NaviSplit: Dynamic Multi-Branch Split DNNs for Efficient Distributed Autonomous Navigation

1st Timothy K Johnsen
University of California Irvine, USA
tjohnsen@uci.edu

2nd Ian Harshbarger
University of California Irvine, USA
iharshba@uci.edu

3rd Zixia Xia
University of California Irvine, USA
zixiax3@uci.edu

4th Marco Levorato
University of California Irvine, USA
levorato@uci.edu

***Abstract*—Lightweight autonomous unmanned aerial vehicles (UAV) are emerging as a central component of a broad range of applications. However, autonomous navigation necessitates the implementation of perception algorithms, often deep neural networks (DNN), that process the input of sensor observations, such as that from cameras and LiDARs, for control logic. The complexity of such algorithms clashes with the severe constraints of these devices in terms of computing power, energy, memory, and execution time. In this paper, we propose NaviSplit, the first instance of a lightweight navigation framework embedding a distributed and dynamic multi-branched neural model. At its core is a DNN split at a compression point, resulting in two model parts: (1) the head model, that is executed at the vehicle, which partially processes and compacts perception from sensors; and (2) the tail model, that is executed at an interconnected compute-capable device, which processes the remainder of the compacted perception and infers navigation commands. Different from prior work, the NaviSplit framework includes a neural gate that dynamically selects a specific head model to minimize channel usage while efficiently supporting the navigation network. In our implementation, the perception model extracts a 2D depth map from a monocular RGB image captured by the drone using the robust simulator Microsoft AirSim. Our results demonstrate that the NaviSplit depth model achieves an extraction accuracy of 72-81% while transmitting an extremely small amount of data (1.2-18 KB) to the edge server. When using the neural gate, as utilized by NaviSplit, we obtain a slightly higher navigation accuracy as compared to a larger static network by 0.3% while significantly reducing the data rate by 95%. To the best of our knowledge, this is the first exemplar of dynamic multi-branched model based on split DNNs for autonomous navigation.**

***Index Terms*—autonomous navigation, split deep neural networks, supervised compression, dynamic deep neural networks**

I. Introduction

Autonomous navigation increasingly relies on the execution of computationally expensive deep neural networks (DNN) that integrate perception tasks (*e.g.,* object detection, segmentation, or depth estimation) with decision making tasks. In some settings (*e.g.,* lightweight airborne drones), the limited onboard hardware can complicate execution of such DNNs. Such applications impose severe execution deadlines needed to improve reaction time to the surrounding environment. Thus, it is beneficial to develop modern solutions that allow execution of DNN models on lightweight autonomous vehicles while minimizing execution time, memory, and energy expenditure.

Typical solutions adopt two approaches. **Model reduction** simplifies DNN models to be executed onboard, such as: quantization [1] [2] [3], knowledge distillation [4] [5], [6], and direct design [7]. The resulting models incur a degradation in task performance, and their continuous execution requires considerable energy expense. **Edge computing** [8], [9] remotely executes the full DNN at a compute-capable device – the edge server. This transmits input data (*e.g.,* images) over volatile and capacity-constrained wireless links, creating problems in efficient channel usage, delay, delay variance, and security.

We focus on edge computing for lightweight autonomous unmanned aerial vehicles (UAV) (*e.g.,* micro-drones), where hardware limitations affect both computing and sensing capacities. We consider the task of navigating a UAV to a target position through an unknown environment. The UAV is equipped with an efficient monocular RGB camera, which provides limited information toward navigation. Path planning and collision avoidance requires information on the 3D structure of the surrounding space. Thus, the autonomy pipeline is composed of: (1) a DNN that extracts depths from an input image; and (2) a DNN that inputs extracted depths and state information to output navigation motions.

A straightforward application of edge computing would require the vehicle to transmit the (compressed) image to the edge server, which then executes both the depth and navigation DNNs. Such an exchange imposes considerable channel usage, while exposing the control loop to latency variations due to erratic capacity patterns typical in airborne systems [10].

Herein, we propose an innovative architecture – *NaviSplit* – that uses split DNNs and supervised compression [11] to build a dynamic and efficient, distributed navigation framework. First, we create a "bottleneck" [12] within the depth DNN, where a lower-dimension tensor representation is trained to support the supervised task. This creates more robust representations than that of typical autoencoders, which are trained to simply reconstruct the input image. Second, the portion of the model up to the bottleneck (head model) is executed onboard the vehicle, and the rest of the model (tail model) is executed at the edge server. We note that our design is the

first example of supervised compression for depth estimation. Third, we train several split DNN models that range in computational complexities, compression rate, and depth accuracy – then encapsulate these models in a gated dynamic network framework. Finally, we train an auxiliary model to select the split DNN that minimizes channel usage while supporting navigation. The resulting architecture tunes the computing requirements, channel usage, and depth estimation accuracy, to the changing navigational needs of the vehicle.

The main contributions of this paper are as follows.

- We present *NaviSplit*, an adaptable multi-branched neural architecture with supervised compression. Our core innovations are: (a) a novel distributed neural architecture, and (b) a novel multi-stage training process that results in an auxiliary model that dynamically selects optimal compression factors.
- We implement *NaviSplit* on the robust simulator Microsoft AirSim [13], where the task is to navigate a drone to a target location while minimizing path length and avoiding collisions.
- We release our simulation tool, that interfaces with Microsoft AirSim, open-source to the public.

Our results demonstrate that the *NaviSplit* depth DNN achieves an extraction accuracy of 72-81% while transmitting an extremely small amount of data (between 1.2 and 18 KB) to the edge server. Compared to a static state-of-the-art (SoA) model, *NaviSplit* obtains a slightly higher mean navigation accuracy (82.5% versus 82.2%) with a mean reduction in 95% transmitted data (43 kilobyte/meter versus 2.1 KB/m).

II. Related Work

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 [8]. Processing typically consists of executing DNNs to extract features, semantics, and control. In computer vision applications such as [14], the mobile device can apply JPEG or MPEG compression to compress input images and reduce the amount of data.

Split computing (SC) [11], also known as split DNN and model partitioning, is a recent class of approaches in mobile computing. DNN architectures are partitioned into two sections – head and tail – that are executed by the mobile device and edge server, respectively. The objective is to balance computing load, energy consumption, and channel usage. While early approaches [15] simply "split" existing DNN models, recent methods involve injecting a "bottleneck" into a trained DNN task model. This alters and trains the DNN's layers to learn a compact set of task-specific features that preserves the task accuracy [12].

A recent approach allowed for dynamic quantization of intermediate activation values with fixed activation sizes at the split point [16]. However, the literature is sparse in approaches to dynamically control the size of the activation values as transmitted at the split point. Herein, we design a gated neural architecture that can dynamically select the compression model given perceived context. We remark how, to the best of our knowledge, this is the first exemplar of such construction, as well as the first application of split computing to both a depth estimation and navigation problem.

III. Methods

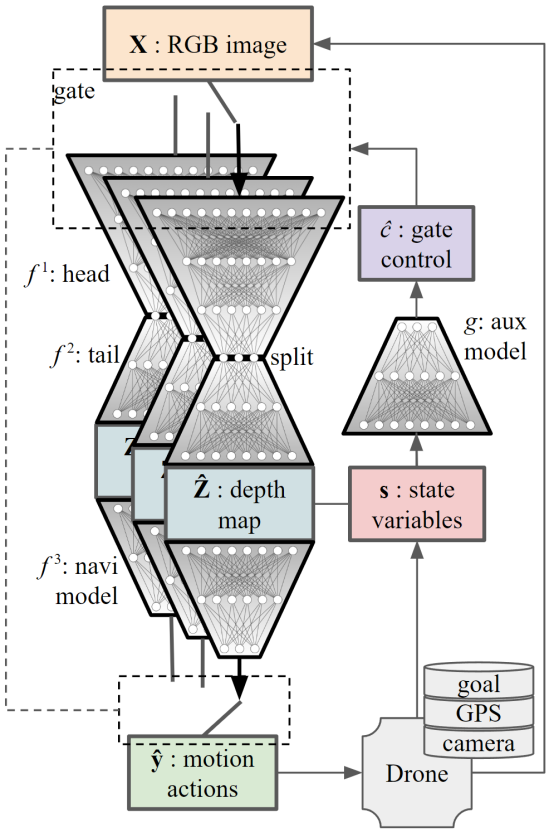

Fig. 1. The framework we propose uses a teacher model that maps a monocular RGB image to a depth map. Several different split points with encoder/decoder are injected into the teacher model to make multiple student model branches capable of split computing – of which an auxiliary model selects from given perceived context. Extracted depth maps are input to the navigation model that outputs motion actions used during drone navigation.

Figure 1 illustrates the system model for *NaviSplit*. We consider a system composed of: a lightweight UAV equipped with a monocular forward-facing RGB camera and small computing chip such as a Jetson Nano or Raspeberry Pi, and an edge server equipped with significantly more computing resources. The objective of the drone is to navigate to a target position in an unknown environment while minimizing path length and avoiding collisions. To this aim, the images captured by the onboard camera are first transformed into a depth map by a depth DNN. Second, the extracted depths are combined with state variables (*e.g.,* current and target positions) and then fed to a navigation DNN to produce motion commands. Given the limited resources available to the drone, we assume that it is either impossible or inefficient to execute the control pipeline at the drone, for instance due to memory constraints, insufficient energy availability, or excessive latency. Thus, we split the depth DNN into a head model that is computed onboard and a tail model that is computed at the edge server.

The most novel component of the presented system is the auxiliary model that controls the gate used to select from the multi-branch split DNN framework.

Edge computing: a viable option is for the drone to compress the images captured by the camera (e.g., using JPEG), that are then used by the edge server to execute the pipeline transforming images to motion commands. However, especially in systems with extreme resource constraints – e.g., a nano drone connected to a mobile base station, the wireless link connecting the drone to the edge server may have a severely constrained capacity, where the achievable data rate has an erratic pattern due to the motion characteristics of the drone.

A. NaviSplit Approach

We seek a methodology to reduce the amount of data transferred over the channel while preserving navigation performance. To this aim, *NaviSplit* develops a new generation of neural models combining split computing and supervised compression with a gated multi-branched model. The gate is driven by a specialized auxiliary module to select encoder/decoder pairs built using a supervised compression approach. The rationale is to select a compression strategy matching the needs of the controller, that is, capable of producing representations suitable to determine control given the operating context. There are five modules composing *NaviSplit*.

• **Sensing (Camera):** we collect sensor data (RGB images) that is sufficient to fulfil mission objectives as accomplished by the downstream task (navigation) model. We use the notation $\mathbf{X}$ to refer to an observation acquired from onboard sensor(s).

• **Depth Maps:** in our implementation, the acquired sensor data is transformed into an intermediate data structure taking the form of a 2D depth map, $\hat{\mathbf{Z}}$, which is a representation of the relative distances between the drone and nearby objects within the field of view of onboard sensors.

• **Task (Navigation):** sensor data and state variables, $\mathbf{s}$, are transformed into task output, $\hat{\mathbf{y}}$. In our implementation, $\hat{\mathbf{y}}$ contains motion actions for the drone to navigate between its current and target location, sensor data is transformed into $\hat{\mathbf{Z}}$, and $\mathbf{s}$ contains the current and target GPS positions.

• **Split Computing:** several sensing-depth-navigation student models are created which each use a different split computing design. This results in a spectrum of models to select from, with various encoded data sizes that are indexed by a gate control value, c. We remark how the drone only needs to store and execute (one per image) the head portion of the model (the encoder), which is built to be of minimal complexity.

• **Adaptation:** An auxiliary model, g_ϕ, is developed to intelligently select an encoder/decoder pair, in response to perceived context at the current time step. To the best of our knowledge, this is the first application of dynamic, adaptive split computing and is the core contribution of this paper.

We aim to solve the following optimization problem:

$$\begin{aligned} &\arg\min_{\phi} \ \langle \hat{c} \rangle \\ \text{s.t. } &\left\langle \eta\left(g_\phi, \{\mathbf{s}^{(t=0)}\}\right)\right\rangle \geq \beta * \left\langle \eta\left(c_{max}, \{\mathbf{s}^{(t=0)}\}\right)\right\rangle \end{aligned} \tag{1}$$

Quantity $\{\mathbf{s}^{(t=0)}\}$ denotes a set of training objectives defined by their initial state variables (starting and target positions). Each value of $\mathbf{s}^{(t=0)}$ will result in a path taken by the drone as decided by both the multi-branch navigation DNN, f, and auxiliary model, g_ϕ, such that one path will be executed using:

$$\begin{aligned} \hat{\mathbf{y}}^{(t)} &= f\left(\mathbf{X}^{(t)}, \mathbf{s}^{(t)}, \hat{c}^{(t)}\right) \\ \hat{c}^{(t)} &= g_\phi\left(\hat{\mathbf{Z}}^{(t-1)}, \mathbf{s}^{(t-1)}\right) \\ \hat{c}^{(t=0)} &= c_{max} \\ \hat{\mathbf{Z}}^{(t)} &= f_2\left(f_1\left(\mathbf{X}^{(t)}, \hat{c}^{(t)}\right), \hat{c}^{(t)}\right) \end{aligned} \tag{2}$$

where time steps will be computed until a termination criteria is met. Thus, $\langle \hat{c} \rangle$ denotes the expected value of the gate control – given that a lower indexed value of c correlates to a smaller data rate used in SC, and c_{max} is the maximum gate value corresponding to the $f_\theta \in f$ with the largest data rate. Quantity η is the task accuracy which is a function of the initial state variables and the auxiliary mechanism, where the auxiliary mechanism that always uses c_{max} corresponds to the SoA static case, and β is a scalar parameter set by the user. A smaller data rate should result in lower accuracy, and thus a typical parameter range is $0 < \beta \leq 1$. Specific to our implementation, we consider η to be the navigation accuracy (*i.e.*, percent of successful paths). Where success is measured by reaching the target position, while avoiding collisions, and within a maximum number of time steps.

The subsequent sections detail each step of a multi-stage procedure we use to train the several model parts of *NaviSlim*. We implement our approach on the robust drone simulator Microsoft AirSim [13], which is integrated with our open-source Python interface to train and evaluate models with[1].

B. Depth Model

Depth maps provide rich information used to calculate precise movements. Given a monocular RGB camera, typical in small drone applications, these depths can not be directly calculated and instead must be extracted. We use a convolutional neural network (CNN) to transform $\mathbf{X} \rightarrow \hat{\mathbf{Z}}$, by minimizing the expected L_1-loss extraction error:

$$\arg\min_{\theta} \ \langle L_1\left(\mathbf{Z}, f_{p,\theta}\left(\mathbf{X}\right)\right)\rangle \tag{3}$$

where we use $f_{p,\theta}$ to refer to the parent depth model – which will later be split into student models. The ground truth depth maps, $\mathbf{Z}$, are obtained directly from AirSim by creating datasets of known mappings $\mathbf{X} \rightarrow \mathbf{Z}$, including 4500 training images and 500 testing images. The shape of an RGB image is [3, 144, 256]; and the shape of a depth map is [144, 256].

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

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 [17], and a scaled exponential linear unit (SELU) [18] 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. We train each depth model using an Adam optimizer [19] and learning rate decay.

C. Split Computing

We evaluate two methods of injecting a compressed split point into $f_{p,\theta}$. The first is a baseline model that simply reduces the number of channels between two blocks around the split point. The second is a more advanced method similar to that presented in [12], which injects a bottleneck around the split point that changes the structure of multiple blocks.

For the baseline student models, we reduce the number of output channels in the second block of $f_{p,\theta}$ down from 128 to either 2, 4, 8, 16, or 32, which subsequently reduces the number of input channels in the third block. In the following discussion, $\mathbf{l}_t = \{1\} \cup \{4, ..., 11\}$, $\mathbf{l}'_t = \{3, ..., 11\}$, and $\mathbf{l}_h = \{2, 3\}$ – which denote sets of block indices.

For the bottleneck student models, we design two custom bottlenecks by creating encoder and decoder sections. We target the decoder's reconstruction to fit the fifth convolution block output from $f_{p,\theta}$. Within the bottleneck, we compress the height and width to a size of either 4x9 (which we refer to as Bottleneck_V1) or 8x14 (Bottleneck_V2). Further, we compress the channel values down from 64 to either 12, 24, or 48. The structure of the injected bottlenecks can be viewed as an entirely separate model replacing the first five convolution blocks of $f_{p,\theta}$. In the following discussion, $\mathbf{l}_t = \{6, ..., 11\}$, $\mathbf{l}'_t = \{5, ..., 11\}$, and $\mathbf{l}_h = \{1, ..., 5\}$.

Head Training. The trainable parameters for blocks on indices $\mathbf{l}_t$ (corresponding to the tail) are directly copied over from the teacher model, where as those for $\mathbf{l}_h$ (corresponding to the head) are randomly initialized. The trainable parameters for blocks at indices $\mathbf{l}_t$ are frozen, and our encoder/decoder is trained as follows. We use knowledge distillation [4] so that the error gradient is calculated with respect to the output of blocks at indices $\mathbf{l}'_t$ in the student model against those in $f_{p,\theta}$. We use an L_2-loss function applied to each layer, an Adam optimizer, learning rate decay, and early stopping [20]. The error gradient is propagated and used to update the trainable parameters from blocks at indices $\mathbf{l}_h$:

$$\nabla = \frac{\partial}{\partial \theta'} \Sigma \{ L_2 \left(f^{(i)}_{p,\theta} \left(\mathbf{X} \right), f^{(i)}_{s,\theta'} \left(\mathbf{X} \right) \right) \ \forall \ i \in \mathbf{l}'_t \} \tag{4}$$

where $f_{s,\theta'}$ is the student model, and $f_\cdot^{(i)}$ denotes the i^{th} block of $f_\cdot$ (either teacher or student model) from the set $\mathbf{l}'_t$.

Tail Training. The trainable parameters in the injected bottleneck, blocks at indices $\mathbf{l}_h$, are then frozen and those in the blocks at indices $\mathbf{l}_t$ are unfrozen for fine tuning as follows. We calculate the error gradient using an L_2-loss on the sum of two terms: the difference between ground truth task output (otherwise called a "hard" target) and the difference between the student and teacher output (otherwise called a "soft" target). The error gradient is propagated and used to update the trainable parameters from blocks at indices $\mathbf{l}_t$:

$$\nabla = \frac{\partial}{\partial \theta'} [L_2 \left(f_{p,\theta} \left(\mathbf{X} \right), f_{s,\theta'} \left(\mathbf{X} \right) \right) + L_2 \left(\mathbf{Z}, f_{s,\theta'} \left(\mathbf{X} \right) \right)] \tag{5}$$

To further reduce the size of encoded data (in kilobytes) before communicating with an edge server, we quantize the compressed encoding to an 8-bit unsigned integer tensor. This way, the memory transmitted is similar to that of JPEG compression of the input RGB image, where one of our 32-channel baseline student models would communicate a tensor of size 18.4 KB as compared to a JPEG compression with 95 quality that would communicate a mean size of 20.6 KB.

D. Navigation Model

The navigation model is tasked to transform $\hat{\mathbf{Z}}$ to $\mathbf{y}$. We aim to keep the size of the navigation model minimal, thus preprocess $\hat{\mathbf{Z}}$ by applying a min pooling layer that reduces the depth map to a size of 8x6, which is then flattened into a vector. We append the relative difference between current and target positions, such that $\mathbf{s} = [\Delta x, \Delta y, \Delta z, \Delta \text{yaw}]$. Data, $\hat{\mathbf{Z}}$ and $\mathbf{s}$, from the four most recent time steps are concatenated in temporal order. This results in a feature vector of length 208, which is fed into a MultiLayer Perceptron (MLP) with 3 layers of 32 Rectified Linear Unit (ReLU) [21] nodes each, followed by an output layer that is squashed with hyperbolic tangent. These output nodes control drone translation and rotation. The navigation neural network is then trained using a TD3 reinforcement learning algorithm [22], utilizing the StableBaselines-3 (SB3) [23] python library with Pytorch [24]. This approach is similar to SoA static approaches [25]–[28].

We use the following reward function:

$$r(t) = \begin{cases} -\alpha_1 & collision \\ \alpha_2 & goal \\ \alpha_3 tanh(d) - \alpha_4 & intermediate \end{cases} \tag{6}$$

where the α terms are positive constants set by the user. The first condition applies a large penalty for colliding with an object, and the second condition applies a large reward for reaching the target position. The third condition is used during intermediate time steps, and applies a reward for moving closer to the target position by using the relative distance, d, and applies a constant time penalty to encourage shorter paths. Further, we apply a curriculum learning schedule to incrementally increase the difficulty. One navigation model is trained for, and paired with, each student depth model.

E. Auxiliary Model

The objective of the auxiliary model, g_ϕ, is to select $f_\theta \in f$ by inferring the gate control value, $\hat{c}$. We use an MLP with 2 ReLU layers of 32 hidden nodes each. The output layer is a hyperbolic tangent node which outputs a normalized value of

$\hat{c}$. As input, we append the four most recent values of $\hat{c}$ to the same feature vector used as input into the navigation model.

We train the auxiliary network g_ϕ using similar methods as the navigation network – a TD3 reinforcement learning algorithm with a similar reward function as outlined in Equation 6 and curriculum learning schedule. Only, we add a penalty for high values of $\hat{c}$ to the intermediate condition, such as: $-\alpha_5\hat{c}$.

IV. Results

We evaluate the teacher depth model used to extract depths from a monocular RGB image. As expected, objects further away receive a higher extraction error. There is a linear increase in the root 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 2 compares the baseline student models, the bottleneck student models, and edge computing that completely offloads the image using JPEG compression. We see that the baseline models perform with equal memory consumption as both the bottleneck and JPEG models, however do not result in a lower error than JPEG. Alternatively, the bottleneck models perform with better error than that of the JPEG models - warranting the bottleneck methodology is more robust.

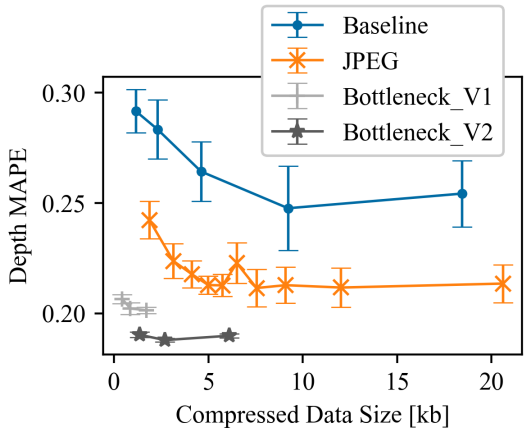

Fig. 2. Comparing various compressed data sizes, corresponding to different models, versus resulting depth extraction error. The markers from left to right: for bottleneck models, range between a reduction in channels of [12, 24, 64]; for baseline models, range between a reduction in channels of [2, 4, 8, 16, 32]; and for JPEG models, range in quality of compression from 5 to 95.

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. We assume this is due to some level of regularization due to the mapping of a 32-bit floating point number to an 8-bit unsigned integer, along with inherent clipping (eliminating possible outliers that may otherwise blow up towards infinity).

Using the student models to extract depth maps, we then train and evaluate each of the navigation models. For navigation accuracy, we use the percent of successful paths – where a successful path is one that reached the target position, without any collisions, and within a maximum number of computational time steps. We evaluate each navigation model against a static set of initial and target positions. We find that the navigation accuracy has no relationship with the distance between initial and target positions – as this is instead a function of the complexity of the path (number and size of objects in the way). However, the number of computational time steps required to execute a successful path has a linear relationship with initial distance – thus is a good normalization value for benchmarks. The auxiliary model is trained to select from these sensing-depth-navigation student model branches.

Table I 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. For the two normalized benchmarks, only the successful paths are considered because unsuccessful ones terminate either early from collision or late from the 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.

TABLE I

Model	Navigation [%]	Path [ts/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

Using the auxiliary model consistently results in a higher navigation accuracy as compared to using the student models independently, and in fact receives a 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.

To evaluate the behavior of *Navisplit*, we consider all successful evaluation paths flown while using the auxiliary model. The gate control index, c, for each student model increases in magnitude with increasing compressed data size. Figure 3 shows the mean gate control value predicted by the auxiliary model, $\hat{c}$, 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 around homes and other objects - warranting that a larger encoded data size is needed for more difficult scenarios.

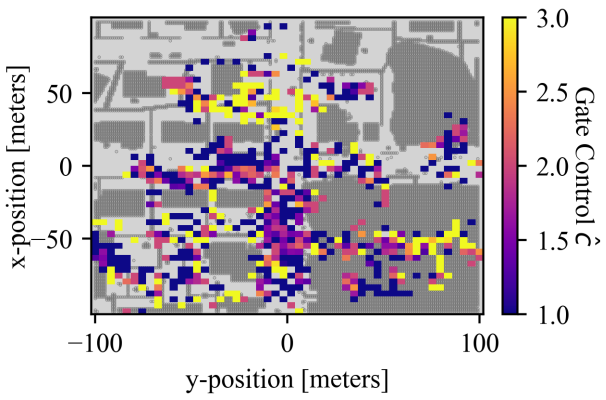

Fig. 3. Mean value of the gate control, $\hat{c}$, for all successful paths while using an auxiliary model to adapt and select $\hat{c}$ at each time step. This a demonstration of *NaviSplit*, which was trained and evaluated in Microsoft AirSim.

V. Conclusions

We have presented *NaviSplit* – an effective multi-branched neural network architecture for drone navigation, that dynamically splits computing between an equipped processing unit and edge server. *NaviSplit* slightly improved navigation accuracy by 0.3% over a larger SoA static model, while significantly reducing the communicated data rate by 95%. Further, our 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 we have a presented a dynamic multi-branch split DNN for efficient distributed autonomous navigation, along with accurate depth map estimation from a monocular camera.

Acknowledgment

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

References

[1] A. Pappalardo, Y. Umuroglu, M. Blott, J. Mitrevski, B. Hawks, N. Tran, V. Loncar, S. Summers, H. Borras, J. Muhizi, M. Trahms, S.-C. Hsu, S. Hauck, and J. Duarte, "Qonnx: Representing arbitrary-precision quantized neural networks," 2022.

[2] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," 2018.

[3] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," 2015.

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

[5] S. Ahn, S. X. Hu, A. Damianou, N. D. Lawrence, and Z. Dai, "Variational information distillation for knowledge transfer," 2019.

[6] A. Mishra and D. Marr, "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy," 2017.

[7] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: Hardware-aware neural network design," 2018.

[8] 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," 05 2016, pp. 1–5.

[9] 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)*, 2018, pp. 115–131.

[10] D. Callegaro, M. Levorato, and F. Restuccia, "Seremas: Self-resilient mobile autonomoussystems through predictive edge computing," *CoRR*, vol. abs/2105.15105, 2021. [Online]. Available: https://arxiv.org/abs/2105.15105

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

[12] 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)*. IEEE, 2022, pp. 337–346.

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

[14] 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*, vol. 10, pp. 91 253–91 262, 2022.

[15] 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*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 615–629. [Online]. Available: https://doi.org/10.1145/3037697.3037698

[16] 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.

[17] Y. Wu and K. He, "Group normalization," in *Computer Vision – ECCV 2018*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, pp. 3–19.

[18] 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*, 2017, pp. 972–981.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the trade*. Springer, 2002, pp. 55–69.

[21] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

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

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

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

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

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

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

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