# Minimalistic Neural Network Architectures for Safe Navigation of Small Mobile Robots

Xinyue Kan and Konstantinos Karydis

*Abstract*— **Small mobile robots can be useful in dangerous and high-risk applications such as disaster response. To this end, small robots must be capable of autonomous navigation. One way to perform autonomous navigation is via learning perception-action cycles. The ability to learn perception-action cycles may enable computationally– and data-efficient ways to transfer navigation policies between robots, and to generalize across operating environments. Learning perception-action cycles currently relies on deep networks. Such networks, however, may not be directly applicable to small robots due to the latter's constrained sensing and computing capacity. To mitigate this challenge, we identify minimalistic neural network architectures to approximate an obstacle prediction function using a robot's observation and action history. We propose a new learning-based algorithm for small robot navigation in partially-known, partially-observable environments. The performance of the algorithm and its ability to generalize are evaluated in simple and complex environments of varying size.**

## I. INTRODUCTION

Small ($< 2$ kg weight, $< 30$ cm largest body dimension) robots enjoy several benefits when compared to their larger counterparts. They have higher agility and maneuverability, and are thus more capable and safer to operate in cluttered environments, even around people. They can be comparably fast and inexpensive to manufacture thus allowing deployment in large numbers. As such, small robots offer promise in various dangerous and high-risk real-world applications such as disaster response and search-and-rescue.

The efficacy of small robots in real-world applications directly depends on their capacity to navigate autonomously. If a map of the environment is known, then autonomous navigation requires integration of planning, control, and estimation. For instance, small unmanned aerial vehicles (UAVs) can perform aerial acrobatics and bio-inspired maneuvers like perching and grasping [1]–[5]. Small UAVs and ground robots can assemble 3D structures [6]–[10], and small legged robots can navigate in complex environments [11]–[13].

However, in many real-world applications we seldom have a detailed map of the operating environment; only partial information about the environment may be available. For example, in the aftermath of a building fire, one can use prior information from the building's blueprints regarding its dimensions and structural elements, but the internal structure conditions are unknown (e.g., floors may have collapsed). To navigate in such partially-known environments, the components of planning, control, and estimation must be complemented with mapping. Simultaneous Localization and Mapping (SLAM) algorithms enable a robot to construct a map of its environment and localize itself within the constructed map. SLAM is a mature field in robotics applications [14]. Yet, most existing algorithms primarily apply to larger robots which are typically equipped with all the necessary sensing and computing hardware to gather and process sufficiently information-rich data in real-time. One exception is OrthoSLAM [15] which is a lightweight SLAM algorithm that can run efficiently on computationally-constrained robots. While OrthoSLAM is particularly suited to indoor and structured environments, it may not perform well in more general environments where the 2.5D and obstacle orthogonality conditions are not met.

Contrary to their larger counterparts, small robots often *lack both the sensing and computing capacity* to run SLAM algorithms for autonomous navigation in partially-known environments. First, they have limited payload capacity to carry for example a LIDAR sensor for mapping. Further, added payload sharply reduces a small robot's operational time. This fact is particularly problematic in small UAVs [16].

A different autonomous navigation paradigm that may circumvent the constrained sensing and computing capacity of small robots hinges on designing navigation policies based on *perception-action cycles* [17]. The robot first perceives its environment to infer the local structure. This can be achieved via lightweight, low power consumption sensors like cameras or infrared (IR) distance sensors. Then the robot acts—i.e. plans its motion—based on the locally-inferred environment, and the cycle continues. The coupling of perception and action, however, can be challenging since the robot needs to keep track how its actions change its sensory input while the same actions direct the acquisition of sensory data [17]. Allowing the robot to learn perception-action cycles instead of hand-engineering desired responses may help address this challenge. Further, learning-based navigation policies may transfer well across different operating environments with minimal modifications while allowing for some online adaptation to improve the robot's performance.

Learning perception-action cycles has been significantly enhanced by recent progress in deep reinforcement learning (RL). For example, A3C [18] and ICM [19] connect planning algorithms with deep networks, and achieve promising performance with visual observations in games. However, both A3C and ICM employ a complex architecture that makes training and prediction time extremely long. Adapted from DNC [20], MACN [21] follows a hierarchical global-

We gratefully acknowledge the support of NSF under grant # IIS-1724341. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

The authors are with the Dept. of Electrical and Computer Engineering, University of California, Riverside. Email: {xkan001, karydis}@ucr.edu

local process, and simplifies the network structure with an external memory module. MACN takes laser data as input and converts the local observation to a partial blueprint which needs three convolution layers for feature extraction. Purely RL methods have been used for navigation in unknown environments (e.g., [22], [23]). However, they often require a map for learning optimal policies, may not incorporate obstacle avoidance, or, if so, utilize rewards (e.g., distance to obstacles) that may cause chattering behaviors in complex environments. Existing approaches to learn perception-action cycles utilize very large networks. For applications with small robots, shallow networks may instead be more useful so that the forward pass evaluation and possible refinement can run under constrained computing capacity. Shallow networks also allow us to recover some guarantees for learning-based systems [24]–[26]. Such guarantees are very important for meaningfully bringing learning into robotics applications.

We note here that employing shallow networks for small robot navigation has been used in the past. Notably, evolving neurocontrollers [27] can be combined with feed-forward and recurrent neural networks to accomplish collision-free corridor following by predicting sensory input [28]. However, these approaches are only tested in simple worlds with a single isolated obstacle and appear to not generalize to unseen environments. As we will show in the following, we follow a systematic approach to identify simple yet useful shallow network architectures, while our proposed approach works and generalizes well to large, complex environments.

### *Contributions of this Paper*

Motivated by the appealing properties and potential of shallow networks, the contribution of this paper is twofold.

**Identification of minimalistic neural network architectures:** We investigate various shallow network architectures and evaluate their capacity to approximate a function for obstacle prediction based on limited history of the robot's observations and actions. We draw similarities and differences between tested architectures and further test those that perform on par with much more complex structures.

**Development of a new learning-based navigation algorithm for robots with constrained sensing and computing capacity:** We propose a navigation algorithm in partially-known, partially-observable environments that combines Temporal-Difference learning with Long-Short Term Memory (LSTM) networks. We demonstrate the generalization capability of the proposed algorithm by testing with various map sizes not presented during training.

## II. Technical Preliminaries

### A. *Reinforcement Learning*

Robot navigation in partially-known environments can be cast as a reinforcement learning problem. Let $q_i = [x_i, y_i]^\mathsf{T} \in \mathbb{R}^2$ denote the robot state at step $i$; $q_0$ and $q_f$ denote the initial and final robot states, respectively. At each step, the robot executes an action $a_i \in A_i$, and receives a reward from the environment. Reaching a desired final state can be viewed as maximizing the total reward. The rewards received after step $i$ form a sequence $R_{i+1}, R_{i+2}, \ldots, R_f$, where index $f$ denotes the terminal state. The return from state $i$ can be expressed as the cumulative discounted reward $G_i = \sum_{k=1}^{f} \gamma^{k-1} R_{i+k}$, where the discount rate $\gamma$ lies in the $[0,1]$ interval. The action-value function for a policy $\pi$ is the expected return $Q(q,a) = \mathbb{E}[G_i | q_i = q, a_i = a]$. The optimal policy thus is to select the sequence of actions that maximize $Q(q,a)$, that is

$$a^* = \arg\max_a Q(q,a) = \arg\max_a \mathbb{E}[G_i | q_i = q, a_i = a] \ .$$

However, the lack of a model for the environment challenges the selection of future rewards and terminal state.

Our work builds upon Temporal-Difference (TD) Learning [29] to obtain estimates of $G_i$. We choose TD Learning as the underlying basis due to its appropriateness in model-free prediction problem. *One-step* TD updates its $G_i$ estimate using only the next observed reward $R_{i+1}$ (instead of waiting until the end of the episode). The standard target of *one-step* TD update is $G_i = \sum_{k=1}^{\infty} \gamma^{k-1} R_{i+k} = R_{i+1} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{i+1+k} = R_{i+1} + \gamma Q(q_{i+1}, a_{i+1})$, when the terminal state $f$ is unknown. However, *one-step* TD methods lack optimality guarantees since estimated rewards after step $i+1$ may be inaccurate.

*N-step* TD methods would instead utilize $n$ number of rewards, $1 < n < f$. The *error reduction property* guarantees that the expectation of *n-step* reward is a better estimate than *one-step* methods. In practice, however, the *n-step* return cannot be used prior to observing $R_{i+n}$ [29]. To obtain $R_{i+n}$, n-step forward sampling of states, actions, and rewards from actual or simulated interaction with an environment is needed. Yet, large values of $n$ may increase the computational complexity to a level that makes real-time execution on a small robot impossible.

In Section III, we propose a method to efficiently predict up to $R_{i+2}$. The proposed method offers trade-offs between the computational complexity associated with $n > 1$ and estimation accuracy of $G_i$. The TD target can then be expressed as $R_{i+1} + \gamma R_{i+2} + \gamma^2 Q(q_{i+2}, a_{i+2})$.

### B. *Environment Maps*

In this work we consider environments that can be sufficiently represented via a grid-based map $\mathcal{M} \subset \mathbb{R}^2$. Each cell $c$ of the map is square with unit length sides, and is uniquely described by the coordinates $(x, y)$ of each center point. Let $\mathcal{M}_{obs}$ and $\mathcal{M}_{free}$ denote the obstacle-occupied and free subspaces of the map, respectively. We require i) $\mathcal{M}_{obs}$ to contain no $2 \times 2$ blocks (i.e. only "walls" are allowed), and ii) $\mathcal{M}_{free}$ to be path-connected (i.e. there is at least one path between any two cells of the obstacle-free subspace). Finally, we consider two types of maps: easy and difficult ones (Fig. 1). A map is classified as difficult when there exist several dead-ends and multiple intersections.

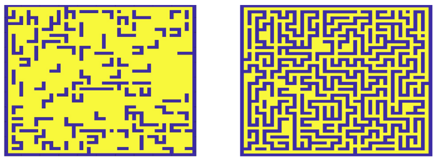


Fig. 1: Sample maps of size of $51 \times 51$. (a) An easy map has no or very few and short dead-ends. (b) A difficult map includes multiple intersections leading to dead-ends after several steps.

## III. Learning-based Navigation

We first formulate the small robot navigation task, and then describe the network architectures we use to predict the presence of obstacles. Then, we propose a new algorithm that combines policy iteration and the considered network structures to achieve small robot navigation tasks.

### A. Problem Formulation

Consider a robot randomly placed at an initial state $q_0 = [x_0, y_0]^\mathsf{T}$. The robot can move in the horizontal and vertical directions (but not diagonally), and is equipped with four distance sensors (e.g., IR sensors) placed at its front, rear, left, and right sides, respectively. The sensors can detect an obstacle in a direction $d \in D = \{East, South, West, North\}$, and have unit length range.[1] Define an action of the robot as moving one unit distance in a direction $d$. The task is to reach some final state $q_f = [x_f, y_f]^\mathsf{T}$ within map $\mathcal{M}$ with a minimum amount of actions while avoiding an unknown number of obstacles encountered along the way.

We group the sensor readings at state $q_i$ in a $4 \times 1$ vector $O(q_i)$ and denote its $d^{\text{th}}$ row as $O(q_i, d) \in \{0, 1\}$, where $0$ and $1$ indicate respectively the presence and absence of an obstacle in direction $d$. As an example, in Fig. 2(a), the sensor report is $O(q_i) = [0, 1, 1, 0]^T$ at $(x_i, y_i)$.

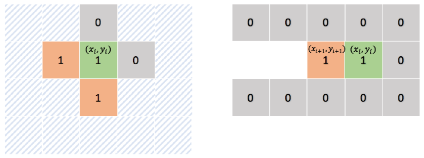


Fig. 2: A sensor report (left), and a dead-end scenario (right).

The action space $A_i$ contains valid (i.e. collision-free) actions for the robot at step $i$. The cardinality of this set is $0 \leq |A_i| \leq 4$. A map cell is then considered inaccessible if it is occupied by an obstacle, or—more importantly—it is associated with cardinality $|A_i| \leq 1$. *The latter is important because it corresponds to dead-end scenarios, and gives us the mechanism to overcome them.* To see this, consider Fig. 2(b). At step $i$ the robot is at cell with coordinates $(x_i, y_i)$ and reports $Q(q_i) = [0, 0, 1, 0]^\mathsf{T}$. Then, $|A_i| = 1$ so $(x_i, y_i)$ will be marked as inaccessible. At step $i + 1$, the robot moves to $(x_{i+1}, y_{i+1})$ and reports $Q(q_{i+1}) = [1, 0, 1, 0]^\mathsf{T}$. However, $(x_i, y_i)$ is already marked as inaccessible. This leads to $|A_{i+1}| = 1$, thus marking $(x_{i+1}, y_{i+1})$ as inaccessible too. By following this approach iteratively, we make sure that the robot will not enter a dead-end again, but instead take another action in the first intersection prior to that dead-end.

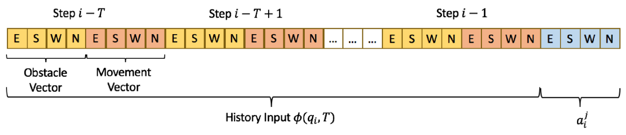
[1]Examples of small robots of this nature include miniature quadrotors and wheeled robots equipped with omni wheels.

### B. Neural Networks for Obstacle Prediction

As discussed in Section II, we train a neural network to approximate a function $f(\cdot)$ that estimates $R_{i+2}$. Given a history of observations, $f(\cdot)$ learns the patterns of a map, and at step $i$ it predicts the next observation the robot is going to make for any valid action it may take. *We seek to determine the simplest network architecture that performs accurate prediction.* While a deep and complex network can achieve better performance, simple (shallow) architectures have important advantages in small robot navigation:

1) The required training time increases with the network size [30]. Using a simple architecture, the training time can remain relatively short.
2) Small robots have limited memory. A simple architecture ensures that the number of network parameters that need to be saved in memory remains manageable.
3) Small robots have limited operational time. The time (and the associated power consumption) for prediction can be low for simple architectures.

Let $\hat{O}(q_{i+1}) = f(\phi(q_i, T), a_i; w_f)$ be the obstacle prediction function. $\phi(q_i, T)$ is a feature encoding function that captures a sliding window of robot actions and observations history of length $T$. $w_f$ represents the weights of the network. $\hat{O}(q_{i+1})$ takes the form of a quadruple that contains the probabilities that $q^d_{i+2} = (x^d_{i+2}, y^d_{i+2})$ is inaccessible in directions $d \in D$ after executing $a_i$. The output ground truth is the accessibility of $q^d_{i+2}$ given the map. We seek to minimize the difference between the predicted and actual obstacles. A unit input to the network is shown in Fig. 3.[2]



Step $i-T$
Step $i-T+1$
Step $i-1$
E S W N E S W N E S W N E S W N ... ... ... E S W N E S W N E S W N
Obstacle Vector
Movement Vector
History Input $\phi(q_i, T)$
$a_i^j$

Fig. 3: The structure of the input given to our network.

We consider two types of networks shown in Fig. 4. A ReLU is used when appropriate to avoid the vanishing gradient problem [31], while a sigmoid unit ensures the networks' outputs follow a Bernoulli distribution [32]. Dropout layers are used between hidden layers to avoid overfitting [33].

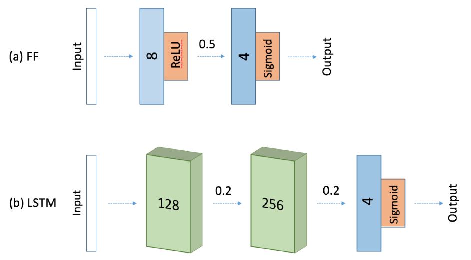


Fig. 4: We consider (a) one 2-layer FF network, and (b) four LSTM networks—one with two layers and three with a single layer.

[2]Note that we only mark a cell as inaccessible based on the robot observation and not the output of the prediction function.

*1) Feedforward (FF) Network:* An FF network with a single hidden layer containing a sufficient number of nonlinear units can approximate any continuous function [34]. Since the output of an FF depends only on the current input and not on any past or future inputs, FF networks are more suitable for pattern classification than for sequence prediction [35]. However, here we consider an FF network because the target function $f(\cdot)$ has $2^4$ possible outputs and can thus be cast as a 16-class classification problem. The structure of our FF network is shown in Fig. 4(a). Its input is an $8T \times 1$ vector. Two fully connected layers have 8 and 4 hidden units, respectively. ReLU and Sigmoid activation layers are added.

*2) Long Short-Term Memory (LSTM) Network:* An FF network can only map from input to output vectors, whereas an LSTM network [36] can in principle map from the entire history of previous inputs to each output [35]. LSTM networks work well in sequential prediction problems such as audio [37] and video data [38] analysis, learning context from languages [39], acoustic modeling of speech [40]. In robot navigation, the LSTM can help making predictions in the presence of long corridors and wide free space.

Our main LSTM network is shown in Fig. 4(b). It consists of two LSTM layers, two dropout layers and one fully-connected layer with a sigmoid activation function. LSTM layers have 128 and 256 memory units. The input size is $T \times 8 + 4$: an $8 \times 1$ vector for each step in the sliding window, and a $4 \times 1$ movement vector. To identify the simplest architecture, we also consider three cases of single-layer LSTM networks with 128, 256, 512 hidden neurons.

### C. Proposed Algorithm

We are now ready to introduce our learning-based algorithm for small robot navigation in partially-known, partially-observable environments. In our algorithm, no prior knowledge of the map is required. We do not save the entire observed map, but only record a sliding window length of observation and action history for prediction purposes.

The algorithm consists of two steps: i) policy evaluation with TD learning, and ii) $\epsilon-$greedy policy improvement. In TD learning, the robot updates its estimation after observing reward $R$. Since an explicit model of the environment is unavailable, $R_{i+n}$, where $n \geq 2$ is difficult to know. In the proposed algorithm, the estimate of $R_{i+2}$ is obtained by the obstacle prediction function described above.

*1) Policy Evaluation:* Let index $j \in \{1, 2, 3, 4\}$ denote a candidate action from the action space $A_i$ at step $i$, for any accessible cell. Then for each candidate action $a_i^j \in A_i$,

$$\mathbb{E}[G_i^j | q_i, a_i^j] = R_{i+1}^j(q_i, a_i^j) + \gamma R_{i+2}^j(q_i, a_i^j) \ .$$

Higher order terms $\sum_{k=3}^{\infty} \gamma^{k-1} R_{i+k}^j$ eventually influence the return very little and are thus neglected. From current state $q_i$, the robot executes an action $a_i^j$ and reaches state $q_{i+1}^j$.

The one step ahead reward $R_{i+1}^j$ is the sum of two parts: i) instant reward, and ii) visit reward. The instant reward $R_{ins,i+1}^j(q_i, a_i^j)$ is given based on the distance to the goal.[3]

[3]This requires only a map outline to pinpoint starting and target locations.

It is positive when the robot is getting closer to the ground and negative when getting away from it. A very large positive reward is given when the target is reached. It is given by

$$R_{ins,i+1}^j(q_i, a_i^j) = \begin{cases} 1, & \text{if } \|q_f - q_{i+1}^j\| < \|q_f - q_i\|, \\ 100, & \text{if } \|q_f - q_{i+1}^j\| = 0, \\ -1, & \text{otherwise.} \end{cases}$$

Let $n_{i+1,m}^j$ represent how many times the robot has been in state $q_{i+1}^j$ in the past $m$ time steps—$m$ is a small number compared to map size. The visit reward is non-positive and penalizes chattering behaviors that often occur in learning-based robot navigation. Note that this penalty is not added when the robot backtracks from a dead-end. Its form is

$$R_{visit,i+1}^j(q_i, a_i^j) = -max(n_{i+1,m}^j - 1, 0) \ .$$

The total one step forward reward $R_{i+1}^j(q_i, a_i^j)$ then is

$$R_{i+1}^j(q_i, a_i^j) = c_1 R_{ins,i+1}^j(q_i, a_i^j) + c_2 R_{visit,i+1}^j(q_i, a_i^j) \ .$$

where constants $c_1$ and $c_2$ weigh the contribution of instant and visit rewards respectively, and are found empirically.

We also seek to estimate $R_{i+2}^j$ for each candidate action $a_i^j \in A_i$. To reach a particular state $q_{i+2}^d$ at step $i+2$, an action $a_{i+1}^d$ has to be executed. If position $(x_{i+2}^d, y_{i+2}^d)$ is accessible, $a_{i+1}^d$ is a feasible action and then $a_{i+1}^d \in A_{i+1}$. The obstacle reward is defined as

$$R_{obs,i+2}^d = \begin{cases} -1, & \text{if } \ a_{i+1}^d \notin A_{i+1} \\ 0, & \text{otherwise.} \end{cases}$$

An example is shown in Fig. 5. At step $i$ the robot takes an action $a_i$ to reach state $q_{i+1}$. The action space at step $i+1$ is $A_{i+1} = \{a_{i+1}^W, a_{i+1}^N\}$. To reach a particular state $q_{i+2}^E$, the robot needs to execute $a_{i+1}^E$, however $a_{i+1}^E \notin A_{i+1}$, so $R_{obs,i+2}^E = -1$. To reach state $q_{i+2}^W$, the robot needs to execute $a_{i+1}^W$. Now $a_{i+1}^W \in A_{i+1}$, so $R_{obs,i+2}^W = 0$.

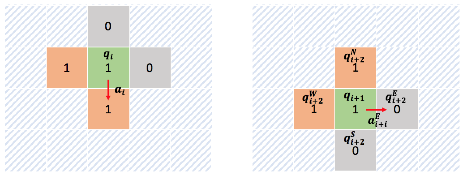


Fig. 5: Actions and observations at steps $i$ (left) and $i+1$ (right).

The obstacle reward determines the two step ahead reward:

$$R_{i+2}^j(q_i, a_i^j) = \sum_{d \in D} R_{obs,i+2}^d * P(d; q_i, a_i^j) \ ,$$

where $P(d; q_i, a_i^j)$ is the probability that $q_{i+2}^d$ is inaccessible after executing $a_i^j$. $P(d; q_i, a_i^j)$ is the output of the prediction function approximated by our neural network. At least one feasible $q_{i+2}^d$ exists, which is same position as $q_i$, $-3 \leq R_{i+2}^j \leq 0$. $R_{i+2}^j = -3$ suggests that the robot will encounter a dead-end if it executes $a_i^j$. With $R_{i+2}^j = -3$ the robot learns to turn around before reaching the dead-end position.

In all, the action that maximizes the expected reward is

$$a_i^* = \arg\max_j \left( R_{i+1}^j(q_i, a_i^j) + \gamma R_{i+2}^j(q_i, a_i^j) \right) \ .$$

*2) Policy Improvement:* Selected $a_i^*$ may not be optimal since the prediction function less than 100% accuracy, and there is insufficient prior knowledge on unseen map areas. Another candidate action may eventually yield a greater total reward than the greedy action [29]. To encourage exploration, the action chosen at each step is either $a_i^*$ with probability $(1-\epsilon)$, or a random action with probability $\epsilon > 0$.

*3) Algorithm:* We propose the robot navigation algorithm 1. In each trial, initial and target states are arbitrary. $N_{navi}$ is the maximum allowed steps for one navigation task. Information used during the process includes the robot observation history of maximum length $T$, a list of inaccessible state $L$ of maximum length $L_{max}$, and target state.

**Algorithm 1** Navigation in partially observable Environment

```
1: Select $q_0$, $q_f$, $T$, $L_{max}$, $\epsilon$
2: initialize $L = \{\ \}$, $W = \{\ \}$, $i \leftarrow 1$
3: get sensor report $O(q_0)$ at initial state
4: action space $A_0 \leftarrow \bigcup_{d \in D} \{a_0^d | O(q_0, d) = 1\}$
5: action $a_0 \leftarrow a_{random} \in A_0$ towards direction $d_0$
6: next state $q^i \leftarrow (q^i | q_0, a_0)$
7: $W \leftarrow W \cup \{O(q_0), a_0\}$
8: repeat
9:   obtain sensor report $O(q_i)$
10:  $A_i \leftarrow \bigcup_{d \in D} \{a_i^d | O(q_i, d) = 1\} \cap \{a_i^d | (q^{i+1} | q_i, a_i^d) \notin L\}$
11:  if $|A_i| = 1$ then
12:    $L \leftarrow L \cup q_i$
13:    $a_i^* \leftarrow a_i \in A_i$
14:  else
15:    for each $a_i^j \in A_i$ do
16:      if $i < T$ then
17:        $R^j(q_i, a_i^j) \leftarrow R_{i+1}^j(q_i, a_i^j)$
18:      else
19:        compute $R_{i+2}^j(q_i, a_i^j)$ with $f(W, a_i^j; w_f)$
20:        $R^j(q_i, a_i^j) \leftarrow R_{i+1}^j(q_i, a_i^j) + \gamma R_{i+2}^j(q_i, a_i^j)$
21:    $a_i^* \leftarrow \arg\max_j R^j(q_i, a_i^j)$
22:  $x \leftarrow X \sim \mathcal{U}(0, 1)$
23:  if $x \le \epsilon$ then
24:    $a_i \leftarrow a_i^*$
25:  else
26:    $a_i \leftarrow a_{random} \in A_i$
27:  $q^{i+1} \leftarrow (q^{i+1} | q_i, a_i)$
28:  $W \leftarrow W \cup \{O(q_i), a_i\}$
29:  $i \leftarrow i + 1$
30: until $i = N_{navi} - 1$ OR $q_i = q_f$
```

## IV. Simulation and Results

The proposed algorithm is tested in simulation. Our implementation is based on Keras with Tensorflow back-end. We use NVIDIA TITAN X GPU and Intel Xeon CPU E5-2603. Obstacles are placed randomly in maps of size $h \times w$. Initial and final states $q_0$ and $q_f$ are picked randomly, with at least one feasible path between them. We set $N_{random} = 512$ and $N_{navi} = h * w * 2$, and use the following hyper parameters in training: batch size 128, optimizer RMSProp [41], initial learning rate 0.001, decay rate 0, samples per epoch 500. The number of epochs, representing one forward pass and one backward pass of all the training examples, is variable.

### *A. Function Approximation*

Algorithm 2 generates data for training, validation, and testing. In each step $i$, we save $O(q_i)$ and $a_i$ as a vector $v_i$. The training input, output, and label are $[v_{i-T+1} \cdots v_{i-1}\ a_i]$, $\hat{O}_{q_i}$, and $O_{q_i}$, respectively. The LSTM defines a distribution $P(d; q_i, a_i, w_f)$; we use binary cross-entropy between training data and model's predictions for the cost function [32].

**Algorithm 2** Data Generator for training Neural Network

```
1: Select $h$, $w$, generate random map, set $q_0$, $q_f$, $N_{random}$.
2: $i \leftarrow 1$
3: get sensor report $O(q_0)$ at initial state
4: action space $A_0 \leftarrow \bigcup_{d \in D} \{a_0^d | O(q_0, d) = 1\}$
5: first action $a_0 \leftarrow a_{random} \in A_0$ towards direction $d_0$
6: next state $q^i \leftarrow (q^i | q_0, a_0)$
7: repeat
8:   obtain sensor report $O(q_i)$
9:   action space $A_i \leftarrow \bigcup_{d \in D} \{a_i^d | O(q_i, d) = 1\}$
10:  if $a_i^{d_{i-1}} \in A_i$ then
11:    $a_i \leftarrow a_i^{d_{i-1}}$
12:  else
13:    $a_i \leftarrow a_{random} \in A_i$ towards direction $d_i$
14:  next state $q^{i+1} \leftarrow (q^{i+1} | q_i, a_i)$
15:  $i \leftarrow i + 1$
16: until $i = N_{random} - 1$ OR $q_i = q_f$
```

The testing accuracy of the FF network is listed in Table I. Prediction accuracies for each direction are consistently around 0.5 for all tested map sizes. The sliding window length has no significant impact on the FF network. Since the accuracy is low, we confirm that a shallow FF network does not appear to be appropriate for this type of applications.

TABLE I: Testing Accuracy for FF network

| Map Size | T | Accuracy East | Accuracy South | Accuracy West | Accuracy North |
|---|---|---|---|---|---|
| ALL | 20 | 0.501 | 0.502 | 0.501 | 0.501 |

We consider four LSTM networks: one 2-layer with 128 and 256 hidden neurons, and three 1-layer with $512, 256$, and 128 neurons, respectively. We present in Table II results on training and validation accuracy from $51 \times 51$ easy maps trained over 125 epochs. All but the 1-layer LSTM with 128 neurons structures give over 90% testing accuracy. The training time per epoch is around 40s for all 1-layer networks irrespectively, and doubles for the 2-layer network.

TABLE II: LSTM testing accuracy on a $51 \times 51$ easy map

| Structure | Accuracy East | Accuracy South | Accuracy West | Accuracy North | time per epoch |
|---|---|---|---|---|---|
| 2 layers | 0.907 | 0.906 | 0.907 | 0.906 | 75s |
| 1-512 layer | 0.904 | 0.902 | 0.905 | 0.900 | 40s |
| 1-256 layer | 0.902 | 0.903 | 0.903 | 0.901 | 40s |
| 1-128 layer | 0.897 | 0.898 | 0.899 | 0.896 | 40s |

TABLE III: Testing Accuracy and time required for training

| | | 2-layer LSTM | | | | | 1-layer LSTM with 256 neurons | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Map Size | Accuracy East | Accuracy South | Accuracy West | Accuracy North | time cost per epoch | Accuracy East | Accuracy South | Accuracy West | Accuracy North | time cost per epoch |
| Easy | $11 \times 11$ | 0.928 | 0.931 | 0.932 | 0.932 | 50s | 0.925 | 0.929 | 0.929 | 0.932 | 30s |
| | $31 \times 31$ | 0.919 | 0.918 | 0.920 | 0.929 | 70s | 0.915 | 0.913 | 0.917 | 0.917 | 40s |
| | $51 \times 51$ | 0.904 | 0.904 | 0.906 | 0.902 | 70s | 0.902 | 0.903 | 0.906 | 0.901 | 40s |
| Diffi-cult | $11 \times 11$ | 0.919 | 0.918 | 0.919 | 0.918 | 50s | 0.918 | 0.917 | 0.917 | 0.917 | 30s |
| | $31 \times 31$ | 0.914 | 0.910 | 0.916 | 0.912 | 70s | 0.912 | 0.910 | 0.913 | 0.914 | 40s |
| | $51 \times 51$ | 0.906 | 0.908 | 0.907 | 0.905 | 70s | 0.908 | 0.907 | 0.903 | 0.907 | 40s |

Figure 6 shows results on training and validation accuracy. The 2-layer LSTM, and the 1-layer LSTM with 256 and 512 neurons achieve similar final accuracy. Also, the network converges faster as the number of neurons increases. The 2-layer LSTM achieves the best overall performance, while the 1-layer LSTM with 256 neurons can get similar testing accuracy with less network parameters and training time.

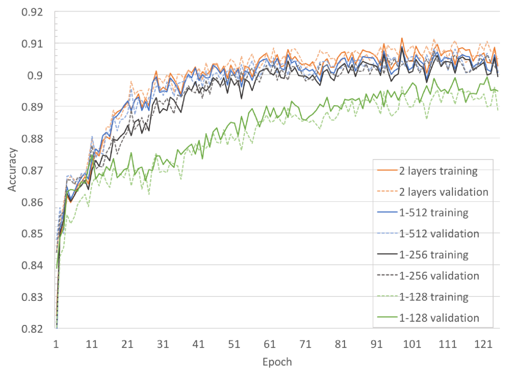


Fig. 6: Training and validation accuracy of different LSTM-based architectures in $51 \times 51$ easy map (cf. Table II).

Figure 7 depicts the training and validation accuracy of the 1-layer LSTM with 256 neurons, and the 2-layer LSTM, respectively in difficult maps. The 2-layer LSTM converges faster. However, since the time per epoch is higher for the 2-layer LSTM, the actual training time between the two architectures remains similar. Table III gives a more detailed picture of the testing accuracy for both architectures in maps of varying size and difficulty. Overall, the 2-layer LSTM requires more training time per epoch. Both structures consistently give us over 90% prediction accuracy.

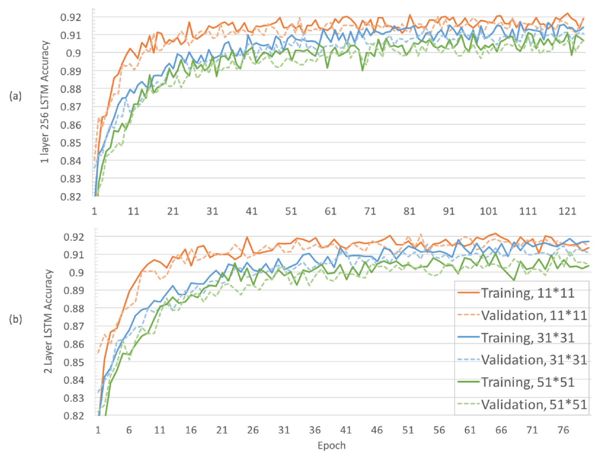


Fig. 7: (a) 1-layer (256) (b) 2-layer LSTM training and validation accuracy in difficult map of varying size.

We also tested $T = 40$ for 2-layer LSTM in $51 \times 51$ maps. Testing accuracy is similar ([0.907, 0.905, 0.908, 0.908]) with longer training time. Thus, we use $T = 20$ for $11 \times 11$ maps, and $T = 30$ for $31 \times 31$ and $51 \times 51$ in both LSTMs.

TABLE IV: The nine conditions considered in simulation

| Ind | Map Size (train) | Map Size (Sim) | $T$ | Ind | Map Size (Train) | Map Size (Sim) | $T$ |
|---|---|---|---|---|---|---|---|
| 1 | $11 \times 11$ | $11 \times 11$ | 20 | 6 | $31 \times 31$ | $51 \times 51$ | 30 |
| 2 | $11 \times 11$ | $31 \times 31$ | 20 | 7 | $51 \times 51$ | $11 \times 11$ | 30 |
| 3 | $11 \times 11$ | $51 \times 51$ | 20 | 8 | $51 \times 51$ | $31 \times 31$ | 30 |
| 4 | $31 \times 31$ | $11 \times 11$ | 30 | 9 | $51 \times 51$ | $51 \times 51$ | 30 |
| 5 | $31 \times 31$ | $31 \times 31$ | 30 | | | | |

## B. Simulation for Navigation Task

We evaluate the performance of the proposed algorithm in simulation. Following Algorithm 2, nine conditions are considered for each architecture; these are shown in Table IV. For each condition, we run the simulation for $N = 30$ times. In each trial, $q_0$ and $q_f$ are chosen randomly. To quantify efficiency, we compare robot movement steps for reaching the target with the number of $A^*$ expand nodes [42]; the $A^*$ algorithm is guaranteed to expand fewer nodes than another search algorithm with the same heuristic if the heuristic is admissible [43]. We use the Euclidean distance in $A^*$.

TABLE V: Fixed map, 1-layer and 2-layer

| 1-layer 256 | | | | |
|---|---|---|---|---|
| | Easy | | Difficult | |
| Index | success | $\leq A^*$ | success | $\leq A^*$ |
| 1 | 30/30 | 19/30 | 20/30 | 2/27 |
| 2 | 26/30 | 22/26 | 25/30 | 12/25 |
| 3 | 29/30 | 24/29 | 24/30 | 12/24 |
| 4 | 28/30 | 10/28 | 25/30 | 1/25 |
| 5 | 27/30 | 18/27 | 25/30 | 1/25 |
| 6 | 29/30 | 25/29 | 22/30 | 10/22 |
| 7 | 30/30 | 9/30 | 27/30 | 1/27 |
| 8 | 30/30 | 20/30 | 25/30 | 4/25 |
| 9 | 29/30 | 27/29 | 26/30 | 12/26 |
| 2 layers | | | | |
| | Easy | | Difficult | |
| Index | success | $\leq A^*$ | success | $\leq A^*$ |
| 1 | 30/30 | 17/30 | 29/30 | 3/29 |
| 2 | 27/30 | 26/27 | 21/30 | 7/21 |
| 3 | 28/30 | 25/28 | 22/30 | 6/22 |
| 4 | 30/30 | 10/30 | 29/30 | 1/29 |
| 5 | 27/30 | 16/27 | 23/30 | 7/23 |
| 6 | 29/30 | 23/29 | 19/30 | 6/19 |
| 7 | 30/30 | 11/30 | 29/30 | 1/29 |
| 8 | 27/30 | 23/27 | 21/30 | 8/21 |
| 9 | 30/30 | 25/30 | 24/30 | 11/24 |

### 1) *Simulation with Fixed Maps:*
In the first set of simulations, the map remains fixed. Simulation results for 2-layer

TABLE VI: Not Fixed Map, 1-layer and 2-layer

| | 1-layer 256 | | | | | | 2-layer | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Easy | | | Difficult | | | Easy | | | Difficult | | |
| Index | $T_p[ms]$ | success | $\leq A^*$ | $T_p[ms]$ | success | $\leq A^*$ | $T_p[ms]$ | success | $\leq A^*$ | $T_p[ms]$ | success | $\leq A^*$ |
| 1 | 16.6 | 30/30 | 18/30 | 14.4 | 27/30 | 5/27 | 27.98 | 29/30 | 11/29 | 22.46 | 29/30 | 2/29 |
| 2 | 18.3 | 27/30 | 13/27 | 14.1 | 19/30 | 7/19 | 29.34 | 27/30 | 23/27 | 22.93 | 22/30 | 4/22 |
| 3 | 17.3 | 29/30 | 24/39 | 13.4 | 25/30 | 10/25 | 27.19 | 28/30 | 24/28 | 22.63 | 19/30 | 5/19 |
| 4 | 25.7 | 30/30 | 3/40 | 18.5 | 25/30 | 2/25 | 35.31 | 30/30 | 10/30 | 30.97 | 22/30 | 0/22 |
| 5 | 26.0 | 27/30 | 20/27 | 18.7 | 23/30 | 7/23 | 34.34 | 29/30 | 21/29 | 33.22 | 24/30 | 8/24 |
| 6 | 25.3 | 28/30 | 20/28 | 18.8 | 24/30 | 10/24 | 33.62 | 26/30 | 19/26 | 31.14 | 23/30 | 5/23 |
| 7 | 24.5 | 30/30 | 30/30 | 18.6 | 25/30 | 4/25 | 33.54 | 30/30 | 11/30 | 31.92 | 27/30 | 0/27 |
| 8 | 24.7 | 29/30 | 9/30 | 17.6 | 19/30 | 5/19 | 33.73 | 30/30 | 21/30 | 32.61 | 20/30 | 4/20 |
| 9 | 24.3 | 28/30 | 28/30 | 25.7 | 24/30 | 12/24 | 32.55 | 28/30 | 21/28 | 33.02 | 20/30 | 10/20 |

LSTM and 1-layer with 256 neurons are given in Table V. Results show that our algorithm can generalize to unseen maps of different size. Training with large simulated maps can be very time consuming; our approach allows training of the prediction function with small maps, and usage for navigation in maps of unknown size.

Regardless of map size, the robot reaches the target successfully over 90% in an easy map. Further, the algorithm considers less nodes than $A^*$ on average over 50% of those times that the robot reaches its target. In difficult maps, the algorithm performs better if trained with larger maps. We hypothesize that this behavior is observed for the following reason. When trained with $11 \times 11$ maps, the prediction function may learn that the maximum length of successive obstacles is smaller than 11. When tested in a $51 \times 51$ map, the robot starts moving toward the target, but after observing 11 successive obstacles it may decide to turn back without being able to ascertain if that path leads to a dead-end. The learned model suggests that the path leads to a dead-end, but it may not be true since the map is larger. The robot then moves to the opposite direction, and does not reach the target within the allocated steps. When trained with $51 \times 51$ maps, the robot will keep moving forward after observing 11 successive obstacles. If it encounters a dead-end, it will mark this path as inaccessible, and continue.

It is worth noticing that the length of the sliding window can be smaller than the length of map size. If the map size is not known, we can simply choose $T = 30$ based on our results. Note that, by construction of the input, we cannot get a prediction of $R_{i+2}$ before the process step reaches the length of the sliding window. When the map size used for simulation is relatively small compared to the length of the sliding window, it is difficult for our algorithm to have less steps compared to $A^*$ expand area. However, this fact indicates that estimates of $R_{i+2}$ in fact help with navigation.

*2) Real-Time Execution Capability:* To provide detailed results on prediction time, a new map is generated in each trial. We run $N = 30$ simulation trials for the same nine conditions with randomly selected $q_0$ and $q_f$.

Table VI shows the results with different maps in each simulation trial. Time needed for prediction is given in both difficult and easy environments. The prediction time is averaged over total steps in 30 trials with different maps, which can represent a generalized result. In both difficult and easy cases, prediction time is related with sliding window length $T$ and number of layers in LSTM. When $T$ increases, it requires more time for prediction. For 2-layer LSTM with $T = 20$ and $T = 30$, the average prediction time is around $20ms$ and $30ms$, respectively, which makes it possible for real-time execution. For 1-256 layer network, prediction time can be reduced to as short as $14ms$. Number of weight parameters in the trained model for 2 layer LSTM network is 0.55 M for $T = 20$ and 0.59 M for $T = 30$. For 1-256 network, it is 0.43 M when $T = 20$ and 0.51 M for $T = 30$.

## V. Conclusions

The paper puts forward the idea that simple, shallow networks may offer promise in small robot learning-based navigation. Such networks can still apply despite the constrained computational and limited payload capacity of small robots, and lend themselves to analysis for deriving certain performance guarantees. Research on identifying minimalistic network architectures that perform well in the context of small robot navigation appears to be limited.

We narrow this gap by investigating five cases of shallow networks, and by delving deeper on those architectures that achieve high levels of accuracy via extensive simulations. We show that a network with only one LSTM layer is able to predict the existence of obstacles given a robot's observation history and a potential action. Our approach scales well as the map size grows: A sliding window of length 30 can be used in $51 \times 51$ maps with over 90% prediction accuracy.

Furthermore, we propose an algorithm which embeds this modest LSTM network to estimate a two-step forward reward in a Temporal-Difference Reinforcement Learning context for robot navigation. We show that in 2D grid worlds the algorithm is capable of adapting to variable-size environments not encountered during training, with randomly selected start and target positions. A simulated robot equipped with a simple four-directional IR sensing module can reach a target with over 95% and 80% success rates in easy and difficult maps, respectively. Combined with the short prediction times of the algorithm, our approach offers promise for real-time execution on small robots in hardware experiments.By employing sensors with different sensing ranges, with the same network architecture, our algorithm can be used to determine n-step forward rewards, where $2 < n < f$. Our future work will compare the proposed method with the lambda-return TD learning and Monte Carlo methods, and validate it with physical experiments.

## References

[1] D. Mellinger, N. Michael, and V. Kumar, "Trajectory generation and control for precise aggressive maneuvers with quadrotors," *The International Journal of Robotics Research*, vol. 31, no. 5, pp. 664–674, 2012.

[2] R. Bapst, R. Ritz, L. Meier, and M. Pollefeys, "Design and implementation of an unmanned tail-sitter," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2015, pp. 1885–1890.

[3] D. Brescianini, M. Hehn, and R. D'Andrea, "Quadrocopter pole acrobatics," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2013, pp. 3472–3479.

[4] K. Mohta, V. Kumar, and K. Daniilidis, "Vision-based control of a quadrotor for perching on lines," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2014, pp. 3130–3136.

[5] J. Thomas, G. Loianno, J. Polin, K. Sreenath, and V. Kumar, "Toward autonomous avian-inspired grasping for micro aerial vehicles," *Bioinspiration & Biomimetics*, vol. 9, no. 2, p. 025010, 2014.

[6] J. Worcester, J. Rogoff, and M. A. Hsieh, "Constrained task partitioning for distributed assembly," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2011, pp. 4790–4796.

[7] D. Stein, R. Schoen, and D. Rus, "Constraint-aware coordinated construction of generic structures," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2011, pp. 4803–4810.

[8] Q. Lindsey, D. Mellinger, and V. Kumar, "Construction with quadrotor teams," *Autonomous Robots*, vol. 33, no. 3, pp. 323–336, 2012.

[9] F. Augugliaro, S. Lupashin, M. Hamer, C. Male, M. Hehn, M. W. Mueller, J. S. Willmann, F. Gramazio, M. Kohler, and R. D'Andrea, "The flight assembled architecture installation: Cooperative construction with flying machines," *IEEE Control Systems*, vol. 34, no. 4, pp. 46–64, 2014.

[10] J. Werfel, K. Petersen, and R. Nagpal, "Designing collective behavior in a termite-inspired robot construction team," *Science*, vol. 343, no. 6172, pp. 754–758, 2014.

[11] A. Mathis, J. Russell, T. Moore, J. Cohen, B. Satterfield, N. Kohut, X.-Y. Fu, and r. S. Fearing, "Autonomous navigation of a 5 gram crawling millirobot in a complex environment," in *Adaptive Mobile Robotics: Proc. 15th Int. Conf. on Climbing and Walking Robots and the Support Technologies for Mobile Machines*, 2012, pp. 121–128.

[12] S. K. Shah, C. D. Pahlajani, and H. G. Tanner, "Optimal navigation for vehicles with stochastic dynamics," *IEEE Transactions on Control Systems Technology*, vol. 23, no. 5, pp. 2003–2009, Sep. 2015.

[13] K. Karydis, I. Poulakakis, and H. G. Tanner, "A navigation and control strategy for miniature legged robots," *IEEE Transactions on Robotics*, vol. 33, no. 1, pp. 214–219, 2017.

[14] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[15] V. Nguyen, A. Harati, A. Martinelli, R. Siegwart, and N. Tomatis, "Orthogonal slam: a step toward lightweight indoor autonomous navigation," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2006, pp. 5007–5012.

[16] K. Karydis and V. Kumar, "Energetics in robotic flight at small scales," *Royal Society Interface Focus*, vol. 7, no. 1, p. 20160088, 2017.

[17] D. Y. Little and F. T. Sommer, "Learning and exploration in action-perception loops," *Frontiers in neural circuits*, vol. 7, p. 37, 2013.

[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. on Machine Learning*, vol. 48, pp. 1928–1937, 2016.

[19] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *Proc. 34th Int. Conf. on Machine Learning (ICML)*, 2017, pp. 2778–2787.

[20] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

[21] A. Khan, C. Zhang, N. Atanasov, K. Karydis, V. Kumar, and D. Lee, "Memory augmented control networks," *CoRR*, vol. abs/1709.05706, 2017.

[22] H. X. Pham, H. M. La, D. Feil-Seifer, and L. V. Nguyen, "Autonomous uav navigation using reinforcement learning," *CoRR*, vol. abs/1801.05086, 2018.

[23] B. Zuo, J. Chen, L. Wang, and Y. Wang, "A reinforcement learning based robotic navigation system," in *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC)*, 2014, pp. 3452–3457.

[24] K. Zhong, Z. Song, P. Jain, P. L. Bartlett, and I. S. Dhillon, "Recovery guarantees for one-hidden-layer neural networks," *CoRR*, 2017.

[25] Y. Li and Y. Yuan, "Convergence analysis of two-layer neural networks with relu activation," in *Proc. Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 597–607.

[26] S. Goel and A. Klivans, "Learning depth-three neural networks in polynomial time," *CoRR*, 2017.

[27] D. Floreano and F. Mondada, "Evolutionary neurocontrollers for autonomous mobile robots," *Neural Networks*, vol. 11, no. 7-8, pp. 1461–1478, 1998.

[28] T. Ziemke, D.-A. Jirenhed, and G. Hesslow, "Internal simulation of perception: a minimal neuro-robotic model," *Neurocomputing*, vol. 68, pp. 85–104, 2005.

[29] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 2014, vol. 1, no. 1.

[30] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[31] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. on Artificial Intelligence and Statistics*, vol. 15, pp. 315–323, 2011.

[32] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[33] 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*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[34] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359 – 366, 1989.

[35] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012, vol. 385.

[36] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber *et al.*, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[37] E. Marchi, G. Ferroni, F. Eyben, L. Gabrielli, S. Squartini, and B. Schuller, "Multi-resolution linear prediction based features for audio onset detection with bidirectional lstm neural networks," in *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2014, pp. 2164–2168.

[38] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 2625–2634.

[39] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, Oct. 2000.

[40] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proc. 15th Annual Conf. of the Int. Speech Communication Association*, 2014, pp. 338–342.

[41] Y. N. Dauphin, H. de Vries, J. Chung, and Y. Bengio, "Rmsprop and equilibrated adaptive learning rates for non-convex optimization," *CoRR*, vol. abs/1502.04390, 2015.

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

[43] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *Journal of the Association for Computing Machinery (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.