ApproxCaliper: Exploiting Application-level Error Resiliency for Optimizing Neural Networks

HASHIM SHARIF*, YIFAN ZHAO*, PETER PAO-HUANG, VATSIN NINAD SHAH, ARUN NARENTHIRAN, MATEUS VALVERDE GASPARINO, and NATHAN ZHAO, University of Illinois at Urbana-Champaign, USA
ABDULRAHMAN MAHMOUD, Harvard University, USA
SARITA ADVE, GIRISH CHOWDHARY, SASA MISAILOVIC, and VIKRAM ADVE, University of Illinois at Urbana-Champaign, USA

ACM Reference Format:
Hashim Sharif, Yifan Zhao, Peter Pao-Huang, Vatsin Ninad Shah, Arun Narenthiran, Mateus Valverde Gasparino, Nathan Zhao, Abdulrahman Mahmoud, Sarita Adve, Girish Chowdhary, Sasa Misailovic, and Vikram Adve. 2021. ApproxCaliper: Exploiting Application-level Error Resiliency for Optimizing Neural Networks. 1, 1 (December 2021), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A wide range of application domains use deep learning models to extract actionable information from sensory data (such as visual, audio, and GPS). For instance, mobile robots are increasingly reliant on visual information and often use neural network (NN) models to process the visual data for perception and control [4, 6, 20, 21, 47, 58, 63]. Today, deep learning models are being deployed across a wide spectrum of hardware devices, ranging from large data center environments with high computational capacities to small IoT and edge compute devices with limited compute capabilities. However, a major challenge is that deep learning models are usually very computationally demanding, which makes it difficult to deploy them on resource-constrained edge systems.

Software optimizations can help bridge the gap between the high computational demands of deep learning workloads and the computational limitations of edge devices. Existing systems for automated NN optimization [9, 22, 29, 51, 53, 60, 65, 66] employ a combination of pruning, quantization, weight compression, and algorithmic approximations techniques, and have shown to be quite effective in reducing model size and improving performance. *Neural network weight pruning* is an especially effective optimization that compresses NN models significantly by removing the less important weights - ones with a relatively low contribution to correctness of final results. While many different techniques for network pruning have recently emerged [15, 18, 22, 23, 30, 32, 36, 37, 40, 41, 64], they generally take the conservative approach of constraining the pruning levels to retain the accuracy of the unpruned networks. In particular, they do not take advantage of *application-level error resilience*: a prediction accuracy reduction may be acceptable to trade for improved compute performance, while avoiding any observable impact on the end-to-end quality of the application. When the error resilience allows the NN accuracy to be relaxed, models can be more aggressively

*Both authors contributed equally to this paper.

Authors' addresses: Hashim Sharif, hsharif3@illinois.edu; Yifan Zhao, yifanz16@illinois.edu; Peter Pao-Huang, ytp2@illinois.edu; Vatsin Ninad Shah, vns2@illinois.edu; Arun Narenthiran, av7@illinois.edu; Mateus Valverde Gasparino, mvalve2@illinois.edu; Nathan Zhao, nz11@illinois.edu, University of Illinois at Urbana-Champaign, Champaign, IL, USA; Abdulrahman Mahmoud, mahmoud@g.harvard.edu, Harvard University, Cambridge, MA, USA; Sarita Adve, sadve@illinois.edu; Girish Chowdhary, girishc@illinois.edu; Sasa Misailovic, misailo@illinois.edu; Vikram Adve, vadve@illinois.edu, University of Illinois at Urbana-Champaign, Champaign, IL, USA.

2021. XXXX-XXXX/2021/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

pruned. For instance, in an autonomously navigating robot that uses NNs for visual perception, the NN accuracy can be relaxed to the extent that the navigation control is not adversely affected.

The existing systems for pruning (and other optimizations) suffer from two major limitations: 1) they optimize models in isolation and do not consider the specific context of the application-pipeline which may have some level of error tolerance, and hence miss out on opportunities for higher computational speedups; and 2) they lack mechanisms to quantify the minimal NN accuracy required to provide the necessary level of end-to-end quality of service (QoS). Moreover, merely picking a design point from the trade-off space of pruned neural network models does not capture how neural networks behave in the end-to-end application pipeline. For instance, it does not capture how simultaneous errors in two different neural network components affect application-level QoS, or how an interplay of performance and accuracy may affect the end-to-end QoS. We hypothesize that realizing NN optimizations in an application-specific manner has the potential to deliver significantly higher performance improvements while still satisfying the application's end goals.

We develop ***ApproxCaliper***, *the first framework that automatically evaluates the potential for neural network model optimization in an end-to-end application context. ApproxCaliper* incorporates a *calibration phase* and a *tuning phase*. The calibration phase quantifies the minimum NN accuracy required to satisfy application-level quality targets specified by developers. It uses statistical error injection in the outputs of the NN sub-component(s) to measure how the end-to-end application quality is affected by varying levels of error in these outputs. The tuning phase then uses these error constraints and optimizes the NN components with approximation techniques that relax some level of accuracy for gains in performance. It considers a tradeoff space of NN models each with different accuracy and performance and selects a model (or a combination of models) that maximizes the overall application performance, while satisfying the learnt error constraints. To generate the tradeoff space of models, our *ApproxCaliper* implementation uses structured weight pruning (a technique that drops low-weight filters), which takes as input a model architecture and iteratively generates progressively smaller pruned networks. *ApproxCaliper* is extensible to other kinds of neural network optimizations (quantization, low-rank factorization, weight compression).

ApproxCaliper is a highly configurable and flexible evaluation system that allows application developers to specify: a) an end-to-end application quality metric, b) neural network architectures to prune, with associated datasets for training, and c) a list of N component-level metrics (error or performance metrics) to consider in the calibration phase. Using these inputs, 1) the *calibration phase* generates analysis results that capture how the end-to-end application quality is impacted by changes in NN accuracy (captured by component-level metrics), and 2) the *tuning phase* uses autotuning over the configuration space of pruned models to discover model configurations that maximize overall application performance with acceptable end-to-end application quality.

The analysis and optimization capabilities of *ApproxCaliper* enable us to study the potential for NN model optimization in the context of two real-world robotics application stacks that use convolutional neural networks (CNNs) for visual perception. The results and findings of our study are summarized below.

1.1 Our Study and Results

We study multiple configurations of two cyberphysical applications that use NNs. The first application, *CropFollow*, is an autonomous software stack on a production agriculture robot, *TerraSentia*, developed by EarthSense [14]. Terrasentia is used for a variety of agriculture tasks such as high-throughput phenotyping and cover crop planting. The robots use a vision-based autonomous navigation system called CropFollow [56], which uses two CNNs, one to predict robot heading angle and another to predict robot's distance from the center of the crop row.

The second application, *Polaris-GEM*, is a Gazebo simulation and lane-following system of the commercially available Polaris GEM e2 electric cart [1]. The simulated cart is configured for lane-following without support for obstacle evasion; the cart navigates in the center of the lane (including curvy pavements) that is demarcated with lane markings. This application stack uses a LaneNet [43] network that detects and returns positions of lane markings.

We evaluate three different NN architectures for CropFollow (ResNet-18 [26], SqueezeNet [27], and DarkNet [48]) and two architectures for Polaris-GEM (LaneNet [43] with 2 different backbones, VGG-16 [55] and DarkNet). Our evaluation using *ApproxCaliper* reveals these findings:

- Across both applications, *ApproxCaliper* discovers significant room for relaxing NN accuracy while still achieving acceptable QoS for the end-to-end task. *ApproxCaliper* finds configurations for CropFollow where the heading regression error can be increased by 2.9× and distance error can be increased by 2.0× without affecting the quality of navigation control (measured by robot collisions). Similarly for Polaris-GEM, *ApproxCaliper* finds that the lane detection accuracy can be relaxed by 19%, while maintaining the correct lane-keeping behaviour (no lane exits).
- Using Gaussian noise as the (injected) error model, the error calibration module in *ApproxCaliper* discovers several insights into the noise tolerance of the feedback control system. We find that a) high bias (i.e., non-zero mean) errors in heading and distance prediction make robot navigation susceptible to collisions, b) an error of high variance can be tolerated if the errors are zero-centered with low bias in the distribution, c) adding errors to only one of heading or distance prediction allows the relatively more accurate model to make the corrections necessary to avoid collisions, while errors in both predictions make the navigation considerably less stable. The latter finding shows the value of simultaneously studying multiple components in the context of the end-to-end application.
- The tuning module in *ApproxCaliper* leverages the error resilience of the CropFollow system to replace the heading and distance prediction models with pruned models that provide lower prediction accuracy but are significantly more compute efficient. Using the autotuning framework in *ApproxCaliper*, we find the most compute-efficient configuration provides a 36× model size reduction for the prediction models and a 5.3× *overall* speedup for the perception module.
- The error resilience of the Polaris-GEM navigation control is leveraged by the autotuning module in *ApproxCaliper* to prune the LaneNet network by up to 4.0× fewer parameters while achieving a performance improvement of 2.2×, without introducing lane departure situations.
- Across both applications, we find that different NN architectures and their pruned variants sometimes yield surprising and unintuitive tradeoff choices. For instance, pruning a larger ResNet-18 model in CropFollow produced models that were smaller, more performant, and yet more accurate than a baseline SqueezeNet and DarkNet model, which are both more efficient than ResNet-18 in the baseline configuration. We observe a similar trend for the LaneNet-VGG16 vs LaneNet-DarkNet models. These cases show the value of *ApproxCaliper* in automatically discovering non-trivial configurations of pruned NN models that maximize overall performance improvements.

Significance of Chosen Applications: Terrasentia is representative of an emerging and important class of small agriculture robots that are performing tasks not possible with large tractors; these include scouting for disease and pests, herbicide-free mechanical weeding, and carbon sequestration through in-season cover-crop planting. Lane-following applications are also an important class of systems since these are being deployed in visually-guided outdoor and indoor navigation systems used in various utility domains, e.g., delivery, mining, healthcare, and home maintenance [8, 19, 50, 57].

Generalizability: *ApproxCaliper* can be applied to any application pipeline that includes feed-forward neural network components and where end-to-end task quality metrics can be measured.

Application developers can use *ApproxCaliper* to better understand the potential for using approximations to trade off component-level accuracy without compromising on end-to-end task outcomes (especially for applications in non-adversarial environments). *ApproxCaliper* implementation uses structured pruning but is equally applicable to other optimization techniques. With *ApproxCaliper*, developers can also make informed choices of application- and system-level design decisions and favor choices that facilitate higher levels of approximation.

1.2 Contributions

Overall, this paper makes the following contributions:

- We present the first framework that assists developers in *automatically evaluating the potential for neural network approximations* in the context of an end-to-end application.
- We propose *ApproxCaliper*, the first system that combines automated model selection and model optimization to discover model configurations that maximize performance benefits, while satisfying application-level quality of service (QoS) constraints.
- We present a novel approach that uses the error constraints discovered in the error calibration phase to guide the automated model optimization phase to more intelligently navigate the search space of configurations. Our findings show that this approach achieves higher performance improvements compared to an empirical autotuning approach that doesn't leverage knowledge about error constraints.
- We show that the *ApproxCaliper* can be used to gather important insights on the error resilience to NN mispredictions in end-to-end applications, by using it to analyze two such applications: 1) a commercial autonomous agricultural robot navigation system using direct field experiments, and 2) a lane-following cart using simulations. Further, we show that *ApproxCaliper* can use the accuracy-performance tradeoff space of pruned models to select configurations of pruned models that provide very significant performance improvements and model footprint reductions.

2 Preliminaries and Terminology

This section introduces terminology we use throughout the paper.

Application-level QoS. An application-level quality of service metric (QoS) is an application-specific and/or domain-specific metric that quantifies how well the application delivers on its desired goals. For instance, in the context of an autonomously navigating robot, a relevant QoS metric is the length (or time) the robot travels before requiring human intervention (e.g., in the case of a collision). For purposes of error calibration and tuning, *ApproxCaliper* allows specifying a custom QoS computation function as well as QoS constraints that are used to distinguish *valid configurations* and *invalid configurations*.

Configuration. We refer to *configuration* as an assignment of a *model* for each NN component in the application. A *model* is a tuple `(architecture, pruneLevel)`, where *architecture* is a NN architecture type (e.g., ResNet) and *pruneLevel* is a pruned variant of the trained model; a higher prune level indicates a higher number of parameters removed from the model.

Component-level Error Metrics. We make a distinction between end-to-end application-level QoS and component-level accuracy. The accuracy of individual components affects the end-to-end QoS but this relationship is usually not straightforward to derive or analytically estimate. *ApproxCaliper* uses artificially injected errors at the level of component outputs to measure how component-level accuracy degradation impacts the application-level QoS. *ApproxCaliper* treats different NN instances in the application as separate components.

N-Dimensional Error Constraint Space. *ApproxCaliper* allows developers to specify a list of N component-level accuracy metrics. Developers can specify one or more accuracy metrics per

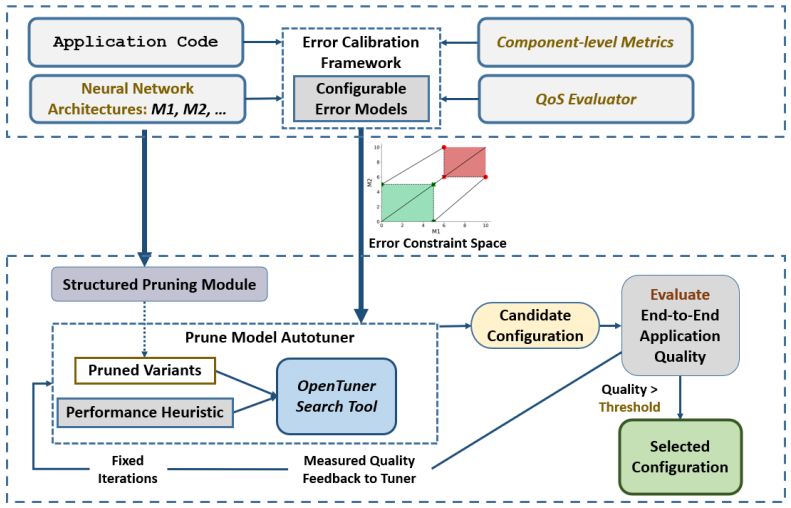

Fig. 1. *ApproxCaliper* System Workflow

component. The output of the *calibration phase* is an N-dimensional error constraint space that captures how an error increase in terms of each individual accuracy metric affects the end-to-end QoS. In case different metrics correspond to different components, the N-dimensional space captures how simultaneously adding errors to multiple components affects the end-to-end QoS. Within this space, *ApproxCaliper* learns the boundary that separates valid and invalid configurations.

Autotuning is an optimization technique that uses design space exploration techniques to find application and/or system-level parameters that maximize an objective function while satisfying some constraints. Since autotuning is a heuristic search performed over intractably large search spaces, it discovers locally optimal configurations and is not guaranteed to find globally optimum.

3 ApproxCaliper

ApproxCaliper splits application-specific NN optimization into two phases: (1) a calibration phase that quantifies application-level error tolerance, followed by (2) an optimization phase that optimizes the NNs leveraging the application's error tolerance discovered by the calibration phase. Figure 1 shows the system workflow for *ApproxCaliper*.

3.1 Error Calibration Framework

The error calibration phase uses a highly configurable framework based on statistical error injection to quantify the error tolerance of the application. This phase captures how the end-to-end application QoS is affected by accuracy degradation of the NNs components in the application pipeline. NN components often dominate the computational costs of edge applications, but there are usually also other application components which interact with the NN inputs and/or outputs in arbitrary ways.

The error calibration framework assumes an application with $K \geq 1$ distinct NNs. The framework takes several parameters as inputs:

Error Model: $f_i(\mathcal{P}_i), 1 \leq i \leq K$, where $\mathcal{P}_i$ is the set of possible parameter values for the i^{th} probability distribution, f_i. In particular, the error model should be parameterized so that it can inject different amounts of error to degrade the NN accuracy by varying levels. The error values that must be injected into the network are computed as described in Section 3.1.1. Alternatively, instead of using predefined error distributions (such as Uniform, Bernoulli,

Binomial), application developers may implement a custom error injection mechanism for some of the NNs, provided to the framework as a function. The framework defaults to using a Gaussian distribution for floating-point values.

Component-level Metrics: $M_j, 1 \leq j \leq N$, which define a bounded N-dimensional error calibration search space. Each NN may have one or more such metrics. A metric may be an error metric (e.g., bias and variance for a regression network estimating heading in the CropFollow pipeline), or a performance metric (e.g., frames-per-second, or FPS). The key assumption of *ApproxCaliper*'s error calibration framework is that *all the metrics must be monotonic with the same directionality*, i.e., a higher metric value must always be more desirable (or always less desirable) than a lower value.

End-to-end QoS evaluator: This can be an automated function, or an interactive interface where users observe the outcome to determine the QoS. The latter is relevant in the context of cyber-physical systems where quality is externally observed, e.g., robot collisions in a crop field.

The goal of the calibration framework is to use these inputs to partition the N-dimensional error constraint space into three disjoint regions, representing valid (QoS above threshold), invalid (QoS below threshold), and undetermined (unknown QoS) regions, using the Error Calibration Algorithm, which we describe next.

Error Calibration Algorithm. Algorithm 1 shows our error calibration algorithm. The core idea of the algorithm is to recursively divide the N-dimensional error space into smaller sub-regions, and traverse each sub-region to find a series of boundary points that mark the boundary.

Algorithm 1 lists and explains the inputs to the `errorCalibrator` function. The target application `app` should be equipped with the best NN variants for each of the NNs, i.e., the ones with highest accuracy among developer-specified network architecture variants, since they will give the highest margin for error injection. The lower and upper bounds, $M_{lb,j}$ and $M_{ub,j}$, for the N metrics, M_j, collectively define two points in the N-dimensional space, M_{lb} and M_{ub} (denoted `lowerBound` and `upperBound` in the Algorithm). Without loss of generality, we assume that the axes of metrics are oriented such that higher metric values are more desirable (e.g., greater error tolerance or higher compute performance).

The algorithm first creates a single N-dimensional rectangular region between M_{lb} and M_{ub} (Line 11). Regions are held in `regionQueue` (Line 12), a priority queue that returns the largest region first. The outer loop retrieves the next region from the queue and traverses the diagonal between the lowest point and highest point of the region (Line 16, 17). The inner loop bisects this diagonal for `maxRegionEvals` times using binary search (Line 18). At each step, the midpoint of the diagonal (`diag.midPoint`) is evaluated, reducing the length of the diagonal by half. It is maintained that the lower boundary of the diagonal is always a valid point, and the higher boundary is always invalid.

To evaluate a point, `getErrorToAdd` (Line 21) computes the amount of error that must be injected such that the N metrics attain the value represented by this point (for example, error bias being equal to 0.1 and error standard deviation 0.05 simultaneously). We discuss this procedure in Section 3.1.1. Then `qosEvaluator` runs the application with this amount of error and returns the observed application-level QoS (Line 22).

After the binary search, the endpoints of the diagonal are added to the set of boundary points, and the algorithm creates new sub-regions to evaluate (Line 28, 29). The algorithm terminates and returns when the queue of regions is empty or total number of QoS evaluation reaches `totalEvals`.

Figure 2 shows the error injection algorithm in a 2-dimensional space and, in particular, how the algorithm creates sub-regions. The user specifies 2 error metrics $M1$ and $M2$, the lower bound of search $(0, 0)$, and upper bound $(10, 10)$. In the first iteration of the outer loop, the first region is the whole space between $(0, 0)$ and upper bound $(10, 10)$, and the diagonal has been reduced to

Algorithm 1: Error Calibration Algorithm.

Inputs:
- app: target application with NNs – errors are injected into these NNs
- errorModel: model for injecting error
- qosEvaluator: function that evaluates app-level QoS via error injection
- qosTarget: user-specified QoS goal
- lowerBound, upperBound: lower/upper threshold for error injection (N-tuple with one entry per metric)
- maxRegionEvals: maximum evaluations per region
- totalEvals: total number of evaluations (across regions)

Output: boundaries: List of tuples with lower and upper bound of valid region boundary

Function *errorCalibrator(*app, errorModel, qosEvaluator, qosTarget, lowerBound, upperBound, maxRegionEvals, totalEvals*)*

```
firstRegion = Region(lowerBound, upperBound);
regionQueue = PriorityQueue([firstRegion ]);
boundaries = Set();
nEvals = 0;
while not regionQueue.empty() do
    region = regionQueue.pop() ;
    diag = Diagonal(region.lowerBound, region.upperBound);
    for rEvals = 0 to maxRegionEvals do
        if nEvals ≥ totalEvals then
            return boundaries;
        errorParams = getErrorToAdd(app, errorModel, diag.midPoint);
        qos = qosEvaluator(app, errorModel, errorParams);
        if qos ≥ qosTarget then
            diag = makeDiagonal(diag.midPoint, diag.upperBound);
        else if qos < qosTarget then
            diag = makeDiagonal(diag.lowerBound, diag.midPoint);
        nEvals += 1;
    boundaries ∪= tuple(diag.lowerBound, diag.upperBound) ;
    regionQueue.push(region.getSubregions(diag));
return boundaries;
```

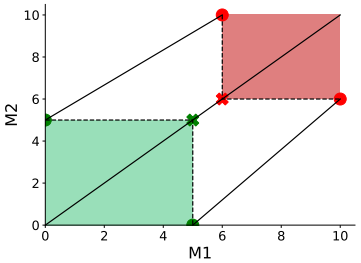

Fig. 2. How the algorithm creates sub-regions after the first diagonal has been traversed in a 2-dimensional search. Black lines represents diagonals of regions, and green and red regions represent valid and invalid regions found by the first diagonal.

```
Configuration: {
  N1: ResNet-prune-level-18,
  N2: SqueezeNet-prune-level-8,
  N3: DarkNet-prune-level-5
}
```

Fig. 3. Example of a configuration for three NN components in a model tuning problem.

between $(5, 5)$ (green cross) and $(6, 6)$ (red cross). Therefore, the region between $(0, 0)$ and $(5, 5)$ is a valid region, and between $(6, 6)$ and $(10, 10)$ is an invalid region. It follows that $(0, 5)$ is a valid point and $(6, 10)$ is an invalid point (marked by green and red dots respectively), and a diagonal (marked by black line) connecting them covers the largest white (to-be-searched) region. Similarly, diagonal between $(5, 0)$ and $(10, 6)$ covers the other half of the white region. Therefore, these 2 regions are selected as next regions to search.

3.1.1 **Error Injection Parameters.** An important step in Algorithm 1 is to inject a specific amount of error such that the output of the NN evaluates to a given metric value. This is an unconstrained optimization problem which we formulate as follows:

$$\min_{P \in \mathcal{P}} ||x_P - x_0|| \tag{1}$$

$$\text{where } x_P = eval(\text{networks}, \text{errorModel}, P); \quad x_P, x_0 \in \mathbb{R}^n$$

where $\mathcal{P}$ is the set of values for parameters of the parameterized error model (which are being searched over), x_0 is the target metric value that Algorithm 1 requests, `networks` and `errorModel` are the neural networks and error model in the error calibration algorithm, and *eval* evaluates the network with amount of error injection defined by P and returns the values of n metrics.

Developers can specify closed-form formula of P to *ApproxCaliper* if such a solution exists. For instance, if error is injected to the output of a regression model that predicts a single scalar value,

there may exist a mathematical function from the value of the regression accuracy metric to the parameters of a distribution (e.g., Gaussian distribution). In cases where such a formula does not exist, *ApproxCaliper* applies existing search-based algorithm to find a solution to this unconstrained optimization problem (described in more detail in §5).

3.2 Model Tuning and Optimization

Approximation: Structured Pruning. The recent literature includes many generic and domain-specific approximation techniques for optimizing NN computations with (potentially) small loss in inference accuracy. Examples include weight pruning (dropping/skipping weights or filters) [22, 31, 33, 49], perforated convolutions (skipping and interpolating some output computations) [17, 53], integer quantization (INT8, INT4) [22], and others. While these approaches do not give theoretical guarantees on accuracy, they have empirically shown promising benefits for NNs examined in isolation.

Algorithm 2: Structured Pruning Algorithm

```
1  Inputs:
2    • model: target model
3    • trainData: training data
4    • numEpochs: epochs to use for training in each iteration
5    • pruneRate: percentage of filters to remove in each iteration
6    • K: number of pruning iterations
7  Output: prunedModels: List of pruned models
8  Function PruneModel(model, trainData, numEpochs, pruneRate,
     K)
9    Set prunedModels;
10   for i = 1 to K do
11     foreach convLayer ∈ convLayers(model) do
12       Set filterList;
13       foreach convFilter ∈ filters(convLayer) do
14         filterLNorm = computeLNorm(convFilter);
15         filterList ∪= (convFilter, filterLNorm);
16       minWeightFilters = getLowestFilters(filterList, pruneRate);
17       restFilters = filterList − minWeightFilters;
18       newConvLayer = setLayerFilters(convLayer, restFilters);
19       model = updateModelLayer(model, newConvLayer);
20     model = trainModel(model, trainData, numEpochs);
21     prunedModels ∪= model;
22   return prunedModels;
```

The *ApproxCaliper* tuning phase incorporates an optimization module that performs structured weight pruning for NNs. We adopted weight pruning since it has shown to be effective in reducing computational cost greatly, while preserving most of the model accuracy through repeated retraining. Weight pruning can be categorized into two types: 1) *unstructured pruning* [22, 31], which removes individual weights that are deemed less important to the overall computational accuracy (e.g, low-magnitude values), and 2) *structured pruning* [33, 35], which removes groups of contiguous weights, for instance, entire filters and channels from convolution layer weights. Unstructured pruning provides much higher reduction in model sizes than structured pruning (e.g., 13× [22] vs. 4.5× [35]), but results in unpredictable sparsity that is much less efficient for highly parallel architectures such as GPUs and CPU vector units (Cortex-A72 vector units in Pi4 [12]) not well suited to irregular computational patterns.

Our implementation is based on the the iterative structured pruning algorithm proposed by Renda et al. [49] and L-norm based filter pruning approach proposed by by Li et al. [33]. The structured pruning workflow is shown in Algorithm 2. The `PruneModel` routine takes as input a trained model, training data (for retraining), number of epochs to retrain, prune rate (fixed fraction of filters to prune in one step), and the total number of pruning iterations (`K`). Each pruning iteration (outer most loop) 1) removes a `pruneRate` fraction of filters with lowest L1-norm from each convolution layer, and 2) retrains the model for `numEpochs` epochs to recover some accuracy. Each subsequent iteration further removes `pruneFraction` filters from the pruned model of the previous iteration.

ApproxCaliper is extensible to other types of approximations such as unstructured pruning, weight quantization, and low-rank factorization. In the *ApproxCaliper* workflow (as shown by

Figure 1), new approximation methods/techniques can be added as independent modules that take as input an NN and generate an optimized variant(s).

Search Space and Configurations of Model Variants. *ApproxCaliper* takes a list of neural network architectures written in Pytorch [44] along with training data for each one, and generates a list of pruned model variants using structured pruning. The search space depends on three parameters:

- *Neural network components.* Each NN component, $C_i, 1 \leq i \leq N$, in the application is a candidate for model optimization using *ApproxCaliper*. For instance, autonomous navigation for our CropFollow robot uses two models, for predicting heading and distance from center.
- *Model architectures.* Different model architectures (e.g., ResNet, DarkNet, SqueezeNet) provide different accuracy-performance tradeoffs. Generally, smaller model architectures with fewer layers and fewer parameters are more compute-efficient but also have lower accuracy than larger ones. *ApproxCaliper* makes model architecture selection an explicit choice for the autotuner, by selecting a particular model architecture and a particular pruned model variant for each component C_i. Different neural architectures may be suitable for different components and can be specified on a per-component level in our framework. We denote specific model architectures for component C_i as A_{ij}.
- *Prune model variants.* For each A_{ij}, structured pruning generates model variants which also become an explicit optimization choice for the autotuner. Number of variants generated (K) is configurable in Algorithm 2 and can be different for each M_{ij}. A particular prune model variant of a model architecture M_{ij} is referred to as V_{ijk}.

Therefore, the total search space is $\{V_{ijk} \mid i, j, k\}$. This can be a ragged space where each NN component has different architectures, and each architectures have different prune levels. An example configuration for three neural network components N_1, N_2, and N_3 is shown in Figure 3; the components are mapped to a combination of model architecture and corresponding prune level.

Autotuning Framework. The search space of possible configurations can grow large enough that an exhaustive approach is not tractable. To enable efficient search, we use OpenTuner [2], an extensible library for building custom autotuning frameworks.

Performance Heuristic. The goal of the autotuner is to find combinations of pruned model variants that minimize the compute cost. Since compute-performance (e.g., frames processed per second) varies across hardware architectures, we use a hardware-independent heuristic to predict the performance of a configuration. The heuristic guides the autotuner towards configurations that minimize overall cost, and is also used for ranking a set of valid configurations (ones that satisfy the end-to-end quality constraints). The total cost of a configuration T_{Total} is computed as a sum of the cost of each selected model (architecture and prune model variant).

$$T_{Total}(Config) = \sum_{i=0}^{N} Cost(Config[i])$$

The cost of each model architecture, A, is computed as a sum of the operation counts (returned by function N_c) across all neural network layers (L is the number of layers), as shown below:

$$Cost(M) = \sum_{i=0}^{L} N_c(M[i])$$

$Cost(M)$ returns a lower cost value for higher prune levels that have a larger fraction of convolution filters pruned compared to a less pruned model (of the same architecture).

As an alternative to the performance heuristic, *ApproxCaliper* can also be configured to use the real performance measured on device as the objective function for tuning. The analytical heuristic is more efficient since it does not require empirical performance measurement on the target machine.

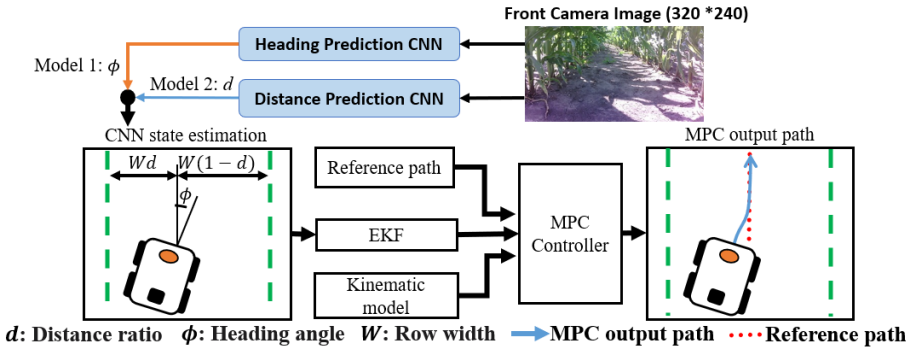

Fig. 4. Workflow for the evaluated CropFollow [56] Navigation System. The front camera images feed into CNNs for distance and heading predictions which are fused with IMU measurements using an Extended Kalman Filter. The fused state estimations are used by MPC for computing angular velocity commands.

Application-level QoS as Feedback to Tuner. The application-level QoS is an important feedback signal to the tuner; a QoS value closer to or higher than the QoS target directs the tuning search (performed by OpenTuner [2]) towards similar configurations, and a low QoS value informs the autotuning search to navigate in a different space.

Using Calibration Analysis to Guide Tuning. We support two model tuning modes:

- ***Unguided tuning*** uses the same workflow shown thus far and does not consider the calibration phase analysis.
- ***Guided tuning*** uses the N-dimensional error constraint space learnt in the calibration phase to better guide the tuner towards higher performance configurations. The error constraint space gives useful guidance since it separates error configurations that succeed (sufficient QoS), or fail (insufficient QoS), or are unvisited.

Our guided tuner is based on the idea that tuning should be directed towards higher error configurations that still provide acceptable application-level QoS. This is because permitting a higher error usually allows a higher level of optimization/pruning. The guided tuner directs the search towards higher error configurations that are either in the *unvisited regions* on the error constraint space, or on the edge of the *valid configurations region*. The unvisited region lies between the valid and invalid configuration regions, therefore includes higher error configurations compared to the valid configurations region. The fraction of configurations to visit in each region is a configurable parameter.

4 Applications

We use *ApproxCaliper* to study two end-to-end systems with 5 neural networks:

- **CropFollow:** A production agriculture robot called Terrasentia, obtained from EarthSense [14], used for high-throughput phenotyping and a variety of other agriculture tasks. This commercial robot is equipped with an autonomous vision-guided navigation system named *CropFollow*, used for navigation through fields of row crops.
- **Polaris-GEM:** A Gazebo-based simulation of Polaris GEM e2 [52] with autonomous lane-following capabilities. The vehicle model contains a simulated forward-facing camera to capture Gazebo-rendered road surface for lane detection and following.

4.1 CropFollow

Hardware Setup. For vision-based navigation, CropFollow uses a forward-facing camera with 720p resolution at 30 FPS (frames per second) for row following using the algorithm in [56]. An

embedded 6 degrees-of-freedom (DoF) Inertial Measurement Unit (IMU) gathers angular velocity and acceleration measurements. The commercial TerraSentia configuration uses an Intel NUC 10i7FNH as the primary computer, and a Raspberry Pi3 for the lower-level control logic. For a compute device that is more representative of edge systems, we replace the expensive and high-power Intel NUC with a significantly lower-cost and lower-power Jetson Nano, with a quad-core ARM A57 and a 128-core Nvidia Maxwell GPU. The Nano costs $99 MSRP (vs. $876 for NUC) and uses 10W of power (vs 90W for NUC).

Autonomous Navigation Stack. Figure 4 shows the workflow. The main components are:

- *Perception Module.* The module [56] contains 2 ResNet-18 NNs, which take as input 320×240 RGB images (resized from 720P) and estimates the robot heading θ and relative distance d respectively. θ is the angle between the direction the robot faces and the direction of the row. d is the ratio between the distance from the robot to left crop row and the total row width, i.e., $d = dL/(dL+dR)$; d is a value between 0 and 1.
- *IMU Fusion with Extended Kalman Filter.* An Extended Kalman Filter (EKF) fuses NN predictions with inertial measurements from IMU. It reduces the chance of abrupt control variations due to vision or IMU noise. EKF takes as input 1) NN predictions θ and d, 2) robot's linear speed and angular speed from the IMU, and 3) the robot state (heading and distance) at the previous time step s_{k-1}, to compute the current state s_k.
- *Model Predictive Controller.* A Model Predictive Controller (MPC) computes angular velocity commands to keep the robot on its intended trajectory using the estimates of (θ, d) from EKF. Specifically, MPC solves a non-linear constrained optimization problem with the maximum curvature radius that the robot can turn (at a particular time step) as constraints and the objective is to find angular velocities over a time horizon (next 20 time steps) to reduce the distance between the predicted path and the *reference path*. Since we evaluate "row following" through crop rows, the reference path is a line through the center of the crop lane.

4.2 Polaris-GEM Vehicle Simulator

Simulated Hardware. The Polaris-GEM simulator models Polaris GEM e2 [52], a commercially-sold electric cart [1]. It consists of a Gazebo environment with realistic scenes and a Stanley controller for path following. A simulated forward-facing camera on the vehicle captures images with 720p resolution at 30 FPS, and resized to 512×256 in post-processing.

Lane-following Software Stack. For vision-based lane following, we combine this simulator with LaneNet [43], a state-of-the-art lane detection network. The main components are:

- *Lane Detection Network.* Polaris-GEM uses LaneNet with VGG-16 backbone for lane detection. LaneNet takes an image of size 512×256 (resized from 720p) and generates 2 masks to detect lane markings: a boolean mask (512×256) marking all lane-pixels, and an *embedding* tensor ($4 \times 512 \times 256$) that helps clustering each *instance* of lane marking. Using a clustering algorithm, LaneNet finds each lane as a cluster of pixels, fits a polynomial curve through the pixels of each lane, and returns these curves as the lane detection result.
- *Postprocessing and Stanley controller.* Stanley controller [59] (similarly to MPC above) uses estimations of current vehicle pose to compute a steering angle to guide the vehicle back to the lane center. To convert LaneNet output to Stanley input, we implement a postprocessing algorithm that (1) identifies the left and right lane markings of the lane the vehicle is currently in, (2) fits a curve through pixels of these two lane markings, and (3) computes θ and d from these two curves using a geometric transform algorithm described in [56]. Given θ and d, Stanley controller issues a steering angle which is: $\delta = -\theta + \arctan\left(k \cdot (0.5 - d) \cdot w \,/\, v\right)$. The first term eliminates

the heading angle θ, and the second term returns d to the lane center (0.5) over time. w and v denotes the width of lane and speed of vehicle respectively.

5 Experimental Methodology

Model Architectures and Training For CropFollow, we evaluate three NN architectures for both heading and distance prediction: ResNet-18 [25] (default), SqueezeNet-v1.1 [28], and DarkNet [48]. We use NNs pretrained on ImageNet [11] and fine-tune them on a dataset containing 25K corn images with manual annotations [56]. The learning rate is 1e-4, with AdamW optimizer.

For Polaris-GEM, we evaluate LaneNet for lane detection with two different backbones: VGG-16 [55] (default) and DarkNet [48]. We use pretrained backbones and fine-tune them on the TuSimple dataset [68] containing 3600+ images with up to five annotated lane markings. The training hyperparameters follow the original LaneNet paper [43]. We use a 4:1 split between training and validation set for both datasets mentioned above.

Model Pruning. We use the structured pruning algorithm (Algorithm 2) with a fixed number of pruning iterations each removing removing an additional 20% filters. We use 20 iterations for NNs in CropFollow and 12 iterations for LaneNet, as the lane detection quality of LaneNet decreases drastically after iteration 12. We refer to the output model of each iteration as a *prune level* and label these from 1 to 20 (or 12) inclusive (0 is the unpruned baseline).

End-to-end QoS Metrics. The end-to-end QoS in CropFollow is measured by the absence of *boundary collisions*. We perform our *row-following* experiments on real-world production crop fields, with each navigation run covering 100 meters. The robot navigates in row-following mode with a linear velocity of 1 m/s. Since the robot has no mechanism to automatically detect collisions with row boundaries, a human observer follows the robot and measures the number of *collisions* with the crop rows. A configuration is valid if the robot has strictly *no* collisions in a run.

For the Polaris-GEM system, we measure the end-to-end QoS by the number of *lane crossings*; the cart in a valid configuration should never go out of its lane (i.e., *zero* lane crossings). The cart navigates a 506-meter long, 2-lane (3 lane markings) road at 2 m/s. We implement a lane crossing detection mechanism by comparing the position of cart and the ground-truth lane markings.

Neural Network Specific Accuracy Metrics As described in Section 4.1, CropFollow contains 2 NNs each predicting a scalar for heading angle θ and distance ratio d. We use two error metrics on these models: (1) *mean error* (or bias), and (2) *standard deviation of error* to capture different kinds of errors in the output. This is motivated by the different impact we observe these two kinds of errors to have on the navigation pattern of the robot.

In Polaris-GEM, the quality of lane detection is measured by *lane detection recall*, a quantity between 0 (worst) and 1 (best): $recall = \sum_{image} \frac{C_{im}}{S_{im}}$, 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 when the average distance between its detected points and its ground-truth points is less than 20 pixels, the threshold used in the TuSimple lane detection challenge.

Error Injection Setup. For both CropFollow and Polaris-GEM, we set `totalEvals` to 20 and `maxRegionEvals` (see Algorithm 1) to 5, and visit a total of $20/5 = 4$ regions. More evaluations lead to more precise error calibration results but increase the experiment time, especially as each evaluation is expensive.

As illustrated in Figure 5, errors in NNs used in CropFollow (approximately) follow Gaussian distributions (same for other architectures and prune levels). Therefore, we use a Gaussian error model $\mathcal{N}(\mu, \sigma^2)$ in error injection: for each NN prediction, an error value is sampled from the Gaussian distribution and added to the prediction. A closed-form formula (e.g. Maximum-likelihood estimation) can compute the parameters (μ, σ) of this error model from the error metrics (§3.1.1).

In Polaris-GEM, we use a Bernoulli distribution $\mathcal{B}(p)$ and a zero-mean Gaussian distribution $\mathcal{N}(0, \sigma)$ to inject error into the boolean mask and the embedding tensor respectively (see §4.2 for these outputs). The resulted error model has 2 parameters (p, σ) and no closed-form formula exists to derive them from lane detection recall. *ApproxCaliper* uses OpenTuner [2] to search for these parameters as described in §3.1.1.

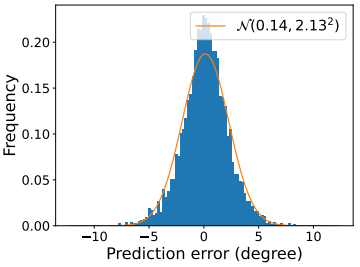

Fig. 5. Distribution of error in the heading angle predicted by baseline ResNet-18 heading model. Errors are measured over the validation dataset and can be approximated well with a Gaussian distribution.

Model Autotuning Setup. *ApproxCaliper* uses the OpenTuner search library for the model autotuning search phase described in §3.2. We apply and compare guided and unguided autotuning in *ApproxCaliper* on CropFollow to find the best model combination for the 2 NNs in CropFollow that has the lowest computation cost (FLOPs) while satisfying QoS requirement (no robot collision). Each NN has 63 candidates – 3 architectures with 21 prune levels each (20 pruned model + 1 baseline), which creates a search space of $63 \times 63 = 3969$ combinations. Each evaluation is a run in real crop fields that takes 5-6 minutes (including setup). Due to the cost of each run, we limit unguided autotuning to 50 iterations that consume approximately 4 hours of physical experiments that require human supervision. For Polaris-GEM, the autotuner searches over 26 model candidates (2 architectures, each 12 prune levels and 1 baseline) for LaneNet and an FPS which LaneNet needs to run at, to find a configuration that reduces overall system resource utilization. FPS is chosen between 0 to 10.5, because the CPU-based clustering algorithm (see §4.2) runs at this FPS and hence limits FPS to 10.5. Resource utilization is reduced if the chosen model has lower computations (measured by FLOPs) and requires a low FPS to meet the end-to-end QoS goals.

Application-agnostic Pruning Baseline. We report our speedups relative to models that are pruned in an application-agnostic manner. The baseline we select represents standard practice for CNN pruning today [13, 34, 38], which is to prune CNN models without accounting for error tolerance in the application (i.e., retaining accuracy close to the unpruned model). For the baseline CropFollow model, based on some trial and error, we consider a 10% relative increase in model error to be close to original, since even slightly pruned models show some increase in error (this benefits the baseline since it exploits our application-aware tuning philosophy). For the Polaris-GEM baseline, we similarly allow 1 percentage point of drop in lane detection accuracy.

Frameworks and Compilation. All models are trained and pruned in PyTorch. For GPU runs on Jetson Nano, we use ONNX [3]. For CropFollow [56], the EKF and MPC are implemented as ROS [45] components in C++ and Python, and their code is not modified in our work. Polaris-GEM is similarly implemented as multiple ROS components. We implement the lane following functionality by combining open-source implementations of LaneNet and Stanley controller; other code in Polaris-GEM is not modified.

6 Evaluation

We experimentally evaluate the following research questions:

RQ1: Does *ApproxCaliper*'s approach of application-aware NN pruning provide higher performance improvements compared to today's practice of application-agnostic pruning (which does not account for application-specific error tolerance, and so aims to retain most of the accuracy of the original NN model)?

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

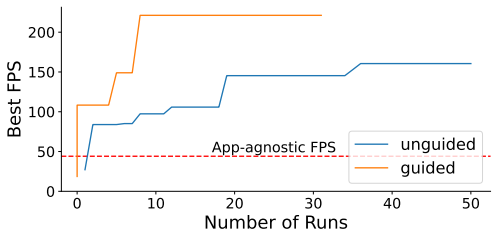

Fig. 6. Performance in FPS (higher is better) of the best valid configuration found vs. number of evaluations in CropFollow autotuning.

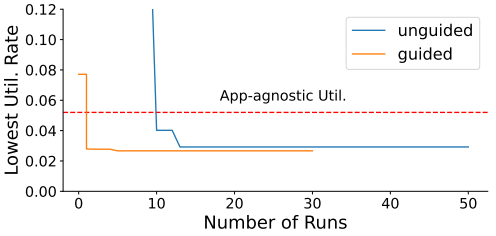

Fig. 7. GPU utilization rate (lower is better) of the best valid configuration found v.s. number of evaluations in Polaris-GEM autotuning.

RQ3: Can *ApproxCaliper* use the error calibration results to better tune the NN components compared to unguided tuning?

6.1 Application-aware Pruning vs Application-agnostic Pruning

To answer RQ1, we compare the best pruning configurations for our two applications, CropFollow and Polaris-GEM, selected by 3 strategies: guided and unguided autotuning in *ApproxCaliper* and the application-agnostic pruning setting.

For CropFollow, Figure 6 shows how the best configuration (found until that point) evolves with the increasing number of tuning iterations. The application-agnostic pruning result is shown as a flat line since this process does not involve autotuning – the best performing pruned model within error threshold is selected for heading and distance predictions. Application-agnostic pruning provides an average FPS (across both models) of 44.0. Application-aware pruning using *ApproxCaliper* provides significantly higher FPS: guided tuning provides an average FPS of 221.2 and unguided tuning provides 160.5 FPS. Hence, guided tuning provides a performance speedup of 5.0× and unguided tuning 3.6×. Section 6.3.2 evaluates in more detail how these large speedups are achieved.

Similarly for Polaris-GEM, Figure 7 shows the GPU utilization rate of configurations decreasing over the course of tuning (here, lower is better). The GPU utilization rate is the usage of GPU due to running LaneNet at an FPS required for successful lane following. Using the LaneNet model application-agnostic pruning provides, lane following requires 5.20% of computing power of GPU. Guided and unguided application-aware pruning reduce this number to 2.92% and 2.66% respectively, which translate to an improvement of 1.8× and 2.0×.

6.2 Error Calibration using *ApproxCaliper*

To answer RQ2, we apply the error calibration framework in *ApproxCaliper* to identify how much error can be introduced to the CNN predictions without affecting the navigation quality of CropFollow and Polaris-GEM. These experiments in turn explain *how ApproxCaliper* is able to achieve the substantial overall gains demonstrated in RQ1 for both applications.

6.2.1 **Error Calibration Results on CropFollow.** In order to evaluate RQ2 in detail, we perform three different kinds of error injection experiments: (1) error injection only to the heading model, (2) error injection only to the distance model, and (3) simultaneous error injection to both models.

Error Injection on Heading Model in Isolation. Figures 8 and 9 show the error constraint space when applying error injection to the heading prediction model at 6 FPS and 10 FPS, respectively, for 20 iterations. The x- and y-axes show mean (aka. bias) and standard deviation of error (§5). The green and red regions are the valid and invalid regions, i.e., they include error combinations

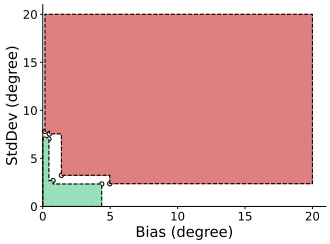

Fig. 8. Error constraints of heading prediction model under Gaussian error injection at 6 FPS.

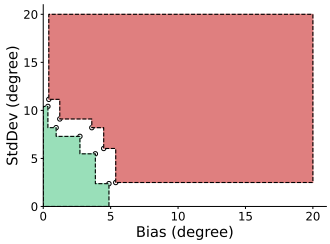

Fig. 9. Error constraints of heading prediction model under Gaussian error injection at 10 FPS.

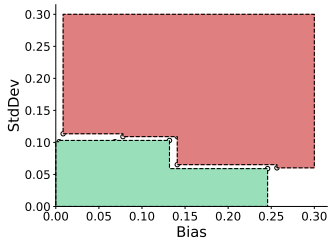

Fig. 10. Error constraints of distance prediction model at 10 FPS. Values have the same "unit" as distance model output – distance ratio – a unit-free quantity.

that produce acceptable QoS (no robot collisions) unacceptable QoS (> 0 robot collisions). The white regions are unevaluated within 20 iterations and can be evaluated if more iterations are used.

The larger green area in Figure 9 compared to Figure 8 shows that higher FPS significantly increases error tolerance. Further investigation (results omitted here) showed that this is because higher FPS allows more control actions per second, so the effect of a bad prediction is shorter-lived and can be counteracted faster.

The bias-variance constraint space in Figures 8 and 9 shows that increasing bias reduces the tolerable standard deviation. At a bias of 0.35 degrees, a high standard deviation of 10.4 degrees is tolerable, without collisions. At a bias of 4.8 degrees, the tolerable standard deviation reduces to 2.4 degrees. At high-bias, the robot steers towards one direction, bringing it closer to the crop boundary; therefore, relatively low variance is acceptable compared to when the robot is centered (i.e., at low bias).

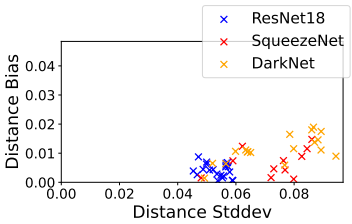

Fig. 11. Standard deviation error v.s. Bias of all distance prediction models.

Error Injection on Distance Model in Isolation. Figure 10 shows the error constraint space generated when Gaussian error is injected to the distance model output at 10 FPS. This graph shows a similar trend where the acceptable standard deviation in errors reduces with high-bias.

Simultaneous Error Injection on Heading and Distance Model. If we consider both bias and variance metrics for both models simultaneously, the error constraint space becomes 4-dimensional. While *ApproxCaliper* supports error injection in any arbitrary number of dimensions, traversing a 4-dimensional space requires a higher number of field evaluations since more error points must be considered. Most of the CNN architectures and their pruned variants introduce much lower bias than variance (distance models shown in Figure 11; heading models follow the same trend); hence, we perform a 2-dimensional error injection experiment that considers standard deviation of errors for each of the models, while keeping the mean bias zero.

Figure 12 shows the results for this experiment. The results reveal that simultaneous errors in both heading and distance prediction networks results in worse QoS compared to errors in either one of the two models. With a distance prediction error at standard deviation of 0.05, heading can tolerate up to 11 degrees of standard deviation error. With increasing stddev of distance prediction error, the tolerable error from the heading model starkly decreases. If the distance error increase to 0.13, the heading error can tolerate no more than 3 degrees' stddev. Conversely, errors in one model

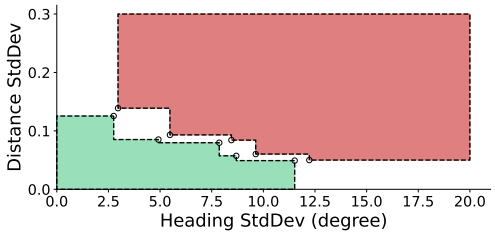

Fig. 12. Joint error constraints of heading and distance prediction models at 10 FPS, where Gaussian error with zero bias are used.

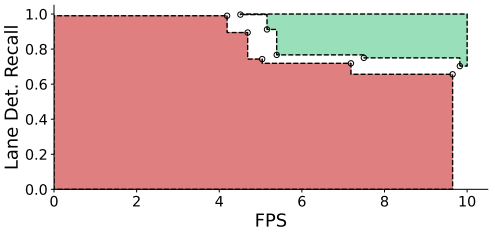

Fig. 13. Error constraints of lane detection recall v.s. FPS on the lane detection network (LaneNet).

are compensated by correct predictions in the other model. High error in both models results in unacceptable navigation quality.

Summary of Insights. The main takeaways from the experiments for RQ2 are:

- FPS, and more generally system performance, has an interplay with accuracy that should be considered when optimizing an application for good performance and acceptable QoS.
- Error calibration with multiple error metrics (e.g., bias and variance of errors for a single component) reveals new insights on how different kinds of error metrics have different impacts on the end-to-end task outcome.
- Simultaneous errors in multiple CNN components (e.g., Heading vs. Distance models) impact the end-to-end QoS differently from errors in one component.

6.2.2 **Error Calibration Results on Polaris-GEM.** For the Polaris-GEM experiment, we use FPS, a performance metric, and lane detection recall, an accuracy to understand the interplay of performance and accuracy towards the end-to-end QoS. Figure 13 shows the results of this experiment with FPS on the x-axis and lane detection recall on the y-axis. We limit the range of FPS search below 10.5 as explained in Section 5 As the FPS increases from 4.5 to 9.8, the application can still provide acceptable navigation (no lane departures) with a lower lane detection recall (99.7% to 70.3%). This reveals that higher FPS increases the system's tolerance of model error (similar to CropFollow). Conversely, this also shows that for a highly accurate model, even a low FPS suffices. This is important because running at a lower FPS reduces the system utilization (which is desirable) Lastly, this result shows that there is a limit to how much FPS and lane detection rate can compensate for each other: no configuration with FPS lower than 4.5 or lane detection rate lower than 70% is feasible.

6.3 Autotuning Results with *ApproxCaliper*

To answer RQ3, we evaluate both guided and unguided tuning strategies supported in *ApproxCaliper*. For CropFollow, the autotuner chooses a pair of pruned models for heading and distance prediction, For Polaris-GEM, autotuner tunes over a pruned model selection and an operating FPS for LaneNet. First, we show the tradeoff space of the different CNN architectures and their pruned variants.

6.3.1 **Tradeoff Space of Pruned Models.** Figure 14 shows the accuracy-performance tradeoff space of CNN architectures (ResNet-18, SqueezeNet, DarkNet) and their pruned variants for the heading (left graph) and distance prediction (right graph) networks in CropFollow. Performance is measured in FPS while accuracy is measured as *L2* error (or MSE) over the validation set (see §5). Points to the upper-left of this space are the better tradeoff points due to lower error and higher FPS.

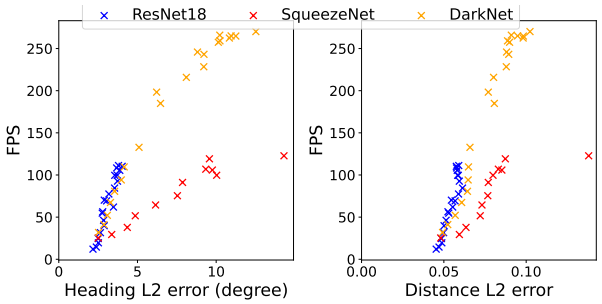

Fig. 14. Regression error (standard deviation) v.s. performance (FPS) of heading (left) and distance (right) prediction CNN in CropFollow. As the pruning level of model increases, its point in figure moves up and to the right.

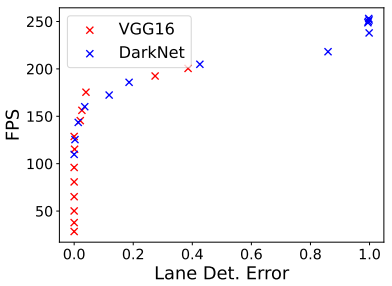

Fig. 15. Lane detection error v.s. performance (FPS) of lane detection CNN in Polaris-GEM. At higher prune levels, error and FPS both increase.

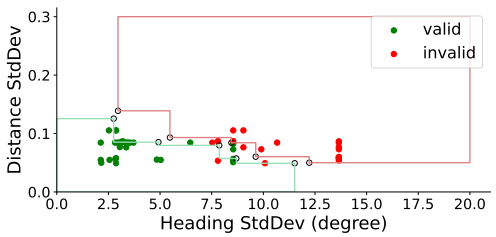

Fig. 16. Configurations evaluated empirically in unguided autotuning v.s. error calibration result used in guided autotuning, on CropFollow.

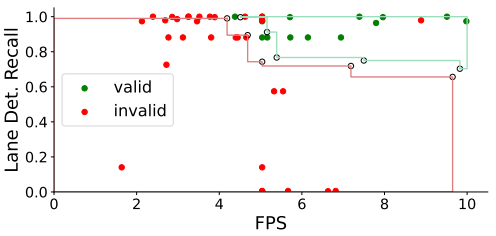

Fig. 17. Configurations evaluated empirically in unguided autotuning v.s. error calibration result used in guided autotuning, on Polaris-GEM.

For both heading and distance prediction, the largest model ResNet-18 (unpruned) has the lowest FPS but is also the most accurate. Interestingly, DarkNet baseline has 1.6x fewer parameters than SqueezeNet, while both have similar error. DarkNet and SqueezeNet baselines have slightly higher error than the ResNet-18 baseline; 1.2x higher error in heading prediction and 10% higher error in distance prediction. This difference in errors grows larger with the higher prune levels; SqueezeNet-18 prune level 20 heading model and DarkNet prune level 20 have 3.6x and 3.3x higher error compared to ResNet-18 prune level 20. Compared to SqueezeNet, DarkNet higher prune levels provides the better tradeoff points since they are lower error and also more performant, e.g., DarkNet heading prune level 10 has similar error as SqueezeNet prune level 7 (both 8.1 degrees) but has 130% higher FPS. Overall, these results show that pruning exposes very interesting tradeoff choices across CNN architectures and their pruned variants.

Figure 15 shows the error-performance tradeoff for LaneNet. Performance is measured in FPS and error measured as drop of lane detection recall ($1 - recall$). Both error and performance increase with higher prune levels (up and to the right in the figure). The DarkNet baseline only has slightly higher error than the VGG-16 baseline but has 10x higher FPS; thus provides a better tradeoff point. Also similar to the CropFollow setting, error of the larger VGG-16 model increases less than DarkNet. The error of DarkNet plummets to 1.0 (0 recall) after prune level 9.

6.3.2 **Guided vs. Unguided Tuning on CropFollow.** As shown in Section 6.1, we apply the tuning component of *ApproxCaliper* to CropFollow to find a pair of pruned models for the heading and distance CNNs, maximizing the performance while maintaining navigation quality.

Comparing Speed of Convergence. Figure 6 shows the model FPS achieved by guided and unguided tuning. We use 50 runs for unguided tuning and 30 for guided tuning for a fair comparison, as the error calibration used to guide guided tuning requires 20 evaluations. In 50 runs, the unguided autotuner converges to the best configuration providing an FPS of 160.5, while in 30 runs, the guided autotuner converges to 221.2 FPS – an improvement of 1.37×. In addition, guided autotuning finds better configuration over the whole search after only the 10th evaluation.

Effectiveness of Using Calibration Result to Guide Autotuning. Figure 16 shows the 50 configurations evaluated in unguided tuning, overlaid on the error calibration result used in guided autotuning. Each configuration is a point in the heading-distance error constraint space, and valid and invalid configuration points are color-coded as green and red respectively. The error calibration result is from previous calibration experiment in Section 6.2 (identical to Figure 12). The green- and red-bordered regions are the spaces of valid and invalid configurations. The purpose of this analysis is two-fold: (1) to study how well the error injection results agree with the behavior of the prune model configurations, and (2) to show that unguided tuning traverses points that are evidently valid or invalid, which guided tuning avoids by using error calibration result. Both must be true for guided tuning to be effective.

In Figure 16, among all 50 points, only two false positives occur (red points in the green region), and there are no false negatives (no green points in the red region). This shows how well the error injection algorithm corresponds to the errors in the pruned model configurations. The exceptions we observe (e.g., red point in the green boundary region) are possibly due to generalization errors; the accuracy of a pruned model in the wild may deviate from the accuracy on the validation set. In addition, 82% (41/50) of configurations shown in Figure 16 (for unguided tuning) would not be evaluated if error calibration result is used (i.e., guided tuning). Guided autotuning converges faster because it can perform more tuning iterations within the same number of empirical evaluations.

6.3.3 **Guided vs. Unguided Tuning on Polaris-GEM.** As shown in Section 6.1, we apply the tuning component of *ApproxCaliper* to Polaris-GEM with the goal to find a configuration of FPS and pruned model selection that minimizes GPU utilization.

Comparing Speed of Convergence. Figure 7 shows the utilization rate achieved by guided and unguided tuning. Similar to CropFollow, we use 50 runs for unguided tuning and 30 for guided tuning. The lowest utilization rates these strategy achieve are 2.92% and 2.66% respectively. Guided tuning provides a 0.26 percentage points improvement, or 1.09× relatively, over unguided tuning. In addition, guided autotuning discovers the best found configuration on only the second empirical evaluation. At tuning time, this is in fact the 16th visited configuration, but only the 2nd configuration that needs evaluation (i.e., does not fall in valid or invalid regions in error calibration results).

Effectiveness of Using Calibration Result to Guide Autotuning. Figure 17 shows the 50 configurations in unguided tuning, overlaid on the error calibration result used in guided autotuning. In Figure 16, 3 false positive (red points in the green region) and no false negative occur, leading to a false positive of 6% and false negative rate of 0%. In addition, 78% (39/50) of the configurations evaluated in unguided tuning would not be evaluated in guided tuning, showing the effectiveness of error calibration filtering out the evidently valid and invalid points.

7 Related Work

We present the first system that evaluates the potential for neural network approximation in the context of an end-to-end application pipeline, and automatically prunes neural network models leveraging the error tolerance of the application.

Systems for Automatic Model Optimization. *DeepCompression* [26] uses a combination of unstructured pruning, weight compression, and weight quantization to achieve significant

reductions in model parameters (up to 49X for VGG-16). DeepCompression tunes neural networks with the constraint to preserve the original accuracy. DeepCompression is used for optimizing neural networks in isolation; not in the context of an end-to-end application. *ApproxCaliper* is complementary to DeepCompression and similar systems: the relaxed error constraints learned through *ApproxCaliper* can be used by frameworks like DeepCompression and others to achieve higher performance benefits when applying approximations in an application-specific context.

ApproxDet [65] and ApproxTuner [53] are dynamic approximation-tuning systems that trade off accuracy at runtime to improve performance. Both these frameworks (like DeepCompression) apply approximations to the neural networks in isolation; they lack mechanisms to measure application's error resilience and have no support for application-specific tuning of neural networks.

Blalock et al [5] show that different neural network architectures and their pruned variants offer an accuracy-performance trade-off space of models to choose from. *ApproxCaliper* automatically navigates this tradeoff space and selects high-performing configurations traversing this space. Moreover, merely picking a point from the trade-off space of pruned models does not capture how the models behave in the end-to-end application pipeline. For instance, it does not capture how simultaneous errors in two different neural network components affect application-level QoS (Section 6.2.1). Also, as shown by our results on the POLARIS-GEM simulator 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 6.2.2). *ApproxCaliper* allows developers to express these different relationships by specifying both performance and accuracy metrics.

Studies of Approximation Impact on End-to-End Applications. Here we discuss papers that study the impact of component-level approximations on the end-to-end task quality beyond deep learning. These are studies that do not propose any new framework or system for evaluating the potential for approximation and are solely presented as empirical studies. De et al. [10] evaluate the impact of approximations applied to the camera image signal processing pipeline used in a simulated lane-keep assist vehicle. Two key differences from our evaluated lane-following application are: a) no approximations are applied to the perception module which is usually a computationally expensive component in vision-guided control systems; and b) for visual perception, it uses classical image processing techniques, whereas we use a neural network (for lane-detection), which is becoming increasingly popular for visual-navigation systems [67].

Similar studies by Raha et al. [46] and Nakhkash et al. [42], and Venkatagiri et al. [62] apply computational approximations on different application subcomponents and measure the impact on end-to-end quality. The main differences from our work are: a) none of these studies propose a general framework for error and/or approximation analysis that can be applied to multiple different applications, and b) these studies do not address deep learning models, or deep learning-specific optimizations, which are increasingly dominant computations in edge computing applications and CPS.

Fault and Error Analysis Frameworks. Randomized fault injection is a well-established approach for error-resiliency analyses. We discuss here some past works that also use an error injection framework and highlight the differences from our approach.

BinFI [7] is a neural network focused framework that uses single bit-flips at the level of tensor operations (convolutions, pool, relu, etc.) to model hardware transient faults (e.g., due to electromagnetic effects) that randomly occur during program execution and may lead to output corruption. FILR [39] is a similar neural network focused technique that also uses single-bit flip errors to identify feature maps that are more vulnerable to transient errors. These systems have different goals from *ApproxCaliper* in the following ways: a) they focus on studying resiliency and not the potential for approximation, b) they focus on analyzing operations within a neural network (at the level of tensor operations and feature maps) and not at the application-level; and c) they study one neural network in isolation, whereas *ApproxCaliper* can also analyze how errors compose across multiple neural

networks (e.g., the heading and distance networks in CropFollow), and d) these are not general configurable frameworks for neural network error injection, whereas *ApproxCaliper* allows developers to configure error models, error metrics, end-to-end QoS metrics and other analysis parameters.

Approxilyzer [61], Relyzer [24], KULFI [54], and AxPIKE [16] inject general bit-flips in arbitrary applications at the architecture level to identify those that lead to program crashes and unacceptable outputs. In contrast, *ApproxCaliper*'s customizable error injection models are tailored for the complex applications with neural network components and is used for guiding the network optimization, with respect to the end-to-end quality.

8 Conclusion

ApproxCaliper is the first system that evaluates the potential for neural network approximation in the context of an end-to-end application pipeline, and automatically prunes feed-forward neural network models leveraging the error tolerance of the application. *ApproxCaliper* is a novel, general framework that can be used by application developers to automatically analyze the impact of neural network prediction errors on end-to-end application quality metrics, and to automatically tune neural network accuracy (of multiple networks) to improve performance or energy while preserving specified quality metrics. We focus on structured model pruning for our optimizations, while our framework is also extensible to other kinds of approximation and optimization techniques Our results for two cyberphysical applications guided by neural networks for visual perception show that *ApproxCaliper* achieves significantly better performance improvements (5.3× and 2.2×, respectively) than tuning model pruning conservatively without allowing accuracy reductions. The experiments also show that *ApproxCaliper* can provide valuable insights into the impact of model choices on application quality metrics and performance, such as the tradeoffs between different choices of models and pruning levels.

References

[1] 2021. GEM e2. https://gem.polaris.com/en-us/e2/.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316.

[3] Junjie Bai, Fang Lu, Ke Zhang, et al. 2020. ONNX: Open Neural Network Exchange. https://www.onnxruntime.ai/about.html.

[4] David Ball, Patrick Ross, Andrew English, Peter Milani, Daniel Richards, Andrew Bate, Ben Upcroft, Gordon Wyeth, and Peter Corke. 2017. Farm workers of the future: Vision-based robotics for broad-acre agriculture. *IEEE Robotics & Automation Magazine* 24, 3 (2017), 97–107.

[5] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033* (2020).

[6] Wei Chen, Ting Qu, Yimin Zhou, Kaijian Weng, Gang Wang, and Guoqiang Fu. 2014. Door recognition and deep learning algorithm for visual based robot navigation. In *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*. IEEE, 1793–1798.

[7] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2019. BinFI: an efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–23.

[8] Wen-Chang Cheng and Chia-Ching Chiang. 2011. The development of the automatic lane following navigation system for the intelligent robotic wheelchair. In *2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011)*. IEEE, 1946–1952.

[9] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2020. Universal Deep Neural Network Compression. *IEEE Journal of Selected Topics in Signal Processing* 14, 4 (2020), 715–726. https://doi.org/10.1109/JSTSP.2020.2975903

[10] Sayandip De, Sajid Mohamed, Konstantinos Bimpisidis, Dip Goswami, Twan Basten, and Henk Corporaal. 2020. Approximation trade offs in an image-based control system. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1680–1685.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[12] ARM Developer. 2021. Cortex-A72. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a72

[13] Xiaohan Ding, Tianxiang Hao, Jianchao Tan, Ji Liu, Jungong Han, Yuchen Guo, and Guiguang Ding. 2021. ResRep: Lossless CNN Pruning via Decoupling Remembering and Forgetting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 4510–4520.

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

[15] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2020. Fast sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 14629–14638.

[16] I. Felzmann, J. Fabrício Filho, and L. Wanner. 2021. AxPIKE: Instruction-level Injection and Evaluation of Approximate Computing. In *2021 Design, Automation and Test in Europe Conference Exhibition (DATE)*.

[17] Michael Figurnov, Aijan Ibraimova, Dmitry Vetrov, and Pushmeet Kohli. 2015. Perforatedcnns: Acceleration through elimination of redundant convolutions. *arXiv preprint arXiv:1504.08362* (2015).

[18] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).

[19] Lorenz Gerstmayr, Frank Röben, Martin Krzykawski, Sven Kreft, Daniel Venjakob, and Ralf Möller. 2009. A vision-based trajectory controller for autonomous cleaning robots. In *Autonome Mobile Systeme 2009*. Springer, 65–72.

[20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

[21] Yili Gu, Zhiqiang Li, Zhen Zhang, Jun Li, and Liqing Chen. 2020. Path tracking control of field information-collecting robot based on improved convolutional neural network algorithm. *Sensors* 20, 3 (2020), 797.

[22] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[23] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626* (2015).

[24] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 123–134.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[26] 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*. 770–778.

[27] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[28] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5MB model size. arXiv:1602.07360 [cs.CV]

[29] Vinu Joseph, Ganesh L. Gopalakrishnan, Saurav Muralidharan, Michael Garland, and Animesh Garg. 2020. A Programmable Approach to Neural Network Compression. *IEEE Micro* 40, 5 (2020), 17–25. https://doi.org/10.1109/MM.2020.3012391

[30] Se Jung Kwon, Dongsoo Lee, Byeongwook Kim, Parichay Kapoor, Baeseong Park, and Gu-Yeon Wei. 2020. Structured compression by weight encryption for unstructured pruning and quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1909–1918.

[31] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

[32] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning Filters for Efficient ConvNets. *CoRR* abs/1608.08710 (2016). arXiv:1608.08710 http://arxiv.org/abs/1608.08710

[33] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient Convnets. *International Conference on Learning Representations (ICLR)* (2017).

[34] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. 2019. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2790–2799.

[35] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David S. Doermann. 2019. Towards Optimal Structured CNN Pruning via Generative Adversarial Learning. *CoRR* abs/1903.09291

(2019). arXiv:1903.09291 http://arxiv.org/abs/1903.09291

[36] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. 2020. AutoCompress: An automatic DNN structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4876–4883.

[37] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270* (2018).

[38] Jian-Hao Luo, Hao Zhang, Hong-Yu Zhou, Chen-Wei Xie, Jianxin Wu, and Weiyao Lin. 2018. Thinet: pruning cnn filters for a thinner net. *IEEE transactions on pattern analysis and machine intelligence* 41, 10 (2018), 2525–2538.

[39] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W Fletcher, Sarita V Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B Sullivan, Timothy Tsai, and Stephen W Keckler. 2021. Optimizing Selective Protection for CNN Resilience. *International Symposium on Software Reliability Engineering* (2021).

[40] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. 2020. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*. PMLR, 6682–6691.

[41] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. *CoRR* abs/1611.06440 (2016). arXiv:1611.06440 http://arxiv.org/abs/1611.06440

[42] Mohammadreza Nakhkash, Anil Kanduri, Amir M Rahmani, and Pasi Liljeberg. 2019. End-to-end approximation for characterizing energy efficiency of IoT applications. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. IEEE, 1–6.

[43] Davy Neven, Bert De Brabandere, Stamatios Georgoulis, Marc Proesmans, and Luc Van Gool. 2018. Towards end-to-end lane detection: an instance segmentation approach. In *2018 IEEE intelligent vehicles symposium (IV)*. IEEE, 286–291.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[45] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[46] Arnab Raha and Vijay Raghunathan. 2018. Approximating beyond the processor: Exploring full-system energy-accuracy tradeoffs in a smart camera system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 12 (2018), 2884–2897.

[47] Lingyan Ran, Yanning Zhang, Qilin Zhang, and Tao Yang. 2017. Convolutional neural network-based robot navigation using uncalibrated spherical images. *Sensors* 17, 6 (2017), 1341.

[48] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[49] Alex Renda, Jonathan Frankle, and Michael Carbin. 2019. Comparing Rewinding and Fine-tuning in Neural Network Pruning. In *International Conference on Learning Representations*.

[50] J.M. Roberts, E.S. Duff, P.I. Corke, P. Sikka, G.J. Winstanley, and J. Cunningham. 2000. Autonomous control of underground mining vehicles using reactive navigation. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, Vol. 4. 3790–3795 vol.4. https://doi.org/10.1109/ROBOT.2000.845322

[51] Xiaofeng Ruan, Yufan Liu, Chunfeng Yuan, Bing Li, Weiming Hu, Yangxi Li, and Stephen Maybank. 2021. EDP: An Efficient Decomposition and Pruning Scheme for Convolutional Neural Network Compression. *IEEE Transactions on Neural Networks and Learning Systems* 32, 10 (2021), 4499–4513. https://doi.org/10.1109/TNNLS.2020.3018177

[52] Matthew Salfer-Hobbs and Matthew Jensen. 2020. Acceleration, Braking, and Steering Controller for a Polaris Gem e2 Vehicle. In *2020 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, 1–6.

[53] Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Akash Kothari, Ben Schreiber, Elizabeth Wang, Yasmin Sarita, Nathan Zhao, Keyur Joshi, Vikram S Adve, Sasa Misailovic, and Sarita V Adve. 2021. ApproxTuner: a compiler and runtime system for adaptive approximations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 262–277.

[54] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2013. Towards Formal Approaches to System Resilience. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*. 41–50. https://doi.org/10.1109/PRDC.2013.14

[55] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[56] Arun Narenthiran Sivakumar, Sahil Modi, Mateus Valverde Gasparino, Che Ellis, Andres Eduardo Baquero Velasquez, Girish Chowdhary, and Saurabh Gupta. 2021. Learned Visual Navigation for Under-Canopy Agricultural Robots. *CoRR* abs/2107.02792 (2021). arXiv:2107.02792 https://arxiv.org/abs/2107.02792

[57] Yunlong Sun, Lianwu Guan, Zhanyuan Chang, Chuanjiang Li, and Yanbin Gao. 2019. Design of a low-cost indoor navigation system for food delivery robot based on multi-sensor information fusion. *Sensors* 19, 22 (2019), 4980.

[58] Niko Sünderhauf, Oliver Brock, Walter Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, et al. 2018. The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research* 37, 4-5 (2018), 405–420.

[59] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. 2006. Stanley: The robot that won the DARPA Grand Challenge. *Journal of field Robotics* 23, 9 (2006), 661–692.

[60] Frederick Tung and Greg Mori. 2020. Deep Neural Network Compression by In-Parallel Pruning-Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 3 (2020), 568–579. https://doi.org/10.1109/TPAMI.2018.2886192

[61] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.

[62] Radha Venkatagiri, Karthik Swaminathan, Chung-Ching Lin, Liang Wang, Alper Buyuktosunoglu, Pradip Bose, and Sarita Adve. 2018. Impact of software approximations on the resiliency of a video summarization system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 598–609.

[63] Stavros G Vougioukas. 2019. Agricultural robotics. *Annual Review of Control, Robotics, and Autonomous Systems* 2 (2019), 365–392.

[64] Xia Xiao and Zigeng Wang. 2019. Autoprune: Automatic network pruning by regularizing auxiliary parameters. *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)* 32 (2019).

[65] Ran Xu, Chen-lin Zhang, Pengcheng Wang, Jayoung Lee, Subrata Mitra, Somali Chaterji, Yin Li, and Saurabh Bagchi. 2020. ApproxDet: content and contention-aware approximate object detection for mobiles. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 449–462.

[66] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. 2020. Automatic Neural Network Compression by Sparsity-Quantization Joint Learning: A Constrained Optimization-Based Approach. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[67] Yuri D. V. Yasuda, Luiz Eduardo G. Martins, and Fabio A. M. Cappabianco. 2020. Autonomous Visual Navigation for Mobile Robots: A Systematic Literature Review. *ACM Comput. Surv.* 53, 1, Article 13 (Feb. 2020), 34 pages. https://doi.org/10.1145/3368961

[68] Kai Zhou. 2018. Tusimple Benchmark Ground Truth. https://github.com/TuSimple/tusimple-benchmark/issues/3