the 2-dimensional error space into *valid* (green) and *invalid* (red) configuration regions. M1 and M2 are error metrics. Valid regions marked number 1, 2, 3 are created in the first 3 iterations, respectively. The green crosses show the MVP (maximal valid points) and red crosses show the MIP (minimal invalid points). The black dots show configurations searched on the boundary of the valid and unvisited regions.

Goal of the Model Optimization and Autotuning Phase. The model autotuner uses heuristic search techniques to navigate the configuration space and *filters* away (1) invalid region (red region) configurations, and (2) valid region (green region) configurations with low error metric values. Configurations with lower error are *not* desirable since *a lower error means less opportunity for optimization.* We refer to valid configurations with high error metric values as *high potential* since a higher metric value indicates a higher error slack, and hence more optimization opportunity. *High potential* configurations are likely found close to the boundary of the valid and unvisited region, as this area includes the highest error values but still produces valid QoS. In Figure 5, the black dots in the green and white regions show the configurations searched in the model tuning phase. The unvisited (white) region is included in the search since it *potentially* includes valid configurations (has unknown QoS). There are no visited configurations on the bottom left of Figure 5, since points with low error metric values are *not* high potential configurations.

3.2 Error Calibration Phase

The error calibration phase uses artificial error injection to add varying levels of errors to NN outputs, and measures the impact on the end-to-end application-level quality.

Error Distribution for Error Injection. The error calibration algorithm injects error to the outputs of NNs as $Y = X + \varepsilon$, where X is the output of the non-approximate NN, Y is the output after error injection, and $\varepsilon \sim \mathcal{D}(p)$ is a sample from an error distribution with parameters p. For instance, if errors are sampled from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$, the error calibration algorithm varies μ and σ to control the error injection. ApproxCaliper varies these dis-

tribution parameters to vary the output error in terms of the specified error metric (Input #2 described below).

ApproxCaliper includes routines that *automatically* select distributions that mimic the errors that manifest with a chosen approximation technique (e.g., low-rank factorization). ApproxCaliper evaluates a set of predefined parametric distributions and returns the one that best fits the distribution of prediction errors. For floating-point outputs, it evaluates Gaussian, log-normal, exponential, and student-T distribution, and for integer outputs, it evaluates binomial, Poisson, and logarithmic distributions. Our current implementation includes these relatively simple distributions, since we find them to appropriately model errors for a variety of different CNNs. Support for more complicated error distributions is trivial to add in the ApproxCaliper framework: users are expected to provide a distribution function with parameters (ApproxCaliper's autotuner automatically varies these to vary the amount of injected output error).

First we describe the key inputs to the error calibration phase and then detail the algorithmic workflow.

Developer Input #1: Baseline Neural Networks. Developers specify one pretrained NN model per NN component to use for error injection. These are the baseline models with no approximation. It is preferred that developers specify models with high accuracy since it gives the highest margin for error injection.

Developer Input #2: Neural Network Metrics. The ApproxCaliper interface provides developers with a flexible way to specify task-specific NN error metrics, which is important since different NN tasks use different metrics such as accuracy or mAP. Developers can use a predefined error metric or provide a custom function $ErrorMetric(Y, G)$, where Y is the output of the NN model after error injection and G is the ground truth. Developers can also specify a performance metric instead such as DNN throughput, and ApproxCaliper can analyze the combined impact of NN compute performance (e.g., FPS) and prediction error (e.g., accuracy, mAP) on end-to-end QoS. The interface allows specifying multiple metrics (e.g., precision and recall) per NN that capture different aspects of the error. For each metric, users also need to provide a *lower bound* and *upper bound* value as the range for error injection.

The error calibration algorithm assumes that *all the metrics must be monotonic with the same directionality*, i.e., a higher metric value must always be more desirable than a lower value. This monotonicity assumption is satisfied by most common metrics such as classification accuracy, F1 score, PSNR, etc. If a metric m is monotonic but in the opposite direction, then one can use $-m$.

Developer Input #3: End-to-end QoS Evaluator. Developers provide the QoS evaluator and the QoS target via the ApproxCaliper interface. The error calibration algorithm uses this QoS evaluator to measure the impact of NN errors on the end-to-end QoS. This function should initialize the application, run the application for some time / iterations, and return the end-to-end QoS.

Output. The output is an N-dimensional *error constraint space* that captures the effect of increasing NN error on the application-level QoS. The error constraint space is a space of the N metrics provided as Input #2. The lower and upper bounds of each metric collectively define two N-dimensional points $M_{lb} = (M_{lb,1}, \ldots, M_{lb,N})$ and $M_{ub} = (M_{ub,1}, \ldots, M_{ub,N})$ and an *N-dimensional box* between them that forms the overall search space. We refer to a union of one or more N-dimensional boxes as a *region*. The algorithm partitions the search space into valid (green) and invalid (red) region, effectively finding the boundary that separates these two regions. Figure 5 shows an example of a 2-dimensional error constraint space where $M_{lb} = (0, 0)$ and $M_{ub} = (8, 8)$.

3.2.1 Algorithm Description.

Algorithm 1 shows the error calibration algorithm. The core idea is to recursively divide the search space into smaller rectangles and search for boundary points on the diagonal of each rectangle. The algorithm exploits *domination relation* in the space to efficiently infer validity of rectangles. If a point P is evaluated as valid, the algorithm infers the rectangle with lower or equal error than P in all metrics $R_{L;P} = \{Q \mid \forall i. \quad Q_i \leq P_i\}$ as **valid**. We say point P *dominates* all points $Q \in R_{L;P}$, and P is the *highest* point of $R_{L;P}$. Conversely, if P is invalid, the rectangle $R_{U;P}$ with higher or equal error than P is **invalid**, P *is dominated by* all points $Q \in R_{U;P}$, and P is the *lowest* point of $R_{U;P}$. By this definition, each rectangle has a unique lowest point and a unique highest point.

For each rectangle, we perform a binary search (Lines 16-26) on its *diagonal*. The algorithm traverses this diagonal to find a *pair of boundary points*: a *maximal valid point* (MVP) and a *minimal invalid point* (MIP). MVP is the highest point on the diagonal that evaluates as valid and MIP is the lowest point invalid. The rectangle dominated by MVP is added into the valid region (Line 27), and the rectangle that dominates MIP is added into the invalid region (Line 28). Then the algorithm computes rectangles to visit next based on the coordinates of MVP and MIP (Line 29). The gap between MVP and MIP also creates an unvisited rectangle (usually small). The volume of this rectangle is controlled by the number of evaluations done per diagonal (`maxDiagEvals`); a higher number of evaluations per diagonal increases analysis time but reduces the size of the unvisited rectangle. The algorithm finishes when the queue of rectangles is empty or the number of QoS evaluations reaches `totalEvals`.

Example. Figure 5 illustrates *three* outer loop iterations of the algorithm on an example. Assuming two error met-

Algorithm 1: Error Calibration Algorithm.

```
Inputs:
  • baselineNNs: pretrained NNs, one per NN component
  • qosEvaluator: function that evaluates app-level QoS
  • qosTarget: QoS target to satisfy
  • lowerBound, upperBound: thresholds for metrics
  • errorDistrib: error distribution for sampling errors
  • maxDiagEvals: maximum evaluations per diagonal
  • totalEvals: total number of evaluations
Output: validRegion, invalidRegion, unvisitedRegion
Function errorCalibrator
    fullRectangle = Rectangle(lowerBound, upperBound);
    rectQueue = PriorityQueue([fullRectangle]);
    nEvals = 0;
    while not rectQueue.empty() and nEvals < totalEvals do
        rectangle = rectQueue.pop();
        diag = rectangle.getDiagonal();
        for rEvals = 0 to maxDiagEvals do
            if nEvals ≥ totalEvals then break;
            qos = qosEvaluator(baselineNNs, errorDistrib,
              diag.midPoint);
            if qos ≥ qosTarget then
                diag = Diagonal(diag.midPoint, diag.MIP);
                diag.MVP = diag.midPoint;
            else
                diag = Diagonal(diag.MVP, diag.midPoint);
                diag.MIP = diag.midPoint;
            nEvals += 1;
        validRegion ∪= DominatedRect(diag.MVP);
        invalidRegion ∪= DominatedRect(diag.MIP);
        rectQueue.push(rectangle.getSubRectsToVisit());
    unvisitedRegion = fullRectangle − validRegion −
      invalidRegion;
    return validRegion, invalidRegion, unvisitedRegion;
```

rics, $M1$ and $M2$, the first iteration traverses the diagonal between $(0, 0)$ (lowest metric values) and $(8, 8)$ (highest metric values). It first evaluates the midpoint $(4, 4)$ (shown as a black cross). Given $(4, 4)$ is found as a *valid point* a new higher midpoint $(6, 6)$ is evaluated. $(6, 6)$ is found as an invalid configuration and hence the search proceeds to evaluate a new midpoint $(5, 5)$. Assuming three evaluations on the diagonal ($maxDiagEvals = 3$), $(5, 5)$ is found to be the diagonal's MVP and $(6, 6)$ is found as the MIP, which are used to create the dominating rectangular green (valid) region and red (invalid) region, respectively. Subsequent iterations create further green and red rectangles – green rectangles labelled 2 and 3 are created in 2nd and 3rd iteration, respectively.

3.3 Model Tuning and Optimization

In the model tuning phase, ApproxCaliper searches for *configurations* that minimize the given performance objective while satisfying the application-level QoS target. A configuration contains choices of optimized NN variants generated by approximation techniques (described later). It may also include NN-specific performance metrics (e.g., model FPS); this allows ApproxCaliper to co-tune for NN performance and error. Each configuration corresponds to a point (shown as black dots in Figure 5) in the error constraint space.

To explore the search space more efficiently, ApproxCaliper uses the error constraint space to skip empirical evaluations for (1) configurations in the invalid regions, and (2) configurations that are well within the valid regions (away from the boundary). Note that we also evaluate configurations in the valid region, but only those within a certain configurable distance to the boundary of valid and unvisited regions, since it is possible that the unvisited regions alone do not yield any valid configuration. Figure 5 shows how ApproxCaliper only searches for *high potential* configurations in these regions. This helps *steer the search* away from both low-potential (too conservative) and known invalid configurations. Our approach ensures that costly empirical evaluations are only spent on configurations that are likely to provide high improvements.

NN Approximation Techniques in ApproxCaliper. ApproxCaliper currently supports two existing approximation techniques (which can be applied in combination): (1) structured pruning based on *Learning Rate Rewinding* (LRR) (Renda et al., 2020) and (2) low-rank factorization (or LR factorization) based on (Tai et al., 2016). We explain the working of these techniques in detail in Appendix §A.1. Developers can also incorporate new approximation techniques (e.g., perforated convolutions (Figurnov et al., 2015)) with the ApproxCaliper interface.

In ApproxCaliper, we apply structured pruning and LR factorization *iteratively*. Each iteration removes a fraction of filters or ranks from convolution layers, producing smaller and more efficient models with (usually) lower accuracy, and creating a tradeoff space of models to choose from. For LR factorization, this iterative usage is not proposed in the original work (Tai et al., 2016). Additionally, in the first factorization iteration, each layer is decomposed into two layers. In further iterations, only the ranks of weight tensors are reduced, and no other layer splitting is performed. We make this choice to avoid an explosion of layers, which can hurt compute performance.

Autotuning to Search Profitable Configurations. The NN variants produced by the approximation techniques and NN-specific performance metrics (if any) together create a large search space of configurations on which an exhaustive approach is intractable. To make the search feasible, we use OpenTuner (Ansel et al., 2014), a library for building custom autotuners. In addition to the QoS evaluator, autotuning also uses the application performance evaluator that developer provided to ApproxCaliper.

3.4 ApproxCaliper Interface

ApproxCaliper is developed as a Python library with an easy-to-use API. Figure 6 shows an example of how the ApproxCaliper interface is used to optimize the heading and distance prediction NNs in the CropFollow autonomous navigation stack. Interface functions involved in the ex-

```
import approx_caliper as ac
# Load NN models; supports Pytorch and ONNX models
heading_nn, distance_nn = ac.load_model("resnet18_h.onnx"), ac.load_model("resnet18_d.onnx")
nns = [heading_nn, distance_nn]
trainset = ac.load_dataset("cropfollow_data/", "cropfollow_labels.json")
# Define metrics for measuring the error of each NN's output; to be used in error calibration.
metrics = [ac.ErrorMetric(heading_nn, ac.l1_error), ac.ErrorMetric(distance_nn, ac.l1_error)]
# Use predefined Structured Pruning method to compress the NNs
# Prune 20 levels with each level pruning 20% of the weights
structured_pruner = ac.nn_approx.StructuredPruner(n_steps=20, prune_fraction=0.2)
# Fits the errors of the optimization technique to an error model -- e.g., Gaussian, Bernoulli
error_model = ac.find_error_model(structured_pruner, nns, trainset)
# User provides method to evaluate end-to-end application quality and application performance
qos_eval, perf_eval = CropFollowQoSEvaluator(), CropFollowPerfEvaluator()
# error_calibrate interface invokes the Error Calibration phase -- computes error constraints
constraints = ac.error_calibrate(
   nns, error_model, metrics, evaluator, qos_target={"collision": 0}, iters=25)
# optimize input NNs using the given optimization scheme (structured pruner)
optimized_nns = ac.optimize(structured_pruner, constraints, nns, trainset, perf_eval)
```

Figure 6: Example of using the ApproxCaliper interface with the NN components in the CropFollow autonomous navigation stack.

ample are described and explained with comments. A few key functions that deserve more explanation are detailed in Appendix §A.4.

3.5 Using ApproxCaliper with a New Application.

As shown in Figure 6, using ApproxCaliper's interface to optimize a new application requires only a few lines of code. The application QoS and performance evaluator are the only function that developers need to implement, which runs the application for some time (iterations) to compute the QoS and measure the performance. Since application test suites usually include such scripts, it requires minimal additional effort in most cases.

Additionally, developers need to specify the number of runs used in error calibration phase and model optimization phase. This is a common practice in existing autotuning works, as automatic approaches (e.g. stop on convergence) are difficult. If there is a total tuning time budget, developers need to manually split it between these two phases.

4 Experimental Methodology

We discuss the important methodology aspects here. Some additional details on experimental methodology can be found in Appendix §A.2.

Optimization Goals and Constraints. We apply ApproxCaliper on **CropFollow** to optimize its 2 NN models simultaneously, maximizing the application-level FPS without introducing collisions. *The QoS target* is that the robot should have 0 collisions with the row boundaries in a 100m run in *real-world* corn fields. The QoS evaluator prompts for the total collisions, which a human observer observes and manually inputs after each run, since the robot has no automatic collision detection mechanism. *The optimization goal* FPS is a proxy for latency, and is relevant for autonomous navigation systems because feedback control systems have *minimal FPS requirements* (Falanga et al., 2019; Anwar & Raychowdhury, 2020) to function correctly. FPS improvements enable execution on low-end compute devices which would otherwise not provide minimal FPS.

Similarly, we apply ApproxCaliper on **Polaris-GEM** to optimize for a combination of NN FPS and NN model. ApproxCaliper searches for an optimized (e.g., pruned) NN model and the FPS that the NN component runs at, to minimize GPU utilization without introducing lane departures. Running the model at a lower FPS reduces GPU utilization as the process sleeps between frames, freeing up GPU cycles. *The QoS target* is the robot should make 0 departures from the current lane in a 506 meter run in simulation. *The optimization goal*, GPU utilization, is a proxy for power usage; we don't report power numbers since there is no easy way to precisely measure power on the target GPU, an Nvidia Quadro P5000.

Approximation Techniques Setup. We develop our own structured pruning and LR factorization implementation based on the original papers ((Renda et al., 2020) for pruning and (Tai et al., 2016) for LR factorization). For both techniques, we set up ApproxCaliper to use a fixed number of iterations each removing an additional 20% filters or ranks. We apply pruning with 20 iterations for NNs in CropFollow and 12 iterations for LaneNet, as the lane detection quality of LaneNet decreases drastically after iteration 12. The output model of each iteration is referred to as a *prune level* and labelled from 1 to 20 (or 12) inclusive, and 0 is the unpruned baseline. We apply LR factorization to LaneNet with also 12 iterations due to accuracy degradation after iteration 12.

Error Calibration and Model Tuning Setup. There are 63 candidates for each of the two NN components in CropFollow, creating a total of $63 \times 63 = 3969$ combinations. Brute

force search is infeasible since each empirical *field evaluation* takes 4-5 minutes. The same holds true for Polaris-GEM which has 2730 configurations and each run takes 5-6 minutes. While we had performed more than 700 hours of field experiments to refine our algorithms, we limit the tuning to 50 navigation runs (∼5 hours). We allot 20 runs to error calibration phase and 30 to model optimization phase, and compare the result against 50 unguided (without ApproxCaliper) autotuning runs using Opentuner.

Neural Network Specific Error Metrics. For both NNs in CropFollow, we use *standard deviation of error* as the error metric in error calibration, because pruned models mostly have zero-centered errors (confirmed by zero-mean Gaussian distribution found by ApproxCaliper).

In Polaris-GEM, the quality of lane detection is measured by *lane detection accuracy*, a quantity between 0 (worst) and 1 (best): $Acc = \sum_{image} \frac{C_{im}}{S_{im}}$, where C_{im} is the number of correctly detected lanes in an image, and S_{im} the number of lanes in ground truth. A lane is correctly detected if the average distance between its detected points and the ground-truth points is less than 20 pixels (the same threshold used in the TuSimple challenge (Zhou, 2018)).

Application-agnostic Baseline. As baselines for comparison, we apply pruning using LRR (Renda et al., 2020) or LR factorization using the technique in (Tai et al., 2016) to all the candidate neural network architectures (e.g., LaneNet-VGG16 and LaneNet-DarkNet for Polaris-GEM) with the constraint to *retain the same accuracy as the original model*, and pick the most efficient pruned variant from this set as our baseline. This represents an intelligent use of today's state-of-the-art model optimization approaches, which accounts for manual selection of neural network architecture and optimized variants, *but does not exploit application resilience to increased inference error*. For instance, for Polaris-GEM, LaneNet-DarkNet prune level 1 (20% weights pruned) is selected as the baseline since it is the most efficient pruned model that retains the original model accuracy of the unpruned LaneNet-DarkNet model, and is more compute-efficient than any of the LaneNet-VGG16 optimized model variants with equivalent accuracy.

5 EVALUATION

In our experiments, we consider the following questions:

RQ1: Does ApproxCaliper's application-aware pruning provide more benefits than today's practice of application agnostic pruning (retaining original model accuracy)?

RQ2: Does the ApproxCaliper error calibration framework identify opportunities for relaxing NN accuracy requirements without impacting the end-to-end QoS?

RQ3: Do the error calibration results guide ApproxCaliper to better tune the NN components than "*unguided tuning*"? Unguided tuning is in-field autotuning without considering the error constraint space computed by the calibration phase, i.e., this requires exploring configurations anywhere within the search space.

RQ4: Do the valid and invalid regions obtained by calibration correspond to the real-world separation between acceptable and unacceptable configurations?

We find that structured pruning provides a **strictly better** accuracy-performance tradeoff than LR factorization. However, LR factorization may outperform pruning in other applications or settings, so the ability to support it is significant. We evaluate LR factorization on Polaris-GEM in §A.3 to show that the benefits of ApproxCaliper's application-aware approach generalize to other approximations.

5.1 Application-aware vs. Application-agnostic Pruning

To answer RQ1, we compare the best pruning configurations for CropFollow and Polaris-GEM, selected by 3 strategies, all using LRR: (1) guided tuning with ApproxCaliper, (2) unguided autotuning with ApproxCaliper, and (3) the application-agnostic pruning baseline. We focus on LRR (i.e., structured pruning) here because it gave dominant results over LR factorization but later experiments include both optimizations.

For CropFollow, Figure 7 shows how the performance (FPS) of the best configuration (found until that point) evolves with the increasing number of tuning iterations. Application-agnostic pruning using LRR gives a flat line (31.8 FPS) since this approach simply uses the best performing pruned model for heading and distance predictions, and does not involve any further tuning in the field. Application-aware pruning using ApproxCaliper provides significantly higher FPS: unguided tuning provides an FPS of 105.8 (i.e., 3.3× faster than application-agnostic pruning) and guided tuning provides 184.9 FPS (i.e., 5.8× faster). These are dramatic speedups obtained with the same optimization techniques, i.e., *purely by accounting for application-level quality goals*.

Similarly, for Polaris-GEM, Figure 8 shows the GPU utilization rate of configurations, which decreases over the course of tuning. Using application-agnostic pruning on LaneNet achieves a GPU utilization of 7.77%. Unguided and guided application-aware pruning using ApproxCaliper reduces the utilization to 2.92% (2.66× reduction) and 2.67% (2.91× reduction) respectively.

5.2 Error Calibration using ApproxCaliper

To answer RQ2, we apply ApproxCaliper's error calibration framework to identify how much error can be introduced to the NN predictions without affecting the navigation quality of CropFollow and Polaris-GEM. These experiments explain **how** ApproxCaliper achieves substantial overall gains.

Error Calibration Results on CropFollow. Figure 9 shows the error constraint space for CropFollow. Each of the 20 error calibration field evaluations are shown as black dots. Our key findings are:

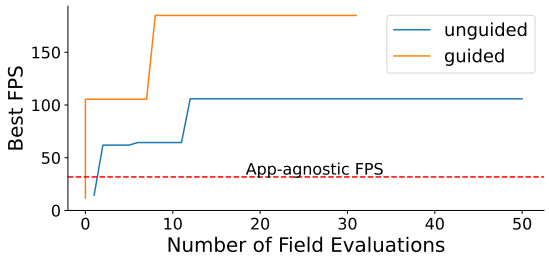

Figure 7: CropFollow tuning result. Graph shows how the performance in FPS (higher is better) of the best configuration evolves with increasing field evaluations. ApproxCaliper guided and unguided tuning is compared with LRR app-agnostic baseline.

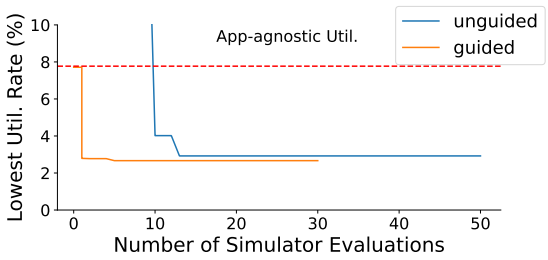

Figure 8: Polaris-GEM tuning result. Graph shows how the GPU utilization rate (lower is better) of the best configuration evolves with increasing simulator evaluations. ApproxCaliper guided and unguided tuning is compared with LRR app-agnostic baseline.

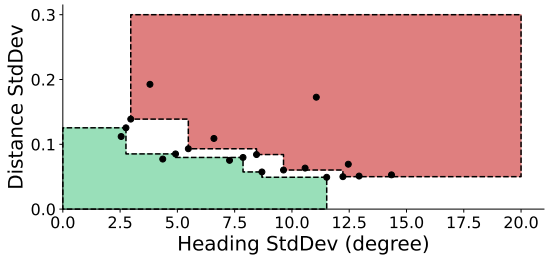

Figure 9: Error constraint space for CropFollow. It captures the interaction of errors in the heading model (x-axis) and the distance model (y-axis). Green region is the valid QoS region, red region is the invalid QoS region, and white regions are unvisited. The black dots on the figure show field evaluations.

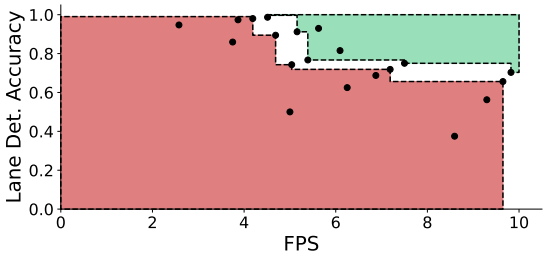

Figure 10: Error constraint space for Polaris-GEM. It captures the interaction of FPS (x-axis) and the errors in the LaneNet model (x-axis). Green region is the valid QoS region, red region is the invalid QoS region, and white regions are unvisited. The black dots on the figure show simulator evaluations.

- The heading model error in isolation (no errors in distance model) can be increased from 2.5 (baseline) to 11.5 degrees, without introducing collisions.
- Distance model error can be increased from 0.05 (baseline) to 0.13, without introducing collisions. In absolute terms, 0.13 is a large standard deviation of error since $0 \leq$ distance ≤ 1 (row center is 0.5).
- Errors can be simultaneously increased in both models. For instance, heading error and distance error can be simultaneously increased by $3\times$ and $1.6\times$, respectively, without introducing collisions.

Overall, these observations show that ApproxCaliper discovers *significant room for relaxing the accuracy*.

Error Calibration Results on Polaris-GEM. For Polaris-GEM, we study the *interplay of model FPS and lane detection accuracy*. We performed a similar analysis (and found similar insights) for CropFollow that we do not include for lack of space. Figure 10 shows the generated error constraint space. The key findings are:

- Higher FPS can counteract the effect of higher errors. As FPS increases from 4.5 to 9.8, lane detection accuracy can be reduced from 99.7% to 70.3%, without introducing any lane departures. Further investigation showed that, at high FPS, the effect of an erroneous prediction is short-lived since it enables more control actions per second.
- FPS and lane detection accuracy compensate for each other to a limited extent. No configuration with FPS below 4.5 or lane detection rate below 70% is feasible.

5.3 Guided vs. Unguided Tuning on CropFollow.

To answer RQ3, we compare guided and unguided tuning with ApproxCaliper.

Comparing Speedups. Figure 7 shows the model FPS achieved by guided and unguided tuning for CropFollow. Guided tuner provides a 1.76 $\times$ speedup (184.9 FPS vs. 105.8 FPS) over unguided tuning. Guided tuning also finds many more high-performance configurations. Across 30 field evaluations of guided tuning, it found 5 configurations that provided higher FPS than the unguided tuner's best result (105.8 FPS) in 50 field evaluations.

Figure 8 shows the utilization rate achieved by guided and unguided tuning for the Polaris-GEM experiment. Guided tuning provides a $1.1\times$ speedup over unguided tuning (GPU utilization of 2.66% vs. 2.92%). In addition, guided autotuning discovers the best found configuration on only the second simulator evaluation, while unguided tuning didn't find a single valid configuration (i.e., one with acceptable QoS) for the first 10 evaluations (hence the blue line starts

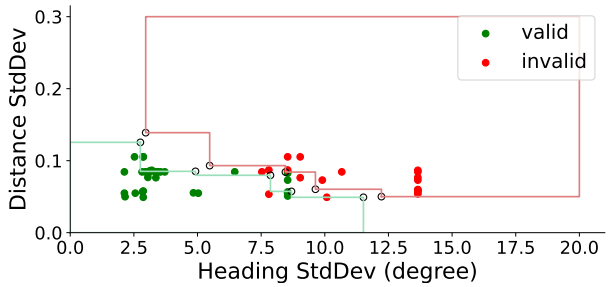

Figure 11: CropFollow: Configurations evaluated empirically in unguided autotuning (points) vs. error calibration regions used by guided autotuning (red and green polygons).

at evaluation 10 in Figure 8).

Effectiveness of Guided Tuning. Guided tuning uses the error calibration result to filter out configurations in the invalid regions and less performant configurations that are not fully pushing the boundary of acceptable error. A large fraction, 82% (41/50) of configurations evaluated in our unguided tuning experiment for CropFollow, and 78% (39/50) of configurations for unguided tuning for Polaris-GEM would be filtered/skipped if the same configurations are considered using guided tuning (i.e., using the error constraint space from calibration phase). Guided tuning is more effective because it focuses more of the tuning budget on configurations that are not known to be (likely) invalid, and likely to be performant.

5.4 Testing the Configurations in the Wild

To evaluate RQ4, Figure 11 shows the 50 configurations evaluated in unguided tuning for CropFollow (Figure 7) overlaid on the error calibration results (Figure 9). We illustrate using unguided tuning since the configurations are more spread out across all 3 regions (valid, invalid, and undetermined regions) than guided tuning. Valid and invalid configurations are shown as green and red points respectively. *Only two of the 50 points are false positives* (red points in the green region), and there are no false negatives (no green points in the red region). The white region is unvisited and is expected to include both valid and invalid points. Similarly, for Polaris-GEM (figure not shown), we observe only *three false positives and no false negatives* out of 50 simulator evaluations for unguided tuning.

These results show that the valid and invalid regions found by the error calibration phase correspond well to the real-world separation between acceptable and unacceptable configurations. The few false positives we observe (e.g., red point in the green boundary region) are likely to be due to out-of-distribution errors, which may be caused by networks being trained on different data (standard neural network training assumption).

6 Related Work

There is a long record of work on approximate computing techniques which relax accuracy to improve application performance. We focus here on techniques for optimizing deep learning models.

Systems for Automatic Model Optimization. A major limitation of existing neural network optimization systems (Han et al., 2016; 2015; Zhu & Gupta, 2018; Frankle & Carbin, 2019; Anwar et al., 2017; He et al., 2017; Li et al., 2017; Molchanov et al., 2017; Hubara et al., 2017; Zhou et al., 2016; 2017; Zhu et al., 2017; Swaminathan et al., 2020; Sainath et al., 2013; Davis & Arel, 2014; Chen et al., 2017) is that they *tune NN models in isolation*, and unlike ApproxCaliper, do not exploit application-level error resilience. ApproxCaliper has complementary goals to these model optimization techniques. ApproxCaliper can use these techniques (as we show for pruning and low-rank factorization) much more aggressively – it relaxes model accuracy to the extent that application-levels goals are not compromised.

Comparison against Existing Accuracy-aware Optimization Systems. While some automated model optimization systems (Tian et al., 2021; Sharif et al., 2021; Xu et al., 2020; 2021; Joseph et al., 2020) allow users to specify an acceptable accuracy loss specification (e.g., 1%, 2% loss) for optimization, these strategies are not *application-aware*. These thresholds are selected *arbitrarily* for experimental evaluation and often do not fully exploit the potential for relaxing accuracy. In this work, we show that application-level error resilience often allows accuracy to be relaxed very significantly to gain higher compute performance.

Moreover, application knowledge is not simply about acceptable accuracy loss. One significant limitation of existing accuracy-aware optimization systems is the lack of mechanisms for jointly optimizing simultaneous errors in multiple neural network components (such as done by ApproxCaliper in Section 5.2) A second key limitation is that none of these systems can model the relationships of performance and accuracy. As shown by our POLARIS-GEM results, the end-to-end QoS is not only dependent on neural network accuracy: it can also be impacted by FPS and other application-specific parameters (Section 5.2).

ApproxCaliper has much broader capabilities than existing accuracy-aware optimization systems. It has a programmable interface for computing application-specific error constraints, and can model error interactions across multiple NN components and between performance and accuracy.

7 Discussion and Conclusion

ApproxCaliper is the first programmable framework for application-aware neural network optimization. Our evaluations on autonomous cyber-physical systems show that application-aware optimization has tremendous potential

in enabling compute-intensive ML models to run on edge hardware. Similar opportunities for approximation exist in AR/VR, data analytics, and robotic manipulation, and many others. Our preliminary results (not included) show that a stem counting data analytics workload and a eye-tracking based foveated rendering also present significant opportunities for application-aware optimization. While some applications (e.g., intermittent systems that require high accuracy inference (Gobieski et al., 2019)) will not be resilient to aggressive accuracy reductions, the error calibration framework in ApproxCaliper can still be useful to help quantify the level of error-resilience.

REFERENCES

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014.

Anwar, A. and Raychowdhury, A. Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning. *IEEE Access*, 8:26549–26560, 2020. doi: 10.1109/ACCESS.2020.2971172.

Anwar, S., Hwang, K., and Sung, W. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3), 2017.

Chen, T., Liu, H., Shen, Q., Yue, T., Cao, X., and Ma, Z. Deepcoder: A deep neural network based video compression. In *2017 IEEE Visual Communications and Image Processing (VCIP)*. IEEE, 2017.

Davis, A. S. and Arel, I. Low-rank approximations for conditional feedforward computation in deep neural networks. In Bengio, Y. and LeCun, Y. (eds.), *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April, 2014, Workshop Track Proceedings*, 2014. URL http://arxiv.org/abs/1312.4461.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009. doi: 10.1109/CVPR.2009.5206848.

Earthsense. A Growing Presence on the Farm: Robots. https://www.nytimes.com/2020/02/13/science/farm-agriculture-robots.html, 2020.

Falanga, D., Kim, S., and Scaramuzza, D. How fast is too fast? the role of perception latency in high-speed sense and avoid. *IEEE Robotics and Automation Letters*, 4(2): 1884–1891, 2019.

Figurnov, M., Ibraimova, A., Vetrov, D., and Kohli, P. Perforatedcnns: Acceleration through elimination of redundant convolutions. *arXiv preprint arXiv:1504.08362*, 2015.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

Gobieski, G., Lucia, B., and Beckmann, N. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 199–213, 2019.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pp. 1398–1406. IEEE Computer Society, 2017. doi: 10.1109/ICCV.2017.155. URL https://doi.org/10.1109/ICCV.2017.155.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5mb model size, 2016.

Joseph, V., Gopalakrishnan, G. L., Muralidharan, S., Garland, M., and Garg, A. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, 2020.

Kim, J.-H., Sharma, G., and Iyengar, S. S. Famper: A fully autonomous mobile robot for pipeline exploration. In *2010 IEEE International Conference on Industrial Technology*. IEEE, 2010.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *International Conference on Learning Representations (ICLR)*, 2017.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April, 2017, Conference Track Proceedings*, 2017.

Neven, D., De Brabandere, B., Georgoulis, S., Proesmans, M., and Van Gool, L. Towards end-to-end lane detection: an instance segmentation approach. In *2018 IEEE intelligent vehicles symposium (IV)*, 2018.

Pedersen, S. M., Fountas, S., Have, H., and Blackmore, B. Agricultural robots—system analysis and economic feasibility. *Precision agriculture*, 7(4), 2006.

Redmon, J. and Farhadi, A. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

Renda, A., Frankle, J., and Carbin, M. Comparing rewinding and fine-tuning in neural network pruning. In *International Conference on Learning Representations*, 2020.

Ruan, X., Liu, Y., Yuan, C., Li, B., Hu, W., Li, Y., and Maybank, S. Edp: An efficient decomposition and pruning scheme for convolutional neural network compression. *IEEE Transactions on Neural Networks and Learning Systems*, 32(10), 2021. doi: 10.1109/TNNLS.2020.3018177.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013.

Salfer-Hobbs, M. and Jensen, M. Acceleration, braking, and steering controller for a polaris gem e2 vehicle. In *2020 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, 2020.

Sharif, H., Zhao, Y., Kotsifakou, M., Kothari, A., Schreiber, B., Wang, E., Sarita, Y., Zhao, N., Joshi, K., Adve, V. S., Misailovic, S., and Adve, S. V. Approxtuner: a compiler and runtime system for adaptive approximations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sivakumar, A. N., Modi, S., Gasparino, M. V., Ellis, C., Velasquez, A. E. B., Chowdhary, G., and Gupta, S. Learned visual navigation for under-canopy agricultural robots. *CoRR*, abs/2107.02792, 2021. URL https://arxiv.org/abs/2107.02792.

Swaminathan, S., Garg, D., Kannan, R., and Andres, F. Sparse low rank factorization for deep neural network compression. *Neurocomputing*, 398, 2020.

Tai, C., Xiao, T., Zhang, Y., Wang, X., et al. Convolutional neural networks with low-rank regularization. *International Conference on Learning Representations (ICLR)*, 2016.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9), 2006.

Tian, Q., Arbel, T., and Clark, J. J. Task dependent deep lda pruning of neural networks. *Computer Vision and Image Understanding*, 203:103154, 2021.

Xu, R., Zhang, C.-l., Wang, P., Lee, J., Mitra, S., Chaterji, S., Li, Y., and Bagchi, S. Approxdet: content and contention-aware approximate object detection for mobiles. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pp. 449–462, 2020.

Xu, R., Kumar, R., Wang, P., Bai, P., Meghanath, G., Chaterji, S., Mitra, S., and Bagchi, S. ApproxNet: Content and Contention-Aware Video Object Classification System for Embedded Clients. *ACM Transactions on Sensor Networks*, pp. 11:1–11:27, 2021.

Zhou, K. Tusimple benchmark ground truth, Oct 2018. URL https://github.com/TuSimple/tusimple-benchmark/issues/3.

Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016. URL http://arxiv.org/abs/1606.06160.

Zhou, S.-C., Wang, Y.-Z., Wen, H., He, Q.-Y., and Zou, Y.-H. Balanced quantization: An effective and efficient approach to quantized neural networks. *Journal of Computer Science and Technology*, 32(4):667–682, 2017.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Conference Track Proceedings*, 2017. URL https://openreview.net/forum?id=S1_pAu9xl.

Zhu, M. and Gupta, S. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *6th International Conference on Learning Representations, ICLR 2018, Workshop Track*, 2018.

A APPENDIX

A.1 NN Approximation Techniques in ApproxCaliper.

As described in §3.3, ApproxCaliper implementation currently supports two approximations: (1) structured pruning based on *Learning Rate Rewinding* (LRR) (Renda et al., 2020) and (2) low-rank factorization (or LR factorization) based on (Tai et al., 2016).

Structured pruning proceeds iteratively, removing a fraction of convolution filters in each iteration. Each iteration removes filters with lowest L1 norms and retrains the model to recover some of the lost accuracy. Each iteration progressively produces smaller and more efficient models with (usually) lower accuracy, creating a tradeoff space of models to choose from.

LR factorization decomposes each (convolution and matrix multiply) weight tensor into 2 lower-rank tensors. For instance, a 2D convolution layer with weights of shape $(C_{\text{out}}, C_{\text{in}}, H_k, W_k)$ can be decomposed into 2 convolution layers with weights of shape $(R, C_{\text{in}}, 1, W_k)$ and $(C_{\text{out}}, R, H_k, 1)$ where the free parameter R controls the rank of the decomposition. A lower R means more approximation, less computation, and less accuracy. Similar to pruning, our strategy is to apply LR factorization iteratively (not proposed in the original work (Tai et al., 2016)), where ranks of weight tensors are reduced in each subsequent iteration.

A.2 Additional Details on Experimental Methodology

Model Architectures and Training. For CropFollow, we evaluate three NN architectures for both heading and distance prediction: ResNet-18 (He et al., 2016) (default), SqueezeNet-v1.1 (Iandola et al., 2016), and DarkNet (Redmon & Farhadi, 2018). We use NNs pretrained on ImageNet (Deng et al., 2009) and fine-tune with 25K corn row images. For Polaris-GEM, we evaluate LaneNet with two backbones: VGG-16 (Simonyan & Zisserman, 2014) (default) and DarkNet (Redmon & Farhadi, 2018). We fine-tune pretrained backbones on the TuSimple dataset (Zhou, 2018) with 3600+ images. We use a 4:1 split between training and validation sets for both datasets.

Error Distributions. ApproxCaliper finds zero-mean Gaussian $\mathcal{N}(0, \sigma^2)$ as the closest error distribution for the NNs in CropFollow (using the `ac.find_err_dist` routine – §3.4). The mean is `0` since the errors are mostly zero-centered; the variance σ is a parametric value that is varied in error injection. In Polaris-GEM, ApproxCaliper finds Bernoulli distribution $\mathcal{B}(p)$ for the Boolean mask and zero-mean Gaussian distribution $\mathcal{N}(0, \sigma^2)$ for the embedding tensor (see §2.3 for these outputs).

Tuning Space and Tuning Time Budgets. For each of the two NN components in CropFollow, there are 63 candidates – 3 (NN architectures) $\times$ (20 (prune levels) + 1 (baseline)), creating a total of $63 \times 63 = 3969$ combinations. For Polaris-

GEM, the autotuner searches over 26 model candidates ($2 \times (12 + 1)$) for LaneNet and 105 FPS values (chosen between 0 to 10.5 at 0.1 intervals), creating 2730 possible configurations. The LaneNet FPS is limited by the clustering algorithm (see §2.3) which runs at a max of 10.5 FPS. Like CropFollow, each evaluation is expensive since it involves a 506-meter long simulator navigation run that consumes 5-6 minutes, and we maintain an ∼5 hour tuning time budget for 50 runs.

Configurations searched in unvisited and valid regions is set to 75% and 25%, respectively. For the valid region, 5% of the area closest to the boundary (computed using Euclidean distance) is included in the search.

A.3 ApproxCaliper with Low-rank Factorization

Our results so far used structured pruning since it provided a strictly better tradeoff (better accuracy and performance) compared to LR factorization. A key contribution of ApproxCaliper, however, is that it can make NN optimizations more effective by allowing them to be used more aggressively, even with reduced accuracy, by ensuring that the overall application use case meets its goals. Here we evaluate the LR factorization technique to show that these benefits of ApproxCaliper generalize to other kinds of approximations. We generate 12 factorization levels of the LaneNet-VGG16 NN used in Polaris-GEM and evaluate the end-to-end QoS (lane departure situations) for each of the optimized model variants. For simplicity, we do not vary FPS and configure the FPS to its maximum of 10.5. Figure 12 shows the different factorization levels laid out on an accuracy vs. performance tradeoff space (figure details in caption).

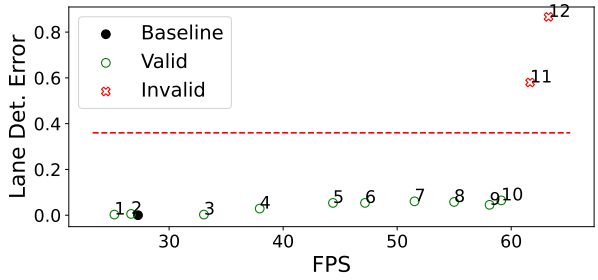

Figure 12: The compute performance (FPS) vs. lane detection error of low-rank factorized models. Green points have valid QoS, and red points have invalid QoS. The red dotted line shows error threshold (36.4%) above which configurations are invalid.

The highest factorization level with acceptable QoS is level 10, which provides an FPS of 59.1, 2.2× higher than factorization level 2, which is the most efficient model with error lower (or equal) to the baseline. The key takeaway is that ApproxCaliper enables both LR factorization and LRR to deliver much higher benefits by considering the error slack in the application; we expect these benefits to extend to other accuracy-relaxing optimizations.

A.4 Important ApproxCaliper APIs

`ac.PerfMetric` specifies a performance metric to use for error calibration (not shown in the example). These metrics are useful when the goal is to model the relationship between NN error and NN performance (e.g., latency at which NN operates). ApproxCaliper supports latency and throughput as performance metrics.

`ac.find_error_model` automatically finds an appropriate error distribution to use for error injection. The algorithm is explained in §3.2.

`ac.error_calibrate` invokes the error calibration phase (§3.2). It takes a QoS evaluator as input, such as `CropFollowQoSEvaluator` in the example that runs the autonomous navigation for a configured time/distance and returns the number of collisions. This routine also takes the QoS target as input (0 collisions in the example). The output is the N-dimensional error constraint space for the N given metrics.

`ac.optimize` uses approximation techniques to optimize the NNs within the constraint space returned by `ac.error_calibrate`. The objective function is configurable: the example uses `ac.objective.latency`; we also support utilization and throughput. The `ac.optimize` interface supports multiple NN architectures/models for each NN component as input, and returns the best (NN architecture, model variant) pair, as explained in §3.3.