Investigating Neural Network Architectures, Techniques, and Datasets for Autonomous Navigation in Simulation

Oliver Chang
Pomona College
Claremont, California, USA
eoca2018@mymail.pomona.edu

Christiana Marchese
Pomona College
Claremont, California, USA
cemb2020@mymail.pomona.edu

Jared Mejia
Pomona College
Claremont, California, USA
jared.mejia@pomona.edu

Anthony J. Clark
Pomona College
Claremont, California, USA
anthony.clark@pomona.edu

***Abstract*—Neural networks (NNs) are becoming an increasingly important part of mobile robot control systems. Compared with traditional methods, NNs (and other data-driven techniques) produce comparable—if not better—results while requiring less engineering knowhow. Training NNs, however, still requires exploration of a significant number of architectural, optimization, and evaluation options. In this study, we build a simulation environment, generate three image datasets using distinct techniques, train 652 models (including replicates) using a variety of architectures and paradigms (e.g., classification, regression, etc.), and evaluate the navigation ability of the model in simulation. Our goal is to explore a large number of model possibilities so that we can select the most promising for future study with a physical device. Training datasets leading to the best performing models were those that included a significant amount of noise from seemingly inefficient actions. The most promising models explicitly incorporated "memory" wherein previous actions were included as an input in the next step. Such models performed as good or better than conventional convolutional NNs, recurrent NNs, and custom architectures including two camera frames. Although trained models perform well in an environment matching the distribution of the training dataset, they fail when the simulation environment is altered in a seemingly insignificant manner. In robotics research it is often taken for granted that a model with good validation characteristics will perform well on the underlying task, but the results presented here show that there can often be a loose relationship between validation metrics and performance.**

***Index Terms*—robotics, neural networks, simulation, computer vision, navigation**

I. Introduction

With increasing frequency, neural networks (NNs) are being incorporated into the control software of mobile robots and providing state of the art results. Primarily, they are used to process camera input to improve navigation, build a map, or identify points of interest [1]. We are interested in how to best incorporate NNs into the decision-making processes of mobile robots. Specifically, we are concerned with how a robot can learn to make effective decisions with respect to its path and goals (e.g., continue on a walking path or cut across a grass covered field) while taking into account energy consumption and trade-offs in terrain difficulty. A NN can be used, for example, to identify how easy the terrain ahead is to navigate [2]. Before addressing this problem, however, we first need to select a NN paradigm (i.e., single image classification, recurrent model classification, regression, multi-image classification, reinforcement learning) and investigate training possibilities (i.e., different training dataset generation techniques and whether pre-training should be used). In this study, we explore the space of possible models by developing a simulation environment and examining a variety of commonly deployed methods. Our motivation is to discover general trends in NN variants so that we can choose an appropriate option to use on our physical device.

Though NNs are conceptually simple—they can learn a function that maps input(s) to output(s)—it is not always obvious how to produce an effective model. One must prepare a training dataset, decide how to arrange or process the input, and then select from among the numerous architectures and modern learning techniques. Although many existing articles compare architectures, our work extends beyond architectures alone and presents a holistic view including different data generation techniques, various methods for constructing network inputs, comparisons with sequence models, and metrics for evaluation.

We systematically explore different possibilities and provide recommendations for using a NN as a decision making component for autonomous navigation. Specifically, we compare (1) three methods for creating synthetic data, (2) the performance of 43 model types, (3) pretrained and non-pretrained models, and (4) the difference between validation accuracy and behavior. We operate under the assumptions that data and model design are more important than hyperparameter tuning, and that our results will carry over when we continue with our future work, which involves crossing the "reality gap" [3]. Tuning hyperparameters [4] will lead to improved results, but such a process is likely more sensitive to the exact composition of the training dataset and may not give insight into which model types are best suited to our problem. In this study, we instead focus on finding general trends with regards to datasets, architectures, and model paradigms, and not on finding the best possible model via hyperparameter tuning.

We built a simulation environment that enables us to effectively generate image datasets and train a variety of different model types. We examine the relationship between validation loss and a model's *in situ* performance in simulation. We then take our evaluation a step further by looking at resulting behaviors when the simulation environment is differently textured, which should represent a different distribution compared to the training dataset—similar to how a real-world dataset will be different than the training dataset. Additionally, we compare training from scratch and that of using transfer learning via pretrained models and fine tuning. Due to modern deep learning techniques, training from scratch may achieve comparable results to transfer learning even when training for only tens of epochs.

Our results indicate that synthetic datasets should be generated using techniques that imitate human-like behavior but with added perturbations. Although it is intuitive to remove inefficient actions from the dataset, doing so makes it difficult for the model to learn how to get back on course if it deviates from the ideal path. Additionally, the best performing models were simple classification networks and network architectures that included an explicit memory where the input to the model combined the current camera frame and the previous output action of the model. Trained models perform very well in an environment matching the distribution of the training dataset. They fail, however, when the simulation environment is updated with new textures even when they are similar in pattern to the originals. The main limitation of this work is that it relies exclusively on synthetic datasets, and the primary contribution is our wide exploration and comparison of NN models applied to a navigation task.

II. Related Work

NNs have gained traction in a variety of fields, and robotics is no exception. The high-dimensional state spaces of realistic, unstructured environments pose problems to traditional methods of control theory which often require near perfect knowledge of the environment and tend to experience scaling difficulty [1].

Though modern deep learning approaches provide clear benefits to robotics, NNs commonly experience issues with small datasets, slow convergence, and overfitting. In order to address these obstacles to learning, deep learning engineers are tasked with making key decisions in the design and methods they choose to utilize in their robotics pipelines. For this reason, we empirically compare a variety of NN architectures whose differences directly affect the aforementioned factors.

A. Visual Navigation

Most commonly, NNs applied to navigation in robotics are strictly vision-based systems due to the various limitations and problems that arise when working with other sensors. In particular, many mobile robots are limited in their carrying and battery capacities due to safety and hardware constraints [5]. Though cameras are limited in providing depth information, other sensors are often useless in certain domains, such as in the presence of shiny or translucent objects. Cameras tend to be cheaper, lighter, and as information-rich (if not more) as alternatives [6], which makes them a natural choice for many applications.

There are a variety of ways to pose the task of navigation for a robot, including methods to devise subgoals that encourage the attainment of the overarching goal. For instance, Richter and Roy [6] designed a system to determine the probability of a future collision given an image and some choice of action, and to recognize novel images (those disparate from the training set) so that a prior control policy could be utilized in unfamiliar circumstances. Similarly, Palossi et al. [7] utilize a system that predicts the steering angle and probability of collision for the robot.

Another approach to navigation and obstacle avoidance demonstrated by Kim et al [8] was to train a system that could predict the optimal linear and angular velocities of the robot's wheels given an input image. Both Bansal et al. [9] and Anjian et al. [10] take hybrid approaches that initially utilize NNs to predict waypoints along the proposed path to a goal, and then implements methods derived from classical feedback control in order to generate smooth trajectories through the waypoints. In our experiments, we compare several architectures which all output discrete directions for each input image—-rotate left, move forward, or rotate right.

B. Neural Network Architectures and Methods

Most articles comparing NN architectures tend to do so by computing metrics on benchmark training datasets. Our work focuses on convolutional NNs (CNNs) since they are the most successful in the domain of image processing. For example, Luo et al. [11] compared ResNet50, InceptionV3, Densenet121, SqueezeNet, MobileNetV2, and MnasNet architectures in terms of accuracy and complexity with the AloTBench framework. Similarly, Bianco et al. [12] use the ImageNet-1k challenge dataset to compare 40 architectures (including VGG, SqueezeNet, ResNet, Inception, DenseNet, ResNeXt, SE-ResNet, and SE-ResNeXt) in terms of accuracy, complexity, memory usage, and inference time. In contrast to this work, we generate three custom datasets and compare the performance of network architectures in terms of their ability to navigate the simulation environment.

In addition to the standard CNN architectures discussed above, we also note that recurrent NNs (RNNs), deep reinforcement learning (DRL), and regression models are often used for image processing and robot navigation. One type of RNN that works well on data with spatial locality (e.g., image data) is a Convolutional LSTM (ConvLSTM), as demonstrated by Shi et al. [13] in their application of ConvLSTM for forecast precipitation. As discussed by Zeng et al. [14], DRL methods have recently become a research hotspot for visual navigation. For a more complete comparison, we include ConvLSTM, DRL, and regression models in this study.

C. Simulation and Data Synthesis

Another important component of NN-based navigation is generating or synthesizing a dataset. Previous approaches to data generation have formulated ideal data generation policies as those which either minimize the distance between the distributions of the simulated and the real data, or maximize the accuracy on the validation task [15]. Richter and Roy [6] develop a SLAM system to generate a geometric map and traverse the environment, acquiring robot configurations within the map along with a corresponding image for each configuration; for each image-configuration pair, many randomly selected actions are examined for safety. In contrast, Kim et al. [8] employed a more laborious approach to data generation, in which humans manually controlled a robot using a joystick while collecting images with the linear and angular velocity of the robot taken as the corresponding label for each image. We create a similar dataset in simulation (our "handmade" dataset—discussed in the next section) and compare it with other approaches. In the learning and optimal control hybrid approaches demonstrated in [9], [16], an obstacle map and goal area is used along with model predictive control (MPC) in order to obtain the optimal waypoint for the robot to navigate towards, which is then used as the ground-truth label for the corresponding tuple containing an image, the linear speed, and the angular speed of the robot.

In prior work, we used ROS and Gazebo [17] to train a model responsible for choosing paths over rough terrain [2]. Another recent tool for data collection is CARLA [18], which uses the Unreal Engine (a video game engine) for rendering a virtual environment. One downside shared by these approaches is amount of time needed to collect data. In this study, we build a fast simulator to test one kind of data synthesis mechanism for imitation learning (creating our "handmade" dataset), similar to the approach in [8], as well as two automated methods (creating our "uniform" and "wandering" datasets) more comparable with the approaches in [6], [9], [16], in which simulation environment information is used to compute intermediary values that determine labels for the training data. Collecting images using our new simulator is roughly 200 times faster than our work with Gazebo—it takes only around 15 minutes to generate $100\,000$ images (we discuss simulation details in the following section), and the process can be parallelized to further improve speed.

For any particular use case, it is *a priori* unclear which of the three components (model architecture, training paradigm, or data generation) can be thought of as the most important factors in the success of the system.

III. Methods

Here we discuss the simulation environment, architectures, training and evaluation methods.

A. Simulation Environment and Data Collection

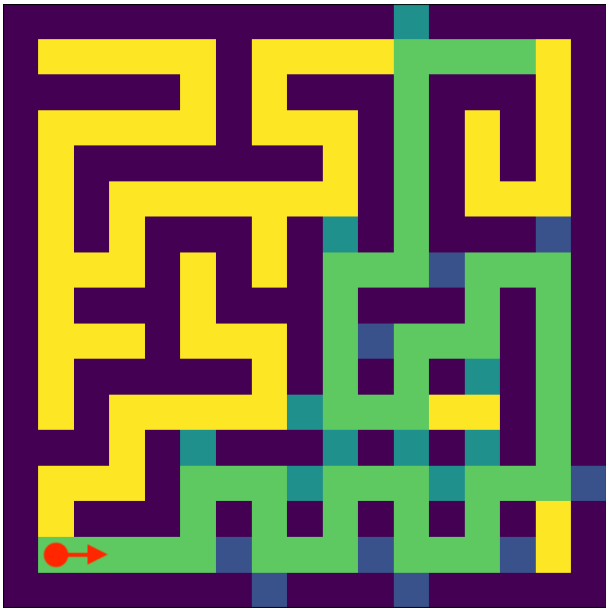
One goal of this work is to compare different methods for generating synthetic data. We built a simulation environment comprising a map generator and a raycasting renderer. The map generator uses the growing tree algorithm [19] to generate maze-like corridors (i.e., no open spaces) with beginning and end positions (see Fig. 1a). Our renderer takes a generate corridor map and produces a virtual environment (see Figs. 1b and 1c) in which an agent can move around and save images. Generating a single frame and saving it to a PNG file takes roughly 10 ms on our server (Intel Xeon E5 CPU; 256 GB RAM). Simulation environments have customizable map sizes, start and goal positions, camera fields of view, camera heights, camera pitches, output image sizes, and surface textures. Code for the simulator and NNs can be found here: https://github.com/anthonyjclark/raycasting-simulation.

As we are interested in image processing and not in mapping or exploration, we label walls with the correct turn in the form of an arrow. The agent can always discern the correct turn by processing the camera frames. For example, the arrow at the end of the corridor in Fig. 1b indicates that an agent should turn left when it reaches the upcoming turn. Therefore, navigating the map is done by processing camera frames and acting accordingly.

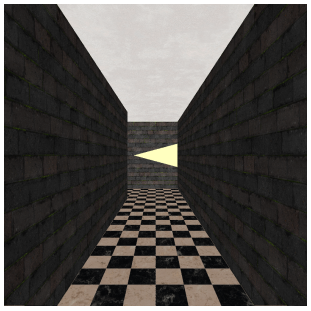
We generated 20 training maps, and from these maps we created three datasets using our simulator:

1) **handmade** ($92\,324$ images): we created this dataset by manually navigating each training map three times and capturing an image after each action (i.e., moving forward or rotating left or right);
2) **uniform** ($100\,000$ images): we created this dataset by positioning the camera uniformly at random along the correct path (shown in green in Fig. 1a), perturbing both the side-to-side position and heading in the corridor, and then automatically computing a "correct" target angle and action; and
3) **wandering** ($113\,702$ images): we created this final dataset by automatically navigating the map from beginning to end; at each step we compute the "correct" target angle and action, and then randomly perturb the position and heading prior to taking the next step.

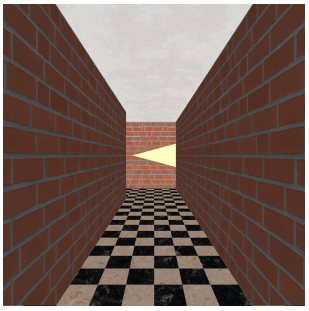
For the two automatically generated datasets (uniform and wandering), we compute an exact angle that the agent should take to reach the next turn at each step. For example, if the agent should move forward down a corridor, then the computed angle is $0°$, if the agent is facing directly toward a turn arrow while occupying a turning cell, then the computed angle is $\pm\ 90°$, and if the agent is approaching a left turn but is currently facing right of center, then the computed angle will be greater than $90°$. The computed angle is used to label each image in these two datasets, which enables us to create regression based models in addition to classification models. Specifically, a network can predict the correct angle of a turn rather than just selecting a discrete action such as *left*, *forward*, or *right*. For the uniform dataset we highlight that, unlike for the others, there is no connection between any two images captured sequentially. Thus, we cannot use the uniform dataset when we are training a sequential model such as an RNN.



(a) Map with Directions



(b) Rendered Frame (Default)



(c) Rendered Frame (Alternate)

Fig. 1. (a) A randomly generated map; colors indicate the agent's position and direction (red), walls (dark), open corridors (yellow), the correct path (green), and walls textured with turning arrows (blue). (b) A rendered frame as seen from the perspective of the agent. (c) The same frame with an alternate texture used during evaluation.

B. Network Architectures

We trained models using seven paradigms: (1) standard CNN classification, (2) stacked input CNN classification, (3) paneled input CNN classification, (4) hybrid image+command classification, (5) RNN classification, (6) DRL classification, and (7) CNN regression, For training models, we use the fastai library [20]. Results from each model paradigm are used to inform subsequent models. For example, we use our results from CNN classification to narrow down the number of models explored when training hybrid models.

For our initial exploration of CNN architectures, we compared the 24 networks found in Table I, including ResNets/X-ResNets/X-SE-ResNets/X-ResNeXts/X-SE-ResNeXts [20], [21], Squeezenets [22], DenseNets [23], VGGs [24], and AlexNet [25]. In the table, model names ending with an "a," "b," or "c" indicate slightly modified variants of the model as specified in the fastai documentation, and any high performance models are shown in bold, where high performance refers to a model's ability to navigate on average 90% through the validation mazes before getting turned around and being unable to find the end goal. We chose to bold all such models in the table as it draws attention to performance differences among columns, and because we are not able to display standard deviations (only averages) for replicates. All classification networks include three output corresponding to the agent actions *left*, *forward*, and *right*.

A common failing of classifiers used in navigation is that they are susceptible to becoming stuck in a loop—particularly when using a deterministic simulation. For example, a model might direct an agent to turn right and then turn left (or vice versa), which results in the agent returning to its previous state and then looping through the same sequence of states. This happens because they do not maintain a memory of past inputs or actions. One method for preventing this is to add a supervisor that disallows such action sequences; another is to use a model that directly maintains some form of memory. We employ both methods here. Specifically, we create custom models that take as input the previous action or input (stacked, paneled, and hybrid models) and an RNN.

Stacked and paneled models are given as input two sequential camera frames combined into single tensors of shape $height \times width \times 6$ (two RGB images stacked by channel) and $2height \times width \times 3$ (two RGB images paneled vertically), respectively. Effectively, by processing two consecutive camera frames at a time, these models should prevent the agent from returning to a previous state and getting stuck in a loop.

Similarly, hybrid models take as input the current image frame and the previous command as a single real-value number (0, 1, and 2 representing *left*, *forward*, and *right*, respectively). The command value is concatenated with the output of the convolutional layers of the model. This is a common method for providing the model with a simple one state memory [26].

The final models include an RNN, two DRL, and four regression models. RNNs maintain an internal memory, unlike stacked, paneled, and hybrid models that required a manually created memory in the form of a different input. Here we are using a ConvLSTM [13], which is specifically created to operate on image inputs. Similar to the work by Justesen et al. [27], we trained models using A2C and PP0 in our procedurally generated environments. Our implementations are from the Stable Baselines3 library [28]. Finally, regression models are identical to the simple classifiers except that these models output an angle in the range $\pm$ 180°, which is then interpreted as a discrete action by the simulation environment.

C. Training and Evaluation

All classification models are trained separately on each of the three datasets; both with and without pretraining; using 8 epochs, 5% of the dataset for validation, and default hyperparameters as specified by *fastai* [20]; and with four replicate models for each configuration. Following the work completed by Müller and Koltun [26], we also explored different loss functions, but these did not result in a change in performance. In total, we train 576 standard classification

models (24 architectures $\times$ 3 datasets $\times$ 2 pretraining options $\times$ 4 replicates); 16 each of the stacked, paneled, and hybrid models (4 architectures $\times$ 4 replicates); 4 RNN models; 8 DRL models (2 methods $\times$ 4 replicates); and 16 regression models (4 architectures $\times$ 4 replicates). All models are trained using a server with 4 NVIDIA Tesla V100 GPUs, each with 32 GB video memory.

Each of these models were evaluated on 20 validation maps, which were not seen during training. A simulation supervisor terminates model evaluation on a single map if the agent gets turned around or stuck in place. Finally, the best performing models were evaluated on 4 alternate validation maps in which the wall textures were altered (see Fig. 1c).

IV. Experiments and Discussion

A. CNN Classification

We start by comparing the performances of 24 classification models (each with and without pretraining) on our three synthetic datasets; Table I shows these results. The table provides the number of parameters (**#**, in megabytes), the average time to execute one epoch (**T**, in minutes and seconds), the validation accuracy (**Valid**, as a percentage), and simulation performance (**Perf**, as an average percentage of the map that was navigated from beginning to end).

Unexpectedly, the handmade dataset leads to drastically poorer performing models (as indicated by values in the Perf columns). Our process for creating this dataset is typical of many imitation learning experiments, but the dataset does not include enough off-path examples to enable a model to learn how to get back on course when it travels off what a human would consider the "ideal" path. The uniform and wandering datasets include examples in which the agent is facing off target or is positioned near the walls—such examples were missing from the handmade dataset.

Models are trained to a high validation accuracy (nearly all above 90%) for all three datasets. However, there is virtually no correlation between accuracy and performance when the table is taken as a whole. The uniform dataset is relatively easy to learn compared to the wandering dataset (signified by the respective validation accuracies), but this does not lead to better performance. In fact, pretrained models are overfit to the uniform dataset (higher validation accuracy is correlated with lower performance), but not to the wandering dataset. Additionally, while pretrained models are better on average for the uniform dataset, scratch models (those without pretraining) are better for wandering.

Based on these results, we select the wandering dataset and the following pretrained architectures for further investigation: XResNeXt18, XResNeXt50, AlexNet, DenseNet121. The XResNeXt models were chosen for their high performance and so that we can explore the differences in model depth when models have similar employed techniques. AlexNet and DenseNet121 were chosen so that we can verify if the relative strengths of these networks remain the same when we change the input or output formats. Specifically, AlexNet has fairly average performance across the datasets, and DenseNet121 performs poorly.

Analyzing model performance on each map suggests that some maps are quite easy to navigate (nearly every model reaches the end goal), whereas other maps are troublesome for a large number of models. Examining maps more closely shows that a rapid succession of turns is the main cause of difficulty. Models were found to frequently get turned around in the same locations of the more difficult maps. Though some maps require more steps to navigate (as indicated by the number of steps required by a human), this does not always lead to poorer model performance. For example, one of the shortest maps was the most difficult and one of the longest maps was navigated by the majority of models.

B. Stacked and Paneled Input

As described in Section III, standard classifiers can become stuck in a "loop." One method for enabling a model to automatically handle this problem is to provide it with information about its past state. Here, we do so by passing stacked and paneled images to the CNN classifiers. Table II shows the performance of these models.

Despite having the same input available as the standard classification models, stacked and paneled models trained on the wandering dataset lead to mediocre results. For our simulation environment, the additional input information (i.e., the previous camera frame) increases training difficulty without any benefit. It may be that training these models for additional epochs leads to performance comparable to the standard classification models, but it is not worth the additional training cost when the standard models perform well.

C. Hybrid Image+Command Input

Similar to the stacked and paneled models, the hybrid input models include information about the previous time step. Hybrid models, however, do so by explicitly including the previous output of a model as an input during the next step. Compared with the corresponding standard classification models, hybrid models improve validation accuracy and performance on average from 91.6 to 92.3 and from 82.5 to 95.3, respectively. The change is particularly dramatic in the performance of the XResNeXt models, which increase by nearly 23.1%. We also note that the while the hybrid models perform better, they have lower validation accuracy when compared to the stacked and paneled models.

D. Recurrent Neural Network

Another method for combining previous inputs and outputs is to use sequence models such as RNNs. Unlike the Hybrid Image+Command architecture, RNNs maintain their own internal memory and do not need a custom input format. Moreover, RNNs like the LSTM "learn" how long they need to remember information. They can theoretically retain historical information for much longer than our stacked, paneled, and hybrid models.

We chose the ConvLSTM architecture, and through experimentation we found that 15 hidden channels and 6 hidden

TABLE I
CLASSIFICATION VALIDATION ACCURACY AND SIMULATION PERFORMANCE

			Handmade				Uniform				Wandering			
			Pretrained		Scratch		Pretrained		Scratch		Pretrained		Scratch	
Model	**#**	**T**	**Valid**	**Perf**	**Valid**	**Perf**	**Valid**	**Perf**	**Valid**	**Perf**	**Valid**	**Perf**	**Valid**	**Perf**
AlexNet	2.6	00:31	91.7	14.1	92.0	22.9	99.6	83.4	99.2	81.7	92.7	**93.4**	92.5	**92.6**
DenseNet121	7.6	06:38	91.5	16.9	91.0	27.6	99.5	63.7	99.3	80.8	91.7	87.5	92.5	**91.8**
DenseNet201	19.1	10:50	91.3	12.2	92.2	28.7	99.6	81.3	99.6	69.7	92.1	**93.0**	92.4	74.9
ResNet18	11.2	01:16	69.1	16.4	92.1	26.4	74.6	76.7	99.4	**91.4**	69.1	**93.6**	93.0	**95.4**
ResNet50	24.4	11:52	91.1	11.9	90.9	21.9	99.5	82.8	99.3	88.2	91.7	**91.9**	92.9	**94.2**
SqueezeNet1-1	1.2	01:00	91.5	10.2	91.5	15.1	99.3	86.8	99.2	86.7	92.3	85.5	92.0	**92.5**
VGG11-BN	9.3	04:09	91.6	9.9	91.9	16.4	99.2	89.9	99.5	76.2	92.3	**96.4**	92.2	**91.8**
VGG19-BN	19.6	09:15	91.4	10.6	91.4	18.8	99.6	83.1	99.4	87.0	91.4	**92.7**	92.8	**94.5**
X-ResNet18a	11.2	02:08	90.5	17.2	92.3	16.1	99.2	75.1	99.6	70.3	91.5	**92.2**	92.4	**96.7**
X-ResNet18b	13.9	03:31	90.6	24.1	91.8	19.5	98.8	77.8	99.6	52.1	91.4	82.1	92.4	**96.8**
X-ResNet18c	10.5	03:37	91.3	27.1	90.6	26.3	99.0	87.3	99.7	72.6	91.4	85.3	92.1	**96.0**
X-ResNet50a	24.4	11:48	90.6	10.2	91.6	21.2	99.3	74.7	99.5	81.2	91.8	**91.7**	92.7	**96.0**
X-ResNet50b	27.8	12:06	90.7	12.1	90.9	18.4	98.9	78.4	99.6	75.1	91.7	88.7	92.7	**96.3**
X-ResNet50c	30.0	10:31	91.4	12.3	91.6	30.0	98.9	68.7	99.5	73.5	91.6	**90.3**	92.8	**96.6**
XResNeXt18a	12.9	04:47	90.7	28.9	91.6	13.4	98.2	**95.9**	99.5	77.8	90.6	74.6	92.6	**96.5**
XResNeXt50a	23.9	07:01	90.5	17.6	92.2	22.3	98.9	**90.7**	99.5	81.4	91.5	72.7	92.3	**98.0**
X-SE-ResNet18	11.3	04:03	90.5	17.0	92.1	17.2	98.0	**91.8**	99.4	77.8	90.9	68.2	93.1	**95.1**
X-SE-ResNet50	121.0	13:36	90.6	22.3	91.8	24.8	98.7	76.5	99.4	76.0	91.9	78.6	92.7	**97.5**
X-SE-ResNeXt18a	13.0	05:09	90.7	25.6	91.7	19.5	98.1	**93.0**	99.5	75.2	91.1	63.8	92.4	**94.8**
X-SE-ResNeXt18b	15.3	05:23	90.4	30.9	91.4	25.9	98.4	87.8	99.7	68.5	90.1	64.2	92.7	**96.6**
X-SE-ResNeXt18c	11.8	04:53	90.2	37.1	91.2	23.8	98.4	**90.8**	99.6	75.1	91.0	68.3	93.3	**96.7**
X-SE-ResNeXt50a	26.4	07:55	90.4	14.2	91.6	17.2	98.6	**91.9**	99.4	71.1	90.6	74.7	92.7	**97.2**
X-SE-ResNeXt50b	30.3	08:12	90.7	19.0	91.7	20.2	98.5	56.1	99.6	80.9	91.8	71.4	91.9	**98.0**
X-SE-ResNeXt50c	26.7	07:32	91.4	22.6	91.7	30.0	98.7	80.5	99.5	78.9	91.9	67.9	92.4	**96.8**

layers were ideal on our navigation task given performance and GPU memory constraints. Table II shows that this type of model can achieve similar performance to that of a feed-forward classification model. However, the ConvLSTM required significant hyperparameter tuning and longer training times before we were able to achieve this result.

E. Deep Reinforcement Learning

DRL models were trained using the PPO and A2C methods with approximately the same number of frames and network updates as the classification models. Table II shows that these models performed quite poorly. In fact, based on our early work with the simulation environment, the DRL models perform only slightly better than a random agent. (DRL methods do not perform validation in the same manner as the supervised learning classification models, so the Valid column is marked as "N/A" in the table.) Models trained for much longer were able to achieve better results (nearing 30% map completion), but for a fair comparison we limited the number of frames seen during training. Regardless, we aim to focus on generalizable vision models, and our initial experimental results with DRL models suggest that they would require much more hyperparameter tuning and would be very specific to a particular environment.

F. CNN Regression

Our final model type was identical to the standard CNN classification networks in all but one aspect: the output was a turn angle instead of a predicted action. The wandering and uniform datasets are labeled with the rotation angle needed to make the agent face the correct heading. For all other networks, this angle was simplified into a discrete action (left, forward, right); for our regression networks, we use the angle as the target output of the network. Table II indicates that while this approach produces decent results, they fall short of hybrid models.

G. Evaluate With New Textures

Finally, we altered the simulation environment by changing wall textures (see Fig. 1c) and then reevaluated four of the top performing models: XResNeXt18a, XResNeXt50a, ConvLSTM, Image+Command (with a XResNeXt50a as the CNN component). Each of these models were able to navigate on average 15% of the distance through the validation mazes. While this indicates that they were able to navigate a turn or two, the performance is clearly much worse than when they operate in the unaltered simulation environment. This result is, in hindsight, unsurprising as the training dataset does not include any variation in textures. To improve performance,

TABLE II
ADVANCED MODEL SIMULATION PERFORMANCE

Model	Valid	Perf
Stacked		
AlexNet	95.8	81.8
DenseNet121	95.3	86.0
XResNeXt18a	95.8	66.0
XResNeXt50a	95.0	79.6
Paneled		
AlexNet	96.0	60.0
DenseNet121	95.5	30.8
XResNeXt18a	92.6	60.9
XResNeXt50a	93.6	39.4
Image+Command		
AlexNet	92.2	**94.3**
DenseNet121	92.4	**93.3**
XResNeXt18a	92.4	**96.3**
XResNeXt50a	92.3	**97.1**
RNN		
ConvLSTM	90.8	87.6
RL		
A2C	N/A	7.7
PPO	N/A	7.6
Regression		
AlexNet	N/A	86.2
DenseNet121	N/A	53.7
XResNeXt18a	N/A	83.0
XResNeXt50a	N/A	**91.9**

models must be explicitly trained to handle such changes to the environment—similar to how they need to able to correct their course if they wander off the ideal path.

V. CONCLUSION

Our goal for this work was to discover training regimes and model architectures that are worth pursuing when using a physical device. Our experiments indicate that, similar to our wandering dataset, we should incorporate both human-like behavior and a significant amount of noise in our training dataset. This is in contrast to our uniform dataset, which was not generated by traversing a map, but rather was generated by randomly selecting locations in the map and computing the corresponding correct action. Moreover, based on our model validation experiments, we believe that the Image+Command hybrid models are most likely to provide generally good performance—they are inexpensive to train when compared to the RNN models, but they still include a form of state memory. The exact underlying CNN architecture is less important, and we should consider secondary aspects such as the training time and the time to process a single input during inference. Examining our results more closely, we also infer that training models from scratch may lead to better results for our particular navigation task, and that it is not important to use models with the largest number of parameters.

Our future work involves three key additions to this study. First, we will test our simulation trained models on a real-world device. Second, we will explore methods to cross the gap between simulation and reality such that our physical device can still benefit from simulation-trained models. Finally, we will incorporate hyperparameter tuning.

ACKNOWLEDGMENT

We gratefully thank Evelyn Hasama, Sean O'Connor, Kevin Ayala, Alex Fay, and Gabriel Konar-Steenberg for their thoughts and feedback, the NVIDIA Corporation for their donation of a Quadro P6000 GPU, and Pomona College for supporting this research.

REFERENCES

[1] J. Shabbir and T. Anwer, "A Survey of Deep Learning Techniques for Mobile Robot Applications," *arXiv:1803.07608 [cs]*, Mar. 2018, arXiv: 1803.07608. [Online]. Available: http://arxiv.org/abs/1803.07608

[2] A. J. Clark, J. Simpson, and J. Hall, "Comparing CNN Inputs for Terrain Classification using Simulation," in *IEEE Transdisciplinary AI*, Laguna Hills, California, USA, Sep. 2019.

[3] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, "Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks," *arXiv:1812.07252 [cs]*, Jul. 2019.

[4] P. Chhikara, R. Tekchandani, N. Kumar, V. Chamola, and M. Guizani, "DCNN-GA: A Deep Neural Net Architecture for Navigation of UAV in Indoor Environment," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4448–4460, Mar. 2021.

[5] D. K. Kim and T. Chen, "Deep Neural Network for Real-Time Autonomous Indoor Navigation," *arXiv:1511.04668 [cs]*, Nov. 2015, arXiv: 1511.04668. [Online]. Available: http://arxiv.org/abs/1511.04668

[6] C. Richter and N. Roy, "Safe Visual Navigation Via Deep Learning and Novelty Detection," *Robotics: Science and Systems*, Jul. 2017.

[7] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "A 64-mw DNN-Based Visual Navigation Engine for Autonomous Nano-Drones," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8357–8371, Oct. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8715489/

[8] Y.-H. Kim, J.-I. Jang, and S. Yun, "End-to-End Deep Learning For Autonomous Navigation of Mobile Robot," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*. Las Vegas, NV: IEEE, Jan. 2018, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/8326229/

[9] S. Bansal, V. Tolani, S. Gupta, J. Malik, and C. Tomlin, "Combining Optimal Control and Learning for Visual Navigation in Novel Environments," in *Conference on Robot Learning*. PMLR, May 2020, pp. 420–429. [Online]. Available: http://proceedings.mlr.press/v100/bansal20a.html

[10] A. Li, S. Bansal, G. Giovanis, V. Tolani, C. Tomlin, and M. Chen, "Generating robust supervision for learning-based visual navigation using hamilton-jacobi reachability," in *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, ser. Proceedings of Machine Learning Research, vol. 120. The Cloud: PMLR, 10–11 Jun 2020, pp. 500–510. [Online]. Available: http://proceedings.mlr.press/v120/li20a.html

[11] C. Luo, X. He, J. Zhan, L. Wang, W. Gao, and J. Dai, "Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices," *arXiv:2005.05085 [cs, eess]*, May 2020, arXiv: 2005.05085. [Online]. Available: http://arxiv.org/abs/2005.05085

[12] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark Analysis of Representative Deep Neural Network Architectures," *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8506339/

[13] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong, and W.-c. Woo, "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting," *arXiv:1506.04214 [cs]*, Sep. 2015.

[14] F. Zeng, C. Wang, and S. S. Ge, "A Survey on Visual Navigation for Artificial Agents With Deep Reinforcement Learning," *IEEE Access*, vol. 8, pp. 135 426–135 442, 2020.

[15] H. S. Behl, A. G. Baydin, R. Gal, P. H. S. Torr, and V. Vineet, "Autosimulate: (Quickly) Learning Synthetic Data Generation," in *Computer Vision – ECCV 2020*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 255–271.

[16] A. Li, S. Bansal, G. Giovanis, V. Tolani, C. Tomlin, and M. Chen, "Generating Robust Supervision for Learning-Based Visual Navigation Using Hamilton-Jacobi Reachability," in *Learning for Dynamics and Control*. PMLR, Jul. 2020, pp. 500–510.

[17] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3. Sendai, Japan: IEEE, 2004, pp. 2149–2154.

[18] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: an Open Urban Driving Simulator," *arXiv:1711.03938 [cs]*, Nov. 2017, arXiv: 1711.03938. [Online]. Available: http://arxiv.org/abs/1711.03938

[19] J. Buck and J. Carter, *Mazes for Programmers: Code Your Own Twisty Little Passages*. Dallas, Texas: The Pragmatic Bookshelf, 2015.

[20] J. Howard and S. Gugger, "Fastai: A Layered API for Deep Learning," *Information*, vol. 11, no. 2, p. 108, Feb. 2020.

[21] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of Tricks for Image Classification with Convolutional Neural Networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, Jun. 2019, pp. 558–567.

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5MB model size," *arXiv:1602.07360 [cs]*, Nov. 2016, arXiv: 1602.07360. [Online]. Available: http://arxiv.org/abs/1602.07360

[23] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," *arXiv:1608.06993 [cs]*, Jan. 2018, arXiv: 1608.06993. [Online]. Available: http://arxiv.org/abs/1608.06993

[24] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, Apr. 2015.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, Red Hook, NY, USA, Dec. 2012, pp. 1097–1105.

[26] M. Müller and V. Koltun, "OpenBot: Turning Smartphones into Robots," *arXiv:2008.10631 [cs]*, Mar. 2021.

[27] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation," *arXiv:1806.10729 [cs, stat]*, Nov. 2018.

[28] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.