2024 IEEE/ACM Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI) | 979-8-3503-7025-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IoTDI61053.2024.00014

NaviSlim: Adaptive Context-Aware Navigation and Sensing via Dynamic Slimmable Networks

1st Timothy K Johnsen
Computational Sciences, Joint Doctoral Program
University of California, Irvine, USA
San Diego State University, USA
tjohnsen@uci.edu

2nd Marco Levorato
Computer Science Department
University of California, Irvine, USA
levorato@uci.edu

***Abstract*—Small-scale autonomous airborne vehicles, such as micro-drones, are expected to be a central component of a broad spectrum of applications ranging from exploration to surveillance and delivery. This class of vehicles is characterized by severe constraints in computing power and energy reservoir, which impairs their ability to support the complex state-of-the-art neural models needed for autonomous operations. The main contribution of this paper is a new class of neural navigation models – NaviSlim – capable of adapting the amount of resources spent on computing and sensing in response to the current context (*i.e.*, difficulty of the environment, current trajectory, and navigation goals). Specifically, NaviSlim is designed as a gated slimmable neural network architecture that, different from existing slimmable networks, can dynamically select a slimming factor to autonomously scale model complexity, which consequently optimizes execution time and energy consumption. Moreover, different from existing sensor fusion approaches, NaviSlim can dynamically select power levels of onboard sensors to autonomously reduce power and time spent during sensor acquisition, without the need to switch between different neural networks. By means of extensive training and testing on the robust simulation environment Microsoft AirSim, we evaluate our NaviSlim models on scenarios with varying difficulty and a test set that showed a dynamic reduced model complexity on average between 57-92%, and between 61-80% sensor utilization, as compared to static neural networks designed to match computing and sensing of that required by the most difficult scenario.**

***Index Terms*—Autonomous Systems, Dynamic Neural Networks, Drone Navigation, Sensor Fusion**

I. Introduction

Drone autonomy is a rapidly developing area of investigation among Internet of Things (IoT) devices, with potential impact to a broad range of applications such as remote exploration, first response, agriculture, and delivery. An extensive survey on hardware and software requirements for developing fully autonomous Unmanned Aerial Vehicles (UAV) can be found in [1]. The most significant issues reported by the authors are: increasing complexity of tasks, operating in unknown and diverse environments, limitations on sensing capabilities, flight time, and energy consumption.

The aforementioned issues become more prominent when considering small-scale vehicles such as airborne micro-drones. This class of vehicles suffers from extreme constraints in terms of computational power and energy reservoir, which severely limits their ability to support the complex machine learning algorithms prevalent in vehicular autonomy. Moreover, the same limitations affect onboard sensors, that require time and energy to achieve adequate resolution to support autonomous functionalities in dynamic environments. Intuitively, the time required to execute sensor acquisition and machine learning algorithms during inference also slow down the reaction time of autonomous vehicles.

The literature related towards developing (micro-)drone autonomy is rich in autonomous drone racing [2]; while other efforts are in developing autonomous swarms [3], or in robust test-beds to evaluate them in [4]. General studies towards drone autonomy have focused on reducing computation required to execute static neural models as in [5], [6], fusing sensors [7], and of course general autonomous navigation logic and modeling such as in [8]. We find the literature lacking in studies that place an emphasis on developing autonomous navigation methods specifically for micro-drone systems with extreme resource constraints. To this aim, we contend that it is necessary to evolve the static nature of state-of-the-art navigation models, and neural models in general, into dynamic algorithms that use the minimum amount of computational complexity required by the difficulty of scenario, and concurrently the minimum time and energy spent in sensor acquisition.

To accomplish such an ambitious objective, in this paper we introduce a new class of navigation models that have a dynamic architecture capable of adapting its structure in real-time to the difficulty of the current operating environment at a fine-time granularity. Specifically, we present *NaviSlim*, a navigational neural network architecture that dynamically scales its own complexity and sensor modalities based on the perceived context. To support this logic, we propose a robust design and multi-stage training approach based primarily on slimmable networks [9], which train partitioned sub-networks within a larger static one, and knowledge distillation [10], which trains different models to exhibit similar behavior. In addition to these two core components, our design utilizes a broad array of advanced algorithms and methods such as shortest path algorithms, supervised learning, deep reinforcement learning, and curriculum learning.

We develop our models using a test bed environment with a

979-8-3503-7025-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IoTDI61053.2024.00014

simulation tool that can be found open-source on our GitHub [1]. Our Python repository interfaces with Microsoft AirSim [11] which is a robust drone simulator rendered in Unreal Engine [12] to handle physics and graphics. Our experiments evaluate several scenarios with varying difficulties. We show that our *NaviSlim* models are dynamically reduced, on average, to: 1) 57-92% model complexity, and 2) 61-80% sensor utilization, of that used by a static network required to otherwise safely navigate the terrain. We further provide evidence that *NaviSlim* adapts the resources used (power, time, and energy) to perceived context.

The remainder of this paper is organized as follows. We first present literature related to dynamic neural networks, and identify the gap we aim to fill within autonomous drone navigation. Then we provide an overview of our approach in Section III, including the general system model design, problem formulation, and inherent challenges. The test bed environment and simulation tool are presented in Section IV. The implementation of *NaviSlim* is described in Section V, while the training procedure is discussed in Section VI and Section VII. We end with results of our experiments implemented in the test bed environment, and conclusions.

II. Related Work

State-of-the-art navigation models employ deep reinforcement learning, as in [13]–[18], in the form of a static neural model processing data from a fixed sensor array – which we use as a comparison to our presented models. The core limitation of such state-of-the-art approaches is that the model and sensing characteristics need to match the most challenging operating situation, which leads to an unnecessarily large resource usage – *i.e.*, the neural network models and sensing requirements are static. Several frameworks are deployed as Simultaneous Localization And Mapping (SLAM) approaches, which includes methods for navigation such as localization, mapping, and tracking [19]. However, SLAM algorithms require extensive sensing and intense computing, and are impractical for microdrones - which are the focus of this paper..

Our proposed architecture falls under the general umbrella of dynamic neural networks, which can scale the depth of a neural network (vertically) with early exits [10] and the width of the network (horizontally) with slimmable networks [9]. The most popular class of dynamic neural models use early exits [10], where low complexity structures (the early exits) are attached to a main model and are sequentially executed. The processing of an input is terminated if the output of the last executed exit has sufficient confidence, so that the overall number of operations depends on the complexity of individual input samples. Our architecture belongs to a new emerging class of dynamic neural networks – dynamic slimmable models – adopted by a small number of recent contributions [20], [21], where an internal module manipulates characteristics of the entire network.

[1]https://github.com/WreckItTim/rl_drone

An alternative approach to ours is to store multiple versions of the same model and swap models at runtime, an approach that requires extended memory availability, as well as a potential context switching latency. Neural architecture search (NAS) [22] embeds a DRL algorithm to select the optimal network structure, however is extremely time consuming as each iteration requires complete training. In our approach, we develop models that realize an advanced form of dynamic slimmable networks [9], designed to seamlessly change its shape with minimal memory and no latency overhead. Technically, this is accomplished by horizontally scaling down the number of active nodes in a set of target layer(s) within a larger super-network. The super-network can be scaled down at various increments, thus creating a series of smaller sub-networks that can be used for inference. Moreover, we employ universally slimmable networks [23] that dynamically and continuously scale down the number of activation nodes in each hidden layer. The design of techniques to select which sub-networks to use during inference is an area of research attracting considerable interest. The very few available solutions (targeting image classification only), [24], [25], use neural gates to intelligently select sub-networks. With NaviSlim, we advance the state-of-the-art by designing and training a dynamic slimmable neural network for navigation whose shape is controlled by a context-aware gate capable of selecting from a continuous array of sub-networks on a sample-by-sample basis.

III. Overview and Problem Formulation

Let us first describe the general setting in which we position our contribution. We consider the task to navigate a microdrone with extremely limited onboard resources, in terms of sensing and computing capabilities, from one location to another while avoiding collisions and minimizing path length. Although the framework and methodology we propose are applicable to more general settings, here we consider a microdrone equipped with: (*a*) multiple depth sensors (*e.g.*, LiDARs) pointing in different directions (*e.g.*, forward and downward), and (*b*) a GPS module returning its position on the map. Depth sensors provide rich information about the environment, which can be used for navigation in settings that do not require high-level reasoning (e.g., semantic features of objects such as the meaning of a traffic sign). Sensor information is input into a neural network which outputs motion commands for the drone.

In the context of micro-drones, the energy associated with sensing and computing represents a non-negligible portion of the overall expenditure. Our measurements show that the continuous execution of a relatively lightweight convolutional model use for object detection, executed on a GPU, can take up to 12% of the total power needed for airborne motion, sensing, and computing. Thus, we consider both dynamic sensors and dynamic neural networks that can be controlled to minimize resource usage. The depth sensors can be tuned to scan a partial field of view, where the smaller the acquired field of view the smaller the time and energy used by the sensor. The relationship between scanned area, resolution, sampling

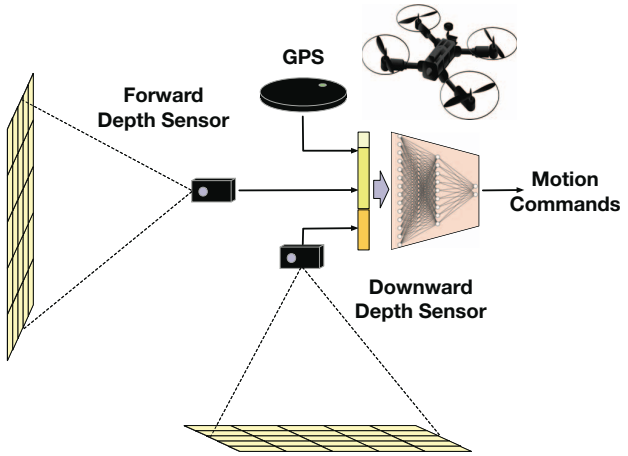

Fig. 1. High-level schematics of the considered sensing-computing-control pipeline. The neural network processes the input of forward and downward facing depth sensors and GPS to produce motion commands that fly the drone on an ideally length-optimal path from point A to B.

time, and energy is exemplified by lightweight LiDAR sensors that require an amount of time and energy proportional to the extent of the variable scanned area and sampling time [26]. In the framework we propose, the neural network used to control motion commands is a universally slimmable network [23], where the number of nodes in each hidden layer can vary to accordingly reduce time and energy spent during computation.

A. Problem Formulation

We consider a drone with an on-board processor tasked to navigate an unknown terrain by utilizing a heterogeneous sensor array and embedded neural model. Given a set of sensor observations, **o**(t), measured at the current timestep, *t*, the overall objectives of the embedded neural model are:

• **Navigation:** Output navigation motions, **n**, required for the controller to drive the drone on a length-optimal path that minimizes flight time and energy, while avoiding collisions.

• **Computing:** Execute operations, **c**, that minimize computing resources used to calculate **n**. We use the number of active sub-network parameters, *m*, as a proxy for computing resource usage – an intuitive rationale that we validate experimentally.

• **Sensing:** Query commands, **s**, that minimize sensing resources used to acquire observations which are then used as inputs to calculate **n**. We allocate to the sensors a discrete power level, *w*, used as a proxy for sensing resource usage.

The pipeline from $\mathbf{o}(t)$ to $[\mathbf{n}(t), \mathbf{c}(t), \mathbf{s}(t+1)]$ is illustrated in Fig. 2. First, the newly acquired set of sensor observations for the current time step, $\mathbf{o}(t)$, is sent into a First In First Out (FIFO) queue. The FIFO queue acts as an attention mechanism, keeping the top *T*-many recent observations as done in [27] for autonomous control of Atari games. The FIFO queue is then input into a context-aware neural model that: 1) uses an intermediate mechanism to predict the minimum $\mathbf{c}(t)$ needed to predict $\mathbf{n}(t)$ given the scenario at the current time step, 2) outputs the predicted $\mathbf{n}(t)$ values to execute at the current time step, and 3) outputs the predicted $\mathbf{s}(t+1)$ values to use during sensor acquisition at the next time step.

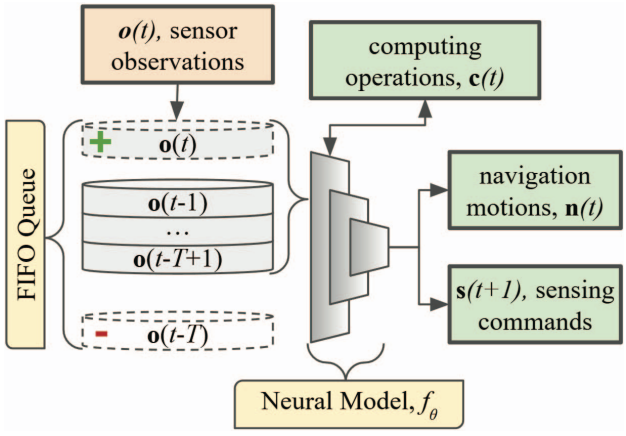

Fig. 2. High-level schematics of the considered sensing-computing-control pipeline, now with an introduced attention mechanism and the ability to predict context-aware computing operations and sensing commands.

Here we formalize the problem that we aim to solve. Let θ be a set of trainable model parameters, such that $\mathbf{a} = f_\theta(FIFO)$, where f_θ is the model and $\mathbf{a}$ is some subset of $[\mathbf{n}, \mathbf{c}, \mathbf{s}]$. Let $\mathbf{p}$ be a path taken by the drone where the length of $\mathbf{p}$ is equal to the number of time steps, P be a set of known length-optimal paths, and $\hat{P}$ be the set of paths taken using f_θ. We optimize θ by minimizing the expected trade off between resource costs of computation, *m*, and resource costs of sensing, *w*, as controlled by a scalar, $0 \leq \alpha \leq 1$, and under the constraint that the length of each path taken by f_θ is no longer than that of that of a scalar, $\beta \geq 1$, times the length of each corresponding optimal path:

$$\begin{aligned} &\min_{\theta} \ < \alpha m + (1-\alpha) w >, \\ &\text{s.t. } length(\hat{P}^{(i)}) <= \beta * length(P^{(i)}) \ \forall \ i \in \{1, ..., b\}, \end{aligned} \tag{1}$$

where $<>$ indicates the expected value over all time steps and paths, b is the total number of paths, $P^{(i)}$ indicates the i^{th} path, and $length(\mathbf{p})$ indicates the number of time steps in path $\mathbf{p}$. We design Equation (1) as to minimize computing and sensing resource usage, while retaining navigation accuracy. The constraint in Equation (1) is required, otherwise the optimization problem would result in the trivial solution where computing and sensing parameters are equal to zero.

B. Challenges and Contributions

The two main challenges in developing *NaviSlim* are:

Test Bed Environment: To train an adaptive model for micro-drone navigation, we need a test bed environment to allow the algorithm to accumulate a large amount of experience across settings with a broad range of complexity and maneuvers. To this end, we developed our open-source software module to interface our models with Microsoft AirSim [11], which is a robust drone simulator for graphics rendering, sensor

acquisition, and physics handling. Our interface obfuscates the environment to the model during training and at deployment time, so that *NaviSlim* can be seamlessly ported to various simulation and real-world environments.

Model Design and Training Procedure: A fundamental question is how to design and train a neural model that can accomplish all three objectives of controlling navigation, computing, and sensing. We base our model design on dynamic slimmable neural networks, and add capabilities to adapt computing and sensing on a sample-by-sample basis. As expected, a single neural network trained from scratch can not converge to any meaningful results that accomplishes all three of these objectives simultaneously. Our solution is to decouple each of these objectives into three respective modules. Each module has with it unique challenges, for which we develop solutions utilizing several methods such as: shortest path algorithms, supervised learning, deep reinforcement learning, curriculum learning, and of most significance knowledge distillation [28].

To the best of our knowledge, the one presented herein is the first neural model that fuses universally slimmable networks with dynamic neural networks to accomplish navigational goals under severe resource constraints, while exploring dynamic sensor scaling, a widely open area of investigation. The core contribution of this paper is *NaviSlim*: a novel framework to design and train neural models that can seamlessly and dynamically adapt their characteristics to environmental context and mission progress to parsimoniously use computation and sensing resources while maintaining high navigation accuracy.

IV. Test Bed Environment

We train and test our models with a simulation framework that utilizes Microsoft AirSim [11], a robust drone simulator that renders physics and graphics in Unreal Engine [12]. AirSim has a Application Programming Interface (API) for Python, that can be used to communicate with the simulation, such as: create sensors and acquire observations, issue drone commands, and detect collisions. We use the AirSim API to interface with our *NaviSlim* repository, also in Python, which includes methods for deep reinforcement learning that partially utilize the Stable-Baselines3 library [29], neural network implementations that partially utilize the PyTorch library [30], and others such as curriculum learning, shortest path algorithms, supervised learning, knowledge distillation, logging, customization, and deployment to other environments including real world drone controllers. Previous studies have shown capabilities of launching models trained in simulation into the real world [31], [32] – thus a simulation tool is a robust means to explore and develop novel model architectures. Using a simulation also mitigates difficulties in training a model with real world hardware that would require mechanisms for episodic deep reinforcement learning.

Fig. 3 shows two maps used by AirSim. The first map is "Blocks", which contains several static objects that have generic shapes and sizes. The second map is "City", which contains various static and dynamic objects that have specific shapes and sizes which reflect real world encounters such as buildings, signs, cars, people, and live traffic. Both maps have varying densities of objects, and we train and evaluate over a wide range of these densities.

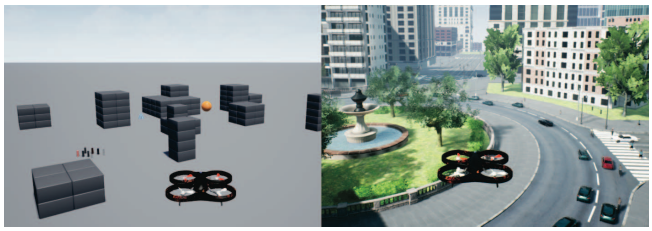

Fig. 3. Two maps from Microsoft AirSim: on the left is "Blocks" which contains static objects with arbitrary shapes and sizes, and on the right is "City" which contains both static and dynamic objects expected to be encountered in the real world.

V. NaviSlim: Design Overview

In this section, we provide an overview of *NaviSlim*, and will detail the specific components (the navigation and auxiliary models) in the next sections. A key novelty is that we design an auxiliary module to control resource expenditure (**c** or **s**), while the navigation module is used to control drone motions (**n**). Thus *NaviSlim*, f_θ, now consists of the navigation model, g_ϕ, and the auxiliary model, h_ψ. If a vanilla approach is taken to train the overall model to simultaneously control both sensing (input) and computing (intermediate calculations used by the model), then the learning process is highly unstable and does not converge to a meaningful control logic – *i.e.*, the navigation paths fail when evaluated in the test bed environment. Thus, our solution is to decouple computing and sensing into two variants of *NaviSlim*. We refer to methods and models related to computing as *NaviSlim-C*, and those related to sensing as *NaviSlim-S* (see Equation (2)):

$$\begin{aligned} &\textit{NaviSlim-C}: \quad \mathbf{n} = g_\phi(FIFO,\ \mathbf{c} = h_\psi(FIFO)) \\ &\textit{NaviSlim-S}: \quad [\mathbf{n}, \mathbf{s}] = [g_\phi(FIFO), h_\psi(FIFO)]. \end{aligned} \tag{2}$$

Note that this structure requires g_ϕ and h_ψ to be executed in series for *NaviSlim-C*, while they can be executed in parallel for *NaviSlim-S*.

The overall system *NaviSlim* model (composed of the navigation and auxiliary models) is illustrated in Fig. 4. The "ToVec()" component converts the data acquired from each depth sensor into preliminary vectors that are then concatenated with the GPS data into one feature vector, **o**, as measured at time *t*. This concatenated feature vector is then inserted onto the FIFO queue as illustrated previously in Fig. 2.

Algorithm 1 shows how a path, also called an episode, is executed using *NaviSlim*. Included in Algorithm 1 are various variables used for deep reinforcement learning, as detailed later in Section VII.

A. Universally Slimmable Networks

The navigation model is a universally slimmable network [23]. Here we introduce a new variable called the slimming

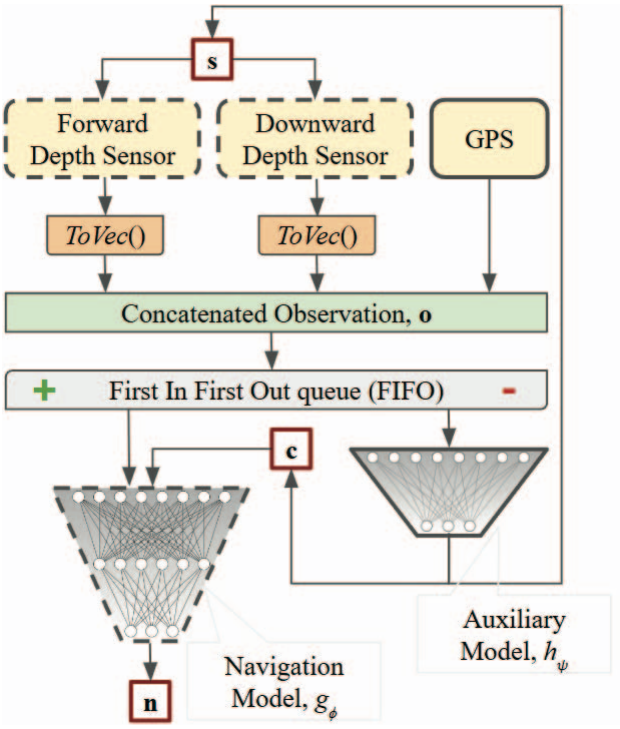

Fig. 4. *NaviSlim*: our novel solution for a context-aware framework capable of adapting resource allocation to that which is required by the difficulty of the current scenario. Shown is our specific implementation. The shapes with dotted lines represent components capable of adaptable resource allocation.

Algorithm 1 Executing an Episode/Path with *NaviSlim*

Input: spawn and goal, max time steps τ, goal tolerance η, navigation model g_ϕ, auxiliary model h_ψ, reward function.

set **c** and **s** to that which corresponds to maximum resources
$t = 1$; $continue = True$
while $continue$ **do**
  acquire sensor observations, **o**, given **s**
  add **o** to $FIFO$
  if *NaviSlim-C* **then**
    $\mathbf{a} \coloneqq \mathbf{c} = h_\psi(FIFO)$
  if *NaviSlim-S* **then**
    $\mathbf{a} \coloneqq \mathbf{s} = h_\psi(FIFO)$
  $\mathbf{n} = g_\phi(FIFO)$, given **c**
  move drone using **n**
  calculate reward, r
  t = t + 1
  if collision detected **then**
    $termination = collision$; $continue = False$
  if distance to target position $< \eta$ **then**
    $termination = goal$; $continue = False$
  if $t > \tau$ **then**
    $termination = time$; $continue = False$

Return: $E = [[\mathbf{o}, \mathbf{n}, \mathbf{a}, r]^{(j)} \; \forall \; j \in \{1, ..., t-1\}, termination]$

factor, ρ, which controls the number of active nodes in each hidden layer, that is, $\mathbf{c} = [\rho]$ since ρ controls the number of operations required to execute the navigation model.

Take an arbitrary hidden layer comprised of a vector of nodes, **h**, indexed from the node at position $k = 1$ to $k = q$ where q is the maximum number of nodes available for that hidden layer. The quantity q corresponds to the number of hidden layer nodes used by the static "super-network", which in our context is a network whose number of parameters must match the most difficult operating scenario. This super-network persists when $\rho = 1$, whereas a sub-network is activated when $\rho < 1$.

The number of active nodes in a layer is equal to $roof(\rho q)$, where $roof()$ rounds up to the nearest integer value, such that the active nodes are those indexed in the range $[1, roof(\rho q))$ and the deactivated nodes are those indexed in the range $[roof(\rho q), q]$. Such a procedural deactivation, as opposed to a random Bernoulli distribution such as used in dropout, is required so that we can select specific sub-networks from the super-network for both training and inference (see Fig. 5).

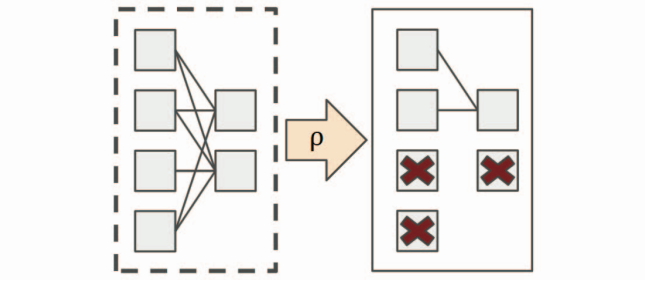

Fig. 5. Procedure used to "slim" a universally slimmable neural network. The variable ρ is used to scale the number of active nodes in each hidden layer. In this example, there are 4 nodes in the first hidden layer and 2 in the second – this is the super-network. Also in this example, we use $\rho = 0.3$ so that 2 nodes are deactivated in the first layer and 1 in the second – this is a sub-network. Note that all weights connected to the deactivated nodes, represented by the lines in between the hidden layers, are also severed.

The relationship between ρ and the total number of active parameters, *m*, in a sub-network is quadratic. Let *u* bet the number of input layer nodes, *l* be the number of hidden layers, **q** be an *l*-length vector containing the number of nodes in each hidden layer, and *v* be the number of output nodes. We consider a fully connected feed-forward multi-layer perceptron with bias terms and at least one hidden layer. We can directly calculate the number of active parameters in a sub-network, *m*, as a function of ρ:

$$\begin{aligned} m(\rho) &= u\rho q_1 + \rho q_1 + \rho q_1 \rho q_2 + \rho q_2 + ... + \rho q_l v + v, \\ m(\rho) &= (\Sigma_{i=1}^{l-1} q_i q_{i+1})\rho^2 + (uq_1 + vq_l + \Sigma_{i=1}^{l} q_i)\rho + v. \end{aligned} \tag{3}$$

Thus, there is a quadratic decrease in the number of active parameters of a sub-network with a decreased ρ.

B. Supervised Learning with Knowledge Distillation

A key method used when training *NaviSlim* is knowledge distillation. From the perspective of a slimmable network, the

main idea of knowledge distillation is to teach sub-networks similar outputs as the super-network. This is accomplished by adding a step to typical supervised learning after the error gradient is calculated between ground truth labels ("hard targets") and the outputs of the super-network , that calculates an additional error gradient between the outputs of the super-network ("soft targets") and outputs of any activated sub-networks. The loss function uses the error between both: A) super-network outputs and ground truth labels and B) sub-network outputs and super-network outputs. This combined supervised learning and knowledge distillation step, what we refer to as the $supervised\_distillation$ function, is defined differently between *NaviSlim-C* and *NaviSlim-S*.

C. NaviSlim-C

The objective of *NaviSlim-C* is to reduce computing resources spent during execution of the navigation model. This is accomplished by defining the $supervised\_distillation$ function with Algorithm 2. Algorithm 2 uses the "sandwich rule" [23], a method for sampling ρ when training with knowledge distillation by calculating the error gradient first with $\rho = 1$, distilling with ρ set to the minimal value, and then distilling with some random intermediate values of ρ.

Algorithm 2 $supervised\_distillation$ for *NaviSlim-C*

Input: navigation model in training g_ϕ, input $FIFO$, target output $\mathbf{n}$, *loss* function.

set running error gradient, *grad*, to zero
$\hat{\mathbf{n}} = g_\phi(FIFO, \rho = 1)$ ▷ Max Resource Cost
aggregate *grad* with *loss* given $\mathbf{n}$ and $\hat{\mathbf{n}}$ ▷ Supervised
$\hat{\mathbf{n}}' = g_\phi(FIFO, \rho = 0.125)$ ▷ Min Resource Cost
aggregate *grad* with *loss* given $\hat{\mathbf{n}}$ and $\hat{\mathbf{n}}'$ ▷ Distillation
for a number of times, *i.e.* 2 **do**
$\hat{\mathbf{n}}' = g_\phi(FIFO, \rho \sim U(0.125, 1))$ ▷ Random Cost
aggregate *grad* with *loss* given $\hat{\mathbf{n}}$ and $\hat{\mathbf{n}}'$ ▷ Distillation

Return: *grad*

D. NaviSlim-S

The objective of *NaviSlim-S* is to minimize the resource usage of sensor acquisition at the next time step. This is accomplished by defining the $supervised\_distillation$ method with Algorithm 3: a novel approach applied to sensing, that distills the network to have similar output regardless of the variable sensor array power level, *w*.

We introduce two variables used to control the respective power levels of the forward, p_f, and downward, p_d, facing depth sensors, with $\mathbf{s} = [p_f, p_d]$. The range of p_f is $[1, 3]$, where the value one corresponds to the minimal power level which uses the smallest area available for scanning. The range of p_d is $[0, 3]$, where zero is the minimal power level corresponding to completely turning off the downward depth sensor. Increasing power levels corresponds to the acquisition of larger areas during scanning with the respective sensor, where a power level of three is the maximum area, and thus the maximum power and time expenditure. Note that $p_f >= 1$ so at-least some sensor information can be acquired at all times. The input layer to the navigation network is designed so that the number of required input nodes corresponds to the magnitude of p_f and p_d. Input nodes are deactivated using a procedural approach, so that sub-networks consisting of subsets of input nodes can be selected during inference and training, similar to *NaviSlim-C*.

Algorithm 3 $supervised\_distillation$ for *NaviSlim-S*

Input: navigation model in training g_ϕ, input $FIFO$ (at max power level), target output $\mathbf{n}$, *loss* function.

set running error gradient, *grad*, to zero
$\hat{\mathbf{n}} = g_\phi(FIFO)$ ▷ Max Resource Cost
aggregate *grad* with *loss* given $\mathbf{n}$ and $\hat{\mathbf{n}}$ ▷ Supervised
$FIFO'$ = re-sampled all $\mathbf{o}$ in $FIFO$ with $p_f = 1, p_d = 0$
$\hat{\mathbf{n}}' = g_\phi(FIFO')$ ▷ Min Resource Cost
aggregate *grad* with *loss* given $\hat{\mathbf{n}}$ and $\hat{\mathbf{n}}'$ ▷ Distillation
for a number of times, *i.e.* 2 **do**
$FIFO'$ = re-sampled with T-many heterogeneous random p_f and p_d values applied to each $\mathbf{o}$ in $FIFO$, where $p_f \sim U(1, 3)$ and $p_d \sim U(0, 3)$
$\hat{\mathbf{n}}' = g_\phi(FIFO')$ ▷ Random Resource Costs
aggregate *grad* with *loss* given $\hat{\mathbf{n}}$ and $\hat{\mathbf{n}}'$ ▷ Distillation

Return: *grad*

Algorithm 2 and Algorithm 3 share a similar structure based on knowledge distillation and the sandwich rule. The difference between the two algorithms is how the respective networks are distilled. Algorithm 2 distills the navigation network to operate with a varying number of nodes in each hidden layer, while Algorithm 3 distills the network to operate with varying power levels used to acquire input sensor observations (which is likened to slimming the input layer). We extrapolate that similar approaches can be used to distill the network to operate with other varying attributes. This is why we use the name *NaviSlim* for both the variants, as *NaviSlim* fuses navigation with a form of "slimming" applied to different attributes of the navigation network.

VI. NaviSlim: Navigation Module

The objectives of the navigation module are two fold: 1) navigate along a length-optimal path from a spawn position to the goal, while avoiding collisions, and 2) execute the underlying navigation model using variable sensing and computing parameters. The process for training the underlying neural network is outlined below:

1) Collect ground truth length-optimal paths, P, using an A-star [33] shortest path algorithm.
2) Acquire sensor observations at each time step in P.
3) Use supervised learning with knowledge distillation to train a dynamic neural network that maps the FIFO queue of recent observations to navigation motions, $\mathbf{n}$.
4) Freeze the navigation module, as to no longer update the trainable network parameters.
5) Evaluate for a successful model by deploying g_ϕ to the test bed environment.

A. A-star Shortest Path Algorithm

Shortest path algorithms are used to solve the problem of finding length-optimal paths between two points. Graph algorithms are a subset of shortest path algorithms where the problem can be constructed in a graph structure with vertices and edges. A-star [33] is a flavor of graph shortest path algorithms, that is guaranteed to find the optimal solution without having to traverse every possible path. We implement A-star by reconstructing the Blocks map into a graph where each vertex corresponds to a spatial point on the map, and edges define if an adjacent position is valid (*i.e.*, does not have an object in it). The cost of a path is simply the total distance traveled.

We partition the Blocks map into regions used for a training, validation, and testing set. We then use A-star to find the optimal paths, *P*, in each set. The test bed environment is used to collect simulated sensor observations from Microsoft AirSim at each timestep in *P*. This results in a dataset of hard targets that map $FIFO \mapsto \mathbf{n}$ for each time step in each path within *P*. These are then used to train the navigation network.

B. Training the Neural Network

Given a set of maps, $FIFO \mapsto \mathbf{n}$, over optimal paths, *P*, the training procedure for the navigation neural network, g_ϕ, is outlined in Algorithm 4. The function $supervised\_distillation$ is an argument to Algorithm 4 that controls how supervised learning works in unison with knowledge distillation as earlier defined in Section V-B. The training set is used to drive the error gradient, the validation set is used to trigger early stopping to mitigate overfit, and the test set is held out to later evaluate the navigation model. We use mean squared error as our loss function.

Algorithm 4 Training a Navigation Neural Network

Input: training set, validation set, randomly initialized g_ϕ, $supervised\_distillation$ function, $loss$ function.

while not converged **do**
  set running error gradient to zero
  sample batch of $[FIFO \mapsto \mathbf{n}]$ pairs from training set
  for each $FIFO \mapsto \mathbf{n}$ in batch... **do**
    grad = $supervised\_distillation(g_\phi, FIFO, \mathbf{n}, loss)$
    aggregate grad into running error gradient
  update ϕ using optimizer with aggregated error gradient
  use the validation set to check for early stopping

Return: trained g_ϕ

After the navigation neural network is trained using Algorithm 4, the optimized parameters, ϕ, are frozen and not updated again. We evaluate if a trained navigation model is successful by deploying g_ϕ into the AirSim environment and measuring the percent of paths in the test set that successfully reach their goal – when using the super-network and Algorithm 1 but without an auxiliary module, h_ψ. Thus, we are only evaluating the navigation prowess of the static super-network without adaptation.

VII. NaviSlim: Auxiliary Module

The objective of the auxiliary module is to control adaptation of either **c** or **s** as applied to the navigation module. The following steps are taken to create the auxiliary module:

1) Successfully train a navigation model, g_ϕ.
2) Create a reward function that penalizes the navigation algorithm for taking sub-optimal paths.
3) Train the auxiliary model, h_ψ, using a Twin Delayed Deep Deterministic Policy Gradient (TD3) [34] deep reinforcement learning algorithm. The objective is to map the FIFO queue to adaptation parameters, **c** or **s**.
4) Evaluate for a successful model by deploying h_ψ with g_ϕ to the test bed environment.

A. Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) consists of an agent that will be episodically traversing an environment during the training process, by taking actions which result in an evaluated reward. The policy is learned during training that maps observations to optimal actions which in essence maximize the rewards. The word "deep" simply refers to using a deep neural network as the policy mechanism.

B. Q-learning

The Q-value, derived from the Bellman equation [35], is an estimation of the aggregation of immediate and long term rewards in an episode – where higher Q-values correspond to more effective actions. Given a reward function, $r(state)$, that inputs the *state* found after executing **n**, Equation (4) shows the optimization problem used to train h_ψ:

$$\begin{aligned} &\max_{\psi} \; < \textit{Q-value} >, \\ &\textit{Q-value}(\mathbf{p}, t) = \Sigma_{i=t}^{length(\mathbf{p})} \gamma^{i-t} * r(state^{(i)}), \end{aligned} \tag{4}$$

where $<>$ denotes the expected Q-value resulting from using g_ψ over all paths and time steps, $state^{(i)}$ corresponds to state variables used to calculate the reward at the i^{th} time step from path $\mathbf{p}$, and γ is a parameter called the discount factor which applies a decaying penalty to long term rewards.

C. Reward Function

Designing a reward function is highly non-trivial, as it directly affects the stability and convergence of the training algorithm, along with the learned policy. We use a random walk algorithm to estimate the behavior of a proposed reward function. Each iteration takes a mean step closer to an arbitrary goal initially set 100 meters away, with Gaussian noise added at each step. We then recursively calculate the rewards at each step to estimate the Q-value corresponding to the theoretical episode. This process is used to design and adjust the coefficients in the reward function by evaluating them against the average of several random walks. We, then, select the following reward function:

$$r = \begin{cases} -\lambda_o & collision \\ \lambda_g & goal \\ \lambda_d tanh(d) - \lambda_t - \lambda_c \rho - \lambda_s (p_f + p_d) & otherwise \end{cases}, \tag{5}$$

where the constants λ_x are non-negative weights applied to different state variables. The first two conditions assign large reward values to terminal states that are encountered either when the drone reaches the goal or collides with an object. The third condition assigns a set of penalties applied to intermediate states to encourage shorter paths. The first term of the third condition applies either a reward or penalty based on d, the change in distance between the drone and goal in between time steps. The second term is a constant penalty for the number of steps taken in an episode. The third term applies a penalty based on the amount of resources allocated to sensing, as controlled by ρ. The fourth term applies a penalty based on the amount of resources allocated to computing, as controlled by p_f and p_d.

D. TD3

Among the many available DRL algorithms, we select a Twin Delayed Deep Deterministic Policy Gradient (TD3) [34] algorithm; as it allows a continuous multivariate observation space, a continuous multivariate action space, and outperforms other DRL algorithms that we tested. TD3 is based on an actor-critic double deep Q-learning paradigm with clipped action noise and delayed policy updates, consisting of a series of six neural networks used in the training process. The actor network, h_ψ, is the policy mechanism that inputs the FIFO queue of recent observations and outputs either $\mathbf{c}$ or $\mathbf{s}$. The target actor network is used to estimate the next action, while noise is added to that action to smooth out training and account for error. The critic network estimates the Q-value, since future states are unknown. The target critic network is used to measure error in the critic network, since we can not calculate all possible future states. The target critic and target actor networks are initially clones of the critic and actor network, respectively, but the weights vary over time and every few episodes the target network weights are updated using a Polyak weighted average.

Two critic networks are used to help stabilize training by selecting the minimum Q-value between both critics, which is referred to as double deep Q-learning [36]. When training a neural network with a TD3 algorithm, initially actions are taken at random to explore the environment. Then the neural network is increasingly used to predict actions as to exploit the policy being learned. Even though six neural networks are used in the training process, only the actor network, h_ψ, is executed at the deployment stage.

E. Curriculum Learning

A common difficulty in training a neural network with DRL arises when the initial task is too difficult. This causes poor rewards early in the learning process which can stagnate training. A solution is to start with an easier task, then progressively increase the difficulty. We implement curriculum learning by starting with a small distance between spawn and goal positions, then incrementally increase this distance as enough evaluation paths successfully reach the goal.

F. Training the Neural Network

We train the auxiliary network, h_ψ, using Algorithm 5. Training the auxiliary network is the main bottleneck for training *NaviSlim*, because it can take several days before finishing. Further, the parameters to Algorithm 5 are highly sensitive (some of which are omitted), which requires ample time to explore them. Having several computers to run training in parallel with different parameters is highly advantageous. We evaluate if a trained auxiliary model is successful by deploying g_ϕ and h_ψ into the AirSim environment and measuring the percent of paths in the test set that successfully reach their goal when using Algorithm 1. Further, we insure the constraint holds in Equation 1 since this is not guaranteed by Equation 4.

VIII. Results

First we explore the size, the number of hidden layers and nodes in each layer, of the navigation super-network. We test each configuration by training the navigation model 10 random times with different seeds, and take the seed with the best error as measured from a static test set. Fig. 6 shows results of a hyper parameter grid search, displaying the Root Mean Squared Error (RMSE) between length-optimal motions as found from A-star, $\mathbf{n}$, and those predicted from g_ϕ, $\hat{\mathbf{n}}$. The darker region of Fig. 6 indicates that, as expected, using a larger super-network results in a reduced navigation error. The navigation model is then deployed in the simulation test bed, using Microsoft AirSim.

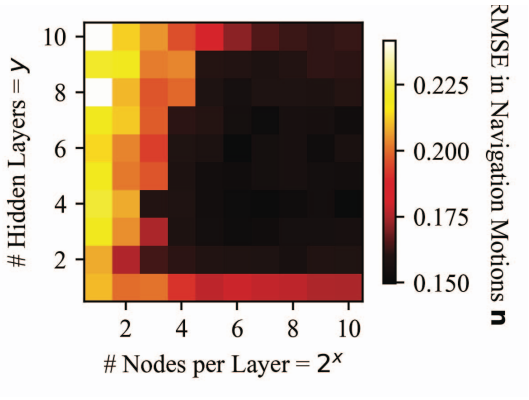

Fig. 6. Results of a hyper-parameter grid search that explores different neural network sizes of the navigation model, g_ϕ. Shown is the Root Mean Squared Error (RMSE) calculated between the length-optimal navigation motions found from A-star and those predicted from a trained g_ϕ.

Fig. 7 shows the percentage of evaluation paths which successfully reach the goal versus distance to goal. The percentage of successful paths drops with increasing distance, as expected. The navigation models perform remarkably well when deployed to the more complex City map even though they are only trained using samples from the simple Blocks map. This illustrates the prowess, and generalization, of our navigation training methods. Typical approaches would stop here, and have a static navigation network with fixed computing and sensing required by that of the most difficult scenario.

Algorithm 5 Training an Arbitrary Auxiliary Network

Input: training region, evaluation set, trained navigation network g_ϕ, randomly initialized auxiliary network h_ψ

create a replay buffer of set length to store episodic data
deep copy h_ψ to make target actor network
if *NaviSlim-C* **then**
  $\mathbf{a} \coloneqq \mathbf{c}$ ▷ Actions will Reduce Computational Costs
if *NaviSlim-S* **then**
  $\mathbf{a} \coloneqq \mathbf{s}$ ▷ Actions will Reduce Sensing Costs
let h_{rand} be a random generator of **a**
randomly initialize 2 critic networks and clone target critics
Each '?'-Boolean bellow is parameterized by i.
for $i = 1$; $i <=$ max episodes; $i = i + 1$ **do**
  generate random spawns and goals from the train region
  if explore? **then**
    E = Algorithm 1 with h_{rand} ▷ exploration
  else
    E = Algorithm 1 with h_ψ ▷ exploitation
  add (E, t) data from each time step in E to replay buffer
  if train? **then**
    sample batch of (E, t) data from replay buffer
    for each (E, t) data in batch **do**
      $\tilde{\mathbf{a}} =$ target_actor($LIFO_{t+1}$)
      $\tilde{\mathbf{a}} = \tilde{\mathbf{a}}$ + clipped Gaussian noise
      $q = r_t + \gamma *$ min(target_critics($LIFO_{t+1}$, $\tilde{\mathbf{a}}$))
      for each critic **do**
        $\hat{q} =$ critic($LIFO_t$, $\mathbf{a}_t$)
        calculate $loss$ between q and $\hat{q}$
        update critic with optimizer and $loss$
      if update? **then**
        $loss$ = -1 * critics[0]($LIFO_t$, $\mathbf{a}_t$).mean()
        update h_ψ with optimizer and $loss$
        Polyak update target actor and target critics
  if evaluate? **then**
    for each spawn and goal pair in evaluation set **do**
      E = Algorithm 1 with h_ψ
    if enough evaluation paths successful **then**
      increase distance between spawn and goal
      terminate training if that distance is large enough

Return: trained h_ψ

However, the following results come from experimentation of our novel approach to adapt computing and sensing with *NaviSlim*.

Next we still evaluate the RMSE between length-optimal motions as found from A-star, $\mathbf{n}$, and those predicted from g_ϕ, $\hat{\mathbf{n}}$; however, we now select one navigation model (using the seed with best RMSE) and evaluate how the error changes with varying values of the slimming factor, ρ. Fig. 9 shows navigation RMSE as a function ρ for each of the training, validation, and testing sets. This shows how the global RMSE increases with decreased ρ, warranting that if we use a static sub-network within the navigation super-network we would receive sub-par accuracy. The novelty of our approach is to intelligently and dynamically select the value of ρ, given context, as to not detrimentally decrease navigation accuracy – which we experiment with next by training and evaluating the auxiliary network, h_ψ.

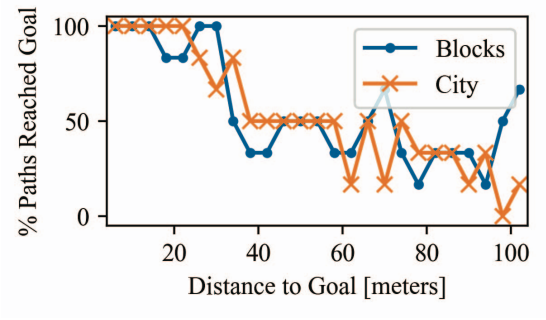

Fig. 7. Results of a trained navigation model, g_ϕ, evaluated in the test bed environment using both AirSim maps. This figure shows the percent of evaluation paths that reach their goal as a function of starting distance.

First we train the auxiliary network, h_ψ, for *NaviSlim-C* which predicts the slimming factor, ρ, that directly controls the computing resources required to execute the navigation network. We evaluate *NaviSlim-C* on three scenarios with increasing difficulty: 1) the Blocks map with only horizontal motion allowed, 2) the Blocks map with vertical motion unlocked, and 3) the City map with vertical motion locked. We compare these scenarios as we hypothesize that the computing resources, represented by ρ, required to run the navigation network will increase with increased difficulty. This hypothesize is warranted with the evidence provided in Table I which shows the average adaptation values used to reduce resource allocation, as learned by *NaviSlim*, and evaluated on the test set. Listed are three scenarios with increasing difficulty. Where the "**Scen.**" column is the scenario being tested for, and the "η" column shows the relative resource expenditure reduction either for the number of active navigation network parameters, m, or sensor power level, w. All values shown in Table I are calculated using the test set. We see a significantly higher mean value of ρ between each of these scenarios. The η values in Table I are percent decreases over the larger super-network. Note that the numbers reported in Table I are for all evaluations done at the end of each curriculum learning step before increasing the distance between spawn and goal in Algorithm 5, and all evaluation paths successfully reached their target position – thus navigation accuracy is maintained.

TABLE I

Scen.	Map	Motion	$\bar{\rho}$	$\eta(m)$	$\bar{p_f}$	$\bar{p_d}$	$\eta(w)$
(1)	Blocks	Horizontal	0.61	57%	-	-	-
(2)	Blocks	Vertical	0.88	86%	2.6	2.2	80%
(3)	City	Horizontal	0.93	92%	2.9	0.73	61%

Fig. 10 shows the learned average values of ρ at each position in the Blocks map, using the test set. This figure is

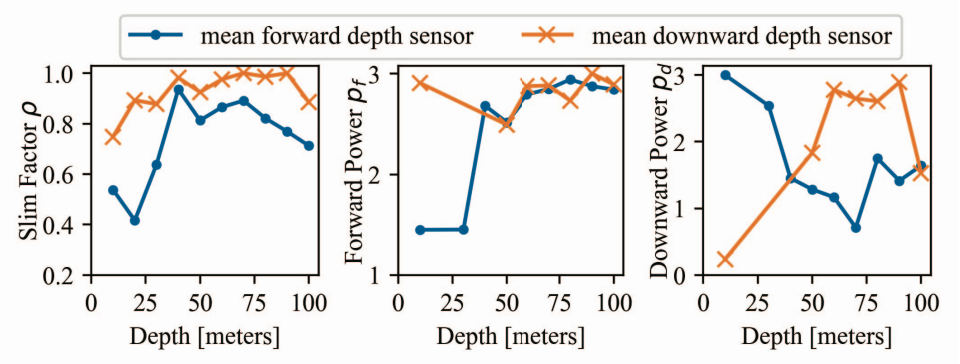

Fig. 8. Mean results of the different adaptability variables that control various resource allocations as predicted from the auxiliary network, h_ψ, over all values from successful paths in the test set. A larger slim factor, ρ, corresponds to increased computation costs (we find this to be time and energy with fixed power) needed to run the navigation network, g_ϕ. A larger power level, either p_f for the forward facing depth sensor or p_d for the downward facing depth sensor, corresponds to increased sensing costs (proven in literature to be both time and power [26]) needed to acquire observations from the sensor array. The x-axis shows the mean values returned from each depth sensor, in meters, and is binned at increments of 10 meters (note that some bins are missing, this is just circumstantial). The mean depth values give context clues to the surrounding environment, and this figure shows how the adaptability variables respond to them. Note that just for this figure, the displayed mean depth values from each sensor are always calculated using the maximum power levels to best and uniformly represent the context, even though different power levels are likely used as input into the auxiliary network, h_ψ, during evaluation.

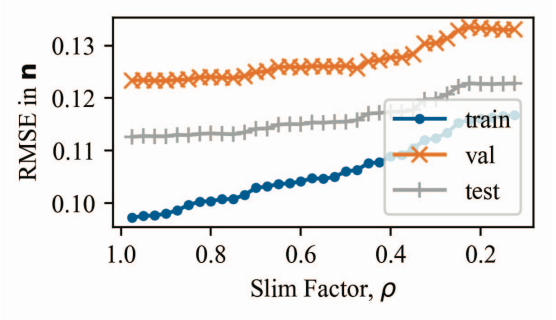

Fig. 9. Results after reducing the resources allocated to computing a trained navigation model, g_ϕ. Shown is the Root Mean Squared Error (RMSE), calculated between the length-optimal navigation motions found from A-star and those predicted from g_ϕ, as a function of the slimming factor, ρ, used to control the number of active parameters, m, in g_ϕ. Not shown here, is that m exhibits a quadratic decrease with ρ.

useful to see the dynamic nature of *NaviSlim* and its context-aware behavior based on both the surrounding environment and maneuvers required to avoid collision with an object. We further isolate the context clues versus learned behavior as illustrated in Fig. 8, where we show the mean depth captured by each sensor versus ρ. Generally, when the navigation path is more convoluted there is a correspondingly high ρ, which is expected behavior for more complex navigation and maneuvers.

Interestingly, as the distance between the drone and objects increases, so does the value predicted for ρ from h_ψ. A possible explanation is that when objects are close, higher-level logic is not needed as the subspace of possible motions, that can be executed without colliding with that nearby object, shrinks. Similarly, after about 50 meters, ρ begins to decrease with further increasing measured depth, likely because the environment becomes more open (less nearby objects and more open physical space) and the subspace of possible motions shrinks as more sophisticated maneuvers are not needed.

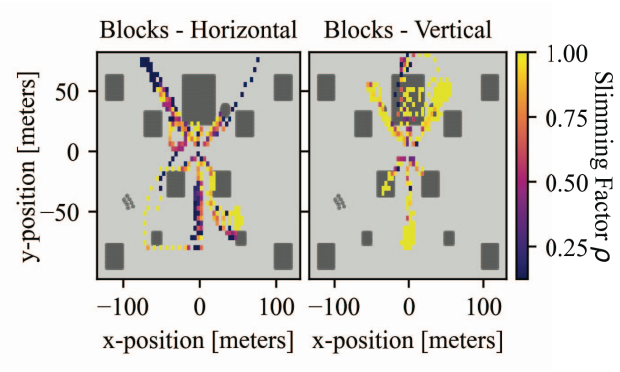

Fig. 10. An aerial 2D view of the AirSim Blocks map overlaid with the average slimming factor, ρ, predicted from the auxiliary network, h_ψ, at each position. Included in the mean are only values from successful paths, and only those from the test set. The left panel shows a scenario that we consider where the drone can only move horizontally, and the right panel shows another scenario where vertical motion is unlocked. The darker gray shapes indicate objects the drone can collide with, however note on the right panel the drone is also flying over these objects.

Next, we measure the actual differences in the resource cost (time, power, and energy) between using and not using *NaviSlim* on a microprocessor similar to that typically deployed on micro-drones. We use a Jetson Nano with a Quad-core ARM Cortex-A57 MPCore processor and 4 GB 64-bit LPDDR4 1600MHz 25.6 GB/s memory. We compare relative resource costs by passing a static set of observations through: (1) *NaviSlim*, including both the auxiliary and navigation modules, with the learned values for ρ; then (2) only the navigation network but with $\rho = 1$ (*i.e.*, just the static super-network without the auxiliary network). We measure the ratio difference between each resource as $\frac{v-u}{v}$, where u is the

resource cost associated with (1) *NaviSlim* and *v* is the resource cost associated with (2) the super-network. The size of the auxiliary hidden layers was fixed at [32, 32], while the size of the navigation hidden layers was varied - as shown in Fig. 11 which shows the relative speedup associated with *NaviSlim*.

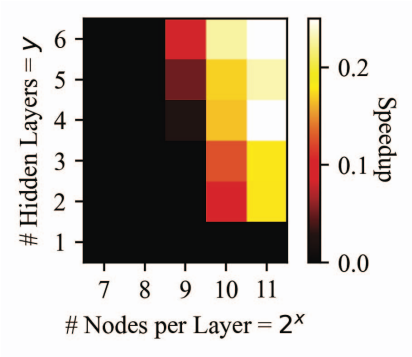

Fig. 11. Test set results ran on a Jetson Nano to measure the relative speedups between using and not using *NaviSlim*, as a function of navigation network size – while using a fixed auxiliary network that has 2 hidden layers with 32 nodes in each. A speedup of zero (black grid spaces) indicates that using *NaviSlim* takes more time to run than not using it, which is expected for smaller navigation network sizes due to the overhead of the auxiliary network.

We find that the difference in power consumption is negligible, with a mean relative difference of 0.005 and standard deviation of 0.07. Since energy consumption is power multiplied by time, this shows that execution time is the dominating factor in energy consumption. Note that reducing execution time of the navigation network also improves reaction time. Fig. 11 shows that smaller networks can actually result in increased execution time when using *NaviSlim* – as indicated by the lower left corner with black grid spaces. This behavior is expected, since the overhead of the auxiliary network does not justify the small size of the navigation network. However, larger networks result in decreased run times - as indicated by the upper right corner with non-black grid spaces. This region is characterized by a positive speedup and also overlaps with the region with lowest navigation error as shown in Fig 6. This proves that we can mitigate the larger execution times inherent with larger neural networks, which is needed to achieve lower navigation error, by using *NaviSlim* – noting that this speedup increases with size of the navigation network.

Next we evaluate *NaviSlim-S* which dynamically controls p_f and p_d, the power levels of the forward facing depth sensor and downward facing depth sensor, respectively. Table I lists the average resolution levels for scenarios (2) and (3) after evaluating the trained auxiliary model, h_ψ, with the test set. The mean value of p_f is greater than p_d for each scenario, which is intuitive because most drone maneuvers involve moving horizontally rather than vertically. When using only horizontal motion, the downward facing depth sensor is almost completely turned off, with a mean p_d value of 0.73.

Fig. 12 and Fig. 8 show the learned sensor power levels between the two scenarios as a function of context. From Fig. 8, we see that p_f is independent of the downward depth sensor observations, but has a clear dependence on the forward depth sensor observations – which is most intuitive. Interestingly, as the values returned from the forward depth sensor increase (indicating forward facing objects are further from the drone) so does p_f, which is a similar relationship we earlier observed with ρ – warranting that less resources are required when an object(s) is very near the drone. We observe another intuitive relationship that p_d increases as the values returned from the downward depth sensor increase (indicating the drone is relatively higher in the air than objects below it). This relationship holds until the mean downward depth reaches some critical point at which p_d then decreases.

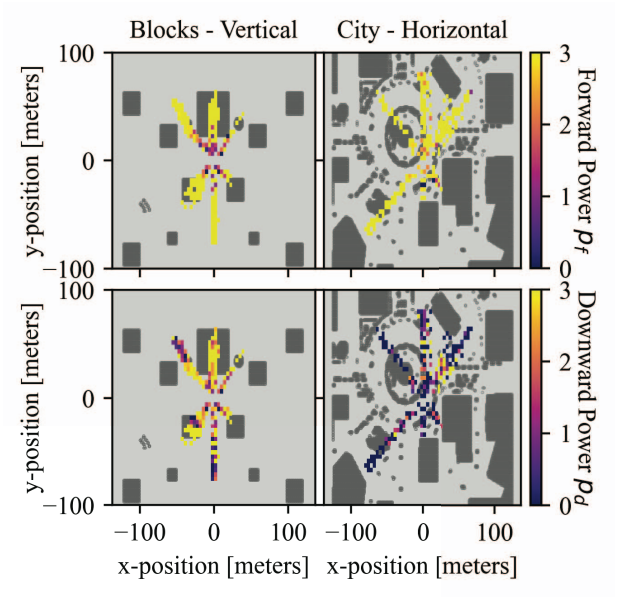

Fig. 12. An aerial 2D view of the two AirSim maps overlaid with the respective power levels, p_f and p_d, corresponding to the forward and downward facing depth sensors as predicted from the auxiliary network, h_ψ, at each position. Included in the calculations of each mean value are only those from successful paths in the test set. The left panels show results on the Blocks map with vertical motion unlocked. The right panels show results on the City map with vertical motion locked. The darker gray shapes indicate objects, however note that on the left panels the drone is also flying over these objects and some objects on the right panels are moving with time.

IX. Conclusions

We presented *NaviSlim*, the first of its kind to dynamically scale computing and sensing used by a neural model for navigation of a (micro-)drone with extreme resource constraints. We detailed the training procedure used to obtain successful models that can safely navigate between points A and B, while using variable computing and sensing. We showed that an auxiliary neural network can successfully learn to map context to computing and sensing required by the difficulty of the current scenario. This is a novel evolution over static networks that must match computing and sensing of that required by the most difficult scenario. We showed that when deploying *NaviSlim* to our test bed environment interfaced with the drone simulation tool Microsoft Airsim, we reduced

average navigation model complexity between 57% and 82%, and sensing power levels between 61% and 80%, as compared to that of the static navigation network required to fulfill the same objectives. We posit that such methods will pave the way in a new evolution of dynamic neural networks used in resource constrained environments.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation under grants CCF 2140154, CNS 2134567, and DUE 1930546.

REFERENCES

[1] T. Elmokadem and A. V. Savkin, "Towards fully autonomous uavs: A survey," *Sensors*, vol. 21, no. 18, p. 6223, 2021.

[2] L. O. Rojas-Perez and J. Martínez-Carranza, "On-board processing for autonomous drone racing: An overview," *Integration*, vol. 80, pp. 46–59, 2021.

[3] K. A. Irizarry, Z. Zhang, C. Stewart, and J. Boubin, "Scalable distributed microservices for autonomous uav swarms," in *Proceedings of the 23rd International Middleware Conference Demos and Posters*, 2022, pp. 1–2.

[4] R. Clark, G. Punzo, G. Dobie, R. Summan, C. N. MacLeod, G. Pierce, and M. Macdonald, "Autonomous swarm testbed with multiple quadcopters," in *1st World Congress on Unmanned Systems Enginenering, 2014-WCUSEng*, 2014.

[5] A. Anwar and A. Raychowdhury, "Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning," *IEEE Access*, vol. 8, pp. 26 549–26 560, 2020.

[6] A. Mehra, M. Mandal, P. Narang, and V. Chamola, "Reviewnet: A fast and resource optimized network for enabling safe autonomous driving in hazy weather conditions," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4256–4266, 2020.

[7] D. J. Yeong, G. Velasco-Hernandez, J. Barry, and J. Walsh, "Sensor and sensor fusion technology in autonomous vehicles: A review," *Sensors*, vol. 21, no. 6, p. 2140, 2021.

[8] K. Amer, M. Samy, M. Shaker, and M. ElHelw, "Deep convolutional neural network based autonomous drone navigation," in *Thirteenth International Conference on Machine Vision*, vol. 11605. SPIE, 2021, pp. 16–24.

[9] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable neural networks," *arXiv preprint arXiv:1812.08928*, 2018.

[10] Y. Matsubara, M. Levorato, and F. Restuccia, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–30, 2022.

[11] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[12] Epic Games, "Unreal engine." [Online]. Available: https://www.unrealengine.com

[13] R. Madaan, N. Gyde, S. Vemprala, M. Brown, K. Nagami, T. Taubner, E. Cristofalo, D. Scaramuzza, M. Schwager, and A. Kapoor, "Airsim drone racing lab," in *Neurips 2019 competition and demonstration track*. PMLR, 2020, pp. 177–191.

[14] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing with deep reinforcement learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 1205–1212.

[15] S. Krishnan, B. Boroujerdian, W. Fu, A. Faust, and V. J. Reddi, "Air learning: a deep reinforcement learning gym for autonomous aerial robot visual navigation," *Machine Learning*, vol. 110, pp. 2501–2540, 2021.

[16] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 7512–7519.

[17] A. Anwar and A. Raychowdhury, "Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning," *IEEE Access*, vol. 8, pp. 26 549–26 560, 2020.

[18] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing with deep reinforcement learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 1205–1212.

[19] H. D. Whyte, "Simultaneous localisation and mapping (slam): Part i the essential algorithms," *Robotics and Automation Magazine*, 2006.

[20] A. V. Malawade, T. Mortlock, and M. A. Al Faruque, "Hydrafusion: Context-aware selective sensor fusion for robust and efficient autonomous vehicle perception," in *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2022, pp. 68–79.

[21] M. Odema, L. Chen, M. Levorato, and M. A. Al Faruque, "Testudo: Collaborative intelligence for latency-critical autonomous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[22] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[23] J. Yu and T. S. Huang, "Universally slimmable networks and improved training techniques," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1803–1811.

[24] C. Li, G. Wang, B. Wang, X. Liang, Z. Li, and X. Chang, "Dynamic slimmable network," in *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, 2021, pp. 8607–8617.

[25] Z. Jiang, C. Li, X. Chang, L. Chen, J. Zhu, and Y. Yang, "Dynamic slimmable denoising network," *IEEE Transactions on Image Processing*, vol. 32, pp. 1583–1598, 2023.

[26] S. Lee and D. Park, "Efficient power control using variable resolution algorithm for lidar sensor-based autonomous vehicle," in *2021 18th International SoC Design Conference (ISOCC)*. IEEE, 2021, pp. 341–342.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[29] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 12 348–12 355, 2021.

[30] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[31] A. Kumar, Z. Li, J. Zeng, D. Pathak, K. Sreenath, and J. Malik, "Adapting rapid motor adaptation for bipedal robots," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 1161–1168.

[32] H. Qi, A. Kumar, R. Calandra, Y. Ma, and J. Malik, "In-hand object rotation via rapid motor adaptation," in *Conference on Robot Learning*. PMLR, 2023, pp. 1722–1732.

[33] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[34] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[35] G. Tesauro *et al.*, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[36] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.