

UNIVERSITY OF CALIFORNIA, IRVINE
& SAN DIEGO STATE UNIVERSITY

Adaptive Dynamic Deep Neural Networks
and Hierarchical Control for Autonomous Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computational Science

by

Timothy Kevin Johnsen

Dissertation Committee:
Professor Marco Levorato (UCI), Co-Chair
Associate Professor Ahmad Bani-Younes (SDSU), Co-Chair
Professor Jasper Vrugt (UCI)
Distinguished Professor Pramod Khargonekar (UCI)
Associate Professor Jun Chen (SDSU)

TABLE OF CONTENTS

	Page
LIST OF FIGURES	iv
LIST OF TABLES	viii
LIST OF ALGORITHMS	ix
ACKNOWLEDGMENTS	x
VITA	xi
ABSTRACT OF THE DISSERTATION	xiv
1 Introduction & Testbed	1
1.1 Simulation and Hardware-In-the-Loop	2
1.2 AI-Accelerated Servers	5
1.3 High Performance Computing Clusters	6
2 Hierarchal Navigation Framework	7
2.1 Low-Level Controller	10
2.2 Mid-Level Policy	10
2.2.1 A* Shortest Path	12
2.2.2 Imitation Learning (IL)	13
2.2.3 Deep Reinforcement Learning (DRL)	14
2.3 Monocular Depth Estimation (MDE)	20
2.4 Split Computing	21
2.5 High-Level Policy	22
2.5.1 World Building & Ray Tracing	22
2.5.2 Multimodal Large Language Models (MLLM)	24
2.6 Experimental Results	28
2.7 Summary	35
3 Adaptive Dynamic Deep Neural Networks	36
3.1 Problem Formulation	37
3.2 Background and Related Work	38
3.3 Theoretical Motivation & Hypothesis Formulation	40

3.4	Key Observations & Proposed Solutions	43
3.5	Dynamic Deep Neural Network Architectures	44
3.5.1	Embedded Networks & Slimmable	44
3.5.2	Multi-Branch Networks	49
3.6	Auxiliary Models	50
3.7	Frameworks for Navigation	52
3.7.1	NaviSlim	53
3.7.2	NaviSplit	62
3.7.3	SmartDepth	68
3.7.4	CADENCE	73
3.8	Summary	83
4	Mitigating Real-World Artifacts	85
4.1	Denoising AutoEncoder with Monte Carlo Dropout	85
4.2	Multimodal Neural Networks	92
4.3	Global and Local Spatio-Temporal Data	96
4.4	Summary	106
5	Conclusions	108
	Bibliography	112

LIST OF FIGURES

	Page
1.1 Microsoft AirSim drone simulator, built on Unreal Engine	2
1.2 Microsoft AirSim maps.	3
1.3 Hardware-In-the-Loop (HIL) system.	4
1.4 High Performance Computing clusters, on the left is The University of Arizona’s HPC resources at the Research Data Center (https://www.datacenterdynamics.com/en/news/asu-to-launch-sol-supercomputer-this-summer/), and on the right is Lawrence Berkeley National Lab’s National Energy Research Scientific Computing Center (https://www.nersc.gov/news-and-events/news/berkeley-lab-deploys-next-generation-supercomputer-perlmutter-bolstering-u-s-scientific-research)	6
2.1 Diagram illustrating the hierarchical navigation framework. The general DNN framework shown is used by either the DQN or TD3 DRL algorithms used to create mid-level policies. Shown in this figure is an example 2D depth map, however the observation space corresponding to O_t may change based on the configured sensor array onboard the robot. Also shown here is a drone. A different robot embodiment would have the same DNN structure but a potentially different action space.	9
3.1 Embedded network with a selectable path in grey within the larger super-network comprised of a finite number of paths.	45
3.2 Procedure used to ”slim” an MLP. The slimming factor 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 (the super-network). Quantity $\rho = 0.3$ so that 2 nodes are deactivated in the first layer and 1 in the second (activating a sub-network). Note that all of the weights connected to any deactivated nodes, represented by the lines in between the hidden layers, are also severed. © [2024] IEEE. Reprinted, with permission, from [42]. . . .	46
3.3 Active channels determined by a slimming factor, $\rho = 0.5$, set during inference of an example slimmable CNN with four maximum channels.	48
3.4 Multibranch network with a switch gate controlling which stem is activated in a forward pass.	49

3.5	High-level schematics of NaviSlim, showing a forward and downward facing depth sensor with a neural network handling navigation actions. © [2024] IEEE. Reprinted, with permission, from [42].	53
3.6	Results of a trained navigation model, evaluated in the testbed environment using the AirSim maps Blocks and CityEnviron. This figure shows the percent of evaluation paths that reach their goal as a function of starting distance. © [2024] IEEE. Reprinted, with permission, from [42].	57
3.7	Shown is the Root Mean Squared Error (RMSE), calculated between the length-optimal navigation motions found from A* and those predicted from the trained navigation network, as a function of the slimming factor, ρ . © [2024] IEEE. Reprinted, with permission, from [42].	58
3.8	An aerial 2D view of the AirSim Blocks map overlaid with the average slimming factor, ρ , predicted from the auxiliary network at each position. 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. © [2024] IEEE. Reprinted, with permission, from [42].	59
3.9	Mean results of adaptive variables as predicted from the auxiliary network, as a function the mean value of the depth sensor data input to the network during predictions. © [2024] IEEE. Reprinted, with permission, from [42].	60
3.10	Jetson Nano benchmarks, measuring the relative speedups between using and not using the auxiliary model, as a function of navigation network size – while using a fixed auxiliary network size. © [2024] IEEE. Reprinted, with permission, from [42].	60
3.11	The navigation action RMSE as a function of the navigation network size. © [2024] IEEE. Reprinted, with permission, from [42].	61
3.12	Several different split points with an encoder and decoder are injected into a teacher model to make multiple student model branches capable of split computing – of which an auxiliary model selects from given perceived context. Extracted depth maps are input to the navigation model that outputs motion actions used during drone navigation. © [2024] IEEE. Reprinted, with permission, from [41].	64
3.13	Comparing various compressed data sizes, corresponding to different models, versus resulting depth extraction error. The markers from left to right: for the Bottleneck_V1 and Bottleneck_V2 models, range between a reduction in channels of [12, 24, 64]; for baseline models, range between a reduction in channels of [2, 4, 8, 16, 32]; and for JPEG models, range in quality of compression from 5 to 95. © [2024] IEEE. Reprinted, with permission, from [41].	65
3.14	Mean value of the gate control, $\hat{c} := G$, for all successful paths while using an auxiliary model to adapt and select $\hat{c}$ at each time step. This a demonstration of <i>NaviSplit</i> , which was trained and evaluated on the AirSimNH map. © [2024] IEEE. Reprinted, with permission, from [41]	67
3.15	High level overview of using MADP to make one depth map prediction at time t , and at a proposed camera pose, based on the previous two steps. © [2025] IEEE. Reprinted, with permission, from [114].	68

3.16	SmartDepth system model, illustrating the various pipelines to select from, either $A \rightarrow C$ or $B \rightarrow C$, in one observe-compute-action step. © [2025] IEEE. Reprinted, with permission, from [114].	69
3.17	<i>SmartDepth</i> results illustrating the trade-off between accuracy and efficiency as a function of X . © [2025] IEEE. Reprinted, with permission, from [114]. .	71
3.18	Illustrated is the flow of data from raw sensor acquisition to intelligent decision-making through the full autonomy stack, CADENCE. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	74
3.19	Example RGB image, ground truth depth map, and predicted depth maps for both static and slimmable networks with varying network sizes. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	77
3.20	Test R^2 -scores of various trained MDE network configurations. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	78
3.21	Learning curve from training the navigation-and-adaptation policy. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	79
3.22	Final evaluations on the static holdout test set. The x-axis shows the energy expended while computing the MDE network over all paths. The hollow blue markers correspond to using a SoTA static MDE network with size α , and a navigation DQN policy that predicts A . The solid orange markers correspond to using a slimmable MDE network with a navigation-and-adaptation DQN that predicts both A and ρ . The green dashed line corresponds to a navigation DQN that bypasses MDE and instead directly uses the AirSim ground truth depth maps. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	80
3.23	Average slimming factor when computing the slimmable MDE network at each of the given (x, y) locations. Compared is the drone moving in an arbitrary direction versus moving only to the left. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	81
3.24	Correlation between the adaptation parameter ρ and two environmental factors: obstacle density, as measured by the mean value of the FIFO queue of recent depth map predictions, and distance to target. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].	82
4.1	Resulting distributions on a single data point after Monte Carlo sampling, for the non-DAE model $h_\psi(\mathbf{x})$ and the DAE model $h_\psi(\tilde{\mathbf{x}})$	89
4.2	Expected versus predicted values for the DAE, used to repair artificially corrupted sensor observations. Each panel above is separated into one of the 15 sensors, which can be artificially corrupted and subsequently repaired. The results shown are from the testing set with sampled artificial corruption (both Gaussian and Bernoulli) and without sampled model uncertainty. © [2025] IEEE. Reprinted, with permission, from [38].	91

4.3	Expected versus predicted values for the ET models, either with or without DAE-treated training methods using artificial corruption. The results shown are from the testing set with sampled artificial corruption (both Gaussian and Bernoulli) and without sampled model uncertainty. © [2025] IEEE. Reprinted, with permission, from [38].	92
4.4	Flow of each modality through either a logistic regression, multi-layer perceptron, or multimodal neural network. © [2025] Reprinted, with permission, from [39].	94
4.5	Learning curves for training mineral classifiers with a multimodal neural network, by varying when the single-modality stems are unfrozen. © [2025] Reprinted, with permission, from [39].	97
4.6	Spatio-temporal averaged values of each variable correlated to ET, and the two ground truth labels of ET.	100
4.7	CNN Unet structure of the 2D model, where data flows from left to right. Below each block is the output dimensions, where c denotes channels, w denotes width, and h denotes height.	102
4.8	CNN structure of the 1D model, where data flows from left to right. Below each block is the output dimensions, where c denotes channels, w denotes width, and h denotes height.	103
4.9	Spatial correlations in R^2 scores (shown on the z-axis in grayscale) from predictions on the test set as evaluated on the pretrained 2d model. At each (x, y) coordinate, is the R^2 score calculated over all times for those coordinates.	105
4.10	Temporal correlations in R^2 scores from predictions on the test set as evaluated on the pretrained 2d model. At each time, t , is the R^2 score calculated over all spatial coordinates for that time.	106
4.11	The resulting ET prediction maps for each of the three models, averaged at each spatial coordinate over time. The data shown is collected by inputting the same predictor observations into each of the models, between the years 2003-2023 during the months May to October.	107

LIST OF TABLES

	Page
2.1 Navigation accuracies using the fully observed map.	30
2.2 Successful recovery rates (<i>SRR</i>) using the fully observed map.	31
2.3 Failed recovery rates (<i>FRR</i>) using the fully observed map.	31
2.4 Recovery rates using the partially observed map.	32
2.5 Ablation study, showing the total recovery rate and relative path length over all AirSim environments.	34
3.1 Adaptive split computing results.	57
3.2 Static student versus adaptive auxiliary models.	66
3.3 Results from varying the efficiency parameter X before runtime. © [2025] IEEE. Reprinted, with permission, from [114].	71
3.4 NVIDIA Jetson Orin Nano benchmarks for MDE	77
4.1 Parameter values used to generate the corrupted input variables (corresponding to sensor observations) and the denoising levels (DNL) of different data corruption treatments: Gaussian noise (G.), Bernoulli perturbation (B.), and both (G.B.). For the Gaussian noise corruption, either the standard deviation, σ , or the coefficient of variation, <i>c.v.</i> , is used to generate the sample, and the choice is according to the manufacturer’s specified error metric on the corresponding sensors. Values of <i>c.v.</i> are identified by a ”%” sign. The parameter ν is the Bernoulli probability in Equation 4.2. The DNL is calculated as $\frac{s_{\hat{U}} - s_U}{s_U}$ or $\frac{c.v.\hat{U} - c.v.U}{c.v.U}$, where $\hat{U}$ is the model-predicted estimator of each original variable, U . Quantities s and <i>c.v.</i> were calculated as in Equation 4.5 and 4.6. Note that ET has two values for σ resulting from the varying instrumental uncertainties arising from the different sites used in this study. © [2025] IEEE. Reprinted, with permission, from [38].	90
4.2 One-Sided Hypothesis Test Results for Single-Mineral Classification	96
4.3 AUC Scores for Multi-Mineral Classification	97
4.4 Pearson Correlation Coefficient (PCC) between each of the predictor variables and the percent error in ET predictions from the pretrained 2D model (w.r.t. MODIS labels).	107

LIST OF ALGORITHMS

	Page
1 Asynchronous Control Loop	11
2 Training a Navigation Neural Network via Imitation Learning	14
3 Calculating an Error Gradient for Training Slimmable Networks	48
4 Calculating the Error Gradient for Variable Sensing	55
5 Fine Tuning the 2D Model with the 1D Data	104

ACKNOWLEDGMENTS

I would like to thank multiple people who have had a significant impact on my PhD journey, whom I know this thesis would not look the same without. To my lab mates, Ian Harshbarger, Sharon Contreras, and Zixia Xia, at UCI who I had many interesting discussions with regarding our work, provided positive motivation and support, and are co-authors on some published works. To my PhD advisor, Marco Levorato, at UCI who has always been available – providing not only guidance but also reasoning into selected paths forward. To one of my first advisors, Virginia Gulick, at NASA Ames Research Center who originally motivated me to pursue a PhD and has always been a source of great support, and someone that I could always fall back and rely on. And to all the other people who have helped guide and motivate me on this journey, thank you!

This work has been partially funded by U.S. National Science Foundation (NSF) grants: CCF 2140154, CNS 2134567, and DUE 1930546.

Fiat Lux

Ad Astra

Veritas Vincit

VITA

Timothy Kevin Johnsen

EDUCATION

Doctor of Philosophy in Computational Science	2026
University of California, Irvine	<i>Irvine, CA</i>
San Diego State University	<i>San Diego, CA</i>
Masters of Science in Data Analytics	2021
San Jose State University	<i>San Jose, CA</i>
Bachelor of Science in Applied Physics	2018
University of California, Irvine	<i>Irvine, CA</i>

RESEARCH EXPERIENCE

Graduate Research Assistant	2022–2026
University of California, Irvine	<i>Irvine, California</i>
Graduate Research Assistant	Summer 2025
Lunar and Planetary Laboratory - University of Arizona	<i>Tuscon, California</i>
Graduate Research Assistant	Summers 2023–2024
Lawrence Berkeley National Laboratory	<i>Berkeley, California</i>
Data Science Intern	Summers 2020–2021
eHealth, inc.	<i>Mountain View, California</i>
Research Assistant	2015–2021
NASA Ames Research Center and SETI Institute	<i>Mountain View, California</i>

TEACHING EXPERIENCE

Teaching Assistant	Fall 2023, Fall 2025
University of California, Irvine	<i>Irvine, CA</i>
Teaching Assistant	2021–2022
San Diego State University	<i>San Diego, CA</i>

REFEREED CONFERENCE PUBLICATIONS

CADENCE: Context-Adaptive Depth Estimation for Navigation and Computational Efficiency 2026
IEEE World AI IoT Congress (AIIoT)

SmartDepth: Motion-Aware Depth Prediction with Intelligent Computing for Navigation 2025
IEEE 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)

An Overview of Adaptive Dynamic Deep Neural Networks via Slimmable and Gated Architectures 2024
IEEE 15th International Conference on Information and Communication Technology Convergence (ICTC)

NaviSplit: Dynamic Multi-Branch Split DNNs for Efficient Distributed Autonomous Navigation 2024
IEEE 25th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)

Navislim: Adaptive context-aware navigation and sensing via dynamic slimmable networks 2024
IEEE Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI)

REFEREED JOURNAL PUBLICATIONS

Single-and multi-mineral classification using dual-band Raman spectroscopy for planetary surface missions 2025
American Mineralogist

Denoising autoencoder for reconstructing sensor observation data and predicting evapotranspiration: Noisy and missing values repair and uncertainty quantification 2025
Water Resources Research

SOFTWARE

OmniNaviPy

<https://github.com/WreckItTim/OmniNaviPy>

A universal Python framework bridging abstract navigation on different robot embodiments and simulators with Vision-Language-Action (VLA) reasoning, Deep Reinforcement Learning (DRL), and real-time motor control for full stack autonomy and static evaluations.

ABSTRACT OF THE DISSERTATION

Adaptive Dynamic Deep Neural Networks
and Hierarchical Control for Autonomous Systems

By

Timothy Kevin Johnsen

Doctor of Philosophy in Computational Science

University of California, Irvine and
San Diego State University, 2026

Professor Marco Levorato (UCI), Co-Chair
Associate Professor Ahmad Bani-Younes (SDSU), Co-Chair, Chair

Autonomous operation of robots and sensors in remote, unknown environments require a rigorous interdisciplinary approach bridging computer science, mathematics, and engineering. A central challenge in deploying full autonomy stacks is managing the cyclic flow of data from raw acquisition to intermediate perception and final predictions. While deep neural networks (DNN) enable advanced perception and logic, they are computationally demanding and static in nature. This clashes with the resource demands of remote systems which are constrained by embedded processors, lightweight sensors, and limited energy reservoirs. Dynamic DNNs (DDNN) provide great promise in mitigating this bottleneck by enabling run-time branching and scaling, but they often lack context awareness. Presented are end-to-end methodologies and applications focused on the development of adaptive DDNNs (ADDNN), which are demonstrated in novel frameworks to augment perception and navigation pipelines with the ability to intelligently select when and how to branch and scale. Ultimately, this research demonstrates that ADDNNs can break the traditional trade-off between efficiency and accuracy, by simultaneously, and significantly, improving both.

Evidence to support this claim is collected by first developing a robust hardware-in-the-loop

(HIL) testbed, that combines high-fidelity simulation (Microsoft AirSim) with deployment on edge computing hardware (NVIDIA Jetson). This setup rigorously benchmarks inference latency, power consumption, and energy expenditure, against navigation accuracy. A flexible navigation framework is designed and deployed to this testbed. At its core, is a generalized mid-level policy that uses a combination of rule-based, initiation learning, and deep reinforcement learning (DRL) algorithms to generate generalized motion actions in response to onboard sensor observations. A key strength of the mid-level policy is in its ability to be deployed to various real-time controllers, closing the loop between sensing and actuation for varying robot embodiments. The modular structure of the hierarchical navigation framework allows for varying sensor configurations and environmental complexities with little retraining and no fine tuning. This is further augmented with a high-level policy by integrating cutting-edge multimodal large language models (MLLMs), which are used to supervise and improve the performance of the entire framework in a zero-shot manner.

Finally, considerations are taken into artifacts that are inherent with real-world scientific data that is processed through a combination of onboard and offboard computing, for two downstream tasks: modeling evapotranspiration in watershed environments, and autonomously classifying geological samples for Martian and lunar missions. Such considerations are taken in poor signal-to-noise ratios, missing data due to instrumental failure and human error, uncertainty quantification and reduction, and heterogeneous sensor fusion.

Chapter 1

Introduction & Testbed

Autonomous vehicles and robotics are becoming increasingly prevalent as they enable remote operations in hazardous or otherwise hard-to-reach environments. The impact of these systems is broad, ranging from space exploration, transportation, and wild-fire prevention to search and rescue, agriculture, and mining. These applications not only ease the burden and mitigate dangers of daily human tasks, but also have a farther-reaching epistemic scope in expanding our knowledge and reach in the universe. This thesis investigates three critical bottlenecks in robotic and scientific autonomy:

1. **Development of modular full-autonomy stacks**, allowing for overlapping logic and training to be applied to varying robot embodiments and environments, especially those that can accommodate new cutting-edge methods as they come to light.
2. **Designing efficient machine learning pipelines**, so that perception and autonomy can be computationally executed on resource-constrained hardware such as compact batteries, lightweight sensors, and embedded computing.
3. **Mitigating real-world artifacts** such as noise, missing features, faulty sensors, un-



Figure 1.1: Microsoft AirSim drone simulator, built on Unreal Engine

certainty, and heterogeneous multimodal spatio-temporal data. Further considerations are investigated in how these artifacts affect downstream autonomy tasks.

The remainder of the dissertation is structured as follows. Several of the experiments are conducted in a similar training and testbed environment, utilizing overlapping hardware and software, all of which is appropriately discussed at the end of this first chapter. The next three chapters are dedicated to each of the bottlenecks outlined above. Chapters 2 and 3 particularly focus on addressing challenges in drone/UAV autonomy [15]. In Chapter 5, final conclusions and implications are drawn that tie together all of the chapters, along with future work.

1.1 Simulation and Hardware-In-the-Loop

The primary focus of this research is to investigate robust methodologies to further develop software modules within an autonomy stack. The software will be deployed to resource-constrained devices, however machine learning training and large-scale data processing is executed by leveraging High Performance Computing (HPC) clusters and lab servers. To ensure that the computational methodologies do not shift too far from realistic constraints, hardware parallels are made by developing a Hardware-In-the-Loop (HIL) system that integrates simulators, datasets, and resource-constrained devices.

Figure 1.1 illustrates the high-fidelity drone simulator, Microsoft AirSim [92]. Physics and graphics are rendered using Unreal Engine [16], which is a cutting edge video game engine

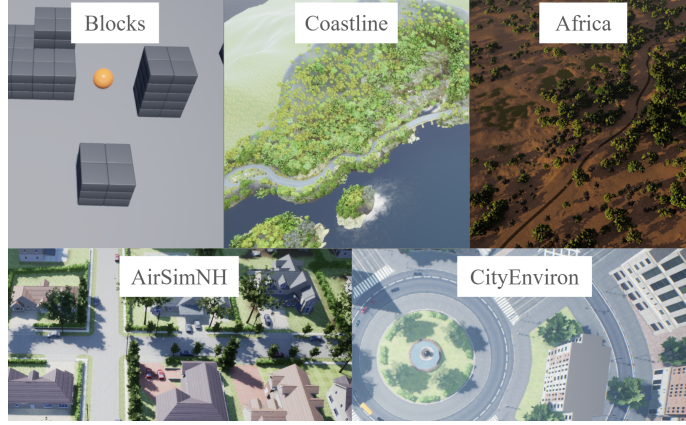


Figure 1.2: Microsoft AirSim maps.

backed by an enormous industry leading to consistent upgrades in realism and computing power throughout the foreseeable future. Key advantages of Airsim, in particular, are:

1. **Environment flexibility** in various contexts such as weather conditions, live traffic, and types of objects rendered into the scene; and complexities such as the density of objects, controllable levels of noise, and robot/vehicle embodiments.
2. **Sensor flexibility** in controlling the resolution, placement, and modality of onboard instruments such as LiDAR, camera, and distance sensors.
3. **Available ground truths** of sensor observations on a continuous basis, where as real-world datasets are limited to the discrete grid dimensions determined during data collection, including RGB images, 2D depth maps, and voxelization [32] of the 3D map.

Some available AirSim maps are shown in Figure 1.2. Each of these maps provide different contexts and complexities, used to evaluate reliability in varying conditions and generalization of models trained in one more many maps while being deployed to the others:

1. **Blocks** is a small, low complexity playground with basic shapes of various sizes, that is mostly used for proof-of-concept, preliminary results, and sanity checks. However also has potential in using the basic shapes as a form of regularization.
2. **AirSimNH** is a medium sized, realistic neighborhood with houses, roads, cars, and trees.

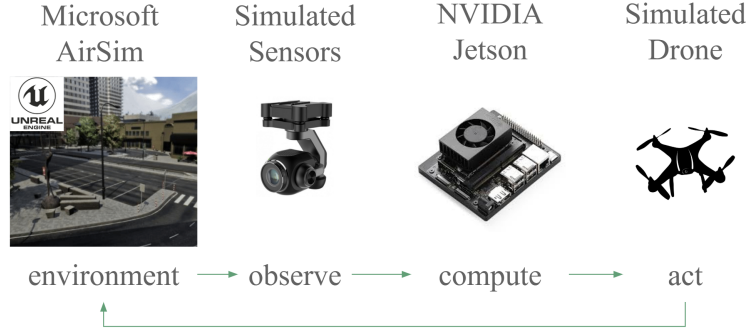


Figure 1.3: Hardware-In-the-Loop (HIL) system.

3. **Coastline** is a large, realistic road cutting through an island with trees, rocks, and water.
4. **CityEnviron** is a very large, realistic cityscape with buildings, roads, cars, people, bicycles, and live traffic.
5. **Africa** is a large, realistic flat map with the largest density of trees.

AirSim is a robust simulator, allowing to vary conditions while evaluating autonomy stacks and collecting data in a much more feasible environment than real world drone data collection. However, to better bridge the gap between simulation and real-world deployments, a HIL approach is taken by integrating hardware into the control loop – as shown in Figure 1.3. AirSim is used to collect observations, calculate physics, detect collisions, and render graphics, and ultimately used to measure mission-oriented task accuracies and intermediate perception-based error rates. All computing of the policies is executed onboard an NVIDIA Jetson [26, 47] device, which is used to benchmark the inference latency, power consumption, and energy expenditure. These devices are compact, embedded computers typical to real-world drones, and other lightweight vehicles, that are equipped with both a CPU and small GPU with unified physical memory (combining CPU, GPU, and RAM). The following NVIDIA Jetson devices vary in computing power and integrated firmware:

1. **NVIDIA Jetson Nano**: 4GB RAM, NVIDIA Maxwell architecture with 128 NVIDIA CUDA cores GPU, Quad-core ARM Cortex-A57 MPCore processor CPU, operating at power levels between 5-10 Watts.

2. **NVIDIA Jetson AGX Xavier:** 16GB RAM, NVIDIA Volta architecture with 512 NVIDIA CUDA cores and 64 Tensor cores GPU, 8-core NVIDIA Carmel Armv8.2 64-bit CPU, operating at power levels between 10-30 Watts.
3. **NVIDIA Jetson Orin Nano:** 8GB RAM, NVIDIA Ampere architecture with 1024 NVIDIA CUDA Cores and 32 Tensor Cores GPU, 6-core Arm Cortex-A78AE v8.2 CPU, operating at power levels between 7-15 Watts.

1.2 AI-Accelerated Servers

The NVIDIA Jetson devices are powerful embedded chips that reflect computing performance onboard small vehicles. The objective is to execute cutting edge perception and logic based models on these devices during inference; however, training such models requires much heavier computing power. The desktop computers at Dr. Levorato’s Intelligent and Autonomous Systems Lab (IASL) are equipped with components that are designed to specifically accelerate training of AI and machine learning algorithms, including NVIDIA 2080 Ti, 3090, 4090, and 5090 RTX GPUs with upgraded RAM up to 128 GB. These servers are used to not only train machine learning models, but also to execute heavy computing algorithms and Multimodal Large Language Models (MLLMs).

To supervise training and other computing, a custom job scheduler and workload manager is developed in Python. This queries jobs in parallel via remote SSH bash commands and scripts to monitor progress, check for available computing resources, and execute various experiments and algorithms. Dropbox [14] cloud storage is used to sync data, models, and code between each of the servers.

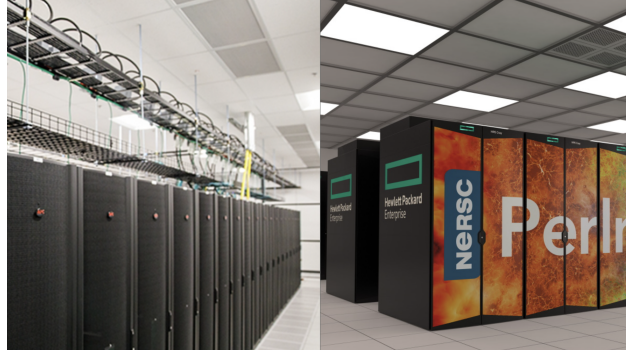


Figure 1.4: High Performance Computing clusters, on the left is The University of Arizona’s HPC resources at the Research Data Center (<https://www.datacenterdynamics.com/en/news/asu-to-launch-sol-supercomputer-this-summer/>), and on the right is Lawrence Berkeley National Lab’s National Energy Research Scientific Computing Center (<https://www.nersc.gov/news-and-events/news/berkeley-lab-deploys-next-generation-supercomputer-perlmutter-bolstering-u-s-scientific-research>)

1.3 High Performance Computing Clusters

The IASL servers are capable of handling most jobs, however some jobs require even heavier computing. Two super-computing clusters with several GPU and CPU components are leveraged to handle parallel computing for large-scale data processing and hyper parameter searches of deep neural networks. Access is granted to Lawrence Berkeley National Lab’s (LBNL’s) National Energy Research Scientific Computing Center (NERSC) [77], and The University of Arizona’s (UoA’s) HPC resources at the Research Data Center (RDC) [100]. Each of these HPC clusters are managed by Simple Linux Utility for Resource Management (SLURM) [116] which is used to query computing jobs and manage resources.

Chapter 2

Hierarchical Navigation Framework

In this chapter, a full-stack navigation framework is proposed that enables robust end-to-end vehicular autonomy. Each level in the 3-tier hierarchical structure is independently developed w.r.t. to the other levels. This creates a highly modular system, so that each component can be adjusted and interchanged with similar functioning components. Thus, the full framework can adapt to variable robot embodiments, sensors, environments, mission objectives, and new cutting-edge techniques – while minimizing the overhead of doing so.

1. **low-level controller:** This tier handles basic actuation, such as motor controllers, allowing the robot to adjust its velocities and accelerations to execute generalized motion commands (*i.e.*, move forward x meters, or rotate y degrees). This is assumed to be integrated into the firmware of the robot, and runs continuously.
2. **mid-level policy:** This policy defines the logic behind generating short-horizon actions, such as motion commands, given a set of mission objectives, robot states, sensor observations, and environmental conditions. Motion commands output from this tier are relayed to the low-level controller to be executed. This is assumed to be light-weight computing, that can be executed on-board the robot at a high frequency.

3. **high-level policy:** This policy oversees the entire operation of the robot. It can dictate new objectives, as relayed to the mid-level policy, and generates long-horizon actions, such as intermediate waypoints. This can further activate different modules in the pipeline, and ultimately plays the role of a human supervisor without requiring human intervention. This is assumed to be heavy-weight computing, that can be executed off-board the robot at a low frequency.

The flow of data through one timestep of the three-tier hierarchal framework is illustrated in Figure 2.1. The mid-level policy generates motion commands that are executed by the low-level controller. The high-level policy is queried when a new waypoint is needed to be generated, while building a world map of the environment. The contributions and advantages of structuring the navigation framework in such a hierarchy are:

1. **Generalization to variable robot embodiments.** The mid- and high-level policies operate on similar logic without needing to know how the low-level controller executes generalized motion commands.
2. **Generalization to variable sensors.** The high-level policy and low-level controller operate under the same logic regardless of the sensor data used by the mid-level policy.
3. **Descritized space leading to quicker training times.** The mid- and high-level policies can operate on more coarse-grained state spaces than the low-level controller which executes continuous movements. This shrinks the space required to be explored during training of otherwise intensive algorithms, such as reinforcement learning.
4. **Modular data flow.** Each tier can be modified and swapped out, allowing for flexible deployment to variable environments and rapid integration of cutting-edge methods.
5. **Closing the sensing-actuation loop.** The resulting feedback loop self-adjusts actuator control in response to sensor observations, enabling real-time autonomy.

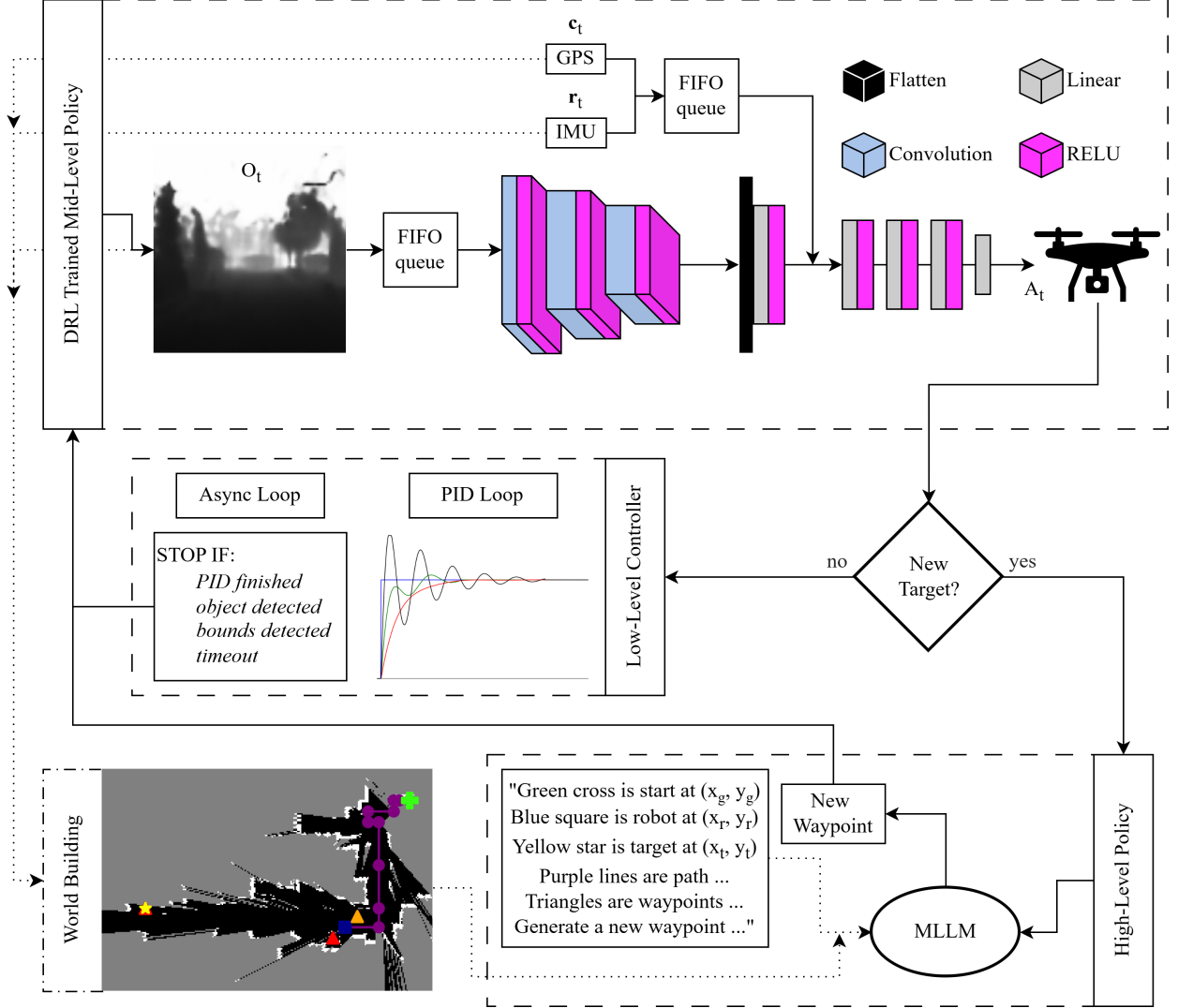


Figure 2.1: Diagram illustrating the hierarchical navigation framework. The general DNN framework shown is used by either the DQN or TD3 DRL algorithms used to create mid-level policies. Shown in this figure is an example 2D depth map, however the observation space corresponding to O_t may change based on the configured sensor array onboard the robot. Also shown here is a drone. A different robot embodiment would have the same DNN structure but a potentially different action space.

2.1 Low-Level Controller

The low-level controller is responsible for executing a generalized motion command, which is intentionally left arbitrary, however are those such as “move forward x meters”, “rotate clockwise y degrees”, *etc.* The motors are controlled by a proportional-integral-derivative (PID) [84] controller to continuously regulate velocities that place the robot on a trajectory to fulfill the command.

If the PID controller is left unregulated, it would not account for environmental factors such as collisions with objects and other conditions which will block the robot from fulfilling the motion command. To this end, an asynchronous loop, Algorithm 1, is developed to monitor progress and determine if the PID controller has either failed or fulfilled the command. At a given low-level frequency, the loop checks to determine if the motion action is fulfilled within a set threshold. This can be represented in meters if the action is translational, or degrees if rotational. Next, the loop checks if the robot is within a given threshold distance to either the nearest detected object (to avoid collisions) or the nearest boundary within the environment. If either of these two thresholds are triggered, then the agent is instructed to halt all motion. Finally, a timeout tolerance is set to halt motion if the action is taking too long to execute, which is used to catch all other errors which may not be explicitly caught above. Calibration of all the threshold values are conditioned on the robot’s actuators, sensors, and kinematic parameters which determine both its reaction and stopping time.

2.2 Mid-Level Policy

The mid-level policy is responsible for generating new motion actions for the low-level controller to execute. It considers the current state of both the robot and the partially observable environment, along with downstream mission tasks and objectives. Considered herein is the task of navigating from the robot’s current position, $p_t \in \mathbb{R}^3$, to a target position, $g_t \in \mathbb{R}^3$.

Algorithm 1 Asynchronous Control Loop

```
1:  $target\_state \leftarrow$  calculate desired state from motion command
2: start stopwatch
3: while true do
4:    $state \leftarrow$  fetch current robot state
5:   if  $state - target\_state < fulfillment\_tolerance$  then
6:     break ▷ Motion Command Fulfilled
7:   end if
8:   if  $state.position - nearest\_object.position < collision\_tolerance$  then
9:     robot.stop()
10:    break ▷ Collision Avoidance Triggered
11:  end if
12:  if  $state.position - nearest\_bounds.position < bounds\_tolerance$  then
13:    robot.stop()
14:    break ▷ Out Of bounds
15:  end if
16:  if  $stopwatch.lap() \geq time\_tolerance$  then
17:    robot.stop()
18:    break ▷ Timeout Triggered
19:  end if
20: end while
```

The robot’s position changes with time t as it progresses towards the goal, as can the target position based on changing mission objectives and any intermediate waypoints.

The mid-level policy must functionally map observations collected from the observation space to motion actions from the action space. There are typically three approaches to this.

Rule Based (RB). This is a heuristic approach that applies finite state machines (FSM) [79, 101] with potentially integrated deterministic path planning algorithms such as A* [27]. A FSM system has the advantage of being simple and highly interpretable. These systems lack generalization to large and complex tasks, and ability to factor uncertainty into the logic. They are also brittle, as they further require complete re-tuning if either the observation or action space changes.

Imitation Learning (IL). This is a machine learning approach that learns by mimicking the actions of an “expert” agent, which can be recorded data from a human teleoperator

or that from a computationally expensive planner algorithm, such as A*. Typically an IL approach is trained using a deep neural network (DNN) [9], allowing for flexibility in the observation space. However, DNNs are “black boxes” which are less interpretable than RB approaches. Further, if the robot embodiment changes then the DNN’s action space no longer aligns with that of the recorded expert data.

Deep Reinforcement Learning (DRL). A machine learning approach that structures the problem as a Markov decision process [6], typically used with DNNs [73, 91, 21], utilizing actor-critic [53] networks or value-based methods [110], that learns through trial an error by maximizing a defined reward function to encourage desired behavior. This allows for more flexibility in both the observation and action space, and ability to learn more complex scenarios while factoring uncertainty into the logic. However, DRL requires the use of a simulator or manipulatable data to have the freedom to explore these spaces.

DRL offers significant advantages through its flexible observation and action spaces. Additionally, as demonstrated in Chapter 3, the DNN backbone of a DRL system allows for various computational efficiency optimizations during execution. While this work primarily employs DRL, IL retains specific benefits: as a supervised learning approach, it typically offers faster training times and a more established research foundation. This maturity enables the use of advanced methods like knowledge distillation for dynamic DNNs, an area where a significant gap remains in current DRL literature.

2.2.1 A* Shortest Path

Shortest path algorithms are used to solve the problem of finding length-optimal paths between two points. A* [27] is guaranteed to find the optimal solution without having to traverse every possible path. A* is implemented by reconstructing a given AirSim map into a discretized 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). Edges are

formed by using the Voxelized data captured using Microsoft AirSim and Unreal Engine, which indicates which $1 \times 1 \times 1 \text{ m}^3$ cells contain a collidable object – effectively creating an occupancy grid.

The cost of a path is defined as the total Euclidean distance traveled. To estimate the remaining cost, the heuristic calculates the minimum distance to the target under a “best-case” scenario, assuming no obstacles are present. This ensures the heuristic remains admissible, meaning it never overestimates the actual cost, which guarantees that the A* algorithm will find an optimal path.

A* is used to generate a set of ground truth length-optimal paths, and utilized by multiple components of this work to:

1. determine if there is a viable path between a proposed start and target location
2. evaluate the difficulty of a path, used for curriculum learning (Section 2.2.3)
3. generate static hold-out sets used for validation and testing
4. determine a set of labels for imitation learning (Section 2.2.2)

2.2.2 Imitation Learning (IL)

The IL procedure for training a navigation network is outlined in Algorithm 2, given a set of observations mapped to ground truth actions, $O_t \mapsto A_t$. The function *calculate_grad* is an argument to Algorithm 2 that controls how supervised learning works in unison with any other mechanisms that affect calculating the error gradient, such as knowledge distillation [29] as later discussed in Section 3.7.1. By default the *calculate_grad* is derived from supervised learning and calculates the gradient through back propagation. 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.

Algorithm 2 Training a Navigation Neural Network via Imitation Learning

Input: training set, validation set, randomly initialized neural network f_θ , *calculate_grad* function, *loss* function.

```
while not converged do
  set running error gradient to zero
  sample batch of  $[O_t \mapsto A_t]$  pairs from training set
  for each  $O_t \mapsto A_t$  in batch... do
     $\text{grad} = \text{calculate\_grad}(f_\theta, O_t, A_t, \text{loss})$ 
    aggregate grad into running error gradient
  end for
  update network parameters  $\theta$  using optimizer with aggregated error gradient
  use the validation set to check for early stopping
end while
Return: trained  $f_\theta$ 
```

Traditionally in robotics, imitation learning uses ground truth paths generated from a human teleoperator. However, this requires a large amount of data collection time and human effort. This is exasperated when considering that robot embodiments and the testbed environments may change. To this end, A* is used to create ground truth optimal paths, which are used in Algorithm 2 as the training, validation, and testing sets to teach and evaluate the navigation network.

2.2.3 Deep Reinforcement Learning (DRL)

DRL has been used in several autonomous robotics applications, particularly for drones [80, 51, 55, 104]. It is fundamentally framed as a Partially Observable Markov Decision Process (POMDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. The state space $\mathcal{S}$ spans a set of global environmental variables, and is realized as S_t at a given time t . The DNN typically inputs a partial representation of S_t , of which it then outputs a realized action A_t from the predetermined action space $\mathcal{A}$. The transition function $\mathcal{T}$ defines how the state advances from S_t to S_{t+1} , given the output action A_t . After which, a reward R_t is measured based on the new state S_{t+1} , as determined from a carefully constructed reward function which typically has full visibility of the state space. The following paragraphs discuss how each

of the $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ components are deployed to train a mid-level policy for autonomous navigation through an unknown environment.

State Space $\mathcal{S}$. To capture dynamic environmental changes, the state space $\mathcal{S}$ utilizes a sliding temporal window of size τ . The state fuses high-dimensional exteroceptive sensors with low-dimensional proprioceptive data:

$$S_t = \left\{ O_k, \mathbf{c}_k, \mathbf{r}_k \right\}_{k=t-\tau}^t \quad (2.1)$$

where O denotes a set of high-dimensional observations acquired at the given time step t from various onboard exteroceptive sensors configured at different orientations (such as RGB images, 2D depth maps, or 3D point clouds), $\mathbf{c} \in \mathbb{R}^3$ denotes the relative position of the robot w.r.t. the target, $\mathbf{g}_t$, and $\mathbf{r} \in \mathbb{R}^3$ represents the IMU heading and tilt angles (yaw, pitch, roll). Note that this work uses relative positions, by zero-centering the origin at the starting point of the robot, as to not overfit to specific coordinates on the map during DRL training.

Action Space $\mathcal{A}$. The action space $\mathcal{A}$ is defined as either a discrete set of relative motion actions, or continuous space of displacement vectors. Specifically, the DRL agent selects from a constrained set of translational movements (meters) and yaw rotations (degrees) relative to its current pose. For discrete space, a deep Q-network (DQN) [73] DRL algorithm is used. For continuous space, a twin delayed deep deterministic policy gradient (TD3) [21] DRL algorithm is used. Discretization drastically reduces the dimensionality of the space needed to explore for DRL, facilitating highly sample-efficient convergence while maintaining precise control. Outputting continuous space is advantageous for more fine-grained control, which is otherwise left for the low-level controller to manage when using a DQN.

Transition Function $\mathcal{T}$. The transition function $\mathcal{T}(S_{t+1}|S_t, A_t)$ dictates how the state space changes given an action. At this stage, the low-level controller is utilized to execute the

output motion action A_t . During simulation, the base kinematics, physics, and graphics are rendered by Unreal Engine and Microsoft AirSim. This simulates actuators and interactions with the environment, utilizing the built-in flight controller to move a drone by the given rotational and/or translational vectors as determined by A_t . During real world deployment, the actual motor commands are executed on the physical actuators.

Reward Function $\mathcal{R}$. Since physical collisions are prevented by the collision avoidance mechanism integrated into the low-level policy, see Algorithm 1, they are entirely removed from the POMDP’s penalty space. The reward function is therefore streamlined to focus strictly on path efficiency and temporal constraints. The episode terminates either when the goal is reached or a maximum step limit, L_{LIMIT} , is exceeded:

$$reward(t) = \lambda_p d_t + \lambda_{step} + reward_{terminal} \quad (2.2)$$

Where $\lambda_p < 0$ and d_t is used to encourage movement toward the immediate target, defined by the Euclidean distance:

$$d_t = \|\mathbf{p}_t - \mathbf{g}_t\|_2 \quad (2.3)$$

The step penalty (λ_{step}) is a constant negative value applied at every timestep. If the robot selects an action that triggers the *stop()* command in Algorithm 1 of the low-level policy, this term inherently penalizes the inefficient, blocked action. A terminal reward/penalty ($reward_{terminal}$) is applied at the end of each episode, and yields a large positive scalar if the robot reaches $\mathbf{g}_t$ within a given target threshold, or a heavy negative penalty if the episode is terminated due to exceeding a maximum number of steps L_{LIMIT} , discouraging infinite loops.

DNN Architecture. A similar DNN architecture is used for both the TD3 (continuous action space) and DQN (discrete action space) approaches, as illustrated in Fig. 2.1. The

DNN processes the multimodal state S_t through a two-stage injection forward pass. All observations are placed into a first-in-first-out (FIFO) queue, consisting of those acquired at the τ most recent steps. A FIFO queue of depth maps is fed through a convolutional neural network (CNN) [59] backbone consisting of multiple convolutional layers (with ReLU activations [75]) to extract spatial features. The output is flattened into a vector and concatenated with another FIFO queue containing the GPS and IMU history. This fused representation is then passed through a multi-layer perceptron (MLP) [88] consisting of multiple linear layers, outputting a realization from the action space. The general structure is inspired by architectures originally used for learning how to play Atari games [73]. However, the specific hyper parameters are explored with a grid search to find optimal settings, which depends on the experiment and application.

For a DQN, the output nodes each correspond to an estimated Q-value, $Q(S_t, A_t)$, for each proposed motion action. The action is taken that corresponds to the node with the highest predicted value, where the Q-value is derived from the Bellman equation [5] to estimate the future rewards for that given action:

$$Q(S_t, A_t) = \max_A [\text{reward}(S_t, A_t) + \gamma V(S_{t+1})] \quad (2.4)$$

Where the term $\max_A$ denotes a theoretical strategy that assumes the mid-level policy will always select the action that maximize the Q-value. The term $V(S_{t+1})$ is the expected long-term reward that will result from taking the given action A_t and state S_t resulting in the future state S_{t+1} , and is scaled down by the discount factor γ to put more priority on maximizing short-term rewards. Quantity V is an intractable value to calculate, as it can only be truly realized by fully exploring the infinite state and action space. It is estimated during training by inputting the next state through the network again and calculating the Q-values at state S_{t+1} after taking action A_t .

For a TD3, the output nodes correspond to continuous values which each control the translational and rotational kinematic vectors. During training, another DNN is used to estimate the Q-values corresponding to each of these output values. This Q-network is something that also has to be learned, which further complicates the training procedure of a TD3 algorithm.

To learn an accurate mapping of the Q-values, for both TD3 and DQN, the weights of the DNN are trained by using an exploration versus exploitation strategy. At first, random motion actions are drawn from a uniform distribution, since the randomly initialized weights of the DNN will otherwise bias action selection and discourage exploration. As the policy learns from sampled actions, exploitation is used to output actions by using forward passes of the DNN. The relative probability of doing exploration rather than exploitation, is annealed as training progresses. This process requires billions of iterations to adequately explore the action and state space. As such, the computational time to execute just one step has a substantial long-term effect on total training time.

Observation Caching and State Discretization. Training vision-based DRL policies in high-fidelity simulators is traditionally bottlenecked by rendering latencies. To ensure robust convergence while maintaining computational feasibility in training times, the computational overhead of synchronous rendering is bypassed by precomputing and caching the environment interactions. Prior to training, the navigable space of the AirSim environments is discretized by collecting and storing realizations of all possible state and observation values. Acquiring an observation directly from this discretized state space reduces the acquisition time by three orders of magnitude. This substantially reduces total training time, and enables:

1. **iterative model development**, where hyper-parameters are empirically explored by measuring their affect on fully trained models.
2. **several random restarts**, which is a key requirement in training DRL algorithms since they are at their core local search algorithms.

3. **training on digital twins**, if enough data is collected in real-world environments.
4. **grid-based path planning algorithms**, such as A* shortest path.

During DRL training, the low-level controller is bypassed and the robot “jumps” to the next valid space, using the same logic as that in Algorithm 1 to determine which spaces are valid. This discretization reduces the granularity of motion actions, as the action must terminate in one of the finite positions that have a cached observation. This is a key motivation of using the multi-tier framework, as the low-level controller is used to control the robot on a finer granularity; however, is not needed during training and only used during final evaluations.

Offline A* Path Generation. During the pre-computing phase, a comprehensive dataset of ground-truth navigation routes is generated using an offline A* search algorithm [27]. The A* algorithm is utilized strictly to quantify the complexity of each path, defining the “difficulty” of a path by the total number of optimal steps required to reach the target from the starting point. This offline dataset is subsequently partitioned into a large training set and two static hold-out sets reserved for validation and testing (which are generated from regions of the AirSim environment unseen during training).

Curriculum Learning. During the DRL training phase, the robot is procedurally exposed to paths of varying complexity via a curriculum learning schedule [7]. The policy first learns simple paths with low A* step counts before progressively advancing to more difficult paths with higher step counts. However, sequential curriculum learning often leaves DNNs vulnerable to “catastrophic forgetting”, in which the learned policy performs well on only recently seen data. To mitigate this, first a high-capacity replay buffer is employed by allocating 100 GB of RAM. Second, a stochastic sampling strategy is used so that while the robot primarily trains on the current difficulty, paths from previously learned difficulties are also sampled from at a lower rate. Finally, an early stopping [83] mechanism is employed that evaluates the policy against the static A* validation set, freezing the DNN parameters

that yield the highest validation accuracy to serve as the final mid-level policy.

2.3 Monocular Depth Estimation (MDE)

The mid-level policy is tasked with both perception, processing raw sensor data, and navigation, mapping it to actionable motion commands. The 2D depth maps used to inform navigation, as shown in Figure 2.1, are directly obtained if the robot is equipped with appropriate sensors such as binocular cameras, LiDARs, and other depth sensors. However, lightweight robots are commonly only equipped with a forward facing monocular camera, requiring a perception mechanism to extract depth information from the RGB images. This is accomplished through a Unet [86] structure, which is a variant of a vanilla CNN that instead maps multi-channel images to other multi-channel images. Several such MDE models exist in literature [25, 60, 31, 67]. The MDE process is highly complex, often requiring larger DNNs than the ones used for downstream navigation logic. The frameworks shown in Section 3.7 further investigate methods for reducing this overhead.

MDE networks are trained with supervised learning by minimizing the error in predictions w.r.t. to ground truth depth maps, by finding the optimal model parameters θ in a neural network f_θ :

$$\operatorname{argmin}_\theta \operatorname{loss}(f_\theta(\mathbf{I}), \mathbf{D}) \quad (2.5)$$

Where $\mathbf{I}$ is the input RGB image corresponding to the ground truth depth maps, $\mathbf{D}$. Development of novel, robust *loss* functions for MDE is of great interest [25, 60, 31, 67], however classical approaches use L_1 loss with decaying learning rates for sharper edges. True values of $\mathbf{D}$ are acquired in the real-world through supplementary sensors such as LiDAR for the KITTI dataset [63] and a Microsoft Kinect RGB-D camera [121] for the NYU-Depth V2 dataset [93]. For Microsoft Airsim, $\mathbf{D}$ is acquired directly through the Unreal Engine

physics.

2.4 Split Computing

In edge computing, data is collected by a mobile device and transferred to a compute-capable edge server over a wireless channel. The server processes the data and in some settings relays them back to the mobile device [89]. Processing typically consists of executing DNNs to extract features, semantics, and control. In computer vision applications, such as [76], the mobile device can apply JPEG or MPEG compression to compress input images and reduce the amount of data.

Split computing [69], also known as split DNN and model partitioning, is a recent class of approaches in mobile computing. DNN architectures are partitioned into two sections that are each executed by the mobile device and edge server. The objective is to balance computing load, energy consumption, and channel usage. While early approaches [45] simply “split” existing DNN models, recent methods involve injecting a “bottleneck” into a trained DNN task model [68]. This alters and trains the DNN’s layers to learn a compact set of task-specific features that preserves the task accuracy.

Split computing is used in this work to split the load of the perception (*e.g.*, MDE) and navigation neural networks during execution of the mid-level policy, between the onboard processor and an offboard server. Generalized compression is designed to reconstruct the original data (MPEG, JPEG). Alternatively, split computing is a form of supervised compression, allowing deeper levels of compression by omitting non-correlated data to desired downstream tasks.

2.5 High-Level Policy

The mid-level policy and low-level controller are crucial requirements for executing real-time navigation. The high-level policy is an optional augmentation that can improve performance of the autonomy stack. It is assumed that this policy can be executed off-board the robot on a nearby server, and is in communication via a wireless link to the robot. This way, there are less restrictions on computing which enables larger algorithms. In this work, the high-level policy utilizes: (1) world building to create a global representation of the environment in which the robot is traversing, and (2) a multimodal large language model to interpret environmental and robotic states along with linguistic instructions to generate new target positions, $\mathbf{g}_t$, for the robot to navigate to next. As shown by the experimental results, this enables restorative waypoint generation used to help navigate the robot out of corners and out-of-distribution errors – which are known problems of DRL-trained navigation policies [18, 71]. This also potentially enables changing mission objectives in response to user based commands, along with adjusting and toggling various modules within the full-stack.

2.5.1 World Building & Ray Tracing

To construct a global map (M_t), the robot’s exteroceptive observations are projected into a global spatial representation. This is achieved by extracting the depth information from each observation and using ray tracing to build a discrete ternary occupancy grid, where each cell represents one of three states: free (0), occupied (1), or unknown (−1).

Consider a 2D depth map $\mathbf{D}$, that represents the relative distances of all nearby objects w.r.t. the position and $\mathbf{p}_t$ and rotational $\mathbf{r}_t$ vectors of the sensor, as acquired at time t . This depth map can either come directly from depth sensors, or extracted from RGB images. Each pixel at index values (u, v) is first mapped to a corresponding 3D coordinate (x', y', z') in the camera frame, where x' corresponds to the width of the depth image, y' to the height, and z' to the depth that is perpendicular to the image frame. Using the assumed camera intrinsic

parameters, focal lengths (f_x, f_y) and principal points (c_x, c_y) , the normalized coordinates are computed as:

$$\mathbf{X}'_{norm} = \frac{\mathbf{U} - c_x}{f_x}, \quad \mathbf{Y}'_{norm} = \frac{\mathbf{V} - c_y}{f_y} \quad (2.6)$$

Where $\mathbf{U}$ and $\mathbf{V}$ are the matrices of index values of $\mathbf{D}$ corresponding to u and v . To account for the spherical distortion of the depth map, the exact ray length is calculated for each pixel and projected to the depth along the optical axis, z' :

$$\mathbf{Z}' = \mathbf{D} \oslash \sqrt{\mathbf{X}'_{norm}{}^2 + \mathbf{Y}'_{norm}{}^2 + 1} \quad (2.7)$$

Where $\oslash$ denotes element wise division. The full 3D Cartesian coordinates can be recovered via $\mathbf{X}' = \mathbf{X}'_{norm} \odot \mathbf{Z}'$ and $\mathbf{Y}' = \mathbf{Y}'_{norm} \odot \mathbf{Z}'$. Where $\odot$ denotes element wise multiplication.

A spatial filter is applied along the vertical Y' -axis, where points outside of a given range $[y'_{min}, y'_{max}]$ are discarded, effectively ignoring the ground and high-hanging obstacles (*e.g.*, telephone wires) that would not impede the vehicle's current trajectory. A horizon threshold is also applied on the depth, z' , to not see outside the predefined bounds of the map. The filtered 3D points are projected onto a local $X'Z'$ -plane.

A local occupancy grid is created by comparing the depth of each cell, z'_{grid} , against the detected obstacle depth, z'_{obs} , of the ray that passes through the cell. A cell is marked as free (0) if it is in front of the nearest obstacle, $z'_{grid} < z'_{obs} - \delta_{occ}$. It is marked as occupied (1) if its depth falls within a tolerance of the obstacle, $(|z'_{grid} - z'_{obs}| \leq \delta_{occ})$. Those lying behind an obstacle, $z'_{grid} > z'_{obs} + \delta_{occ}$, remain initialized as unknown (-1).

The resulting occupancy grid is rotated by the camera's rotational vector $\mathbf{r}_t$ and translated by its position $\mathbf{p}_t$, and then used to update the world map M_t in global coordinates (x, y, z) . A masking operation ensures that only newly observed free or occupied cells overwrite the

world map, preserving the memory of previously explored regions while updating it with more recent ones. Figure 2.1 shows a partially observed representation of M_t , along with linguistic commands and other state information as discussed further in Section 2.5.2.

2.5.2 Multimodal Large Language Models (MLLM)

Vision-Language-Action (VLA) [48, 49] models offer unprecedented levels of generalization by leveraging large-scale data and models. However, they require extensive fine-tuning to specific robot embodiments, sensors, objectives, and environments. Alternatively, MLLMs [115] are advantageous in their ability for zero-shot inference, meaning they do not require additional training when switching tasks, embodiments, and sensors. This is accomplished through the use of transformers [103] which tokenize the input data so that it does not need to adhere to specific dimensions and modalities. Further, they are trained on large-scale web data which encapsulates variable inputs and tasks over petabytes of data. The cost of having such high flexibility is in the computational overhead during both training and inference, because the size of the models requires several gigabytes of dedicated VRAM. Only a few GPUs are on the market which can execute some of the most cutting-edge MLLMs. Thus it is advantageous to leverage the large-scale knowledge and generalization of MLLMs for robotics, however is disadvantageous to the limited computing typically available to robotics.

To enable MLLM inference in remote robotics, it is assumed that the robot is in communication with a nearby server that can host an MLLM. This is typical in most applications which are connected via the edge through wireless links. The bottleneck in an MLLM/edge based policy is then in the latency which arises from: (a) collecting raw sensor data onboard the robot, (b) compressing and sending the data via a wireless link to the server, (c) processing the data on the server, (d) compressing and sending the data back to the robot, and (e) decompressing and interpreting the data onboard the robot. For this reason, the high-level policy is executed at a low frequency, or only when it is needed.

A trigger condition $T_t = \text{True}$ is used to determine if the high-level policy should be executed. If satisfied, the system halts the mid-level policy and constructs a multimodal query to interface with the offboard MLLM. A sparse trigger condition is implemented that evaluates when the progress towards the target is stalled, such as when the robot is stuck in a dead end or an infinite loop. The vehicle’s maximum forward progress toward the target is evaluated over a rolling discrete window of N waypoints. At the current time step t , the path history is analyzed over a temporal window spanning from step $t - N$ to t . The spatial progress metric, $\Delta\rho_t$, is calculated as the difference between the initial distance to the goal at the start of the window (d_{t-N}) and the absolute minimum distance to the goal achieved at any point within that window:

$$\Delta\rho_t = d_{t-N} - \min_{k \in [t-N, t]} (d_k). \quad (2.8)$$

The system requires a minimum history of N waypoints before evaluation can begin. Once populated, the high-level MLLM policy is invoked if the maximum forward progress falls below a predefined stagnation threshold, ϵ . The trigger condition T_t is formally defined as a Boolean status:

$$T_t = \begin{cases} \text{True}, & \text{if } (\Delta\rho_t < \epsilon) \\ \text{False}, & \text{otherwise.} \end{cases} \quad (2.9)$$

When $T_t = \text{True}$, the system flags a stalled status, the lower tiers of the hierarchical framework are temporarily suspended, the robot’s current state maps and linguistic context are transmitted to the offboard MLLM to generate a restorative waypoint, and the temporal window is reset to not trigger a subsequent false positive when detecting a stalled status.

This MLLM query consists of two primary components, as illustrated in Figure 2.1: a synthesized visual state map with clearly indicated key features, and a structured linguistic

prompt with commands and context that links visual features to precise coordinates. An intuitive alternative to this proposed multimodal query is to establish a deterministic mapping between specific spatial coordinates and pixel indices within the image array, by providing a visual XY axis system in M_t . However, state-of-the-art MLLMs such as Gemma 3 [96] frequently employ dynamic rescaling and tiling techniques during visual pre-processing, such as the Pan & Scan (P&S) algorithm as inspired by LLaVA [64], to handle varying input resolutions. This dynamic rescaling breaks the mapping between pixel indices and spatial coordinates, leading to severe precision loss when generating waypoints. This mapping is further complicated by visual language model (VLM) encoders, such as SigLIP [119], which tokenize the input.

Instead of passing raw, high-dimensional sensor data (*i.e.*, RGB-D) directly to the MLLM, the perceived environment is extracted into a state map which is a 2D aerial view represented in RGB color space, $M_t \in \mathbb{R}^{H \times W \times 3}$. This design choice is specifically tailored to the architecture of contemporary MLLMs, such as Gemma 3 [96], which utilize VLMs like SigLIP [119] to extract visual tokens. This visual feature extraction process is properly leveraged by projecting obstacles and mission-critical coordinates using highly distinct RGB color channels and primitive geometric shapes. These stark contrast features are more efficiently tokenized by the VLM’s visual encoder, ensuring that high-fidelity spatial representations are passed to the language model. The semantic markers projected onto M_t include, in bottom-to-top order of projection onto the map:

1. Free space is indicated by black pixels, which represents traversable space.
2. Obstacles are indicated by white pixels, which represents non-traversable space.
3. Undetermined space is indicated by grey pixels, which represents unexplored or uncertain regions which may or may not be traversable.
4. The robot’s historical trajectory, up to time t , is indicated by purple lines and circles.

5. Any previously generated waypoints are indicated by red triangles, which provides context into attempted restorations in the past, for higher level reasoning.
6. The start position at $t = 0$ is shown by a green cross.
7. The robot’s position $\mathbf{p}_t$ is indicated by a blue square.
8. The global target is indicated by a yellow star.

To bridge the visual representation with semantic reasoning, M_t is paired with a structured text prompt. Since MLLMs rely on cross-modal attention mechanisms to fuse visual tokens with text, pairing the distinct shapes and colors in M_t with explicit linguistic definitions and spatial coordinates forces a strong semantic alignment. A standard query follows the template in the following paragraph.

“You are the supervisor for a robot tasked to navigate to a target position. The robot can only see what is in front of it, but you have access to the full map and path history. The robot can navigate on its own, but it has just gotten stuck on a complex set of obstacles and is asking for your help to generate an intermediate waypoint to get unstuck and make progress towards the target. See the given image for reference, which is an illustration of the current map and robot state. The provided image was created by updating an occupancy grid through ray tracing of the robots exteroceptive depth sensors, IMU, and GPS, thus is only partially observed. The robot (blue square) is at (x_r, y_r, θ_r) as measured in (meters, meters, degrees). The target (yellow star) is at (x_t, y_t) . The start (green cross) is at (x_o, y_o) . The robot has traveled with a path history (purple lines and circles) of (x, y, theta) coordinates: $[(x_1, y_1, \theta_1), \dots, (x_n, y_n, \theta_n)]$. Consider the previous attempted waypoints (red triangles), which were generated to try and unstuck the robot to no avail, are at the following (x, y) coordinates: $[(x_1, y_1), \dots, (x_m, y_m)]$. White pixels are obstacles. Black pixels are safe. Gray pixels are unknown. Only generate a waypoint at a safe (black space) (x, y) coordinate, and one that has not been attempted yet. Identify any complex obstacles

that may block the robot from reaching the target. Specifically format your response as: [STRATEGY]: reason for generating the waypoint. [WAYPOINT]: (x, y). Do not include any conversational filler.”

By explicitly defining the geometric relationships, the MLLM’s visual encoder is grounded in text, reducing spatial ambiguity. The pre-trained MLLM processes this multimodal input pair and generates a new spatial coordinate. This output acts as a strategic intermediate waypoint, routing the robot out of the current deadlock scenario. Once generated, this new waypoint replaces the target $\mathbf{g}_t$ for the mid-level policy, the trigger condition T_t is reset to False, and high-frequency control using the lower tiers continues.

2.6 Experimental Results

To evaluate the navigation reliability and computational efficiency of the entire framework, navigation experiments are conducted within the HIL environment. To ensure unbiased evaluation, the offline A* algorithm is utilized to generate a static, hold-out testing set of 4,400 paths. There are 1,100 paths for each of four maps, and each varying in path complexity. The A* algorithm verifies that a valid route exists for every path in the test set. Performance is quantified by the navigation accuracy, which is the percentage of test episodes where the robot successfully navigates to within the target radius before exceeding the maximum step limit, L_{LIMIT} . Statistical variance is mitigated by using a constant random seed during evaluation, and setting the “temperature” parameter of the MLLM to zero.

The mid-level policy is trained following the procedures outlined in Section 2.2.3, by training on the AirSimNH map, over 14 distinct difficulty levels, using 2,000,000 total training episodes, and retraining with 10 random restarts. This takes four days to execute, as distributed across the IASL AI-accelerated servers. During training, the descritized cache

system (Section 2.2.3 was used rather than direct queries to the simulator. This reduced training time by a factor of 184, which would have otherwise taken 736 days to finish. During evaluations, the low-level controller is used instead of the cache system to simulate action commands in real-time. This requires some calibration of the low-level policy, to not overshoot or undershoot waypoints, collisions, or out-of-bounds positions.

Over the given evaluation set for the AirSimNH map, using the low-level controller with the mid-level policy, and sampling 10 paths for each of the first 10 difficulties (for feasible test times), 94% of the paths successfully reach the target position within the given time constraint. When evaluating on the mid-level controller without the low-level controller (using only the cache-based system), the same percent of paths (94%) reach the target position. This demonstrates the robustness of the training procedures by splitting it into a mid-level policy that leverages the cache-based systems, and deploying to a real time low-level controller (which can take up to three orders of magnitude longer to execute) without any loss in accuracy. More thorough evaluations of the DRL-trained mid-level policy are demonstrated in Chapter 3.

The remainder of this section is dedicated to integrating the high-level policy into the framework. Specifically, the Gemma 3 MLLM, released by Google Gemini, is used. To quantify the tradeoff between zero-shot spatial reasoning and inference latency, the high-level policy is benchmarked using three distinct parameter scales of the Gemma 3 architecture (4B, 12B, and 27B). The navigation accuracy is first investigated, as a function of the Gemma 3 model size and the AirSim environment. In this baseline experiment, the entire map is fully observable to the high-level policy. The ground truth occupancy grid is taken directly from AirSim, bypassing the ray-tracing component, to establish an upper performance bound for the MLLM. Table 2.1 lists the results, which is evaluated against the entire evaluation set, and “none” indicates using only the mid-level policy.

The mid-level DQN policy was trained exclusively within the AirSimNH environment. Con-

Table 2.1: Navigation accuracies using the fully observed map.

MLLM model	AirSimNH	Blocks	Coastline	CityEnviron
none	84.91%	83.71%	63.73%	44.55%
gemma3:4b	92.91%	89.57%	69.64%	57.91%
gemma3:12b	94.55%	88.71%	73.00%	64.27%
gemma3:27b	96.09%	92.43%	77.73%	70.73%

sequently, the baseline accuracy, labeled as “none” in table 2.1, drops significantly in the other three environments due to out-of-distribution generalization errors. The integration of the high-level MLLM policy successfully mitigates this overfitting, and overall yields substantial accuracy improvements, establishing its efficacy in resolving deployment errors. The accuracy increases monotonically with model size across all environments, highlighting a clear tradeoff between computational efficiency and navigational reliability.

While the asynchronous stall detection mechanism is robust, it remains susceptible to false positives, occasionally triggering the MLLM when the robot is navigating around large obstacles. This can be mitigated by expanding the temporal window (N) used to calculate progress; however, doing so consequently delays stall detection and requires relaxation of the maximum time limit, L_{LIMIT} . Since the optimal value of N is a function of specific mission requirements, the performance of the MLLM is isolated by directly measuring its recovery rate. Thus, the successful recover rate, SRR , is defined as the percentage of episodes that successfully reached the target after initially failing under the sole execution of the mid-level policy. Conversely, the failed recovery rate, FRR , tracks instances where MLLM-generated waypoints actively diverted the robot away from a viable trajectory or introduced new failure modes. Specifically:

$$SRR = \frac{\sum_{i=1}^N (F_{MLP,i} \wedge S_{MLLM,i})}{\sum_{i=1}^N F_{MLP,i}} \times 100\% \quad (2.10)$$

Where N is the total number of episodes. $F_{MLP,i}$ is a binary indicator (1 if the episode failed

Table 2.2: Successful recovery rates (SSR) using the fully observed map.

MLLM model	AirSimNH	Blocks	Coastline	CityEnviron
gemma3:4b	+59.64%	+35.96%	+33.08%	+25.57%
gemma3:12b	+67.47%	+31.58%	+40.85%	+37.70%
gemma3:27b	+77.71%	+53.51%	+52.38%	+48.03%

Table 2.3: Failed recovery rates (FRR) using the fully observed map.

MLLM model	AirSimNH	Blocks	Coastline	CityEnviron
gemma3:4b	-1.18%	-0.00%	-9.56%	-1.84%
gemma3:12b	-0.64%	-0.17%	-8.70%	-2.65%
gemma3:27b	-0.64%	-0.00%	-7.85%	-1.02%

under the mid-level policy, 0 otherwise). $S_{MLLM,i}$ is a binary indicator (1 if the episode succeeded after MLLM intervention, 0 otherwise). The $\wedge$ is the logical “AND” operator, denoting when both $F_{MLP,i}$ and $S_{MLLM,i}$ are equal to a value of 1.

$$FRR = \frac{\sum_{i=1}^N (S_{MLP,i} \wedge F_{MLLM,i})}{\sum_{i=1}^N S_{MLP,i}} \times 100\% \quad (2.11)$$

Where $S_{MLP,i}$ is a binary indicator (1 if the episode succeeded under the sole mid-level policy, 0 otherwise). $F_{MLLM,i}$ is a binary indicator (1 if the episode failed after the MLLM was introduced, 0 otherwise).

Tables 2.2 and 2.3 list the successful and failed recovery rates. Scaling the MLLM parameter size not only increases SSR but also decreases FRR . Notably, failure rates are most pronounced in the Coastline environment, which inherently has the largest distribution shift from the AirSimNH environment, the largest spatial area, and most open space.

Having established the theoretical upper bound using fully observed maps, evaluations are next made under practical edge-deployment constraints. In this configuration, the MLLM relies entirely on the global map, M_t , that is dynamically constructed via the robot’s depth sensor and ray-tracing, as opposed to using the ground truth fully explored map. This results

Table 2.4: Recovery rates using the partially observed map.

MLLM model	AirSimNH	Blocks	Coastline	CityEnviron
gemma3:4b	+54.82%	+28.95%	+25.81%	+25.08%
gemma3:12b	+65.06%	+36.84%	+41.60%	+36.07%
gemma3:27b	+80.72%	+64.91%	+53.63%	+50.82%

in a partially observed, uncertain representation of the environment. The results listed in table 2.4 show the recovery rates for using the partially observed global map M_t with the ray-traced occupancy grid. Note that only the *SSR* is reported going forward for two main reasons: (1) the failed recovery rates can be mitigated by improving trigger conditions to query the MLLM, and (2) evaluating against the full sample size (4400 paths) takes almost a full day to compute, however only computing the recovery rate significantly reduces the sample size (to 1289 paths) and brings compute time down to hours.

Table 2.4 shows that using the partially observed map results in small changes in recovery rates, as compared to Table 2.2, with the smallest drops in the recovery rates coming from the largest Gemma 3 model (27b). This demonstrates the robustness of the partial maps, and that the full map outside of view of the robot is not as advantageous to generating recovery waypoints. Counter-intuitively, the recovery rates slightly improve for some maps, as compared to using the fully observed maps, which is a phenomenon more explored in the following paragraphs.

An ablation study is conducted to evaluate the contribution of individual components within the high-level policy, specifically focusing on the multimodal prompt formulation. Table 2.5 reports the aggregated *SRR* and relative path lengths across all AirSim maps for various system configurations. Unless otherwise specified, the default configuration utilizes the Gemma 3 (27B) model, partially observed maps, and comprehensive visual-language inputs (including path and waypoint histories). The relative path length is defined as the average ratio, calculated over all successful episodes, of the total steps required to reach the target under

a specific configuration compared to the default setup:

$$RPL = \frac{1}{|S|} \sum_{i \in S} \left(\frac{L_{configuration,i}}{L_{default,i}} \right) \quad (2.12)$$

Where S is the set of episodes that successfully reached the target under both configurations. $|S|$ is the total count of those successful episodes. $L_{configuration,i}$: The total steps (path length) taken in episode i using the specific configuration. $L_{default,i}$ is the total steps taken in episode i using the default setup. This metric is critical because it directly correlates with navigational efficiency, which is paramount for mission success in edge robotics where strict time, power, and energy constraints apply.

Configuration *(I)* establishes the baseline performance of the default framework. Components of the prompt are systematically modified or ablated to isolate their impact. These independent modifications include: *(II-III)* scaling down the MLLM parameter size; *(IV)* omitting explicit spatial coordinates from the linguistic prompt, forcing the model to rely entirely on the visual state map guided only by drawn x and y axes with labeled tick marks; *(V)* removing the waypoint history, preventing the model from recalling previously failed recovery attempts; *(VI)* removing the visual state map M_t , forcing the model to rely strictly on linguistic, coordinate-based instructions; *(VII)* removing the historical trajectory, blinding the MLLM to previously traveled paths; and *(VIII)* replacing the ray-traced maps with ground-truth occupancy grids extracted directly from AirSim’s voxel data.

Configurations *II-III* warrant that performance is strongly correlated with the MLLM parameter size, demonstrating that advanced reasoning directly scales with model size.

There is a large degradation in performance resulting from omitting the explicit linguistic coordinates, as in configuration *IV*, forcing the MLLM to rely solely on the coordinate system drawn onto M_t . This generates more erratic waypoints that are further from the agent’s position, and warrants that linking linguistic coordinates to visual map features is a

Table 2.5: Ablation study, showing the total recovery rate and relative path length over all AirSim environments.

Configuration	<i>SSR</i>	<i>RPL</i>
(I) WayFinder (default settings)	56.79%	1.00
(II) Using Gemma3:4b	30.88%	1.17
(III) Using Gemma3:12b	41.74%	1.17
(IV) No Coordinates in Language	12.26%	1.46
(V) No Waypoint History	48.02%	0.99
(VI) No Visual Component	55.62%	1.10
(VII) No Path History	52.37%	1.12
(VIII) Ground Truth Occupancy Grid	53.45%	1.05

robust prompt engineering technique. It is posited that this is a general trend, that can be applied to many types of spatial reasoning problems.

Omitting the waypoint history can trap the agent in an infinite loop, as the ungrounded MLLM repeatedly generates the same failed waypoint without memory of prior attempts. Configuration *V* reflects this with a corresponding drop in the evaluated recovery rate.

Interestingly, removing the global map M_t , as done in configuration *VI*, only moderately degrades performance. It is posited that the MLLM is capable of extrapolating global spatial structures from the linguistic coordinate arrays provided for the start, current, target, and historical path positions. This is further warranted by the degradation in performance that results from removing the historical path points from the language prompt (configuration *VII*).

Counterintuitively, substituting the ray-traced maps with perfect, ground-truth occupancy grids from AirSim, configuration *VIII*, results in a performance degradation. The ray-traced maps provide a localized, partially observed view that reflects the explored vicinity. It is posited that providing the localized maps, as opposed to ground truth global ones, explicitly forces the MLLM to reason based on the robot’s perspective thus allowing for exteroceptive-based logic.

2.7 Summary

This chapter demonstrates an end-to-end 3-tier hierarchal framework for autonomous navigation of a robot through an unknown environment. Results show that the mid-level policy can be trained, via DRL, independently of a low-level controller, by demonstrating zero degradation in navigation accuracy. This enables the formulation of general logic, imbued into the mid-level policy, which can then be deployed to robots with varying embodiments without the need for further training. Further, by decoupling low-level motor control from mid-level logic and reasoning, the training time of the DRL algorithm is shortened by a factor of 184 – bringing down estimated training times from 763 days to 4. Evaluations show that the low-level controller can be integrated back into the control stack with zero loss in accuracy, given some level of calibration of the PID and sensitivity variables. A high-level policy can then be integrated into the stack. A visual state map is constructed as the robot traverses the unknown environment, through ray tracing. Offboard, a MLLM is used to then input the state map with linguistic commands to help navigate the drone out of dead-locks during deployment. This shows an increase in accuracy of up to 27.5%, as evaluated against diverse AirSim maps which are susceptible to out-of-distribution errors. Notably, the high-level policy is deployed without the need for additional training, further showing the flexibility of the hierarchal framework.

Ultimately, such a hierarchal structure enables robust deployment to diverse environments, robot embodiments, and sensor configurations. This is accomplished by its modular structure, allowing for low-level control to be defined by the firmware, mid-level policies to be rapidly trained, and high-level policies to augment the entire stack with additional capabilities, optimizations, and higher level logic.

Chapter 3

Adaptive Dynamic Deep Neural Networks

Chapters 1 and 2 demonstrate a full autonomy stack to evaluate navigation methodologies and benchmark resource costs. This chapter leverages that testbed to further develop adaptive dynamic deep neural networks (ADDNN). The hierarchical approach in chapter 2 demonstrates using more complex logic as necessary. ADDNNs follow the same root theory by considering a neural network as a set of hierarchal building blocks.

By contrast, deep neural networks (DNNs) enable advanced perception and logic, but are inherently static, only having one level of complexity. Dynamic DNNs (DDNNs) allow for runtime branching and scaling, providing multiple levels of complexity, but lack context awareness to rationalize when to activate what levels.

This chapter demonstrates that ADDNNs are uniquely robust in that they change their structure at run-time, activating specific levels, in response to perceived context. Significantly, this enables ADDNNs to break the traditional trade-off between efficiency and accuracy by simultaneously improving both.

3.1 Problem Formulation

Consider the development of robot policy ϕ that generates sequences of actions to both efficiently and accurately reach a target position within an unknown terrain. Executing the policy results in stepping through consecutive sensing-computing-acting stages that continue until the vehicle reaches a target (success) or a time limit is exceeded (failure). During the sensing stage, a set of observations are acquired from onboard sensors that are added to a first-in-first-out (FIFO) queue, X , of temporal length τ . During the computing stage, X is processed via an adaptive pipeline to predict: (1) motion actions, A , that place the vehicle on a length-optimal trajectory towards the target; and (2) a gate value, G , that corresponds to a specific scale or subset of model parameters, $\hat{\theta} \subseteq \theta$, to be utilized by the pipeline at the next computing stage. During the acting stage, the vehicle executes A and allocates the proper computing resources as required by G .

We wish to develop the policy ϕ such that it minimizes resource costs while maximizing navigation accuracy:

$$\begin{aligned} \operatorname{argmin}_{\phi} \quad & \langle E_{\phi} \rangle \\ \text{s.t.} \quad & \langle \eta_{\phi} \rangle \geq \eta_0 \end{aligned} \tag{3.1}$$

where $\langle E_{\phi} \rangle$ is the expected efficiency when operating under policy ϕ , η denotes navigation accuracy, and η_0 is a constant representing a baseline accuracy. In this work, η_0 is set to the maximum accuracy achieved from some static, high-fidelity baseline – as measured using the simulator component of the testbed described in Section 1.1 with Microsoft AirSim. Efficiency E is defined as the energy expended during the computing stage, which is correlated to power consumption and inference latency – as calculated by the HIL component of the full-stack as described in Section 1.1 with an NVIDIA Jetson device.

3.2 Background and Related Work

Small autonomous vehicles, such as drones, rely heavily on lightweight sensors that frequently lack the precise 3D environmental data necessary to navigate unknown terrains. To compensate, modern systems employ Deep Neural Networks (DNNs) for complex computer vision tasks, including monocular depth estimation (MDE, Section 2.3) [25, 60, 31, 67], object detection [122], segmentation [70], and classification [56]. Among these, depth maps are particularly essential for perception-based decision-making. RGB-Depth data are widely used by State-of-the-Art (SotA) approaches to enable autonomous robotics via Deep Reinforcement Learning (DRL) [80, 51, 55, 104], typically processed through Convolutional Neural Networks (CNNs). Such CNN policies constitute a mid-level policy, as discussed in Chapter 2. However, executing these models requires processing high-resolution inputs for every frame. The substantial computational demands of these networks conflict with the strict limitations of onboard resources, risking mission failure due to rapid battery depletion or high inference latency that degrades the vehicle’s reaction time. Consequently, there is a critical need for low-complexity approaches that balance computational efficiency, perception fidelity, and navigation accuracy.

Current research addresses these computational bottlenecks through two primary methodologies. The first, static model reduction, simplifies DNNs via pruning [8], knowledge distillation [29, 1, 72], quantization [81, 4, 12], and direct design [24]. While these techniques reduce resource consumption, they often incur a degradation in task accuracy. Although static model reduction can occasionally yield negligible accuracy improvements due to inherent regularization effects, achieving simultaneous, significant gains in both accuracy and efficiency remains largely unfeasible. Furthermore, because static models must be computationally complex enough to handle the most challenging operational scenarios, they exhibit suboptimal efficiency when navigating simpler, less demanding environments.

The second methodology, edge computing [89, 33], mitigates onboard limitations by offload-

ing perception tasks to remote, high-performance servers. While this enables lightweight robots to leverage complex, higher-level logic, it introduces critical vulnerabilities. Relying on remote servers requires volatile communication links, exposing the system to unacceptable latency, compromised data privacy, and sudden connectivity loss in unpredictable environments.

To address the limitations of both static model reduction and edge computing, this work investigates a third paradigm: adapting dynamic DNNs [40] to scale computations at runtime in response to contextual demands. Dynamic DNNs adjust their architecture through mechanisms such as early exits [97] to scale active layers, slimmable networks [118, 117] to scale active nodes per layer, or multi-branch architectures [40] that route inputs through unique network pathways. Crucially, unlike methods that swap entire models at runtime and incur severe context-switching latency, dynamic DNNs are loaded into memory prior to deployment. This framework eliminates the reliance on volatile communication links while mitigating the inefficiency and accuracy degradation inherent in static networks.

Currently, SotA MDE networks [31] adapted for lightweight devices primarily rely on static model reduction [120] or knowledge distillation [113]. Although dynamic DNNs have seen isolated applications in MDE (*e.g.*, utilizing early exits [11]), comprehensive methodologies for selecting onboard scaling factors based on real-time navigation goals and environmental factors remain notably absent from the literature. Only a few approaches adapt scaling in response to intrinsic latent features or extrinsic environmental factors [66, 78, 62, 35].

Developing the adaptive logic necessary to respond to specific mission parameters and environments represents the primary challenge of this framework.

To bridge these gaps, this chapter presents adaptive and dynamic approaches wherein computational load is scaled at runtime. By maintaining high-fidelity perception only when strictly necessary for navigation reliability, this work establishes a framework for effectively deploying adaptive dynamic DNNs (ADDNN) in environments that demand both uncom-

promising accuracy and optimal energy efficiency.

3.3 Theoretical Motivation & Hypothesis Formulation

The advancement of robust machine learning models has been hindered by the need to sacrifice either efficiency or accuracy in favor of one over the other.

Jared Kaplan and OpenAI conducted an empirical investigation into the relationship between model size, data size, and training time [46]. This paper is now considered a foundational paper in machine learning science, finding that “performance depends strongly on scale, weakly on model shape”. Where performance refers to the cross-entropy loss of a trained language model. Their results demonstrate that the total number of parameters, N , amount of data, D , and compute training time, C , has a more significant impact on performance than how the parameters are ordered and structured in the neural network. Where order dictates either the depth (the number of layers) or width (the number of nodes in a given layer) of the neural network. Recalling that early exits scale depth and that slimmable networks scale width, this leads to the following logical conclusion:

Conjecture 1. *Under the assumption that scale dominates order, both early exits and slimmable networks can demonstrate the same performance with large enough scale.*

Kaplan and OpenAI further qualified the relationship between performance and scale as a power-law, such that diminishing returns in performance are received when increasing one of the three scales (N , D , C) while keeping the others fixed. The threshold of diminishing returns was quantified at an 8:5 ratio between $N:D$, as N relatively increases. The 8:5 ratio is a significant finding as it explores what happens after a critical threshold is reached as N approaches D . The classical bias versus variance tradeoff in statistics [28], commonly referred to in machine learning science as underfit versus overfit [23], tells us that the model overfits as N approaches D – lowering the performance on a test set with a different data distribution

as the training set. However, the empirical results from Kaplan and OpenAI [46] remarkably demonstrate that additional performance gains can still be obtained on the test set as $N:D$ increases past this threshold, tapering off at around 8:5. In fact, they found the difference in loss between training and testing sets to be a constant offset at this large scale of DNN training – eliminating problems of overfitting with increased scale assuming $N > D$.

These findings are paramount when arguing the validity and robustness of ADDNNs. Consider first, that the results in [46] indicate that as $N \rightarrow \infty$ and $D \rightarrow \infty$, under the assumption that $N > D$, performance approaches its optimal value. Now consider, second, that these results also demonstrate that performance improves with a power law relationship, receiving diminishing returns at around $N : D = 8 : 5$ – indicating that there are fewer data samples at this level which have potential to lead to improved performance. With these two considerations in mind, we can logically conclude that:

Conjecture 2. *Larger scales are needed for improved performance, but only needed for a unique set of problems corresponding to a decreasing percentage of the data population.*

The first part of conjecture 2 is being leveraged by modern research scientists. Robust vision models can transform raw images into complex features required for downstream logic, requiring sequential execution of 1-10 GigaFLOPs (Floating Point Operations) per inference. Extracting higher-level logic from heterogeneous data using Multimodal Large Language Models (MLLMs) is significantly more expensive, requiring approximately 50 GigaFLOPs to generate a single token – where one inference requires generating a variable sized sequence of several tokens. Current trends indicate the advent of even larger scale MLLMs requiring High Performance Computing (HPC) clusters to execute.

Conjecture 2 is troublesome when considering the computing hardware constraints imposed on autonomous robotics. The BAE RAD5500 is a radiation-hardened processor designed for deep space, that operates at a mere 20 Watts to ensure resilience against bit-flipping and

thermal limits, but this reliability comes at the cost of a low GigaFLOPS (Floating Point Operations per Second) range throughput and 1GB of SDRAM. Conversely, an NVIDIA RTX 5090 offers massive TeraFLOPS computing speed and a moderate 32GB VRAM but demands 575 Watts and substantial infrastructure, astronomically clashing with the constraints of deep space deployment. A more down-to-earth comparison is the NVIDIA Jetson Orin Nano, an embedded chip designed for lightweight robotics with 8GB RAM, that scales between 7–15 Watts and operates at roughly one-hundredth the speed of the flagship 5090.

This illustrates a clear dichotomy between the need for larger models and the constraints in being able to host them on available hardware. Conjecture 2 is the theoretical background into designing ADDNNs as aggregated building blocks, with accumulative hierarchal levels of complexity that can be scaled at runtime in response to perceived context. The second part of Conjecture 2 implies that higher level building blocks of an ADDNN (larger scales) are only needed to be activated in response to the more sparsely populated data distributions as they become encountered. This leads to the following hypothesis:

Hypothesis 1. *As data distributions shift into different domains throughout the life-cycle of a model, so will the corresponding performance-optimal model scales.*

The remainder of the chapter is dedicated to experimentation to provide evidence that supports Hypothesis 1. Embedded slimmable and multi-branch neural networks that scale the width of the networks are investigated, of which, under the assumption of Conjecture 1, the results can be extrapolated to other DDNN structures which focus on different ordering paradigms, such as early exits and depth scaling. The practical results of these findings will warrant the validity of ADDNNs, as especially applied to resource-constrained environments such as remote robotics where there is a significant clash between computing hardware and model scale. If Hypothesis 1 is true, then it is possible to obtain improvements in both accuracy and efficiency by changing the active scale of a network in response to the perceived data distribution shifts.

Evidence is already provided in literature [54] to warrant the validity of Hypothesis 1 in terms of compacting input data as applied to wireless sensor networks; showing a level of optimized “data aggregation” as a function “resource-constrained event-based systems” that trigger specific data routing paths in response to “when they detect the indicated phenomenon.” The idea of data aggregation is similarly presented in Section 3.7.1 by adaptively ablating input sensor data in response to context, as applied to a neural network used for navigation. However, the majority of this chapter focuses on collecting evidence to support a novel idea of “latent aggregation”, in which features of the learned latent space of a neural network are instead ablated, or otherwise compacted, in response to context.

3.4 Key Observations & Proposed Solutions

There are two key observations made during this work, while attempting to deploy ADDNN frameworks. The first leads to the conclusion that time series problems are required to prove Hypothesis 1. This is due to the following paradox:

Observation 1 (The Intrinsic Optimization Paradox). The latent space of a DNN is rich with intrinsic features that can be useful to determine the optimal scale given the current data distribution. However, once the latent space is realized there is no further need to optimize calculations as they have already been computed.

However, consider that two sets of input observations, X , are parametrically related to variable t , such that $X^{(t+1)}$ is correlated to $X^{(t)}$. Then the latent features realized during processing of $X^{(t)}$ can be leveraged to determine the optimal scale to use when processing $X^{(t+1)}$. This assumption is extremely valid for robotics applications, when considering t denotes time, which have small changes in the state space from one timestep to the next. Thus parametric relationships provide a solution to the paradox demonstrated in Observation 1.

The second observation was made while analyzing the learning curves yielded from training

multiple failed ADDNN frameworks:

Observation 2 (Unstable Dynamic Gradients). The error gradient becomes highly unstable when attempting to predict the gate values G that control that same network’s active parameters $\hat{\theta} \subseteq \theta$. This is likely because the random initialization of the weights leads to highly suboptimal predictions of G , trapping the gradient from approaching any local optima.

Observation 2 leads to several design choices to mitigate ADDNNs’ unstable gradients, which has resulted in several published frameworks [42, 41, 40, 43]. These frameworks use an overlapping methodology which is to decouple the adaptive mechanism from the dynamic mechanism into an auxiliary model. The dynamic mechanism defines how the neural network can be scaled at runtime, either by slimmable networks, multi-branch networks, or split computing (see Section 3.5). While the adaptive mechanism controls the dynamic portion, intelligently deciding when to scale and by how much. Dynamic neural networks are trained first, after which the weights are frozen and then the auxiliary model is trained second. Separating the adaptive-dynamic processes into a two-stage training process effectively mitigates the unstable gradients problem, as demonstrated through experimentation.

3.5 Dynamic Deep Neural Network Architectures

This section describes the architectures used as DDNNs, which define how the networks can be scaled at runtime. These networks have independent training procedures, typically utilizing a combination of self-distillation and supervised learning.

3.5.1 Embedded Networks & Slimmable

Consider a large neural network as a super-network, which contains a finite set of sub-networks that represent unique paths through the network. Each path corresponds to a subset of parameters that are embedded within the super-set of parameters, such that they

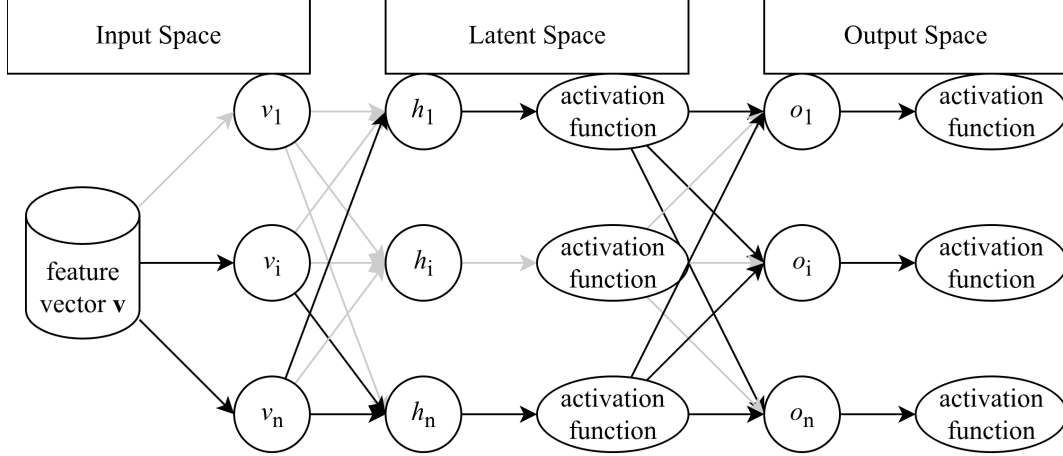


Figure 3.1: Embedded network with a selectable path in grey within the larger super-network comprised of a finite number of paths.

are aggregated from other subsets. Figure 3.1 illustrates this idea, showing a selectable sub-path within one network. In this example, one input node v_1 is deactivated illustrating the idea of data aggregation [54], and one hidden node h_i is deactivated illustrating the idea of latent aggregation. Some notable embedded networks in literature are used for semantic reasoning to create unique pathways for each cluster of data [90], and for early exits to stop execution of the network midway through a forward pass and skip to the final layers [97].

Slimmable networks [118, 117] are the most popular form of embedded networks, which consider the paths are cascading so that higher levels include: (1) all parameters at lower levels, and (2) some additional active parameters which will be part of all higher levels. The slimming factor, ρ , controls the percent of active nodes in each hidden layer of a neural network. Let us first consider this as applied to a multi-layer perceptron (MLP) [87, 88], and then to a convolutional neural network (CNN) [59] which requires some slight adjustments to address problems encountered in deep learning.

Consider an arbitrary hidden layer composed of a vector of nodes, $\mathbf{h}$, indexed from $k = 1$ to q , where q is the maximum number of nodes available in that layer. Here, q represents the capacity of the static “super-network”—defined in our context as a network provisioned to

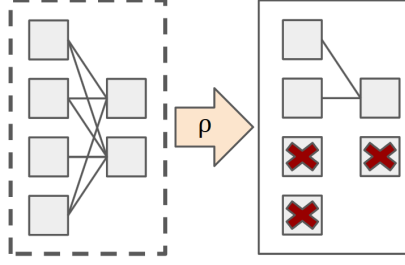


Figure 3.2: Procedure used to "slim" an MLP. The slimming factor ρ is used to scale the number of active nodes in each hidden layer. In this example, there are 4 nodes in the first hidden layer and 2 in the second (the super-network). Quantity $\rho = 0.3$ so that 2 nodes are deactivated in the first layer and 1 in the second (activating a sub-network). Note that all of the weights connected to any deactivated nodes, represented by the lines in between the hidden layers, are also severed. © [2024] IEEE. Reprinted, with permission, from [42].

handle the most complex operating scenario. This complete super-network is utilized when the width multiplier $\rho = 1$, whereas a reduced sub-network is activated when $\rho < 1$. For a given ρ , the number of active nodes in the layer is defined as $\lceil \rho q \rceil$, where $\lceil \cdot \rceil$ denotes the ceiling function. Consequently, the active nodes are those with indices $k \in \{1, \dots, \lceil \rho q \rceil\}$, while the deactivated nodes correspond to $k \in \{\lceil \rho q \rceil + 1, \dots, q\}$. This deterministic procedural deactivation, unlike the random Bernoulli masking employed in dropout [94], is essential to consistently extract specific sub-networks for both training and inference (see Figure 3.2).

Through this procedural deactivation, slimmable networks effectively employ a cascading deactivation strategy. In contrast, a non-cascading approach offers greater architectural flexibility, as a given configuration is not strictly required to encompass the nodes of smaller sub-networks. Ultimately, cascading procedures (e.g., slimmable networks) primarily aim to scale computational efficiency and maintain downstream task fidelity by activating a targeted percentage of the network. Non-cascading methods, however, are tailored to specific clusters or classes within the input data distribution.

The relationship between ρ and the total number of active parameters, m , in a sub-network is quadratic. Let u be the number of input layer nodes, l be the number of hidden layers, $\mathbf{q}$ 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 :

$$\begin{aligned} m(\rho) &= u\rho q_1 + \rho q_1 + \rho q_1 \rho q_2 + \rho q_2 + \dots + \rho q_l v + v, \\ m(\rho) &= (\sum_{i=1}^{l-1} q_i q_{i+1}) \rho^2 + (u q_1 + v q_l + \sum_{i=1}^l q_i) \rho + v. \end{aligned} \tag{3.2}$$

Thus, there is a quadratic decrease in the number of active parameters of a sub-network with a decreased value of the slimming factor ρ [42].

Knowledge distillation [29] is used, in the form of self-distillation, with supervised learning to train an embedded DDDN. Typically knowledge distillation is used between a teacher and student model to distill knowledge learned from the teacher to the student, however self-distillation is used to distill knowledge between the same latent space of a network during training. Algorithm 3 demonstrates how to properly calculate the error gradient during training of a slimmable network. The main idea of self-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. Specific to slimmable networks, the “sandwich rule” [117] is used as a method for sampling values of ρ by calculating the error gradient first with $\rho = 1$, distilling with ρ set to the minimal value, and then distilling with some intermediate values of ρ . For a classical slimmable network [118], the intermediate values of ρ are discrete values set before runtime – enabling discrete scaling at runtime. For a universally slimmable networks (US-Nets) [117], the intermediate values of ρ are sampled from a uniform distribution – enabling continuous scaling at runtime.

Algorithm 3 Calculating an Error Gradient for Training Slimmable Networks

Input: neural network f_θ , input data X , labels Y , *loss* function, *continuous* boolean.

set running error gradient, *grad*, to zero

$\hat{Y} = h_\psi(X, \rho = 1)$ ▷ Largest Slim

aggregate *grad* with *loss* given Y and $\hat{Y}$ ▷ Supervised Learning

$\hat{Y}' = h_\psi(X, \rho = \rho_0)$ ▷ Smallest Slim

aggregate *grad* with *loss* given $\hat{Y}$ and $\hat{Y}'$ ▷ Knowledge Distillation

for iterate over $i \in [0, 1, \dots,]$ **do**

if *continuous* **then**

$\hat{Y}' = h_\psi(X, \rho \sim U(\rho_0, 1))$ ▷ Continuous Slim

else

$\hat{Y}' = h_\psi(X, \rho = \rho[i])$ ▷ Discrete Slim

end if

 aggregate *grad* with *loss* given $\hat{Y}$ and $\hat{Y}'$ ▷ Knowledge Distillation

end for

Return: *grad*

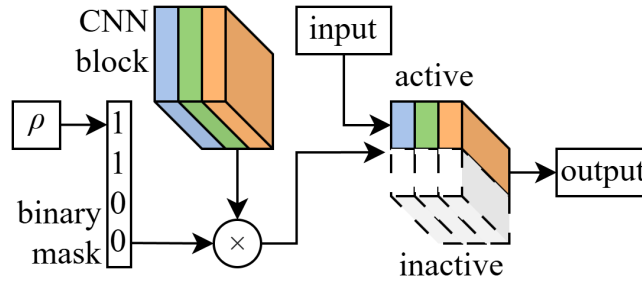


Figure 3.3: Active channels determined by a slimming factor, $\rho = 0.5$, set during inference of an example slimmable CNN with four maximum channels.

The same idea of self-distillation is later used to train other forms of embedded networks in Section 3.7.1, slightly modifying the hard and soft targets in Algorithm 3.

Figure 3.3 illustrates how a block with a convolution layer is transformed into a slimmable layer, for CNNs. Recall that in a convolution block, each convolution consists of applying a 2D filter (*width*, *height*) over the input feature map to the block. This results in an output feature map for a given number of channels (nodes), which is what is slimmed during a forward pass of the CNN. To train a slimmable CNN, an adjustment has to be made in how statistics are calculated during forward and backwards passes of the network. Typically, batch [34] or group [112] normalization is used to self-normalize the flow of data at each layer

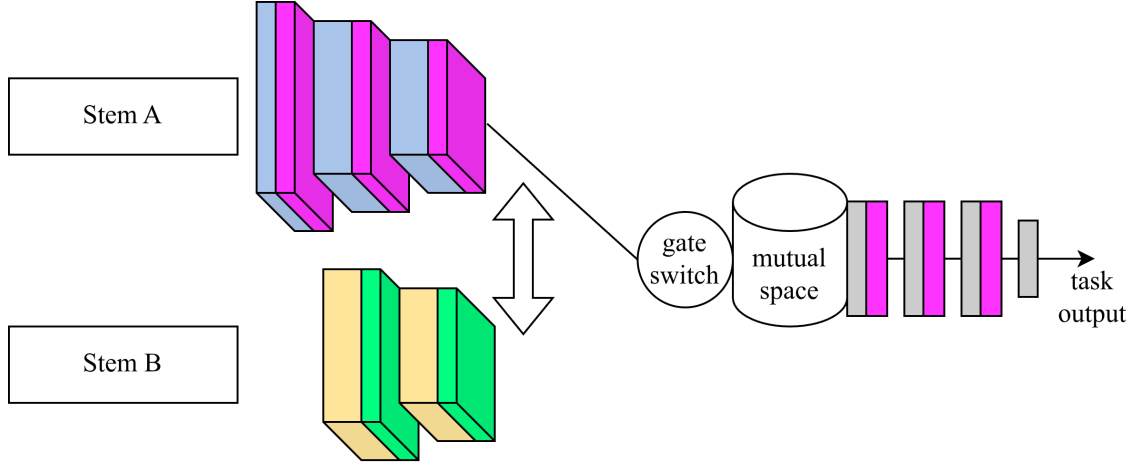


Figure 3.4: Multibranch network with a switch gate controlling which stem is activated in a forward pass.

to zero-mean and unit variance. However, this paradigm collapses for slimmable networks since the number of channels are changing during each forward pass – drastically altering the dimensionality and calculated statistics. To this end, switch norm [118] was proposed to calculate the statistics separately for each unique value of ρ . Although affective, note that this requires a discrete mapping, which is why universally slimmable networks for CNNs are not considered in this work, and all values of ρ are drawn from a discrete set. Further, it is empirically found that switch group norm leads to more stable gradients and converges to better global optima than batch does.

3.5.2 Multi-Branch Networks

Gated multi-branch networks are composed of several independent or intertwined paths, each indicating a unique structure of how to process data. Figure 3.4 illustrates a multi-branch network controlled by a gate, that toggles which input stem is used into a downstream task head. This example shows two stems, and one level, for the gate to select from; however, in practice the number of levels, stems, and accordingly selectable branches can be any value, creating complex intertwined paths.

There are a number of applications, in which a gated multi-branch neural network is devel-

oped for the purpose of optimizing resource usage. Testudo [78] considers wireless communication resources. HydraFusion [66] considers sensor selection for object detection. Slimmable encoders [2] consider levels of quantization used in split computing.

The largest advantage of multi-branch networks, is in their flexibility. Each path can handle unique input modalities (varying in resolution, dimensionality, and scope), network architectures (CNNs, RNNs, MLPs, transformers ...), and network structures (varying in complexity and order). Each branch can be loaded into the memory for seamless switching, if the memory capacity allows it. However, the largest disadvantage of multi-branch networks is in the context switching latency which arises when the memory capacity does not allow for seamless switching.

3.6 Auxiliary Models

The governing dual-optimization problem outlined in Equation 3.1 operates under the assumption of an optimal robotic control policy ϕ . To solve this problem, we must define how the optimal sub-parameters, $\hat{\theta} \subseteq \theta$, to a DDNN f_{θ} are selected at each time step. Further considering Observations 1 and 2, early generations of the presented ADDNN frameworks (Section 3.7) approach this by decoupling the architecture. An auxiliary model is trained independently of the DDNN to adaptively predict optimal sub-parameters for the next timestep:

$$G^{(t+1)} = g_{\psi}(X^{(t)}) \quad (3.3)$$

Where g_{ψ} is the “auxiliary model” with its own learnable parameters ψ , and G controls the gate of the DDNN f_{θ} used for both perception and navigation:

$$A^{(t)} = f_{\theta}(X^{(t)}, G^{(t)}) \quad (3.4)$$

If the optimization problem is tractable, then the parameters ψ of the auxiliary model can be learned through supervised learning. The number of parameters in a DDNN are large, upwards to billions; however, finite subsets of these can be used to create a small number of selectable embedded or branched paths in the network. The intractable portion of this problem is rooted in the space required to effectively explore time-series data; of which, in drone navigation, there are infinite possibilities. To this end, DRL is employed to leverage its exploration vs. exploitation strategy so that the entire space does not need to be explored. Thus, DRL is used to train the auxiliary models.

Later generations of the framework evolve this pipeline by separating the DDNN into an initial perception network, followed by a single joint action-adaptation head. Instead of the DDNN handling both perception and navigation, it acts as an intermediate feature extractor:

$$Y^{(t)} = f_{\theta}(X^{(t)}, G^{(t)}) \quad (3.5)$$

This intermediate perception output, $Y^{(t)}$, is then passed alongside the original input to the joint auxiliary model, g_{ψ} , which simultaneously predicts the next timestep’s gate and the current timestep’s action:

$$G^{(t+1)}, A^{(t)} = g_{\psi}(X^{(t)}, Y^{(t)}) \quad (3.6)$$

A key advantage of generating a motion action A paired with the gate control G is that interdependent relationships can be exploited. This is an intuitive relationship, since G affects downstream fidelity which affects the quality of A , and A changes the data distribution shift used to predict G . A disadvantage is that the navigation portion of the network is not being scaled. However, as outlined in Section 2.3, perception (particularly MDE) typically requires the use of a dominantly larger DNN than navigation. Thus scaling computations of

the perception network has a significantly larger impact, as later evidenced in Section 3.7.4.

3.7 Frameworks for Navigation

The frameworks presented in this section consider the task of processing onboard sensor data into motion actions and gate controls. The frameworks are designed as mid-level policies (Section 2.2) with DRL (Section 2.2.3) or IL (Section 2.2.2) and tested in the HIL environment (Section 1.1). Varied is the sensor configuration, the pipeline used to process data, and the dynamic-adaptive mechanisms used to optimize efficiency and accuracy.

Constrained robots, such as aerial vehicles, are typically equipped with lightweight sensors. This reduces the overall mass of the robot, decreases costs to build it, and reduces the footprint of the sensors in terms of power and energy. However, such sensors can provide limited information of the 3D environment which is otherwise needed to fulfill downstream mission tasks such as navigation. This is why various pipelines are considered that have different sensor configurations, along with the exact type of DNNs and other algorithms used to process the raw data into useful intermediate features and downstream actions.

To ensure the processing pipelines are properly scaled, their efficiency is optimized in terms of inference latency (test-time compute), power demands, and energy expenditure; each of which have profound affects on fulfilling mission objectives, especially when considering resource-constrained robots with compact batteries and embedded processors. Reducing inference latency results in improved reaction time of the robot to the environment, enabling fewer collisions and quicker times to reach the target position. Reducing power demands enables other asynchronous processes to be hosted onboard the embedded processor, which is otherwise severely limited in available power. Reducing energy expenditure consequently improves the battery life, and makes progress towards green IoT initiatives. In practice, the neural operations used to transform raw data into motion actions can take up to one third

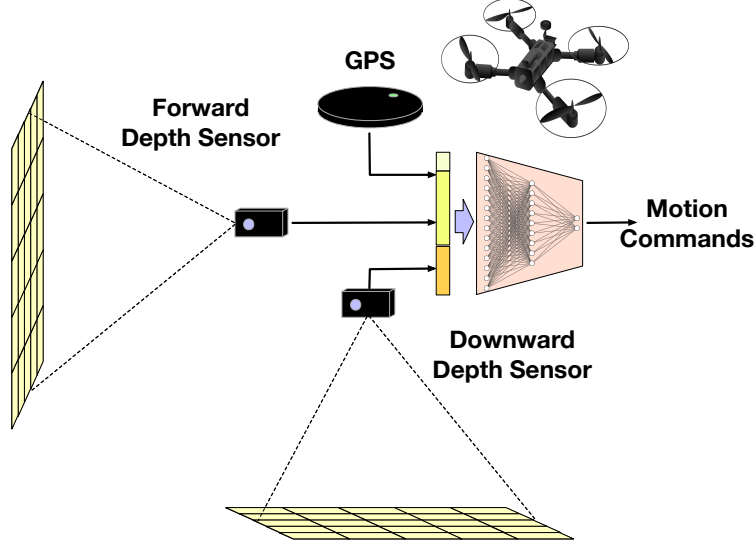


Figure 3.5: High-level schematics of NaviSlim, showing a forward and downward facing depth sensor with a neural network handling navigation actions. © [2024] IEEE. Reprinted, with permission, from [42].

of a drone’s energy reservoir. Since energy is directly correlated to power and time, it is the primary target variable that is minimized in the governing dual-optimization problem outlined in Equation 3.1 and the below frameworks.

3.7.1 NaviSlim

The work discussed in this section was published in [42].

The sensor configuration of NaviSlim [42] is illustrated in Figure 3.5. Considered is a drone 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. 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 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 [61].

The dynamic mechanisms that the auxiliary model controls are: (NaviSlim-S) which scales the power levels of the sensors used to capture raw depth data, and (NaviSlim-C) which scales the computing via slimming of the neural network used to transform raw data into motion actions.

The objective of *NaviSlim-C* is to minimize the resource usage of processing raw sensor data at the current timestep when executing the navigation network. This is accomplished by training the navigation network as a slimmable one, by utilizing Algorithm 3.

The objective of *NaviSlim-S* is to minimize the resource usage of sensor acquisition. This is accomplished by defining a self-distilled error gradient, Algorithm 4. This is a novel approach applied to sensing, that distills the network to have similar output regardless of what value the variable sensor array power levels, w_f and w_d , are set. Where w_f controls the power level of the forward facing depth sensor, is in the range $[1, 3]$, and a value of 1 corresponds to the minimal power level which uses the smallest area available for scanning. The range of w_d , corresponding to the downward facing depth sensor, 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 $w_f \geq 1$ 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 magnitudes of w_f and w_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.

Both NaviSlim-C and NaviSlim-S require supervised learning based training algorithms, to most effectively apply the knowledge distillation based gradient calculations. To this end, imitation learning is used to train the navigation network with the respective error gradient calculations.

Algorithm 4 Calculating the Error Gradient for Variable Sensing

Input: navigation model in training f_θ , set of inputs $FIFO$ (depth sensor observations at max power level placed into FIFO queues at each timestep), set of target outputs A , $loss$ function.

set running error gradient, $grad$, to zero

$\hat{A} = g_\psi(X)$

▷ Largest Power Levels

aggregate $grad$ with $loss$ given A and $\hat{A}$

▷ Imitation Learning

$FIFO' =$ re-sampled all observations in $FIFO$ with $w_f = 1, w_d = 0$

$\hat{A}' = f_\theta(FIFO')$

▷ Smallest Power Levels

aggregate $grad$ with $loss$ given $\hat{A}$ and $\hat{A}'$

▷ Knowledge Distillation

for a number of times, *i.e.* 2 **do**

$FIFO' =$ re-sampled FIFO queue with random w_f and w_d values applied to each observation, where $w_f \sim U(1, 3)$ and $w_d \sim U(0, 3)$

$\hat{A}' = f_\theta(FIFO')$

▷ Random Power Levels

aggregate $grad$ with $loss$ given $\hat{A}$ and $\hat{A}'$

▷ Knowledge Distillation

end for

Return: $grad$

The auxiliary model is trained to control adaptation of the dynamic navigation model. Specifically a DRL TD3 algorithm [21] is used to map the continuous multivariate observation space to a continuous multivariate action space. The output nodes correspond to the adaptive variables, either ρ (NaviSlim-C) or w_f and w_d (NaviSlim-S). A similar approach is used to train the model as in Section 2.2.3 for purely navigation, with a few exceptions. First, the navigation action A is not in the output space, instead only the gate value G is. Second, there is an additional term added to the reward function which penalizes using high adaptive variables. Third, there is no baked-in collision algorithm and avoidance relies purely on the generated actions from the navigation model. The resulting reward function is:

$$reward = \begin{cases} -\lambda_o & collision \\ \lambda_g & goal \\ \lambda_d \tanh(d) - \lambda_t - \lambda_c \rho - \lambda_s (w_f + w_d) & otherwise \end{cases}, \quad (3.7)$$

Where the λ terms are tuneable constants that weight each penalty and reward. The entire training process is:

1. Collect ground truth length-optimal paths using A* (Section 2.2.1)
2. Acquire sensor observations at each timestep along optimal paths
3. Use imitation learning (Section 2.2.2) to train a dynamic neural network (with Algorithm 3 or 4) that maps the collected observations to optimal A* motion actions
4. Freeze the navigation model, as to no longer update the trainable network parameters
5. Evaluate the trained navigation model in the testbed environment to measure the baseline navigation accuracy η_0 and efficiency
6. Use a TD3 DRL algorithm to train the auxiliary model
7. Evaluate the trained navigation-and-adaptation model in the testbed environment to measure the adaptive navigation accuracy η and efficiency

Figure 3.6 shows the percentage of evaluation paths which successfully reach the goal versus distance to goal, when using the navigation model and maximum adaptive variables – corresponding to the baseline accuracy η_0 . The percentage of successful paths drops with increasing distance, as expected.

The root mean squared error (RMSE) is calculated between length-optimal motions, from A*, and those predicted from the trained navigation network. Evaluated is how the error changes with varying values of the slimming factor, ρ , but without adaptation. Meaning, the values of ρ are set before runtime and are static throughout the evaluations. Figure 3.7 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 sub-par accuracy is obtained if a static sub-network is activated within the navigation super-network.

The core novelty of NaviSlim is to intelligently and dynamically adapt the value of ρ , given context, as to not detrimentally decrease navigation accuracy. To this end, the auxiliary

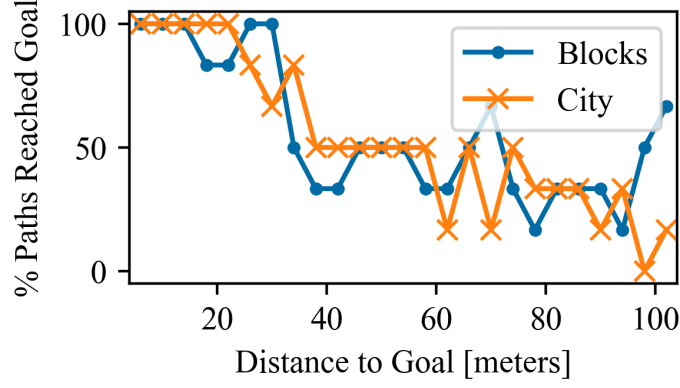


Figure 3.6: Results of a trained navigation model, evaluated in the testbed environment using the AirSim maps Blocks and CityEnviron. This figure shows the percent of evaluation paths that reach their goal as a function of starting distance. © [2024] IEEE. Reprinted, with permission, from [42].

Table 3.1: Adaptive split computing results.

Scenario	Map	Motion	$\bar{\rho}$	δ_C	$\bar{w}_f$	$\bar{w}_d$	δ_S
(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%

network is trained and evaluated 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 CityEnviron map with vertical motion locked. Table 3.1 lists the average predicted adaptation values from the auxiliary model. Where the “ δ ” column shows the relative resource expenditure reduction either for NaviSlim-C or NaviSlim-S. All values shown in Table 3.1 are calculated using the test set. We see a significantly higher mean value of ρ between each of these scenarios. The δ values in Table 3.1 are percent decreases measured w.r.t. the larger super-network (*i.e.*, the navigation network using the maximum adaptive values and corresponding to η_0). All evaluation paths that reached their target position with the baseline policy also reached their target position with the adaptive policy, demonstrating a zero-loss in navigation accuracy.

Figure 3.8 shows the learned average values of ρ at each position in the Blocks map, using

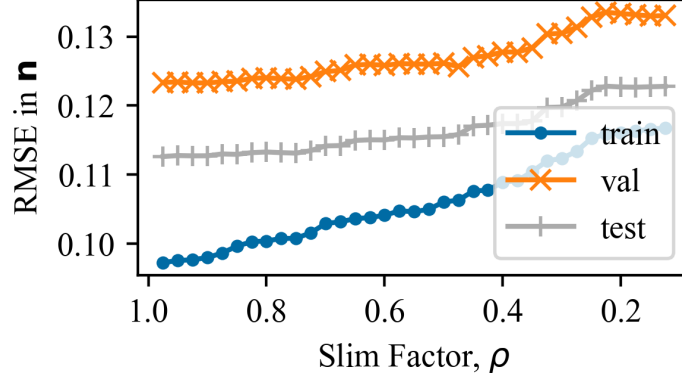


Figure 3.7: Shown is the Root Mean Squared Error (RMSE), calculated between the length-optimal navigation motions found from A* and those predicted from the trained navigation network, as a function of the slimming factor, ρ . © [2024] IEEE. Reprinted, with permission, from [42].

the test set. This figure is 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. The context clues are further isolated versus learned behavior in Figure 3.9, which shows the mean depth captured by each sensor versus ρ . Generally, when the navigation path is more complex there is a correspondingly high value predicted for ρ , which is expected behavior for correspondingly complex navigation and maneuvers. Interestingly, as the distance between the drone and objects increases, so does the value predicted for ρ . 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.

Resource costs (time, power, and energy) are next measured when using and not using the adaptive component of *NaviSlim* on a microprocessor similar to that typically deployed on micro-drones: a Jetson Nano (Section 1.1). Compared are the relative resource costs by

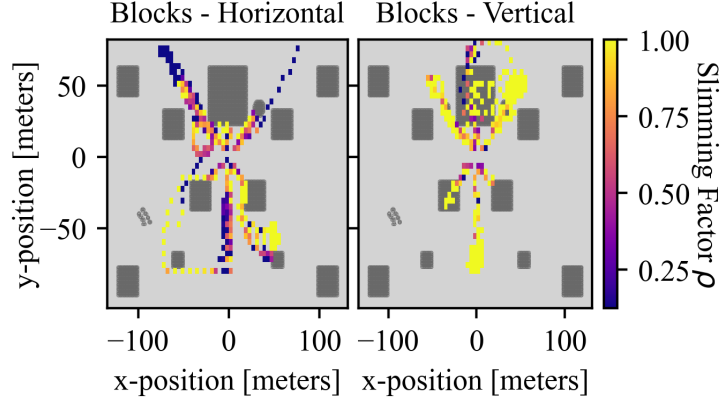


Figure 3.8: An aerial 2D view of the AirSim Blocks map overlaid with the average slimming factor, ρ , predicted from the auxiliary network at each position. The darker gray shapes indicate objects the drone can collide with, however note on the right panel the drone is also flying over these objects. © [2024] IEEE. Reprinted, with permission, from [42].

passing a static set of observations through: (1) *NaviSlim*, including both the auxiliary and navigation modules; then (2) only the navigation network but with $\rho = 1$ (*i.e.*, just the static super-network without the auxiliary network). The ratio difference is measured 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 Figure 3.10 which shows the relative speedup associated with *NaviSlim*.

The difference in power consumption is negligible, with a mean relative difference of 0.005 and standard deviation of 0.07, due to a static power level dictated by the Jetson Nano. 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. Figure 3.10 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

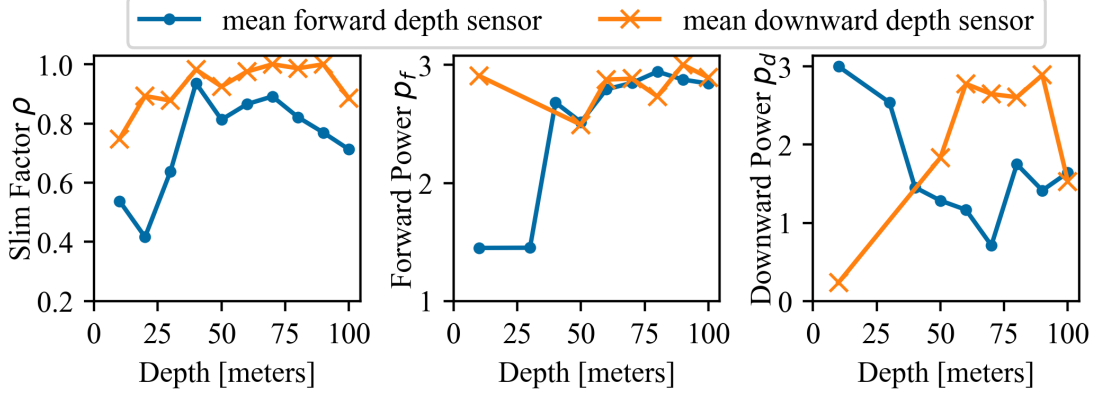


Figure 3.9: Mean results of adaptive variables as predicted from the auxiliary network, as a function the mean value of the depth sensor data input to the network during predictions. © [2024] IEEE. Reprinted, with permission, from [42].

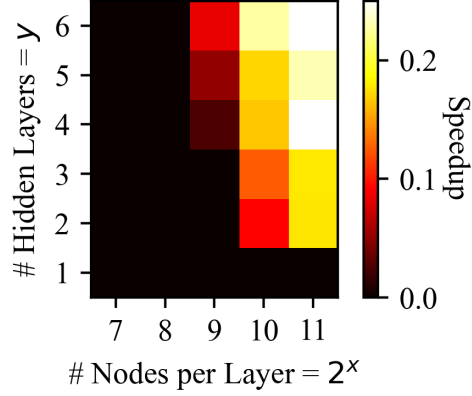


Figure 3.10: Jetson Nano benchmarks, measuring the relative speedups between using and not using the auxiliary model, as a function of navigation network size – while using a fixed auxiliary network size. © [2024] IEEE. Reprinted, with permission, from [42].

non-black grid spaces. This region is characterized by a positive speedup.

A question arises: is the size of the navigation network that results in a speedup justified? This is tested by conducting a hyper parameter search over the size of the navigation network. This is shown in Fig 3.11, which indicates that there is an area of overlap in which both the speedup is positive and RMSE is minimized – as a function of network size. There is a general trend that larger network sizes result in lower RMSE, corresponding to larger speedups, to some extent before overfitting occurs.

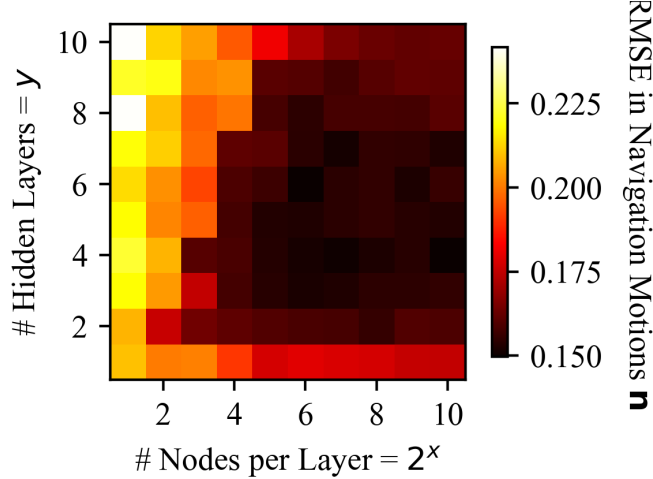


Figure 3.11: The navigation action RMSE as a function of the navigation network size. © [2024] IEEE. Reprinted, with permission, from [42].

Table 3.1 also lists the average power levels for scenarios (2) and (3) after evaluating the trained auxiliary model with the test set for NaviSlim-S. The mean value of w_f is greater than w_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 w_d value of 0.73.

Figure 3.9 shows that w_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 w_f , which is a similar relationship earlier observed with ρ – warranting that less resources are required when an object(s) is very near the drone. Another intuitive relationship is that w_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, after which w_d decreases.

NaviSlim is the first framework to dynamically scale computing and sensing used by a neural model for navigation of a resource-constrained drone. The training procedure obtains suc-

successful models that can safely navigate between two points, while using adaptive computing and sensing. An auxiliary 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. When deploying to the testbed environment, *NaviSlim* 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. Note that these efficiency improvements come without a degradation in accuracy, however do not yet demonstrate an improvement in accuracy.

3.7.2 NaviSplit

The work discussed in this section was published in [41].

While *NaviSlim* considers sensors which directly provide depth information, *NaviSplit* [41] considers a more difficult problem of perception by using MDE to extract the depth information from monocular RGB images. A further consideration is taken by considering the drone is on the edge and in communication with a nearby server. To this end, split computing (Section 2.4), is used to split part of the perception neural network onboard then relay that latent space to the server which is used to compute the rest of perception and navigation. The generated motion actions are then relayed back to the drone.

Figure 3.12 illustrates the system model for *NaviSplit*. An MDE DNN is trained as described in Section 2.3 using Microsoft AirSim data. The depth model consists of ten feature extraction blocks followed by a depth prediction block. Every feature extraction block includes a convolution layer, a group normalization layer [112], and a scaled exponential linear unit (SELU) [52] activation layer. The depth prediction block includes three convolution layers, one group normalization layer, two SELU activation layers, and a Sigmoid activation layer – where a value of one corresponds to over 100 meters away and a value of zero is directly

in front of the camera.

A split point is injected after the second block of the MDE network. This layer is further compacted using 8-bit quantization then communicated to the server to complete processing. Compared are three approaches for injection of the split point which all use a bottleneck approach [68] that first trains a teacher MDE network, then distills that learned knowledge into several smaller student networks with an injected and compressed split point:

1. (Baseline): The number of channels are scaled down at the split point at discrete values of 2, 4, 8, 16, or 32.
2. (Bottleneck_V1): The number of channels are scaled down at the split point at discrete values of 12, 24, or 48. Further, the kernel size is scaled down to 4x9 and an additional linear layer is added for further compression.
3. (Bottleneck_V2): The number of channels are scaled down at the split point at discrete values of 12, 24, or 48. Further, the kernel size is scaled down to 8x14 and an additional linear layer is added for further compression.

On the server, the intermediate perception variable (the predicted depth map $\mathbf{Y}$) is fused with state variables comprised of the IMU, GPS, and target coordinates. This is then input into a late-injection CNN for navigation, following the structure in Figure 2.1, and trained with a TD3 DRL algorithm. An auxiliary model is trained to input the predicted depth map along with the state variables, and predicted actions, to generate the gate control value G used decide which branch of the split network to use at the next timestep. The auxiliary network is trained with a TD3 DRL algorithm, adding a penalty to the reward function proportional to the size of the split point used by the given gate control value.

The teacher depth model, used to extract depths from a monocular RGB image, predicts objects further away with a higher extraction error. There is a linear increase in the root

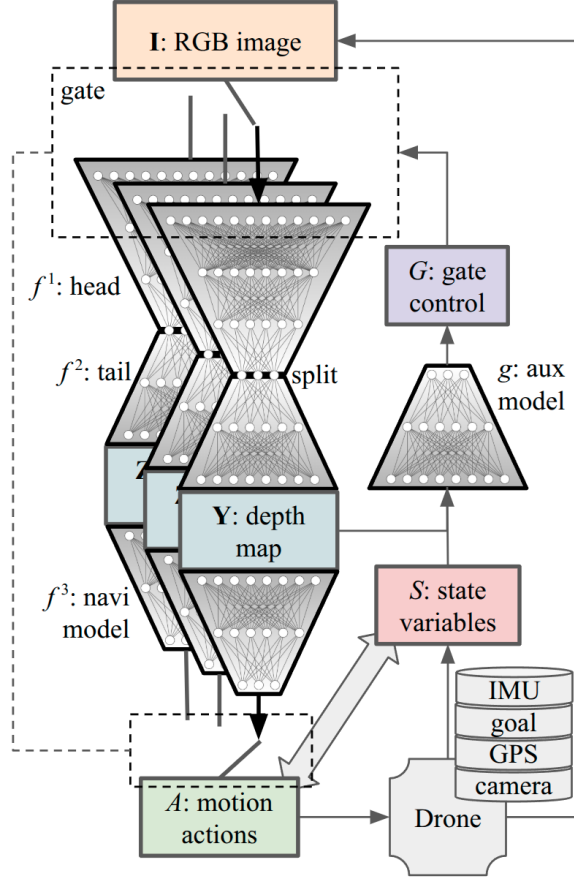


Figure 3.12: Several different split points with an encoder and decoder are injected into a teacher model to make multiple student model branches capable of split computing – of which an auxiliary model selects from given perceived context. Extracted depth maps are input to the navigation model that outputs motion actions used during drone navigation. © [2024] IEEE. Reprinted, with permission, from [41].

mean squared error (RMSE) from 9.4 to 21.2 meters, as the ground truth depth bins range from 10 to 90 meters. RMSE drops to 16.1 meters when extracting depths at the horizon (those clipped to 100 meters away).

Figure 3.13 compares the different student models against an edge computing approach that completely offloads the RGB image using JPEG compression (so that there is no neural network execution onboard the drone). The baseline models perform with equal memory consumption as the other models, however do not result in a lower error than JPEG. Alternatively, the Bottleneck_V1 and Bottleneck_V2 models perform with better error than that

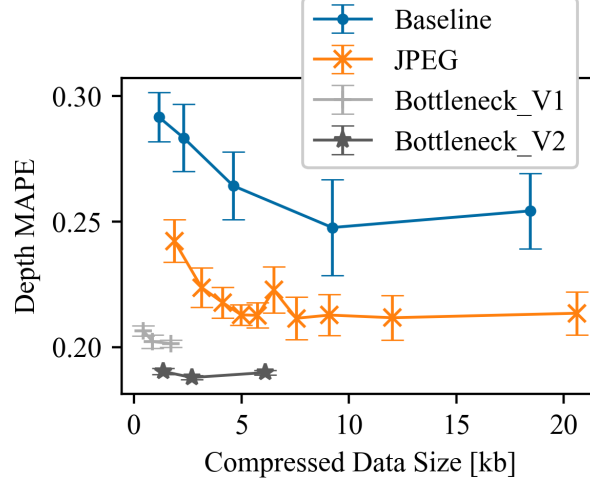


Figure 3.13: Comparing various compressed data sizes, corresponding to different models, versus resulting depth extraction error. The markers from left to right: for the Bottleneck_V1 and Bottleneck_V2 models, range between a reduction in channels of [12, 24, 64]; for baseline models, range between a reduction in channels of [2, 4, 8, 16, 32]; and for JPEG models, range in quality of compression from 5 to 95. © [2024] IEEE. Reprinted, with permission, from [41].

of the JPEG models - warranting that split computing with a reduced kernel size and linear layer is more robust than JPEG compression (particularly for MDE).

Using linear quantization slightly improves the depth extraction error, ranging in a decrease of mean absolute percent error (MAPE) of 0.8 to 4% – improving with increased compression. This is likely due to some level of regularization resulting from mapping the 32-bit floating point numbers to 8-bit unsigned integers, along with inherent clipping (eliminating possible outliers that may otherwise blow up towards infinity). This is a common, but counter-intuitive, trend in data reduction.

Table 3.2 lists the following benchmarks for the teacher, student, and auxiliary models: navigation accuracy, the normalized number of time steps per meter of initial distance, and the normalized compressed data size (communicated during SC) per meter of initial distance between start to target. For the two normalized benchmarks, only the successful paths are considered because unsuccessful ones terminate either early from collision or late from the

Table 3.2: Static student versus adaptive auxiliary models.

Model	Navigation [%]	Path [timesteps/m]	Encoded [KB/m]
Teacher (SoA)	82.2	0.1446	42.64
Student-1	81.6	0.1487	0.6856
Student-2	78.6	0.1504	2.772
Student-3	79.3	0.1436	10.58
Auxiliary	82.5	0.1489	2.12

max time step requirement. Normalization is needed, because different student models result in a different set of evaluation paths that successfully reach the target position.

Using the auxiliary model consistently results in a higher navigation accuracy as compared to using the student models independently, and in fact receives a slightly higher navigation accuracy than when using just the SoA teacher model. Further, the auxiliary model obtains this navigation accuracy using a substantially smaller encoded feature space. The only model which was executed using a smaller encoded space received a lower navigation accuracy due to it being over-compressed. The benefit of *Navisplit*, with an auxiliary model, is being able to select from several student models – thus improving navigation accuracy while using a dynamic encoded data size. **This is a significant finding, as it contradicts the previous assumption in model reduction that accuracy is either maintained or degraded.**

To evaluate the behavior of *Navisplit*, considered are all successful evaluation paths flown while using the auxiliary model. The gate control value for each student model is intentionally defined to increase in magnitude with increasing compressed data size. Figure 3.14 shows the mean gate control value predicted by the auxiliary model, $\hat{c} := G$, at each position. The apparent learned behavior from the auxiliary model is to use the smallest student model as often as possible – reducing the size of compressed data at the split point – and only activating more expensive student models as needed. Thus, the auxiliary model effectively increases navigation accuracy while minimizing the compressed data size used throughout the path. We see a low value of $\hat{c}$ for open roads and a high value of $\hat{c}$ when navigating

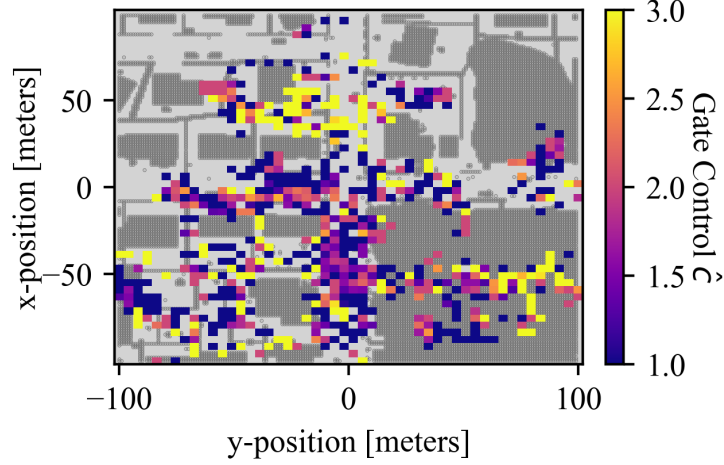


Figure 3.14: Mean value of the gate control, $\hat{c} := G$, for all successful paths while using an auxiliary model to adapt and select $\hat{c}$ at each time step. This is a demonstration of *NaviSplit*, which was trained and evaluated on the AirSimNH map. © [2024] IEEE. Reprinted, with permission, from [41]

around homes and other objects - warranting that a larger encoded data size is needed for more difficult scenarios.

The results warrant that *NaviSplit* is an effective multi-branched neural network architecture for drone navigation, that dynamically splits computing between an equipped processing unit and edge server. *NaviSplit* slightly improves navigation accuracy by 0.3% over a larger SoTA static model, while significantly reducing the communicated data rate by 95%. Further, the depth models used to extract intermediate features needed for the downstream task of navigation, performed with up to 81 mean absolute percent error when constructing 2D depth maps from a monocular RGB camera. These models were split and outperformed JPEG compression, which would otherwise communicate the entire image directly to the edge server rather than computing part of the DNN onboard the drone. Thus reinforcing that split computing is more robust than generalized JPEG compression, and **warranting that ADDNNs can result in both improved accuracy and efficiency.**

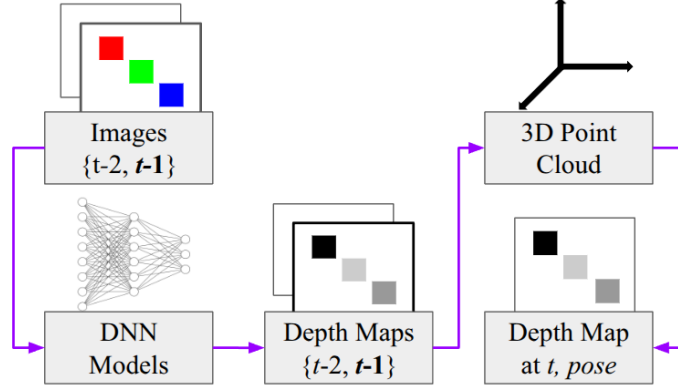


Figure 3.15: High level overview of using MADP to make one depth map prediction at time t , and at a proposed camera pose, based on the previous two steps. © [2025] IEEE. Reprinted, with permission, from [114].

3.7.3 SmartDepth

The work discussed in this section was published in [114].

SmartDepth [114] introduces a novel technique that leverages sensory input from previous frames to predict the future depth map before the camera reaches a specific location. This capability can be used both to reduce resource usage and to improve navigation performance. Figure 3.15 illustrates a single prediction based on the proposed framework, which leverages previous 2D depth map(s) to extrapolate a depth map at any given time and location. A downstream navigation model takes as input a combination of DNN and extrapolated depth maps to navigate to a target position. The tradeoff between resource usage and navigation performance can be scaled by intelligently selecting an efficiency level before runtime – as determined by considerations in expected energy expenditure, execution time, mean power, depth error, navigation accuracy, and path length.

Considered is a vehicle with a forward-facing monocular camera, navigating from a start to a target position while determining its relative location (*e.g.*, via GPS). Depth maps and pose data inform the navigation model, which selects motion actions at each step. SmartDepth aims to optimize path length, avoid collisions, and minimize computation. At each step, the

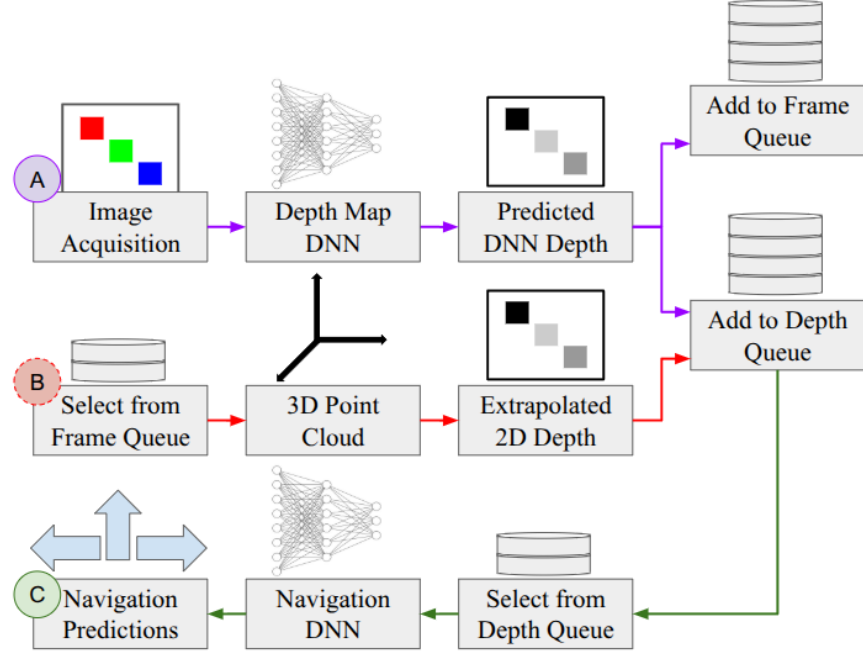


Figure 3.16: SmartDepth system model, illustrating the various pipelines to select from, either $A \rightarrow C$ or $B \rightarrow C$, in one observe-compute-action step. © [2025] IEEE. Reprinted, with permission, from [114].

vehicle either uses (a) an accurate but costly DNN-based depth estimation or (b) an efficient extrapolation algorithm. Figure 3.16 illustrates the system model. SmartDepth chooses between two pipelines, $A \rightarrow C$ or $B \rightarrow C$, at each step. Each pipeline has associated resource costs and depth prediction errors. Pipeline $A \rightarrow C$ corresponds to a navigation approach that predicts depth maps from an input camera image using a DNN. Pipeline $B \rightarrow C$ is a lower-complexity option that uses the extrapolation algorithm, reducing computational costs which may affect navigation performance. Both pipelines input the predicted depth maps into a navigation DRL-trained mid-level policy such as in Section 2.2.3. SmartDepth’s logic determines, at each step, which pipeline to use and how to select from each queue.

The extrapolation algorithm first considers a reference MDE depth frame as extracted from a DNN, as in Section 2.3. Given reference 2D depths $\mathbf{D}^{t'}$ at frame t' , the value at each pixel, (j, k) , as indexed by (row, column) of the image, is mapped from the 2D image space to 3D

global space using:

$$P_i^{(t')} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{R}_{t' \rightarrow \mathcal{O}} \cdot \mathbf{K}^{-1} \cdot \mathbf{D}_{j',k'}^{(t')} \cdot \begin{bmatrix} j' \\ k' \\ 1 \end{bmatrix} + \mathbf{r}_{t' \rightarrow \mathcal{O}} \quad (3.8)$$

where $\mathbf{K}$ is the constant intrinsic camera matrix, and P is the set of coordinates to the corresponding pixels as observed from 3D global space – such that the i^{th} item in the set corresponds to a set of $[x, y, z]$ coordinates that has some physical entity occupying that space with a resolution determined by $\mathbf{K}$. Quantities $\mathbf{R}$ and $\mathbf{r}$ are the rotation matrix and translation vector needed to transform the camera pose at frame t' into the global coordinate system – given the global origin $\mathcal{O}$. The 3D points are then transformed into a future depth map at frame t , by setting the pixel values at $\mathbf{D}_{j,k}^{(t)}$ for all points in the set $P^{(t')}$ that are still in scope, using:

$$\begin{bmatrix} j \\ k \\ 1 \end{bmatrix} = \text{round} \left(\mathbf{K} \cdot \left(\mathbf{R}_{\mathcal{O} \rightarrow t} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \mathbf{r}_{\mathcal{O} \rightarrow t} \right) \right) \quad (3.9)$$

where j and k are rounded to the nearest integer. Distant objects will intuitively occupy fewer pixels. As the camera moves forward, these objects appear closer but remain under-sampled, leading to a loss of detail. Increasing the depth map resolution or incorporating additional depth maps during the 3D point-gathering phase can mitigate this effect.

The extrapolation algorithm can be used to significantly reduce computing resources allocated to depth extraction, as needed by downstream navigation. Benchmarking on an NVIDIA Jetson AGX Xavier reveals that using a DNN for MDE, specifically the one in [67], results in a mean execution time of 0.3882 seconds and a power consumption of 23.88 Watts, per frame. Using the extrapolation algorithm has a significantly lower execution

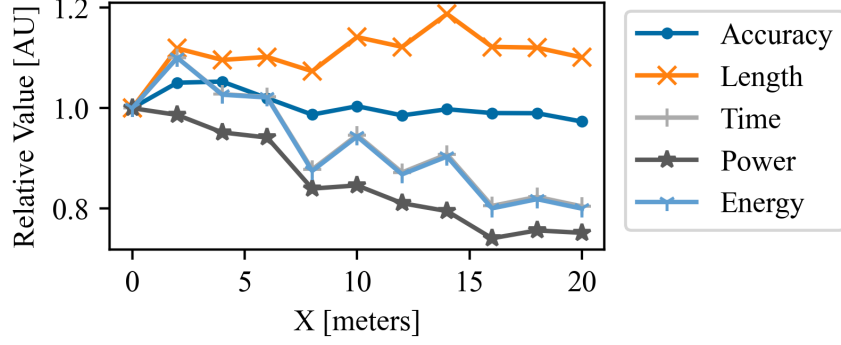


Figure 3.17: *SmartDepth* results illustrating the trade-off between accuracy and efficiency as a function of X . © [2025] IEEE. Reprinted, with permission, from [114].

Table 3.3: Results from varying the efficiency parameter X before runtime. © [2025] IEEE. Reprinted, with permission, from [114].

X [m]	β , accuracy	length	time	power	$\dot{E}$, energy
0	1.0000	1.0000	1.0000	1.0000	1.0000
2	1.0506	1.1189	1.1004	0.9865	1.1002
4	1.0529	1.0959	1.0278	0.9517	1.0267
6	1.0198	1.1018	1.0224	0.9414	1.0212
8	0.9871	1.0734	0.8779	0.8399	0.8749
10	1.0038	1.1414	0.9466	0.8462	0.9436
12	0.9855	1.1219	0.8721	0.8109	0.8682
14	0.9978	1.1875	0.9080	0.7951	0.9037
16	0.9902	1.1219	0.8055	0.7411	0.8006
18	0.9899	1.1204	0.8234	0.7567	0.8188
20	0.9733	1.1008	0.8045	0.7518	0.7999

time of 0.0103 seconds and power consumption of 10.34 Watts. This results in a substantial reduction in energy expenditure, per frame, on average, by 98.85%. Thus, switching from a DNN to extrapolation, for even one frame, has a significant reduction in computing costs.

Considered is an efficiency parameter that is set before runtime, X , which is the number of meters the robot can travel forward before fetching a new reference frame using the DNN. There are two other trigger conditions which immediately result in fetching a new reference frame: starting a new episode and making any yaw rotations.

Figure 3.18 shows, and Table 3.3 lists, navigation accuracy, path length, execution time,

mean power consumption, and expended energy versus X . Where β is the relative value η/η_0 from Equation 3.1 where η is the navigation accuracy resulting from using X and η_0 is the baseline accuracy at $X = 0$. All other metrics were measured using only the set of test paths that were successful in both the baseline and model at $X = 0$, using Equation 3.10:

$$\dot{u} = \frac{1}{N} \sum_i^N \frac{u^{(i)}}{u_0^{(i)}} \quad (3.10)$$

where u is a variable representing either length, time, power, or energy, N is the number of test paths determined from the baseline model, $u^{(i)}$ is the value of a given variable measured when using an arbitrary model to navigate the i^{th} path, and $u_0^{(i)}$ is the value of a given variable when using the baseline model to navigate the i^{th} path. This normalization is used to mitigate biases resulting from various path characteristics, and to show relative performance w.r.t. the baseline model.

The results found in Table 3.3 and Figure 3.18 use depth sensors returned by AirSim as the reference frames for the extrapolation algorithm, rather than directly using a MDE network. This is because there are various depth prediction models in literature that vary in accuracy – and these models will only improve. By assuming the depth model is a perfect predictor, the metrics shown here are theoretical upper bounds which will be approached as MDE networks become more robust.

Figure 3.18 shows that the displayed metrics converge around $X = 16$ meters, after which the navigation accuracy drops below 1% of that from the baseline model. This corresponds to a value of $\beta = 0.9902$, which results in an overall decrease in navigation accuracy by 0.98%. As for the other error metrics that are measured on a path-by-path basis, setting $X = 16$ meters results in: increased path length by 12.19%, decreased execution time by 19.45%, decreased average power consumption by 25.89%, and decreased energy expenditure by 19.94%. These values represent the expected computational costs associated with each

approach, which can be adjusted before runtime by selecting a different value for X . The user has the flexibility to prioritize higher efficiency, which comes with the trade-off of some degradation in navigation accuracy, or alternatively, choose to maximize navigation accuracy. This choice results in a 5.29% boost in navigation accuracy but at the cost of a 2.67% increase in energy expenditure. Interestingly, this increase in accuracy is sometimes observed when the data quality decreases, as is commonly seen in cases like quantization, likely due to some inherent form of regularization which reduces overfit. It’s also worth considering that the extrapolation algorithm might be introducing new information that the navigation neural network can leverage to improve its performance.

3.7.4 CADENCE

The work discussed in this section was accepted for publication, and the preprint is [43].

Considered is onboard, and scalable, computing, w.r.t. to processing raw RGB images into intermediate 2D depth maps. The depth maps are then input to a navigation neural network with a similar structure as that used in previous frameworks: a CNN feature extractor that is late-injected with GPS and IMU data, which is then used by the flattened layers to output motion actions. Notably different, the flattened layers also output the gate value used to scale computing. This positions CADENCE as the first single joint action-adaptation head framework, as discussed in Section 3.6 and Equations 3.6 and 3.5. See Figure 3.18 for an illustration of the full end-to-end implementation. Unique to CADENCE:

1. The expensive CNN bottleneck used for perception is slimmed rather than the smaller navigation network, such as in NaviSlim.
2. All computing is done onboard the drone rather than splitting computing between an onboard processor and an offboard server, such as in NaviSplit.
3. An adaptive mechanism is used to determine when to bypass perception rather than

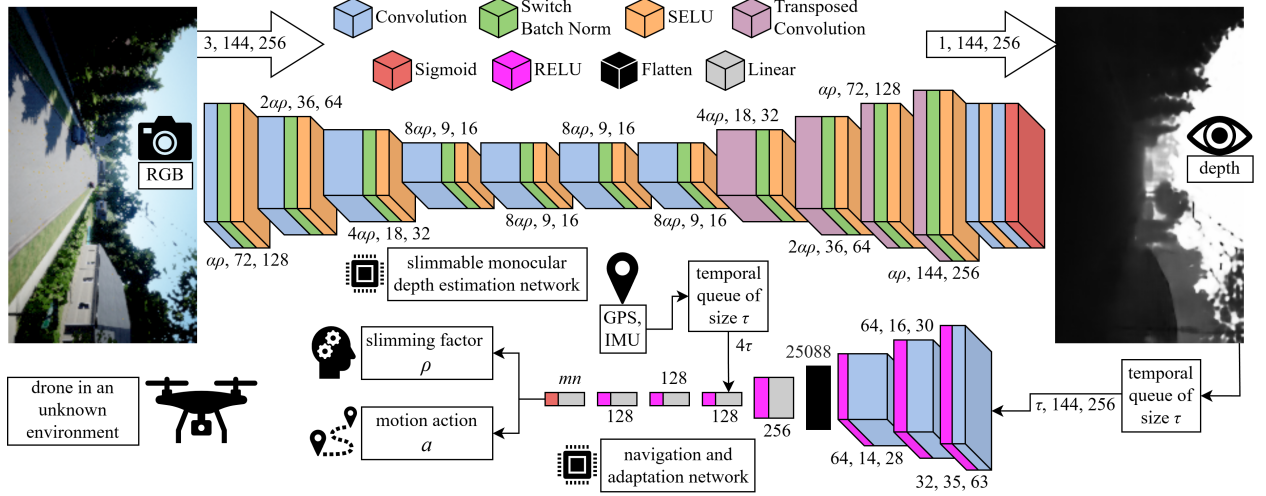


Figure 3.18: Illustrated is the flow of data from raw sensor acquisition to intelligent decision-making through the full autonomy stack, CADENCE. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

using an efficiency parameter set before runtime, such as in SmartDepth.

Perception extracts a 2D depth map, $\hat{\mathbf{D}}$, from an RGB image, $\mathbf{I}$, using a CNN hyper-parameter structure like that used in DGNNNet [31]. Uniquely, this architecture is transformed into a slimmable network that can dynamically scale its computational overhead at runtime. Let $\boldsymbol{\rho} = [\rho_1, \rho_i, \dots, \rho_n]$ be a set of selectable slimming factors whose values are each in the range $(0, 1]$, representing the percent of active channels in each hidden layer. This dynamic portion of the computing stage, f , is defined as:

$$\hat{\mathbf{D}}^{(t)} = f_{\theta}(\mathbf{I}^{(t)}, \boldsymbol{\rho}^{(t)}), \quad (3.11)$$

where θ denotes the learnable model parameters. When $\rho = 0$, the system entirely bypasses image acquisition and network execution, instead providing a zero-filled depth map. This state represents the maximum energy-saving mode, quickest inference time, and smallest

power consumption. The slimmable MDE network is trained by:

$$\operatorname{argmin}_{\theta} \left\langle \operatorname{loss}(\mathbf{D}, \hat{\mathbf{D}}^{(\rho)}) \forall \rho \in \boldsymbol{\rho} \right\rangle, \quad (3.12)$$

where $\hat{\mathbf{D}}^{(\rho)}$ is the predicted depth map when using the given slimming factor ρ , and $\mathbf{D}$ is the ground truth depth map. The maximum number of channels in each hidden layer is controlled by a constant scaling value, α , as shown in Figure 3.18, which is a static value set before runtime. Experimental results vary this maximum value, α . At runtime, the dynamic value of ρ is changed during inference as output by an auxiliary network.

A unified navigation-and-adaptation policy (3.13) is proposed that jointly predicts the motion action $A^{(t)}$ and slimming factor $\rho^{(t+1)}$. This joint formulation addresses the inherent interdependence between perception and control: ρ changes the quality of $\hat{\mathbf{D}}$ which directly impacts the prediction of optimal actions A . Conversely, A alters the drone's trajectory and future environmental context, which may alter optimal values of ρ .

$$[A^{(t)}, \rho^{(t+1)}] = g_{\psi}(\hat{\mathbf{D}}^{(t)}, \mathbf{p}^{(t)}), \quad (3.13)$$

where $\mathbf{p}$ is the relative pose of the drone w.r.t. the target as measured using the onboard IMU and GPS. Realizing the optimal slimming factors is an intractable problem, due to the infinite state-space resulting from the inherent interdependence between $A^{(t)}$ and $\rho^{(t+1)}$. Thus the policy is trained using DRL, specifically with a double Deep Q-Network (DQN) [102]. This DNN structure maps a continuous input to discrete output. A robust reward function is found through empirical findings with surrogate models:

$$\operatorname{reward} = \begin{cases} -10 & \text{time constraint} \\ 40 & \text{goal} \\ -d - E(\rho) - 1 & \text{otherwise} \end{cases} \quad (3.14)$$

This applies terminal conditions based on task success: a penalty if the episode violates the time constraint, and a reward if the drone reaches the target. For non-terminal steps, it applies penalties to encourage efficient progress: d penalizes the Euclidean distance to the target; $E(\rho)$ penalizes the energy expended by the slimmable MDE network on an NVIDIA Jetson Orin Nano (constituting the HIL component of CADENCE); and a constant penalty discourages long episodes.

Each output node of the DQN corresponds to a paired motion action and slimming factor. There are $n * m$ many output nodes, where n corresponds to the number of selectable values of ρ and m corresponds to that of A . The DQN selects from several discrete magnitudes in either direction, and in rotations. Translational moves with larger magnitudes, measured in meters, will correspond to longer intervals in-between steps. The drone executes the action in real-time until either the magnitude of the action is fulfilled or a rigid surface is detected via a short-range distance sensor, thus the magnitude determines a variable navigation frequency.

The primary objective of this framework is to reduce the computational bottleneck imposed by MDE. Table 3.4 summarizes the performance of the MDE network across its various sizes, as defined by α , with a fixed value of $\rho = 1$. The power consumption is directly queried using the native API onboard an NVIDIA Jetson Orin Nano and is measured across the entire board, including effects from static memory storage and other auxiliary computational costs. At α values between 64 to 256, the maximum power is being used; however, there are quite significant changes in inference latency. Alternatively, at α values between 4 to 32, there are more significant changes in power consumption rather than in inference latency. At α values below 4, there are nominal changes in both power and latency. This displays an interesting set of profiles that depend on the network size.

To provide context, executing the MDE network at maximum power, $\alpha > 32$, consumes nearly one-third of the total energy capacity onboard a commercially available drone on today’s market. This overhead significantly reduces flight duration and increases the risk of

Table 3.4: NVIDIA Jetson Orin Nano benchmarks for MDE

α	power [mW]	latency [ms]	energy [mJ]
256	17740	117.1	2078.1
128	18351	30.0	550.0
64	17808	11.0	196.4
32	12961	6.5	83.9
16	7375	7.0	51.6
8	6123	6.7	41.1
4	5224	6.5	34.1
2	4968	6.0	29.8
1	4788	6.0	28.7

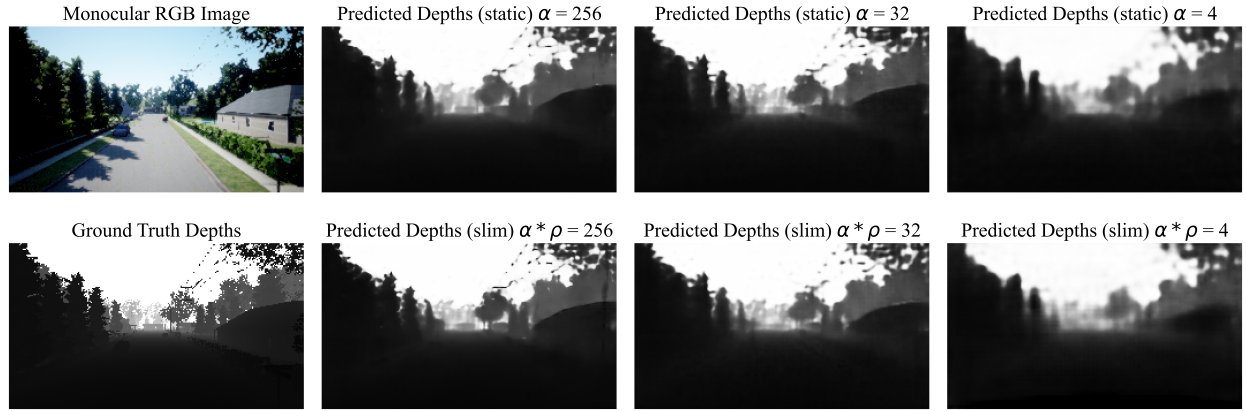


Figure 3.19: Example RGB image, ground truth depth map, and predicted depth maps for both static and slimmable networks with varying network sizes. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

mission failure. However, scaling down the α value reduces inference latency, which simultaneously lowers energy expenditure and improves reaction time. Furthermore, CADENCE’s variable navigation frequency minimizes the total number of MDE executions required, further preserving the energy reservoir.

Figure 3.19 compares an RGB image, the ground truth depth, and depths estimated from either a static or slimmable network. Network size is calculated by multiplying $\alpha * \rho$. Depth predictions are more blurred around the edges of objects at lower network sizes – illustrating advantages of higher fidelity MDE.

Results indicate a strong linear correlation between expected and predicted depth values,

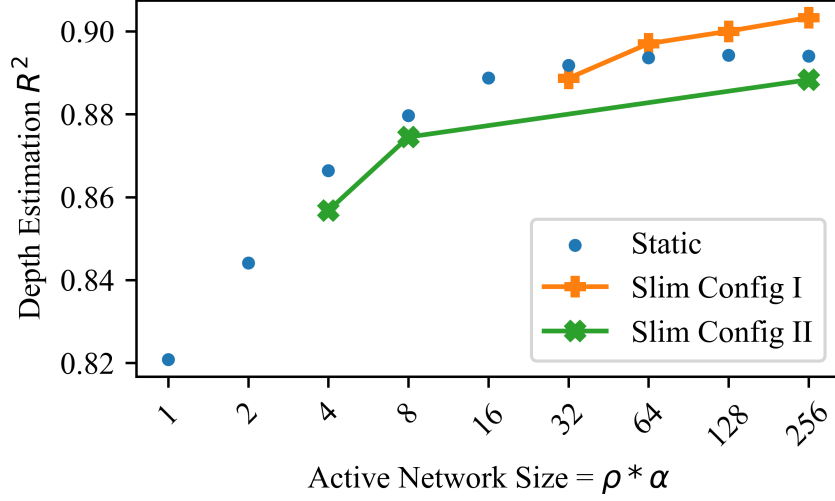


Figure 3.20: Test R^2 -scores of various trained MDE network configurations. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

achieving R^2 -scores up to 0.9033. High errors occasionally occur at large depth magnitudes, primarily along object edges and within sparse structures like leaves, causing the blurring artifacts shown in Figure 3.19. Figure 3.20 compares the R^2 -scores of static and slimmable MDE networks across two configurations (at a fixed $\alpha = 256$). Configuration I utilizes the four largest sizes with slimming factors $\boldsymbol{\rho} = [1, 1/2, 1/4, 1/8]$, while Configuration II adopts an aggressive reduction strategy using $\boldsymbol{\rho} = [1, 1/32, 1/64]$. Consistent with findings in [118], the slimmable networks achieve higher R^2 -scores than static baselines at their largest sizes. Expanding the range of slimming factors (as seen in Configuration II), along with the number of slimming factors (not shown for brevity), degrades overall accuracy relative to the static networks.

Figure 3.21 shows a learning curve after training the navigation-and-adaptation policy with DRL. The validation accuracy converges as it begins to overfit to the training data. Early stopping is applied by selecting the model state corresponding to the highest validation accuracy.

Figure 3.22 highlights a key outcome of CADENCE, presenting two solutions to the opti-

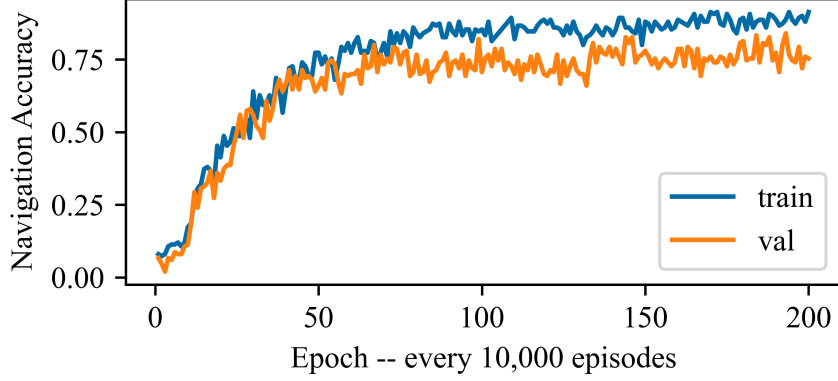


Figure 3.21: Learning curve from training the navigation-and-adaptation policy. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

mization problem in (3.1). Prioritizing navigation accuracy using solely static SoTA models necessitates selecting the largest variant ($\alpha = 256$), which incurs substantial energy costs. Both of the adaptive-dynamic configurations outperform the largest static network in terms of energy efficiency and navigation accuracy. Furthermore, the accuracy decline w.r.t. smaller static networks warrants the need for complexity adaptation. Notably, Configuration I achieves higher accuracy than even the ground truth depth data. It is posited that the partial blurring inherent in smaller network outputs acts similarly to Gaussian noise injection, enhancing model robustness and generalization. Consequently, blurring presents a trade-off: it risks underfitting when tight edge navigation is required, but mitigates overfitting on novel data. By providing multiple network sizes, CADENCE empowers the adaptation logic to dynamically balance these effects based on the immediate environmental context.

To validate the multiple network sizes theory, and accordingly providing evidence for Hypothesis 1, the distinct paths successfully navigated by different static DQN policies are evaluated. Let P be the complete set of test paths, and $A_i \subset P$ be the subset of paths successfully completed by the i -th model. We define $B = \bigcup_{i=1}^n A_i$ as the union of unique successful paths across n independent policies. Our goal is to saturate B utilizing the fewest number of policies (i.e., where increasing n yields only marginal gains). Iteratively adding models in descending order of network size achieves saturation at $\alpha = 32$, requiring policies

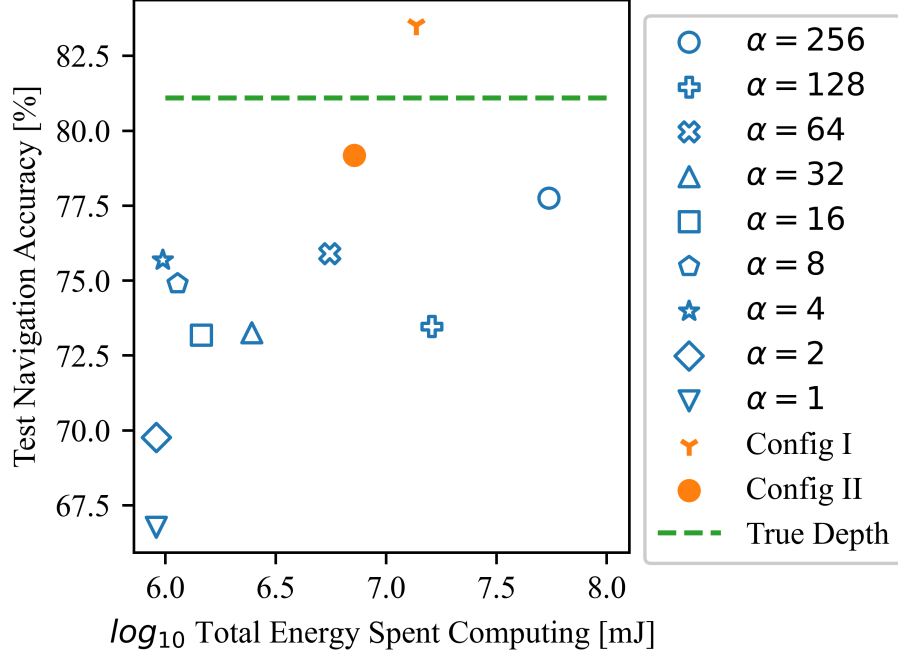


Figure 3.22: Final evaluations on the static holdout test set. The x-axis shows the energy expended while computing the MDE network over all paths. The hollow blue markers correspond to using a SoTA static MDE network with size α , and a navigation DQN policy that predicts A . The solid orange markers correspond to using a slimmable MDE network with a navigation-and-adaptation DQN that predicts both A and ρ . The green dashed line corresponds to a navigation DQN that bypasses MDE and instead directly uses the AirSim ground truth depth maps. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

with $\alpha \in \{256, 128, 64, 32\}$. This is Configuration I (see Figure 3.22). A greedy approach, iteratively adding the model that most significantly expands B , regardless of network size, saturates with just three policies: $\alpha \in \{256, 8, 4\}$, mirroring Configuration II. These saturation dynamics prove that while two policies may exhibit similar overall accuracies, their path-solving behaviors differ drastically. It is posited that this behavioral variance stems from size-dependent blurring, which the adaptation policy controls across diverse scenarios. **This corresponds to optimal network scales as a function of shifting data distributions.**

DNNs inherently have a lack of understanding into why predictions are made. This leads to safety and reliability issues when deploying to control algorithms used by autonomous

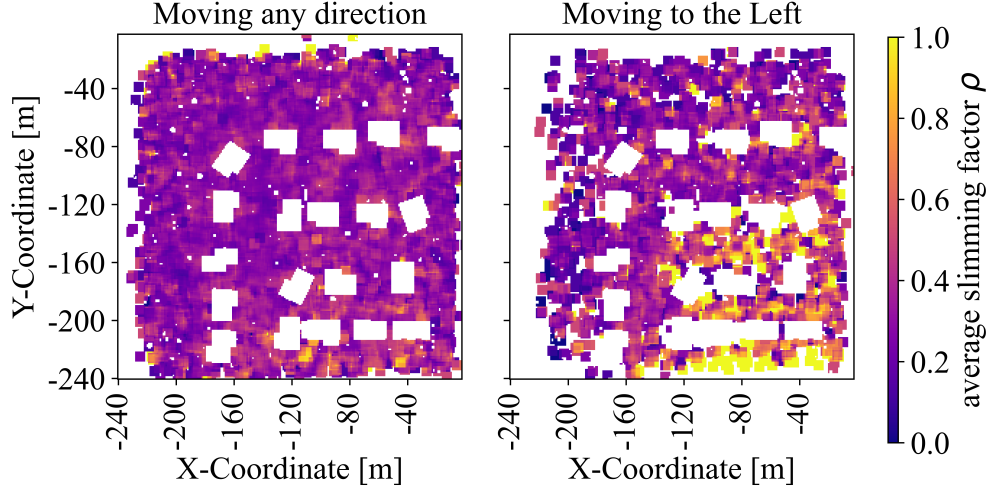


Figure 3.23: Average slimming factor when computing the slimmable MDE network at each of the given (x, y) locations. Compared is the drone moving in an arbitrary direction versus moving only to the left. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

vehicles. Correlations are examined, that may help illuminate why the trained navigation-and-adaption policy (Configuration I) predicts particular values of A and ρ . Figure 3.23 shows the average predicted slimming factor throughout the test region, indicating that areas with higher densities of houses correspond to higher values of ρ . This correlation becomes more prominent when the drone is on a trajectory moving through small alleyways in-between houses (moving to the left). This warrants the advantage of having higher fidelity depth maps to better navigate between objects, and that the trained policy is considering navigation objectives along with the current trajectory when predicting the optimal adaptation value ρ .

Figure 3.24 indicates that CADENCE is predicting larger slimming factors when there is a higher density of nearby objects – which further warrants that higher fidelity perception is needed to navigate through more complex scenarios. Conversely, smaller values of ρ are predicted when the drone is closer to the target. This is likely because detailed path planning becomes less important as the target is more directly visible. There are several other environmental and navigation factors correlated to the predicted values of ρ . When collision

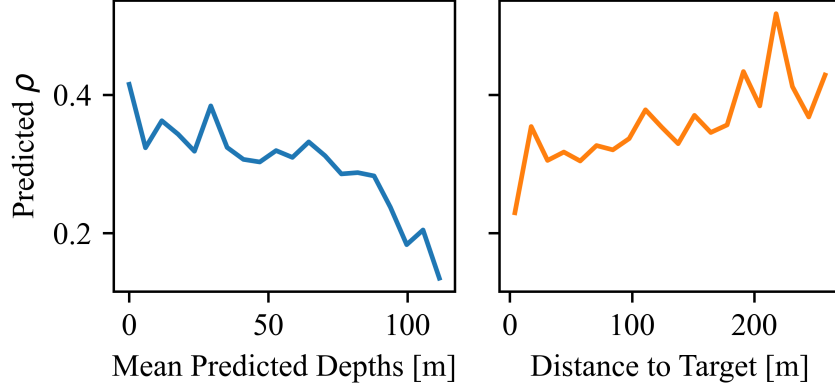


Figure 3.24: Correlation between the adaptation parameter ρ and two environmental factors: obstacle density, as measured by the mean value of the FIFO queue of recent depth map predictions, and distance to target. © [2025] IEEE. Reprinted, with permission, from the preprint at [43].

avoidance is triggered, the adaptation model tends to predict a value of $\rho = 0$ more than when avoidance is not triggered. This is possibly because the policy prioritizes moving around the object in front of it, and there is previous depth maps in the FIFO queue that can be used to navigate around it. When the navigation component is predicting large steps, corresponding to moving down a large open corridor (*i.e.*, a road or between trees), it tends to also predict a smaller value of ρ . The predicted value of ρ often matches the one from the previous step, implying that the model bases its predictions on persistent environmental conditions. The relationship between optimal values of ρ , navigation features, and environmental factors is quite complex. However, these correlations demonstrate that the adaptation logic is driven by learned context features rather than stochastic ones, warranting that the policy demonstrates a robust and interpretable adaptation strategy.

CADENCE is an adaptive perception-and-control system designed to overcome the computational bottleneck of monocular depth estimation (MDE) on resource-constrained autonomous vehicles. By optimizing the trade-off between efficiency and reliability through utilizing a slimmable MDE network and a unified navigation-and-adaptation policy, CADENCE successfully closes the loop between sensing and actuation. Experimental results

on a hardware-in-the-loop (HIL) testbed, using an NVIDIA Jetson Orin Nano with Microsoft AirSim, demonstrate that CADENCE achieves a 75.0% reduction in energy expenditure, 16.1% improvement in power consumption, and a 74.8% decrease in inference latency compared to static state-of-the-art (SoTA) baselines. **Crucially, these efficiency gains are accompanied by a 7.43% improvement in navigation accuracy, proving that ADDNN frameworks can significantly improve both accuracy and efficiency by decoupling the perception and navigation-adaptation components.** These findings highlight the potential for green IoT frameworks to enable autonomy on lightweight vehicles. This has implications in not only reducing mission failures due to depleted energy reserves or violated time constraints, but also improves flight time by reducing the total energy expenditure, reaction time to the environment by reducing the time spent computing, and power consumption which enables the device to concurrently host other operations.

3.8 Summary

This chapter introduced the idea of dynamic networks, that can change their structure at run-time, and augmenting them with adaptive mechanisms, to change their structure in response to both perceived data distribution shifts and downstream mission objectives. Observation 1 lead to taking a time-series approach to bypass the challenges of an innate paradox observed between realizing latent features and using those features to adapt computing. Observation 2 lead to a solution of decoupling the training of the dynamic and adaptive components into two distinct stages, to mitigate the problem of unstable gradients.

Such adaptive dynamic deep neural networks (ADDNN) were shown to minimize the resource overhead of computing DNNs while maximizing the downstream task accuracy. Prior to this work, typically resource reduction came at the cost of maintained or degraded task accuracy. The navigation frameworks presented in this chapter provide evidence to Hypothesis 1, by showing that as data distributions shift within the AirSim environments, while the drone is

navigating through the map, so do the optimal sub-paths of the dynamic neural networks used for perception.

Chapter 4

Mitigating Real-World Artifacts

This chapter presents various deep learning techniques used to mitigate artifacts in real world scientific data:

1. Noisy data resulting from uncertainty in sensor instrumentation and environmental factors.
2. Missing features in the data due to sensor failure, data corruption, or human error.
3. Uncertainty quantification and reduction.
4. Multimodal data.
5. Spatio-Temporal data that varies in resolution, scope, and dimensions.

4.1 Denoising AutoEncoder with Monte Carlo Dropout

The work discussed in this section was published in [38].

Considered is that real world data is imperfect, in terms of: (1) noisy data with a low signal-

to-noise ratio that can be represented as a Gaussian distribution [105, 57], and (2) missing data from sensor failure or human error that can be represented as a Bernoulli distribution [19]. The two data corruption schemes are implemented together on observed data X so that they can be resampled through artificial corruption. The corrupted random vector of sensor observations, $\tilde{X}$, can be mapped from X as:

$$\tilde{X} = B \cdot (X + \boldsymbol{\varepsilon}) \quad (4.1)$$

$$B \sim \mathcal{B}(\boldsymbol{\nu}^{(i)}) \quad (4.2)$$

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}) \quad (4.3)$$

where $\boldsymbol{\varepsilon} = \begin{bmatrix} \epsilon_1 & \dots & \epsilon_m \end{bmatrix}^T \in \mathbf{R}^m$ is the random vector of Gaussian errors in all sensor observations; the covariance matrix $\boldsymbol{\Sigma} \in \mathbf{R}^{m \times m}$ takes the form of $\mathbf{diag}(\begin{bmatrix} \sigma_1^2 & \dots & \sigma_m^2 \end{bmatrix}^T)$. Where the σ values are the expected uncertainty from each of the sensor instruments. Correspondingly, the elements of vector $\boldsymbol{\nu}$ denote the probabilities of a missing feature – which are empirically measured over a given set of data.

Artificial corruption is used to train Denoising AutoEncoder (DAE) [30, 106] models for two purposes: (1) to directly denoise and gap-fill real-world data for general use cases, and (2) to consider denoising and gap-filling methods as learned in the latent space of a downstream model for a specific use case. For both purposes, training is augmented by resampling the input data with Equation 4.1 during each forward pass. During training of a traditional DAE, for purpose (1), the output space is set to the original input data before resampling:

$$\operatorname{argmin}_{\phi} \left\langle \operatorname{loss}(h_{\phi}(\tilde{X}), X) \right\rangle, \quad (4.4)$$

Theoretically, artificial corruption during training allows the latent space to learn the original distribution from which the observations were sampled from. Through a learned representation of the covariance matrix, the DAE can “denoise” and gap-fill the corrupted data.

When artificially corrupting the training data for a downstream task model, further latent compactions can be obtained by learning correlations between how the noise and missing data affects the supervised task. This can lead to advantages similar to those observed when comparing generalized to supervised compression, which enables split computing to obtain more robust results than JPEG compression (Section 3.7.2).

When considering the robustness of the DAE and any downstream task models, the uncertainty must be considered from both a bias and variance perspective. Bias is measured as the reconstruction error over a static set of corrupted variables, or the downstream task error resulting from both before and after applying the DAE to the pipeline. Variance is measured through two Monte Carlo sampling techniques. One technique is perturbing the input space (*i.e.*, sensitivity analysis), providing an estimate into the variance resulting from the input space. The second technique is Monte Carlo dropout [22]. This randomly samples the neural network nodes during a forward pass of the pipeline, providing an estimate into the variance resulting from the latent space. Of interest, is how both the bias and variance is affected by corrupted inputs, and consequently how they are reduced by applying DAE-based methods.

The sensor instrument manufacturers typically report expected observation uncertainty as either a fixed standard deviation (σ) or coefficient of variation (σ/μ), where μ is the expected value of the observation (*e.g.*, mean). Thus, the variance of a sensor or target variable is measured as either S as the estimator to σ , or $C.V.$ as the estimator to σ/μ . Given a Monte Carlo sampling size, n_{MC} , and the number of data points in the test set, n_{test} :

$$s_u = \sqrt{\frac{1}{\sum_{i=1}^{n_{test}} n_{MC}^{(i)} - n_{test}} \sum_{i=1}^{n_{test}} \sum_{l=1}^{n_{MC}^{(i)}} \left(u_l^{(i)} - \bar{u}^{(i)}\right)^2} \quad (4.5)$$

$$c.v.u = \sqrt{\frac{1}{\sum_{i=1}^{n_{\text{test}}} n_{\text{MC}}^{(i)} - n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} \sum_{l=1}^{n_{\text{MC}}^{(i)}} \left(\frac{u_l^{(i)} - \bar{u}^{(i)}}{\bar{u}^{(i)}} \right)^2} \quad (4.6)$$

Where $\bar{u}^{(i)}$ is the average of all values sampled for the given data point at index $i \in [1, \dots, n_{\text{test}}]$ and variable u . Thus the objective of a DAE model is to: (a) reduce the bias as represented by deterministic error metrics (*e.g.*, MSE, MAPE, accuracy, *etc*) which depend on the task, and (b) reduce the variance as represented by the stochastic metrics defined in Equations 4.5 or 4.6.

It was shown in [38] that this is accomplished in a hydrology application. Hydrological processes that are difficult to measure and upscale are typically estimated using extensively parameterized models that make use of readily available data for environmental variables. Extracting features that capture the essential data structure underlying the target quantity is crucial for model predictions. Evapotranspiration (ET) – the flux of water vapor from land to atmosphere via soil evaporation and plant transpiration – is a significant process in the global hydrological cycle that is typically estimated through data-based models. Considered is a dataset collected over several years of predictor variables from satellites, local weather towers, and meteorological forcing data that are correlated to ET, along with ET target variables from embedded soil sensors.

Consider a matrix of observed variables, $\mathbf{X} \in \mathbf{R}^{1000 \times 20}$, corresponding to 1,000 data points collected for 20 variables – 19 of which correspond to predictors such as temperature, radiation, vegetation, and other climate variables, and 1 corresponds to the target variable ET. First, the DAE is used to denoise and gap-fill the predictor variables via a DNN. Second, an ET-predicting DNN is trained with input noisy data. In both cases, the input data is artificially corrupted during training. Figure 4.1 shows the affects of the DAE when inputting one data-point of predictor variables, $\mathbf{x} \in \mathbf{R}^{19}$, and predicting the target variable

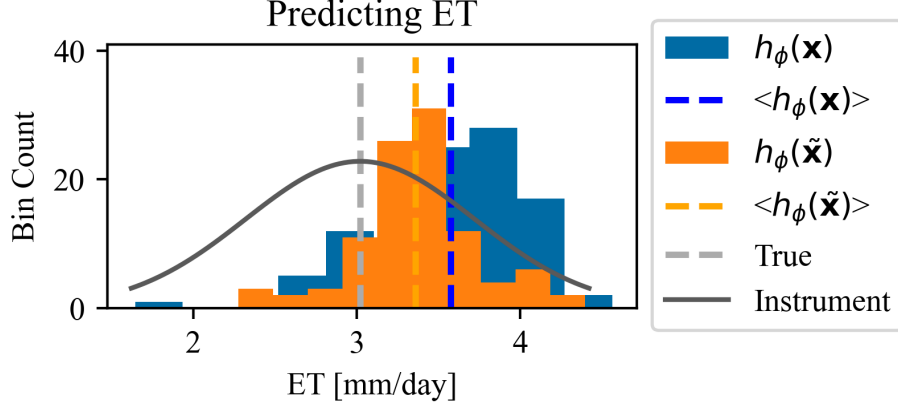


Figure 4.1: Resulting distributions on a single data point after Monte Carlo sampling, for the non-DAE model $h_\psi(\mathbf{x})$ and the DAE model $h_\psi(\tilde{\mathbf{x}})$.

ET. The distribution corresponding to $h_\psi(\mathbf{x})$ is from making predictions with a DNN that did not use artificial corruption during training (non-DAE). The distribution corresponding to $h_\psi(\tilde{\mathbf{x}})$ is from making predictions with a DNN that did use artificial corruption during training (DAE). The mean predicted values over a given Monte Carlo sample is indicated by a vertical dotted line, and used to visualize the predictive bias. First, the bias corresponding to the DAE trained model falls in-between the true value and that of the non-DAE model. The variance is illustrated by the histograms resulting from Monte Carlo sampling applied to either of the models. Second, the width of the histogram corresponding to the DAE trained model is visibly smaller than that of the non-DAE one. These two observations visualize an intended result of reducing both bias and variance with DAE-treated methods.

Results from comparing non-DAE to DAE treated models are aggregated over a static hold-out test set, 20% of $\mathbf{X}$, and are listed in Table 4.1. Note that four of these variables, pa , $precip$, xT , xu , do not carry a quantifiable uncertainty and thus have no denoising methods applied. All of the variables result in a significant decrease in variance. On average, the DAE treated model reduces inherent variance in ET predictions by 69%, and in reconstructing the noisy variables by 71%.

Table 4.1: Parameter values used to generate the corrupted input variables (corresponding to sensor observations) and the denoising levels (DNL) of different data corruption treatments: Gaussian noise (G.), Bernoulli perturbation (B.), and both (G.B.). For the Gaussian noise corruption, either the standard deviation, σ , or the coefficient of variation, $c.v.$, is used to generate the sample, and the choice is according to the manufacturer’s specified error metric on the corresponding sensors. Values of $c.v.$ are identified by a ”%” sign. The parameter ν is the Bernoulli probability in Equation 4.2. The DNL is calculated as $\frac{s_{\hat{U}} - s_U}{s_U}$ or $\frac{c.v. \cdot \hat{U} - c.v. \cdot U}{c.v. \cdot U}$, where $\hat{U}$ is the model-predicted estimator of each original variable, U . Quantities s and $c.v.$ were calculated as in Equation 4.5 and 4.6. Note that ET has two values for σ resulting from the varying instrumental uncertainties arising from the different sites used in this study. © [2025] IEEE. Reprinted, with permission, from [38].

Symbol	Name	Units	σ	ν	DNL G. (%)	DNL B. (%)	DNL G.B. (%)
txmax	Air Temperature (max)	K	0.6	0.07	-66	-64	-64
txmin	Air Temperature (min)	K	0.6	0.07	-58	-59	-59
tomax	Skin Temperature (max)	K	0.2	0.07	46	-62	-60
tomin	Skin Temperature (min)	K	0.2	0.07	27	-57	-56
u	Wind Speed	ms^{-1}	0.3	0.08	-91	-66	-78
vpd	Vapor Pressure Deficit	Pa	200	0.07	-89	-68	-81
srad	Shortwave Radiation	Wm^{-2}	5%	0.08	-66	-72	-72
nsrad	Net Shortwave Radiation	Wm^{-2}	5%	0.08	-68	-72	-72
nrad	Longwave Radiation	Wm^{-2}	5%	0.08	-75	-89	-88
rn	Net Radiation	Wm^{-2}	5%	0.08	-67	-76	-76
g	Soil Heat Flux	Wm^{-2}	3%	0.11	-80	-95	-94
lai	Leaf Area Index	AU	0.3	0.02	-62	-28	-52
swc	Soil Water Content	m^3m^{-3}	3%	0.07	107	-51	-47
swp	Soil Water Potential	kPa	10%	0.07	-89	-89	-89
h	Vegetation Height	m	0.1	0.02	-81	-36	-74
pa	Air Pressure	Pa	-	-	-	-	-
precip	Precipitation (sum)	mm	-	-	-	-	-
xT	Air Temperature Height	m	-	-	-	-	-
xu	Wind Speed Height	m	-	-	-	-	-
ET	Evapotranspiration	$mm\ day^{-1}$	0.7	-	-77	-78	-69

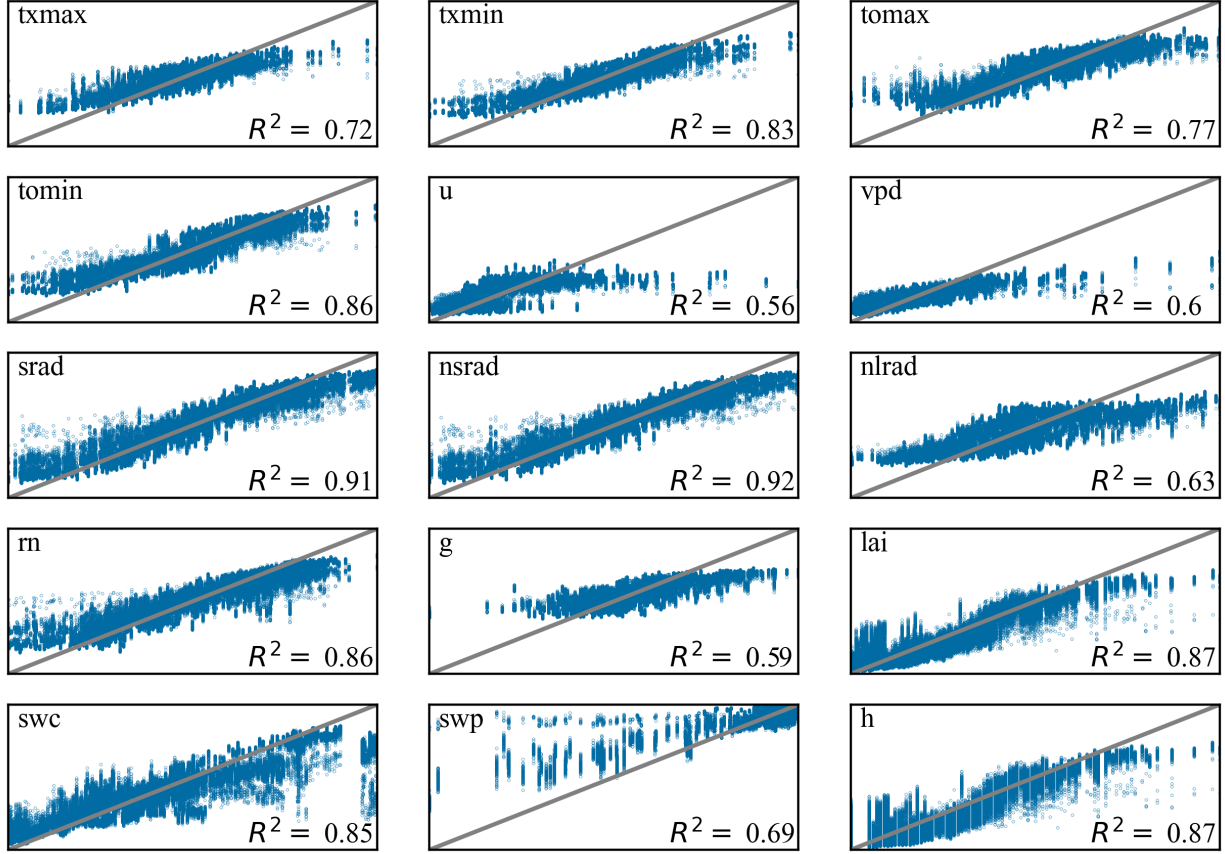


Figure 4.2: Expected versus predicted values for the DAE, used to repair artificially corrupted sensor observations. Each panel above is separated into one of the 15 sensors, which can be artificially corrupted and subsequently repaired. The results shown are from the testing set with sampled artificial corruption (both Gaussian and Bernoulli) and without sampled model uncertainty. © [2025] IEEE. Reprinted, with permission, from [38].

Figures 4.2 and 4.3 show the affects of the DAE treated model on the bias. When reconstructing the predictor variables, the coefficient of determination, R^2 , values, range from 0.56 to 0.92 depending on the variable. Some of the variables have visible plateaus in the R^2 plots (*e.g.*, *swp*, *u*), which may indicate degeneracies [37] in the data such that the covariance matrix realized between these 19 variables is not strong enough to reconstruct degenerate variables. For ET predictions, the values are mostly under predicted with larger magnitudes indicating either sparse labels or further degeneracies. However, there is an increase in R^2 values from 0.69 to 0.72, indicating that the DAE treated methods are robust – allowing the predictive DNN to better account for noise and missing features in the observed 19 variables.

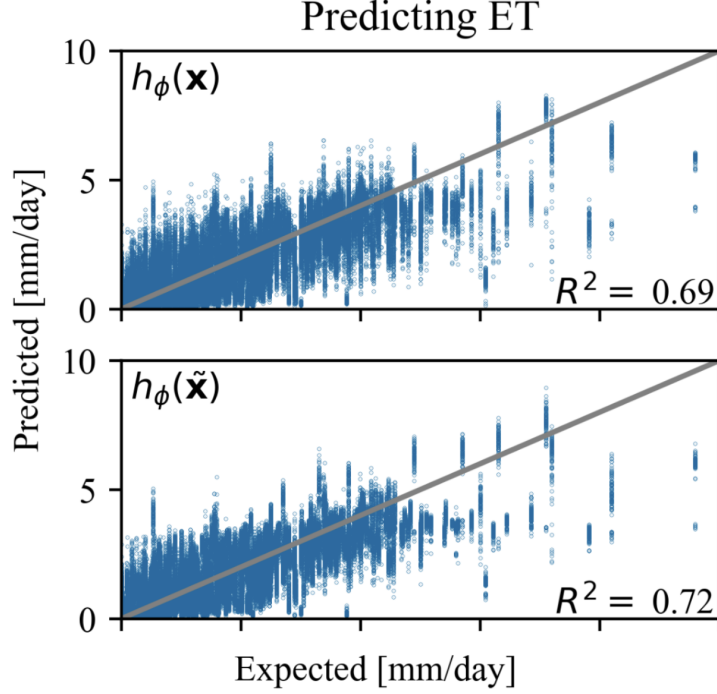


Figure 4.3: Expected versus predicted values for the ET models, either with or without DAE-treated training methods using artificial corruption. The results shown are from the testing set with sampled artificial corruption (both Gaussian and Bernoulli) and without sampled model uncertainty. © [2025] IEEE. Reprinted, with permission, from [38].

4.2 Multimodal Neural Networks

The work discussed in this section was published in [39].

Considered are heterogeneous sensors which vary by both modality and dimensions. This creates two unique problems w.r.t. neural network frameworks: (1) lining up the different dimensions for each modality such that they are compatible for input into a neural network, and (2) fusing the data such that the latent space learned by the neural network considers both single and multi modal relationships. Methods are tested using a dataset measured from rock and mineral samples at Dr. Virginia Gulick’s group at the Lunar and Planetary Laboratory at the University of Arizona, which is a combination of Raman spectra and RGB images. The novel Raman spectrometer is dual-band co-registered, such that two wavelengths (532 nm and 785 nm) consecutively irradiate the same spot on the sample.

Images are captured at a set height above the samples, using a high-resolution camera. The objective is to autonomously classify minerals and rocks common on Lunar and Martian surfaces via lightweight neural networks. Considered first is fusion of the dual-band Raman spectrometer, then integration of the RGB images.

The different modes of the dual-band Raman spectrometer are split into two main groups. The first main group, *A*, uses a 532 nm excitation laser and is split into two sub groups, *A1* which considers the wavenumber range from 200 to 1200 cm^{-1} that corresponds to a region of the spectrum dense in mineral diagnostic peaks [20, 107, 108], and *A2* which considers the wavenumber range from 3300 to 3700 cm^{-1} that corresponds to hydration diagnostic peaks [65]. The second main group, *B*, uses a 785 nm excitation laser and only considers the wavenumber range from 200 to 1200 cm^{-1} , however is measured with a higher resolution than modality *A*.

There are three neural network architectures compared, as to analyze the affects of more complex structures:

1. **Logistic regression (LR)** which has the lowest level of complexity with no hidden layers, and considers all modalities concatenated and fed in as one input feature vector.
2. A **multi-layer perceptron (MLP)** with one fully connected hidden layer, which also considers all modalities concatenated.
3. A novel **multimodal neural network (MNN)** which has individual stems for each modality that are then fused with a mutual hidden layer before final classifications. The MNN independently trains a stem for each input modality. Then the stems are frozen, and the output latent space is fed to a mutual hidden layer that is trained to learn cross-modal features.

Figure 4.4 illustrates the flow of data for each of the modalities through each of three neural

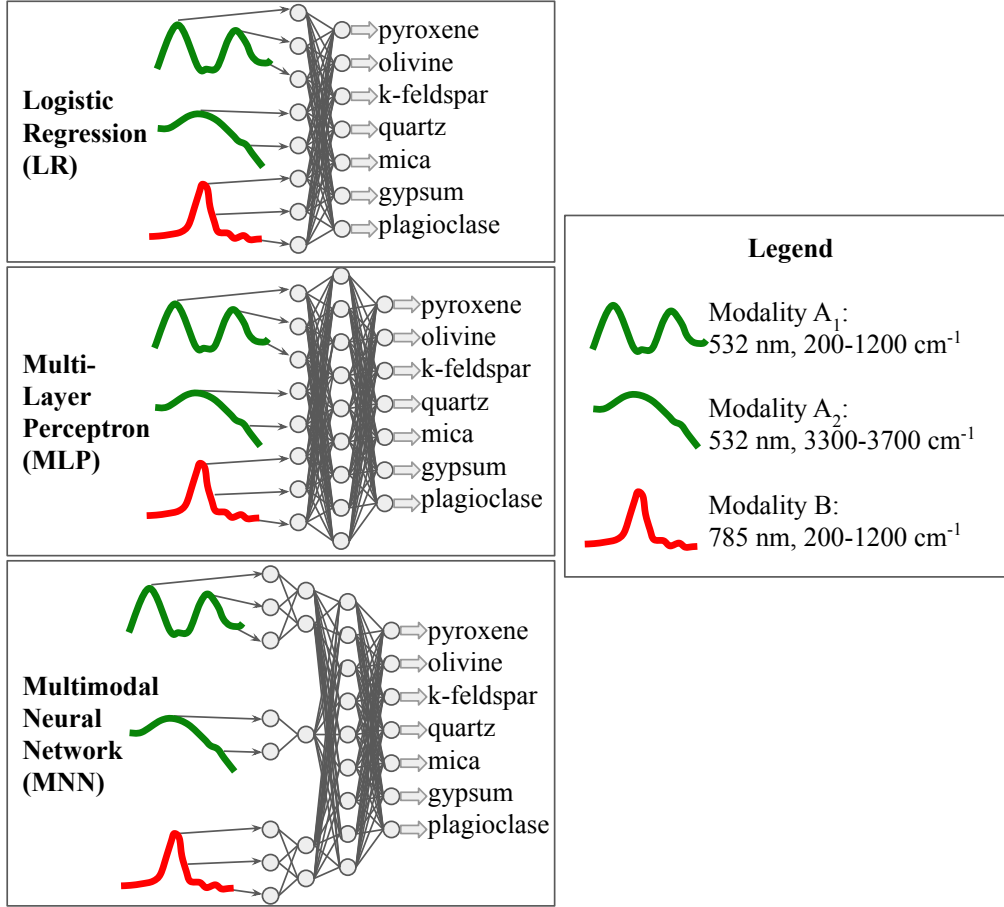


Figure 4.4: Flow of each modality through either a logistic regression, multi-layer perceptron, or multimodal neural network. © [2025] Reprinted, with permission, from [39].

network architectures, where each model is a classifier that outputs scores for seven common rock-forming minerals.

A comparative study is conducted that varies all possible combinations of the two main modalities considered (A and B) as input into each of the three models. For each configuration (modalities, model, and classification type), 100 random runs were executed that trained and evaluated the model with different random seeds and initialized model parameters. Training and evaluating was done via leave-one-out cross-validation [95]. AUC is the area under the receiver operating characteristic curve [10], and is used to evaluate classification problems with no set cutoff. Where the cutoff value is used to partition the neural network output scores into positive or negative classifications for each mineral. Cutoff val-

ues are highly mission dependent, as they rely on a cost-function which defines some value attributed to false and true positives and negatives. An AUC score of 0.5 corresponds to a random classifier, and a value of 1.0 corresponds to a classifier with perfect separation.

Table 2.5 shows the results of a one-sided hypothesis test comparing various leave-one-out distributions over all 100 random seeds, for single-mineral classification. Shown is the p-value along with both the means of the AUC scores and standard deviations, σ , corresponding to each of the two distributions in each row. All p-values were near-zero, warranting a near-zero probability that the mean AUC score of one distribution is higher than the other by chance. These results demonstrate two key findings:

1. Using both spectral modes results in significantly higher AUC than using either one of the modes. This demonstrates the robustness of multimodal sensor fusion, warranting that one mode has unique information or lower SNR as compared to the other.
2. Using an MNN has higher AUC than either LR or an MLP. This demonstrates that the multimodal neural network structure, using frozen independent stems that are fused together, is more robust over more traditional approaches that simply concatenate the modalities together and use a fully-connected neural network.

Figure 4.5 shows the learning curves for all 100 random seeds of the MNN. Towards the end of each of the learning curve, the lines become highly erratic (visible by large perturbations in the cross-entropy loss at later epochs). When training an MNN, first the single-modal stems are independency trained. Second, the stems are frozen as feature extractors and connected to a mutual hidden layer that is then trained. The erratic part of the learning curves correspond to a third stage that was investigated – in which the stems are unfrozen and then the entire MNN is fine-tuned by propagating the gradient backwards into each of the stems. This is a phenomenon observed generally when fine-tuning machine learning models [58], and a specific challenge to multimodal machine learning [109]. Due to the chaos

Table 4.2: One-Sided Hypothesis Test Results for Single-Mineral Classification

Compared Distributions	p-val from T-test $H_a : \overline{\text{AUC}}_1 > \overline{\text{AUC}}_2$	$\overline{\text{AUC}}_1$	$\overline{\text{AUC}}_2$	σ_1	σ_2
mode A (1)	0.0	0.95	0.88	0.022	0.050
mode B (2)					
modes A and B (1)	0.0	0.96	0.95	0.023	0.022
mode A (2)					
modes A and B (1)	0.0	0.96	0.88	0.023	0.050
mode B (2)					
MLP (1)	0.0	0.94	0.92	0.033	0.060
LR (2)					
MNN (1)	0.0	0.97	0.92	0.017	0.060
LR (2)					
MNN (1)	0.0	0.97	0.94	0.017	0.035
MLP (2)					

induced in the gradient when unfreezing the stems, results reported in Table 2.5 consider only the first two stages of training – using frozen stems.

Considered next is the integration of RGB images with the Raman spectral modes. A separate CNN stem is trained for the images, then frozen and fused with the frozen spectral stems. Table 4.3 lists the results from using the MNN with frozen stems, for multi-mineral classification. These AUC scores warrant the robustness of the MNN, by showing an increase in AUC from single-modality based classifiers to multi-modal ones, where the highest AUC score is obtained by fusing all spectral and image modalities.

4.3 Global and Local Spatio-Temporal Data

The work discussed in this section was presented at [36], is related to a publication in preparation with Lawrence Berkeley National Labs (LBNL), and is a continuation of the

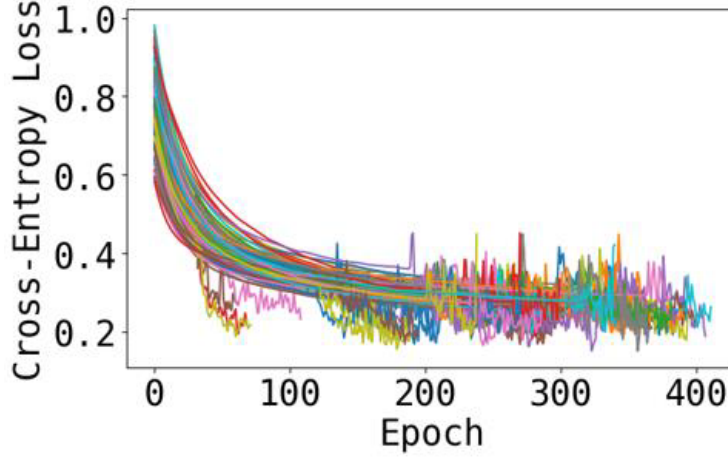


Figure 4.5: Learning curves for training mineral classifiers with a multimodal neural network, by varying when the single-modality stems are unfrozen. © [2025] Reprinted, with permission, from [39].

Table 4.3: AUC Scores for Multi-Mineral Classification

modality	all	quartz	gypsum	mica	olivine	plagioclase	kspar	pyroxene
Raman-A1	0.8638	0.8022	0.9149	0.5387	0.6177	0.6516	0.7265	0.4650
Raman-A2	0.8420	0.5117	0.7218	0.4984	0.4893	0.4328	0.4060	0.3755
Raman-B	0.8582	0.7283	0.9899	0.5152	0.5635	0.7536	0.6668	0.4103
Image	0.8529	0.5717	0.5185	0.6169	0.6196	0.6562	0.5428	0.7022
All Raman	0.8831	0.7177	0.8363	0.7311	0.6043	0.7419	0.7230	0.2618
All Modes	0.8988	0.8017	0.9699	0.7531	0.6155	0.8438	0.7092	0.5262

work published in [38].

Considered again is the task of predicting ET, now fusing heterogeneous spatio-temporal data on a larger scale that varies in modality, dimension, scope, and resolution:

1. Daymet [99, 98] uses interpolation algorithms based on ground sensor observations to measure the variables: minimum and maximum temperature, precipitation, solar radiation, and vapor pressure, using a resolution of 1km x 1km and period of one day.
2. National Oceanic and Atmospheric Administration’s (NOAA) Real-Time Mesoscale Analysis (RTMA) [13] uses interpolation algorithms based on satellite observations to measure the variables: wind speed, at a resolution of 2.5km x 2.5km and period of one

day.

3. NASA’s Moderate Resolution Imaging Spectroradiometer (MODIS) [44] uses Penman–Monteith [82, 74] and interpolation algorithms based on satellite observations to measure the variables: ET, Normalized Difference Vegetation Index (NDVI), snow cover, and Leaf Area Index (LAI), at a resolution of either 250m x 250m for NDVI or 500m x 500m for all other variables, and at a period of one day for snow cover, four days for LAI, and eight days for both ET and NDVI.
4. Surface data is collected from the United States Geological Survey [17] and contains measurements of the variables: elevation, slope, aspect, leaf cover, and foliage type, at a resolution of 30m x 30m and all variables are considered static in time.
5. LBNL’s high fidelity local sensor towers [111] equipped with meteorological, energy flux, and soil sensors – providing a small range of data at high resolution, accuracy, and precision. The three towers were located at the grasslands in the East River watershed near Crested Butte, Colorado operated between 2020 and 2023. Each tower includes a weather station, a soil water potential sensor, and soil water content sensors.

Section 4.1 considers an MLP with 1D point-by-point predictions of ET, where as this new dataset considers large-scale 2D spatial data. Frozen single-modality stems are not used, such as in Section 4.2, because they would break spatial correlations that would otherwise be learned jointly by the CNN [3]. Specifically, two structures are considered: (1) a CNN inputting a small spatial area followed by a flattened layer to predict single-pixel values in the center pixel of the input space, and (2) a Unet [86] to predict ET over a larger 2D space with the same area as the input space.

Data is resampled to both spatially and temporally align the different modalities. Spatially,

this is accomplished by applying a bilinear interpolation algorithm:

$$\hat{v}_{x,y}^{(t)} = \frac{\sum v_{i,j}^{(t)} * A_{i,j}^{(t)}}{\sum A_{i,j}^{(t)}} \quad \forall \text{ intersecting rectangles} \quad (4.7)$$

Where $\hat{v}_{x,y}^{(t)}$ is the resampled observation, $v_{i,j}^{(t)}$ is an original observation, and $A_{i,j}^{(t)}$ is the respective area of intersection between the spatial resolution and coordinates used to measure the original observation and that of the resampled observation. If there are no overlapping original observations with a resampled one, then the value is set to not-a-number (NaN). This algorithm is used to resample all variables to a uniform grid with 30m x 30m resolution.

All variables are temporally resampled to an 8-day period, the same as that used to measure ET. All variables had sampling periods of either $m = 1$, $m = 4$, or $m = 8$ days, and started on the first day of each year. Each temporally resampled observation is aggregated over an 8-day viewing window by using the mean of all valid observations within that window. Note that the number of valid observations can vary within that window if there is missing data. If no valid observations are within the window, for a given variable, then the resampled value is set to NaN. During resampling, the rolling window starts on the first day of the year, and does not consider overlapping windows. If a full 8-day window does not fit at the end of the year, then that data is truncated. This is repeated for all years between 2000 and 2023, inclusive. Figure 4.6 illustrates the spatio-temporal averaged values for each variable.

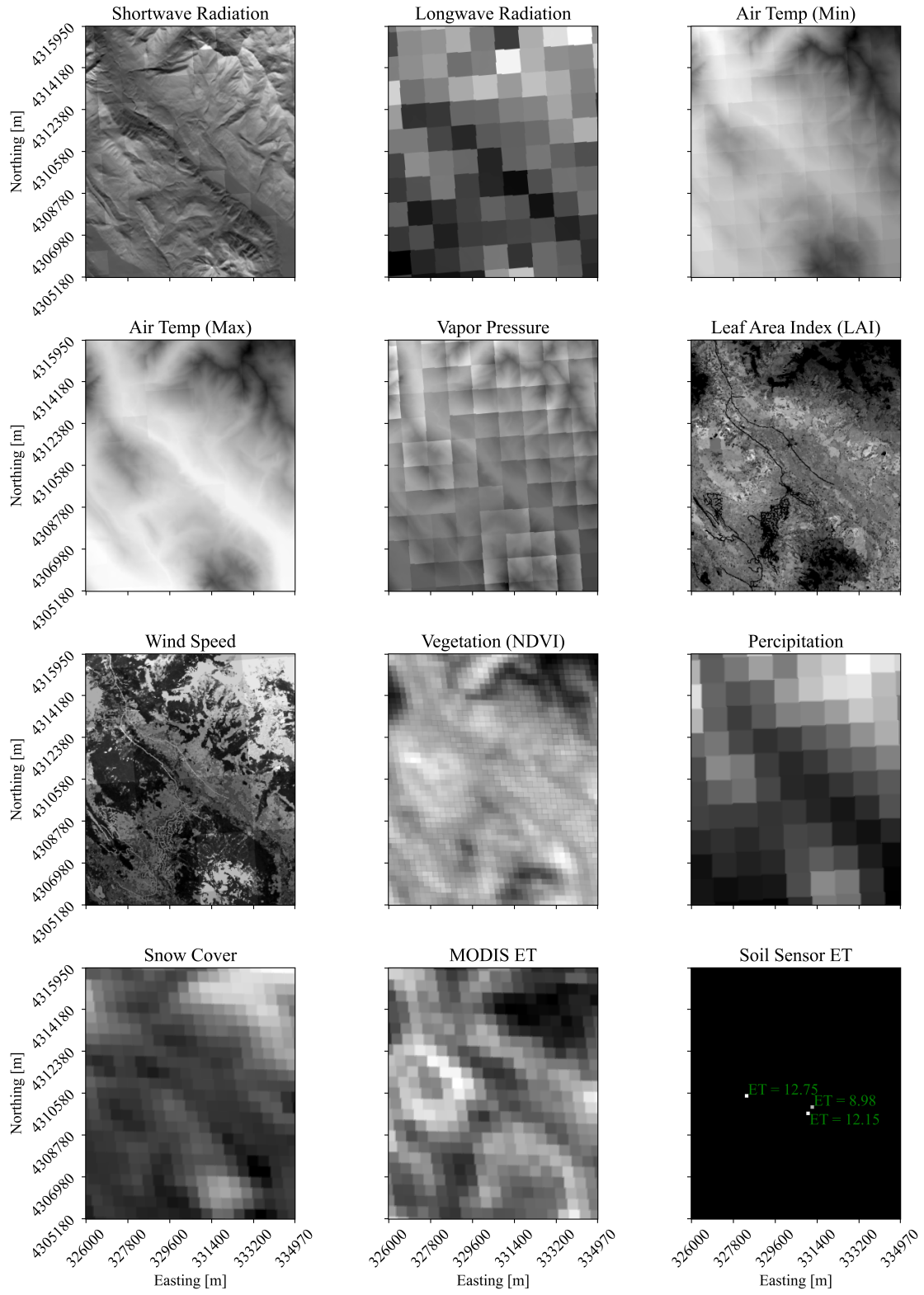


Figure 4.6: Spatio-temporal averaged values of each variable correlated to ET, and the two ground truth labels of ET.

All data is standardized to zero-mean, unit-variance. A neural network is innately incompatible with NaN values, thus all NaN values are replaced with zero, after standardization. This is the equivalent of filling in all missing values with the mean, since standardization transforms the data to zero-mean. Further, when doing the matrix multiplication of a forward pass through a neural network, using zeros are equivalent to removing that input node from the network. Filling missing values with zero in this way, forces the trained neural networks to learn a latent space with a form of inherent gap filling by indicating these values are missing (similar to the DAE used in Section 4.1). Alternatively, this missing values can be filled with interpolation schemes; however, the information that those values were originally missing then becomes lost when training the network. Cross-validation is used to train and evaluate each model.

The architecture of the 2D model is illustrated in Figure 4.7, and functionality is defined as:

$$\hat{\mathbf{P}}^{(t)} = f_{\theta}(\mathbf{D}^{(t)}) \quad (4.8)$$

Where $\mathbf{D}^{(t)}$ is a data map of the predictor variables after spatially resampling to a 300x360 spatial grid of 30m x 30m patches, as observed at the temporarily resampled time t . Quantity $\hat{\mathbf{P}}$ is a predicted ET matrix corresponding to the same spatial grid at the same time t , where $\mathbf{P}^{(t)}$ is the ground truth ET map at measured from MODIS. The model parameters, θ , are trained with an Adam optimizer [50] and stochastic gradient descent [85] by minimizing the the mean squared error between $\hat{\mathbf{P}}$ and $\mathbf{P}^{(t)}$ over the given training fold of cross-validation.

The architecture of the 1D model is illustrated in Figure 4.8, and functionality is defined as:

$$\hat{p}_{x,y}^{(t)} = g_{\phi}(\mathbf{D}_{i_1:i_2,j_1:j_2}^{(t)}) \quad (4.9)$$

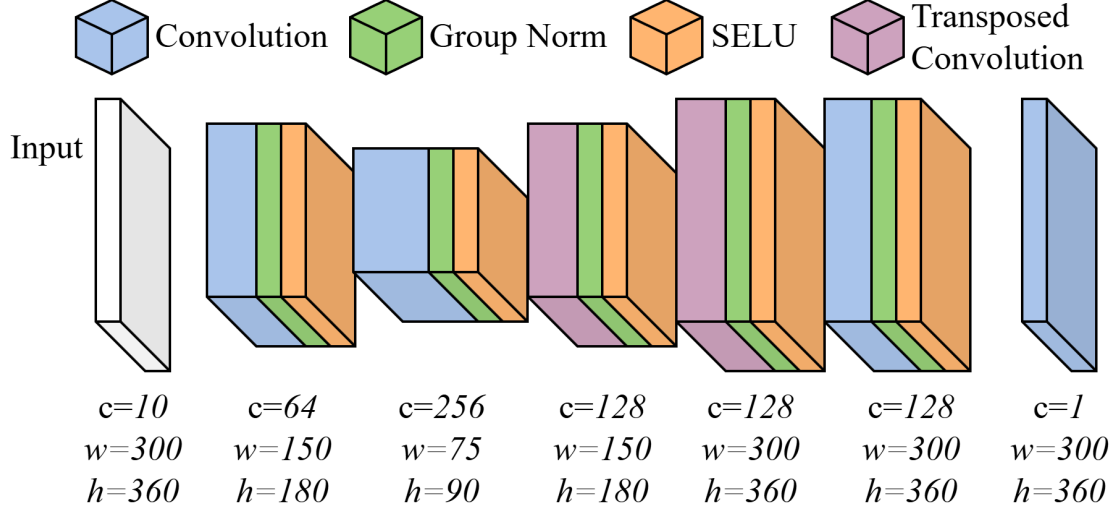


Figure 4.7: CNN Unet structure of the 2D model, where data flows from left to right. Below each block is the output dimensions, where c denotes channels, w denotes width, and h denotes height.

Where the function g_ϕ is a CNN with trainable model parameters ϕ . Quantity $\hat{p}_{x,y}^{(t)}$ is a predicted ET scalar value at time t and spatial position (x, y) , and p is the ground truth ET measured from the soil sensor placed at the same coordinates. The model parameters, ϕ , are trained by minimizing the the mean squared error between $\hat{p}$ and p over the given training set. Quantity $\mathbf{D}_{i_1:i_2, j_1:j_2}^{(t)}$ is now a sub-matrix of $\mathbf{D}^{(t)}$. Index values i_1 and i_2 are the minimum and maximum range of index values (inclusive) as calculated from the index value i that corresponds to the spatial coordinate at $x = X_i$, such as:

$$i_1, i_2 = i - w, i + w \quad (4.10)$$

Where w is the width of the sub-matrix, and h is the height so that:

$$j_1, j_2 = j - h, j + h \quad (4.11)$$

Where j , j_1 , and j_2 are similarly the index values corresponding to the spatial coordinate at $y = Y_j$. For this study, $w = h = 3$.

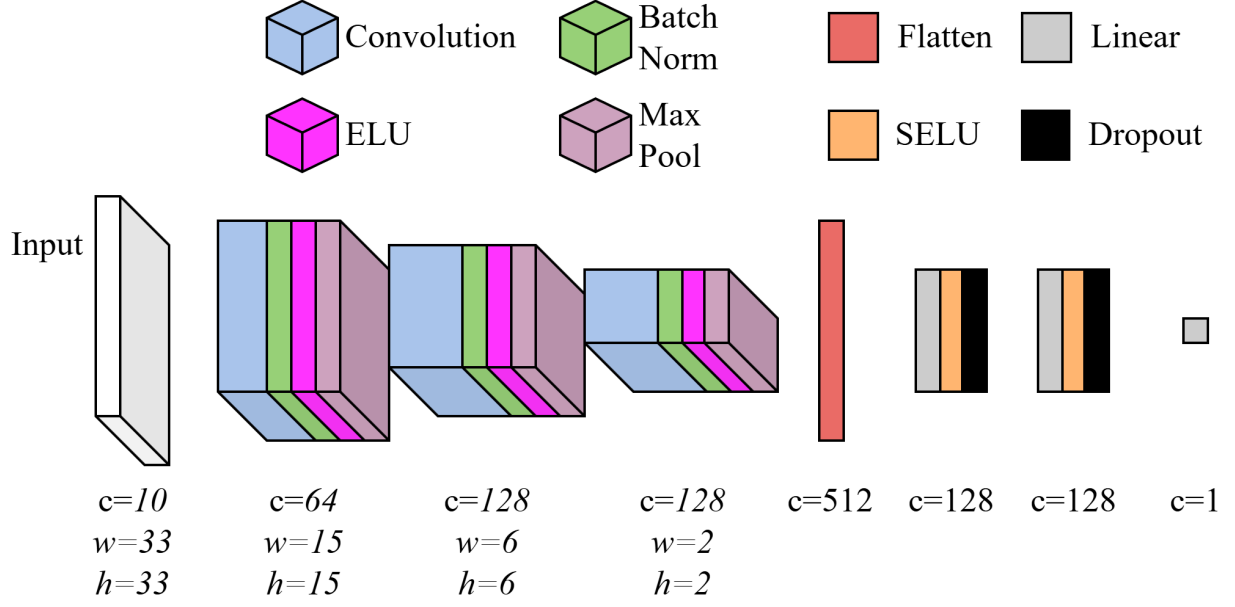


Figure 4.8: CNN structure of the 1D model, where data flows from left to right. Below each block is the output dimensions, where c denotes channels, w denotes width, and h denotes height.

The 2D model leverages robust spatial correlations in both predictors and target variables, over a large region of ground truth ET values from the MODIS data. However, the 1D model leverages higher accuracy ET ground truth values over a significantly smaller region from the soil sensors. A fine tuning approach is taken to integrate large-scale spatial correlations extracted from global sensors with the high-accuracy inherent in small-scale local sensors. The core challenge is that the dimensionalities of the ET data collected from the 2D MODIS and 1D soil sensors are incompatible. For this reason, a bias-correction algorithm, Algorithm 5, is adopted to finetune the 2D model with 1D soil sensor data. The fine tuned model is defined as:

$$\hat{\mathbf{P}}^{(t)} = f_{\theta_2}(\mathbf{D}^{(t)}) \quad (4.12)$$

Where now the post-trainable parameters are denoted by θ_2 which are derived from the pretrained parameters θ in Equation 4.8, but the structure and functionality of f is otherwise identical. Algorithm 5 describes the fine tuning approach, which uses bias-correction to shift

Algorithm 5 Fine Tuning the 2D Model with the 1D Data

Require: θ - a set of pretrained parameters for the 2D model using the MODIS labels

Require: P - a set of 1D ET labels from the soil sensors

Require: S - a set of spatial coordinates, (x, y) , corresponding to each label in P

Require: D - a set of predictor data matrices

Require: T - a set of times, t , to train on

$\hat{P} \leftarrow \{f_{\theta}(D_t) \mid t \in T\}$ ▷ make an initial set of 2D ET predictions

$B \leftarrow \{\hat{P}_{x,y}^{(t)} - P_{x,y}^{(t)} \mid (x, y) \in S, t \in T\}$ ▷ calculate a set of scalar bias errors

$M \leftarrow \{\hat{P}^{(t)} - B_{x,y}^{(t)} \mid (x, y) \in S, t \in T\}$ ▷ generate bias-shifted 2D maps

$\theta_2 \leftarrow \text{train}(f, \theta, D, M)$ ▷ Post-train the 2D model on bias-shifted maps

return θ_2

the predicted maps from the 2D model and subsequently retrain it.

The error on the test set from the 2D model is an R^2 score of 0.8872, and root mean squared error (RMSE) of 2.3630 mm/8-day. Figure 4.9 shows the spatial correlations in prediction errors on the test set after training the 2D model. The R^2 value at each spatial coordinate is averaged over all temporal coordinates. Figure 4.10 plots temporal correlations in ET predictions, instead averaging over all spatial dimensions at a given time.

The error on the test set from the 1D model is an R^2 score of 0.8513, and RMSE of 4.2637 mm/8-day. The error on the test set from fine-tuning is an R^2 score of 0.9802, and RMSE of 0.7789 mm/8-day, as evaluated on the bias-shifted ET maps. Figure 4.11 plots the predicted ET values for each of the models, illustrating the spatial patterns learned for each one as they progress to the final prediction map. Visibly, the fine tuning spatial structures are retained from both the 2D and 1D models. However, the actual values of the ET lay somewhere in between the two models. It is difficult to determine which of these is the most “ground truth” value.

Fine tuning with the bias-shifted maps is able to properly adjust to the higher accuracy of the soil sensors, as warranted by the significant increase in R^2 scores from 0.8872 to 0.9804. This suggests that bias-shifted maps are more correlated to the given predictor variables, demonstrating this as a potentially more robust ground truth.

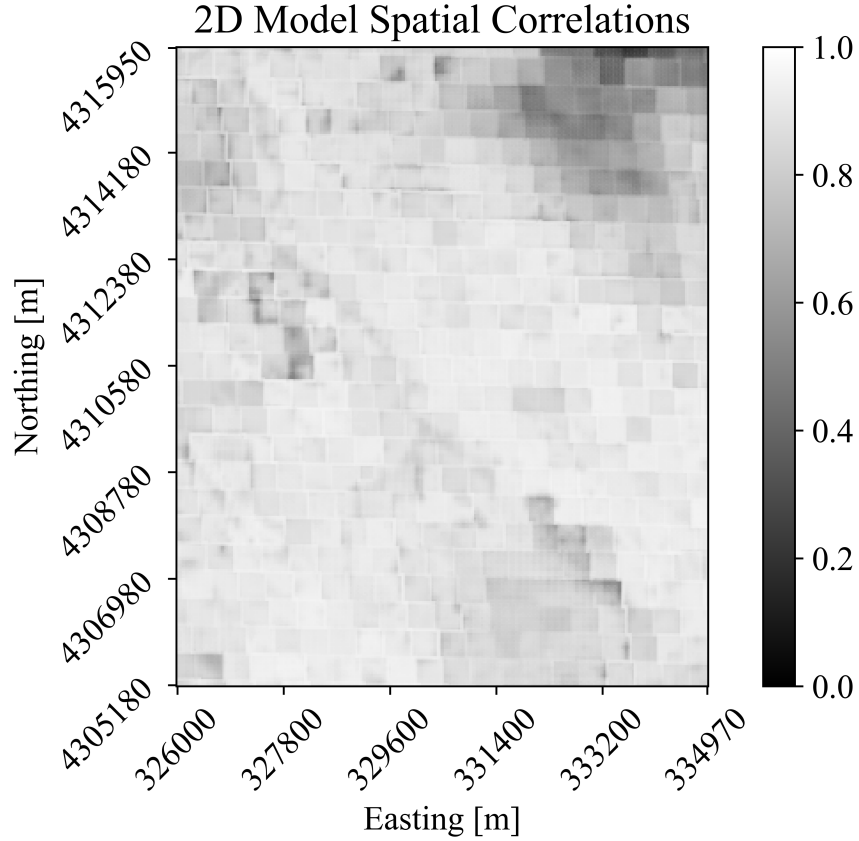


Figure 4.9: Spatial correlations in R^2 scores (shown on the z-axis in grayscale) from predictions on the test set as evaluated on the pretrained 2d model. At each (x, y) coordinate, is the R^2 score calculated over all times for those coordinates.

Figure 4.9 illustrates spatial regions of high and low R^2 scores, where the dark regions indicate spatial coordinates where the ET predictions should be accepted with lower confidence. These darker regions also correspond to similar regions visible in the predictor variables, as shown in Figure 4.6. The high accuracy in summer time, as shown in Figure 4.10, is an expected phenomenon typical in water sheds, as other seasons correspond to higher snow cover and rough weather conditions which make it difficult to model ET. Table 4.4 quantifies these relationships between R^2 scores and each of the predictor variables, by measuring the Pearson Correlation Coefficient (PCC) between the percent error and each of the predictor variables. The highest correlations come from the variables: snow cover (0.11), NDVI (-0.09), and vapor pressure (-0.09).

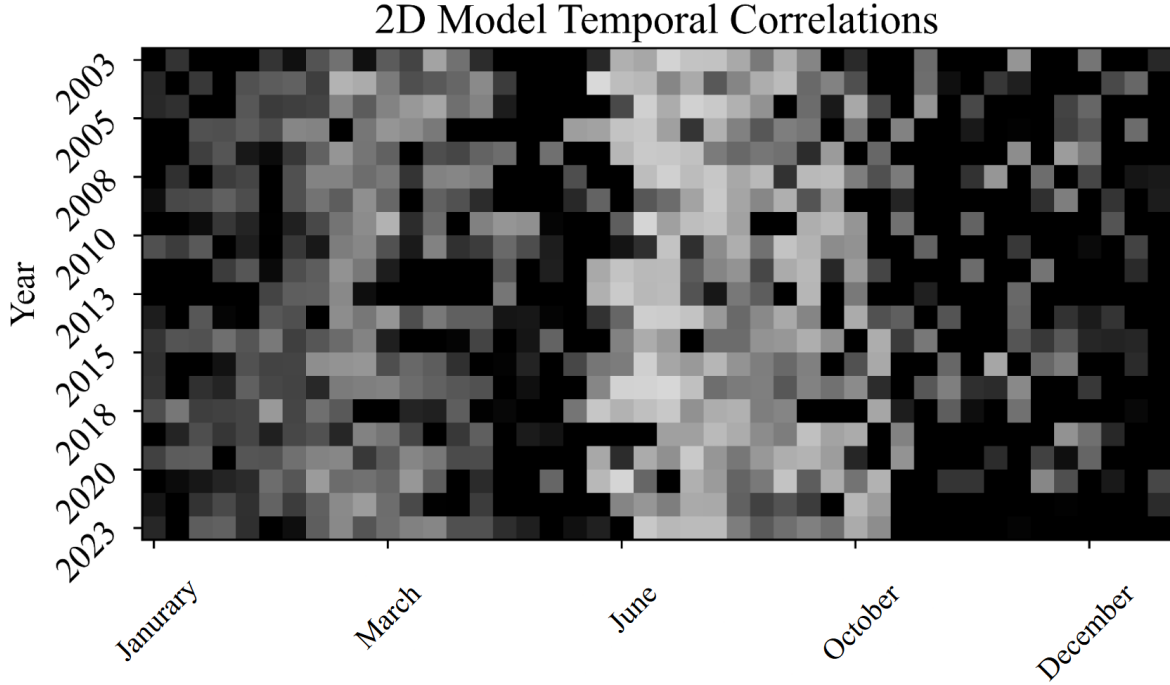


Figure 4.10: Temporal correlations in R^2 scores from predictions on the test set as evaluated on the pretrained 2d model. At each time, t , is the R^2 score calculated over all spatial coordinates for that time.

4.4 Summary

This chapter presented two main applications of multimodal neural networks used to fuse real-world scientific data, for studying EvapoTranspiration (ET) and classifying minerals. It was shown how a Denoising AutoEncoder (DAE) can be used with Monte Carlo Dropout (MCD) to denoise, gap-fill, and sample from a DNN to reduce bias and variance while quantifying uncertainty in predictions and input data. It was further shown how multimodal data can be fused in two ways: (1) by training independent stems that are frozen and then integrated via a mutual hidden layer, and (2) by spatio-temporal resampling to a uniform grid and using a CNN or Unet structure to leverage unified spatial information. A fine tuning approach was then presented that integrated advantages of the large-scale spatial correlations with small-scale high-fidelity data.

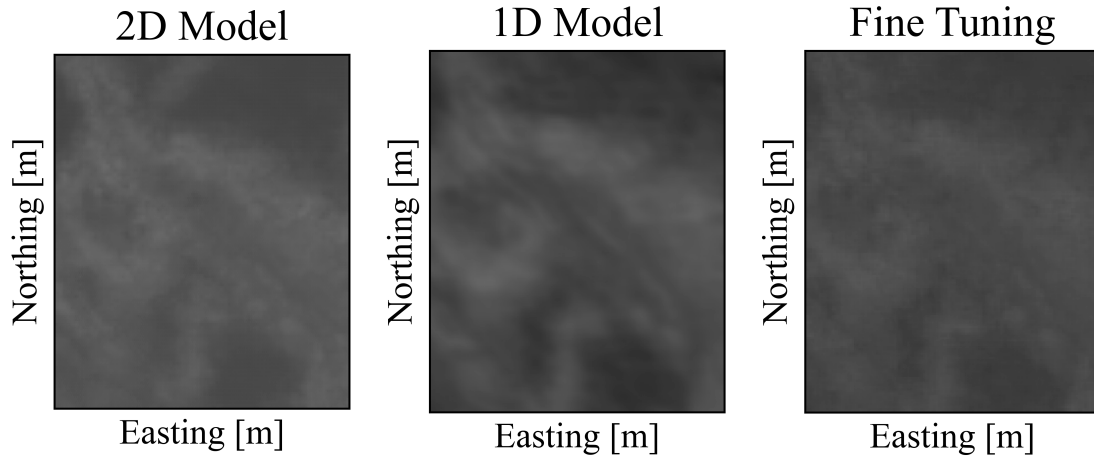


Figure 4.11: The resulting ET prediction maps for each of the three models, averaged at each spatial coordinate over time. The data shown is collected by inputting the same predictor observations into each of the models, between the years 2003-2023 during the months May to October.

Table 4.4: Pearson Correlation Coefficient (PCC) between each of the predictor variables and the percent error in ET predictions from the pretrained 2D model (w.r.t. MODIS labels).

Variable	PCC
Shortwave Radiation	0.03
Longwave Radiation	-0.00
Air Temp (Min)	-0.07
Air Temp (Max)	-0.08
Vapor Pressure	-0.09
Leaf Area Index (LAI)	-0.05
Wind Speed	0.08
Vegetation (NDVI)	-0.09
Percipitation	0.07
Snow Cover	0.11

Chapter 5

Conclusions

This thesis presented end-to-end methods for autonomous systems, focusing on advancements in rigorous hardware-in-the-loop (HIL) testbeds with integrated high performance computing, flexible navigation frameworks, adaptive systems optimizing both resource expenditure and task accuracy, and real-world scientific deployments.

First was developed a rigorous testbed environment. This combines the robust drone and physics simulator Microsoft AirSim built on Unreal Engine, with NVIDIA Jetson embedded processors. Together, these provided a HIL testbed that was used to evaluate both navigation reliability and computing efficiency of various sense-compute-act pipelines. High performance computing clusters and AI-accelerated servers were leveraged to train deep neural networks (DNN) for robot navigation, real-time perception, and onboard science. Considerations were taken to reduce the overhead of executing those DNNs on resource constrained devices such as NVIDIA Jetson processors deployed to lightweight vehicles.

Second was demonstrated a flexible hierarchal navigation framework, that is capable of adapting to generalized embodiments, sensors, and mission objectives through three highly modular and dynamic levels. The low-level controller allows robot-specific firmware to dictate

the execution of actuators in response to generalized motion commands. These commands are generated by a mid-level policy which learns a mapping from on-board sensors via rule-based, imitation learning, and deep reinforcement learning (DRL) algorithms. Of which, a robust DRL methodology was developed that imbues curriculum learning with A* shortest paths, and a caching mechanism to drastically reduce training and evaluation times. A higher level of augmentation was enabled by overseeing operations of the autonomous system, and intervening when the robot requires new instructions. This was accomplished by integrating global world building from local sensors, via ray tracing and state updates, to form visual-language prompts to input into multimodal large language models such as Gemma3. This enabled the generation of intermediate waypoints to be fed to the mid-level policy. It was shown that the high-level policy integration can be used to escape out-of-distribution errors which plague DRL-based policies. The MLLM can be used to further obfuscate general commands which can be decomposed into a set of waypoints to give to a robot, as enabled through flexible user-defined linguistic instructions. The end-to-end 3-tier hierarchal framework closes the gap between sensing and actuation, as validated by successfully deploying a mid-level DRL policy trained with the cache mechanism to a low-level real-time controller, with zero accuracy loss, and further augmenting it with the high-level policy with up to 27% navigation accuracy gains (varying the environment complexity and the degree of inherent out-of-distribution errors).

Third was introduced the methodology behind adaptive dynamic deep neural networks (ADDNN). Dynamic DNNs (DDNN) are defined by their ability to scale computing at runtime, or otherwise activate different sub-paths used to route data processing from input to output. The DDNNs took on the form of embedded slimmable and gated multi-branch networks. Adaptive mechanisms then augmented the DDNNs so that they can scale in response to perceived context and changing mission objectives. It was shown that this approach enabled the networks to adjust to observed data distribution shifts, resulting in drastic reductions in resource costs (computer power, energy expenditure, inference latency,

communicated data rate, number of sensor acquisitions, and sensor power levels). Remarkably, the adaptive methods also resulted in significant improvements to navigation accuracy, where model reduction strategies typically come with negligible accuracy boosts or slight degradations. Pertinent observations were made to properly train ADDNN frameworks by adjusting to unstable gradients through decoupling the dynamic and adaptive processes, and overcoming a latent-observation paradox by considering the problem as a time-series system. These observations and corresponding solutions were found robust by validating them through ADDNN-based navigation frameworks that resulted in 61-80% reductions in sensor power levels, 57-82% reductions in DNN complexity, 95% reductions in communicated data rates, and significant reductions in resources allocated to computing for perception based DNNs: 75% energy expenditure, 16% power consumption, and 75% inference latency. These frameworks resulted in either negligible loss in navigation accuracy, or more profoundly in a 7% increase in navigation accuracy when properly decoupling the adaptive and dynamic mechanisms through a scalable perception backbone and a joint adaptive-task head. These results warrant a more epistemic conclusion, that DNNs can be constructed as hierarchical structures with overlapping and interleaving latent spaces that have optimal scales embedded within them (dynamic) – which can further be activated in response to perceived data distribution shifts encountered throughout the model’s life cycle (adaptive).

Fourth was presented mitigation and data fusing strategies when considering the artifacts inherent in real-world scientific data. Uncertainty in predictive models can be quantified via sampling methods that utilize Monte Carlo Dropout and sensitivity analysis techniques. The uncertainty arising in the form of both bias and variance can be reduced, for both input data and output predictions, by employing denoising autoencoder methods into the training procedures. Heterogeneous data that varies in the modality, dimensions, scope, and resolutions can be fused through employing either: frozen stems which learn independent features first, followed by codependent features second; or resampling methods to jointly learn robust spatial features.

The HIL testbed; hierarchal navigation framework with abstract robot agent layers; ADDNN methods; training procedures; trained models; collected datasets; static hold-out evaluation sets; are all publicly released for open-source use and can be found at ¹. A goal of publishing the work done in this thesis is to foster an environment that accelerates generalized robot autonomy and robust perception modules, and to encourage student/researcher involvement.

Future work will leverage the generalizations of the hierarchal navigation framework to deploy to real-world drones and rovers, through proper definitions of the abstract low-level control layers and retraining of the DRL based mid-level policies. Further generalizations will be explored through the use of varying user-based instructions as fed as linguistic modalities to the MLLM models, and investigations into more cohesive visual-language-action (VLA) models.

Adaptive dynamic deep neural networks will pave the way towards a new generation of ever-changing machine learning models, enabling profound levels of generalization and optimization in autonomous systems.

¹<https://www.github.com/WreckItTim/OmniNaviPy>

Bibliography

- [1] S. Ahn, S. X. Hu, A. Damianou, N. D. Lawrence, and Z. Dai. Variational information distillation for knowledge transfer, 2019. 38
- [2] J. S. Assine, E. Valle, M. Levorato, et al. Slimmable encoders for flexible split dnns in bandwidth and resource constrained iot systems. *arXiv preprint arXiv:2306.12691*, 2023. 50
- [3] T. Baltrušaitis, C. Ahuja, and L.-P. Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018. 98
- [4] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. Scalable methods for 8-bit training of neural networks, 2018. 38
- [5] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957. 17
- [6] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. 12
- [7] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009. 19
- [8] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Gutttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020. 38
- [9] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 12
- [10] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997. 94
- [11] A. Cipolletta, V. Peluso, A. Calimera, M. Poggi, F. Tosi, F. Aleotti, and S. Mattoccia. Energy-quality scalable monocular depth estimation on low-power cpus. *IEEE Internet of Things Journal*, 9(1):25–36, 2021. 39

- [12] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications, 2015. 38
- [13] M. S. De Pondeva, G. S. Manikin, G. DiMego, S. G. Benjamin, D. F. Parrish, R. J. Purser, W.-S. Wu, J. D. Horel, D. T. Myrick, Y. Lin, et al. The real-time mesoscale analysis at noaa’s national centers for environmental prediction: current status and development. *Weather and Forecasting*, 26(5):593–612, 2011. 97
- [14] Dropbox, Inc. Dropbox. <https://www.dropbox.com>, 2024. Accessed: 2024-05-20. 5
- [15] T. Elmokadem and A. V. Savkin. Towards fully autonomous uavs: A survey. *Sensors*, 21(18):6223, 2021. 2
- [16] Epic Games. Unreal engine. 2
- [17] J. Faundeen, T. E. Burley, J. A. Carlino, D. L. Govoni, H. S. Henkel, S. L. Holl, V. B. Hutchison, E. Martín, E. T. Montgomery, C. Ladino, et al. The united states geological survey science data lifecycle model. Technical report, US Geological Survey, 2014. 98
- [18] A. Faust, K. Oslund, O. Ramirez, A. Francis, L. Tapia, M. Fiser, and J. Davidson. Prm-rl: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 5113–5120. IEEE, 2018. 22
- [19] B. Faybishenko, R. Versteeg, G. Pastorello, D. Dwivedi, C. Varadharajan, and D. Agarwal. Challenging problems of quality assurance and quality control (QA/QC) of meteorological time series data. *Stochastic Environ. Res. Risk Assess.*, 36(4):1049–1062, Apr. 2022. 86
- [20] J. J. Freeman, A. Wang, K. E. Kuebler, B. L. Jolliff, and L. A. Haskin. Characterization of natural feldspars by raman spectroscopy for future planetary exploration. *The Canadian Mineralogist*, 46(6):1477–1500, 2008. 93
- [21] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018. 12, 15, 55
- [22] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016. 87
- [23] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992. 40
- [24] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. Squeezenext: Hardware-aware neural network design, 2018. 38
- [25] C. Godard, O. Mac Aodha, M. Firman, and G. J. Brostow. Digging into self-supervised monocular depth estimation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3828–3838, 2019. 20, 38

- [26] H. Halawa, H. A. Abdelhafez, A. Boktor, and M. Ripeanu. Nvidia jetson platform characterization. In *European Conference on Parallel Processing*, pages 92–105. Springer, 2017. 4
- [27] 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*, 4(2):100–107, 1968. 11, 12, 19
- [28] T. Hastie. The elements of statistical learning: data mining, inference, and prediction, 2009. 40
- [29] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 13, 38, 47
- [30] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006. 86
- [31] X. Hu, L. Zhu, T. Wang, C.-W. Fu, and P.-A. Heng. Single-image real-time rain removal based on depth-guided non-local features. *IEEE Transactions on Image Processing*, 30:1759–1770, 2021. 20, 38, 39, 74
- [32] J. Huang, R. Yagel, V. Filippov, and Y. Kurzion. An accurate method for voxelizing polygon meshes. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 119–126, 1998. 3
- [33] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131, 2018. 38
- [34] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015. 48
- [35] Z. Jiang, C. Li, X. Chang, L. Chen, J. Zhu, and Y. Yang. Dynamic slimmable denoising network. *IEEE Transactions on Image Processing*, 32:1583–1598, 2023. 39
- [36] T. Johnsen, X. Bi, C. Chou, C. Varadharajan, Y. Wu, J. Skone, and L. Ramakrishnan. Data fusion with deep learning for mapping evapotranspiration and error analysis. In *AGU24 Annual Meeting*, Washington, D.C., United States, dec 2024. Conference presentation. 96
- [37] T. Johnsen and V. Gulick. Artificial intelligence to classify minerals and rocks with raman spectra and image analysis. In *AGU Fall Meeting Abstracts*, volume 2019, pages P43E–3507, 2019. 91
- [38] T. K. Johnsen, X. Bi, C. Chou, C. Varadharajan, Y. Wu, J. Skone, and L. Ramakrishnan. Denoising autoencoder for reconstructing sensor observation data and predicting evapotranspiration: Noisy and missing values repair and uncertainty quantification. *Water Resources Research*, 61(10):e2024WR039831, 2025. vi, vii, viii, 85, 88, 90, 91, 92, 97

- [39] T. K. Johnsen and V. C. Gulick. Single-and multi-mineral classification using dual-band raman spectroscopy for planetary surface missions. *American Mineralogist*, 110(5):685–698, 2025. vii, 92, 94, 97
- [40] T. K. Johnsen, I. Harshbarger, and M. Levorato. An overview of adaptive dynamic deep neural networks via slimmable and gated architectures. In *2024 15th International Conference on Information and Communication Technology Convergence (ICTC)*, pages 252–256. IEEE, 2024. 39, 44
- [41] T. K. Johnsen, I. Harshbarger, Z. Xia, and M. Levorato. Navisplit: Dynamic multi-branch split dnns for efficient distributed autonomous navigation. In *2024 IEEE 25th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 196–201. IEEE, 2024. v, 44, 62, 64, 65, 67
- [42] T. K. Johnsen and M. Levorato. Navislim: Adaptive context-aware navigation and sensing via dynamic slimmable networks. In *2024 IEEE/ACM Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 110–121. IEEE, 2024. iv, v, 44, 46, 47, 53, 57, 58, 59, 60, 61
- [43] T. K. Johnsen and M. Levorato. Cadence: Context-adaptive depth estimation for navigation and computational efficiency, 2026. vi, 44, 73, 74, 77, 78, 79, 80, 81, 82
- [44] C. O. Justice, E. Vermote, J. R. Townshend, R. Defries, D. P. Roy, D. K. Hall, V. V. Salomonson, J. L. Privette, G. Riggs, A. Strahler, et al. The moderate resolution imaging spectroradiometer (modis): Land remote sensing for global change research. *IEEE transactions on geoscience and remote sensing*, 36(4):1228–1249, 1998. 98
- [45] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, page 615–629, New York, NY, USA, 2017. Association for Computing Machinery. 21
- [46] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 40, 41
- [47] L. S. Karumbunathan. Nvidia jetson agx orin series. *A Giant leap forward for robotics and edge AI applications. Technical Brief*, 2022. 4
- [48] K. Kawaharazuka, J. Oh, J. Yamada, I. Posner, and Y. Zhu. Vision-language-action models for robotics: A review towards real-world applications. *IEEE Access*, 2025. 24
- [49] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024. 24
- [50] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 101

- [51] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2021. 14, 38
- [52] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems*, pages 972–981, 2017. 62
- [53] V. Konda and J. Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999. 12
- [54] L. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings 22nd international conference on distributed computing systems workshops*, pages 575–578. IEEE, 2002. 43, 45
- [55] 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*, 110(9):2501–2540, 2021. 14, 38
- [56] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 2012. 38
- [57] T. Krueger. Bayesian inference of uncertainty in freshwater quality caused by low-resolution monitoring. *Water Res.*, 115:138–148, May 2017. 86
- [58] A. Kumar, A. Raghunathan, R. Jones, T. Ma, and P. Liang. Fine-tuning can distort pretrained features and underperform out-of-distribution. *arXiv preprint arXiv:2202.10054*, 2022. 95
- [59] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 2002. 17, 45
- [60] J. H. Lee, M.-K. Han, D. W. Ko, and I. H. Suh. From big to small: Multi-scale local planar guidance for monocular depth estimation. *arXiv preprint arXiv:1907.10326*, 2019. 20, 38
- [61] 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)*, pages 341–342. IEEE, 2021. 53
- [62] 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*, pages 8607–8617, 2021. 39
- [63] Y. Liao, J. Xie, and A. Geiger. Kitti-360: A novel dataset and benchmarks for urban scene understanding in 2d and 3d, 2022. 20

- [64] H. Liu, C. Li, Q. Wu, and Y. J. Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023. 26
- [65] Y. Liu, A. Wang, and J. Freeman. Raman, mir, and nir spectroscopic study of calcium sulfates: gypsum, bassanite, and anhydrite. In *40th Annual Lunar and Planetary Science Conference*, page 2128, 2009. 93
- [66] A. V. Malawade, T. Mortlock, and M. A. Al Faruque. Hydrافusion: Context-aware selective sensor fusion for robust and efficient autonomous vehicle perception. In *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPs)*, pages 68–79. IEEE, 2022. 39, 50
- [67] A. Masoumian, H. A. Rashwan, S. Abdulwahab, J. Cristiano, M. S. Asif, and D. Puig. Gcndepth: Self-supervised monocular depth estimation based on graph convolutional network. *Neurocomputing*, 517:81–92, 2023. 20, 38, 70
- [68] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia. Bottlefit: Learning compressed representations in deep neural networks for effective and efficient split computing. In *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 337–346. IEEE, 2022. 21, 63
- [69] Y. Matsubara, M. Levorato, and F. Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Computing Surveys*, 55(5):1–30, 2022. 21
- [70] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos. Image segmentation using deep learning: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3523–3542, 2021. 38
- [71] V. R. Miranda, A. A. Neto, G. M. Freitas, and L. A. Mozelli. Generalization in deep reinforcement learning for robotic navigation by reward shaping. *IEEE Transactions on Industrial Electronics*, 71(6):6013–6020, 2023. 22
- [72] A. Mishra and D. Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy, 2017. 38
- [73] 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*, 518(7540):529–533, 2015. 12, 15, 17
- [74] J. L. Montie. Evaporation and environment. *Symposia of the Society for Experimental Biology*, 19:205–234, 1965. 98
- [75] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. 17

- [76] M. Nakahara, M. Nishimura, Y. Ushiku, T. Nishio, K. Maruta, Y. Nakayama, and D. Hisano. Edge computing-assisted dnn image recognition system with progressive image retransmission. *IEEE Access*, 10:91253–91262, 2022. 21
- [77] National Energy Research Scientific Computing Center. *Perlmutter System Documentation*, 2026. Accessed: 2026-04-08. 6
- [78] 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. 39, 50
- [79] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016. 11
- [80] 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)*, pages 7512–7519. IEEE, 2021. 14, 38
- [81] A. Pappalardo, Y. Umuroglu, M. Blott, J. Mitrevski, B. Hawks, N. Tran, V. Loncar, S. Summers, H. Borrás, J. Muhizi, et al. Qonnx: Representing arbitrary-precision quantized neural networks. *arXiv preprint arXiv:2206.07527*, 2022. 38
- [82] H. L. Penman. Natural evaporation from open water, bare soil and grass. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 193:120–145, 1948. 98
- [83] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 2002. 19
- [84] K. J. Åström and T. Hägglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, Research Triangle Park, NC, 2nd edition, 1995. 10
- [85] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951. 101
- [86] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention—MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015. 20, 98
- [87] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 45
- [88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. 17, 45

- [89] M. Sapienza, E. Guardo, M. Cavallo, G. Torre, G. Leombruno, and O. Tomarchio. Solving critical events through mobile edge computing: An approach for smart cities. pages 1–5, 05 2016. 21, 38
- [90] A. S. Sayyed and F. Restuccia. Sinf: Semantic inference in neural networks with semantic subgraphs. In *ICML 2025 Workshop on Machine Learning for Wireless Communication and Networks (ML4Wireless)*, 2025. 45
- [91] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 12
- [92] 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*, pages 621–635. Springer, 2018. 2
- [93] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus. Indoor segmentation and support inference from rgb-d images. In *European conference on computer vision*, pages 746–760. Springer, 2012. 20
- [94] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. 46
- [95] M. Stone. Cross-validators choice and assessment of statistical predictions. *Journal of the royal statistical society: Series B (Methodological)*, 36(2):111–133, 1974. 94
- [96] K. Team, A. Du, B. Yin, B. Xing, B. Qu, B. Wang, C. Chen, C. Zhang, C. Du, C. Wei, et al. Kimi-vl technical report. *arXiv preprint arXiv:2504.07491*, 2025. 26
- [97] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2464–2469. IEEE, 2016. 39, 45
- [98] P. E. Thornton, S. W. Running, and M. A. White. Generating surfaces of daily meteorological variables over large regions of complex terrain. *Journal of hydrology*, 190(3-4):214–251, 1997. 97
- [99] P. E. Thornton, R. Shrestha, M. Thornton, S.-C. Kao, Y. Wei, and B. E. Wilson. Gridded daily weather data for north america with comprehensive uncertainty quantification. *Scientific Data*, 8(1):190, 2021. 97
- [100] University of Arizona High Performance Computing. Research data center (rdc) resources. https://hpcdocs.hpc.arizona.edu/resources/data_center/, 2026. Accessed: 2026-04-08. 6
- [101] N. D. Van, M. Sualeh, D. Kim, and G.-W. Kim. A hierarchical control system for autonomous driving towards urban challenges. *Applied Sciences*, 10(10):3543, 2020. 11

- [102] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016. 75
- [103] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 24
- [104] S. Vemprala, S. Mian, and A. Kapoor. Representation learning for event-based visuomotor policies. *Advances in Neural Information Processing Systems*, 34:4712–4724, 2021. 14, 38
- [105] G. Villarini, P. V. Mandapaka, W. F. Krajewski, and R. J. Moore. Rainfall and sampling uncertainties: A rain gauge perspective. *J. Geophys. Res. Atmos.*, 113(D11), June 2008. 86
- [106] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008. 86
- [107] A. Wang, J. J. Freeman, B. L. Jolliff, and I.-M. Chou. Sulfates on mars: A systematic raman spectroscopic study of hydration states of magnesium sulfates. *Geochimica et cosmochimica acta*, 70(24):6118–6135, 2006. 93
- [108] A. Wang, B. L. Jolliff, and L. A. Haskin. Raman spectroscopy as a method for mineral identification on lunar robotic exploration missions. *Journal of Geophysical Research: Planets*, 100(E10):21189–21199, 1995. 93
- [109] W. Wang, D. Tran, and M. Feiszli. What makes training multi-modal classification networks hard? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12695–12705, 2020. 95
- [110] C. J. C. H. Watkins et al. Learning from delayed rewards. 1989. 12
- [111] Y. Wu, C. Chou, C. Beutler, A. Henderson, B. Dafflon, S. Wielandt, A. Newman, E. Brodie, B. Gilbert, L. Peruzzo, et al. Meteorological and soil data from ecohydrology sensor towers at pump house and snodgrass mountain in east river watershed, colorado, 2019-2025. Technical report, Watershed Function SFA, 2025. 98
- [112] Y. Wu and K. He. Group normalization. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision – ECCV 2018*, pages 3–19, Cham, 2018. Springer International Publishing. 48, 62
- [113] J. Xiao, K. Zhang, X. Xu, S. Liu, S. Wu, Z. Huang, and L. Li. Improving accuracy and efficiency of monocular depth estimation in power grid environments using point cloud optimization and knowledge distillation. *Energies*, 17(16):4068, 2024. 39

- [114] M. Yang, T. K. Johnsen, I. Harshbarger, M. Mendula, and M. Levorato. Smart-depth: Motion-aware depth prediction with intelligent computing for navigation. In *2025 IEEE 21st International Conference on INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING IN SMART SYSTEMS AND THE INTERNET OF THINGS (DCOSS-IoT)*. IEEE, 2025. v, vi, viii, 68, 69, 71
- [115] S. Yin, C. Fu, S. Zhao, K. Li, X. Sun, T. Xu, and E. Chen. A survey on multimodal large language models. *National Science Review*, 11(12):nwae403, 2024. 24
- [116] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*, pages 44–60. Springer, 2003. 6
- [117] J. Yu and T. S. Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1803–1811, 2019. 39, 45, 47
- [118] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018. 39, 45, 47, 49, 78
- [119] X. Zhai, B. Mustafa, A. Kolesnikov, and L. Beyer. Sigmoid loss for language image pre-training. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 11975–11986, 2023. 26
- [120] G. Zhang, X. Tang, L. Wang, H. Cui, T. Fei, H. Tang, and S. Jiang. Repmono: a lightweight self-supervised monocular depth estimation architecture for high-speed inference. *Complex & Intelligent Systems*, 10(6):7927–7941, 2024. 39
- [121] Z. Zhang. Microsoft kinect sensor and its effect. *IEEE MultiMedia*, 19(2):4–10, 2012. 20
- [122] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye. Object detection in 20 years: A survey. *Proceedings of the IEEE*, 111(3):257–276, 2023. 38

ProQuest Number: 32736402

INFORMATION TO ALL USERS

The quality and completeness of this reproduction is dependent on the quality and completeness of the copy made available to ProQuest.



Distributed by
ProQuest LLC a part of Clarivate (2026).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

ProQuest LLC
789 East Eisenhower Parkway
Ann Arbor, MI 48108 USA