MEMORY AUGMENTED CONTROL NETWORKS

Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar, Daniel D. Lee
GRASP Laboratory, University of Pennsylvania

ABSTRACT

Planning problems in partially observable environments cannot be solved directly with convolutional networks and require some form of memory. But, even memory networks with sophisticated addressing schemes are unable to learn intelligent reasoning satisfactorily due to the complexity of simultaneously learning to access memory and plan. To mitigate these challenges we propose the Memory Augmented Control Network (MACN). The network splits planning into a hierarchical process. At a lower level, it learns to plan in the locally observed space. At a higher level, it uses a collection of policies computed on locally observed spaces to learn the optimal plan in the global environment it is operating in. The performance of the network is evaluated in discrete grid world environments for path planning in the presence of simple and complex obstacles and in addition is tested for its ability to generalize to new environments not seen in the training set.

1 INTRODUCTION

A planning task can be thought of as a loop process that contains a two-step problem. In the first step, the agent perceives its environment. In the second step, the agent executes an action based on the perceived environment. In the past such perception-action loops have been learned using deep learning Chebotar et al. (2016), Lee et al. (2017). Popular approaches in this spirit are often end-to-end (i.e. mapping sensor readings directly to motion commands) and manage to solve problems in which the underlying dynamics of the environment or the agent are too complex to model. Approaches to learn end-to-end perception-action loops have been extended to complex reinforcement learning tasks such as learning how to play Atari games Mnih et al. (2013a), as well as to imitation learning tasks like controlling a robot arm Levine et al. (2015).

However, in planning problems, the optimal actions are often sequential in nature. Purely convolutional architectures (CNNs) perform poorly when applied to planning problems due to the reactive nature of the policies learned by them Zhang et al. (2016b), Giusti et al. (2016), Daftry et al. (2016). The complexity of this problem is compounded when the environment is only partially observable as is the case with most real world tasks. Zhang et al. (2016a) showed that when learning how to plan in such environments, it becomes necessary to use memory to retain information about states visited. Using recurrent networks to store past information and learn optimal control has been explored before in Levine (2013). While Siegelmann & Sontag (1995) have shown that recurrent networks are Turing complete and are hence capable of generating any arbitrary sequence in theory, this does not always translate into practice. Recent advances in memory augmented networks have shown that it is beneficial to use external memory with write operators that can be learned by a neural network over recurrent neural networks Graves et al. (2014b), Graves et al. (2016). Specifically, we are interested in the Differentiable Neural Computer (DNC) Graves et al. (2016) which uses an external memory and a network controller to learn how to read, write and access locations in the external memory. The DNC is structured such that computation and memory operations are separated from each other. Such a memory network can in principle be plugged into the convolutional architectures described above, and be trained end to end since the read and write operations are differentiable. However, as we show in our work, directly using such a memory scheme with CNNs performs poorly for partially observable planning problems and also does not generalize well to new environments.

To address the aforementioned challenges we propose the Memory Augmented Control Network (MACN), a novel architecture specifically designed to learn how to plan in partially observable
environments under sparse rewards. Environments with sparse rewards are harder to navigate since there is no immediate feedback. The intuition behind this architecture is that planning problem can be split into two levels of hierarchy. At a lower level, a planning module computes optimal policies using a feature rich representation of the locally observed environment. This local policy along with a sparse feature representation of the partially observed environment is part of the optimal solution in the global environment. Thus, the key to our approach is using a planning module to output a local policy which is used to augment the neural memory to produce an optimal policy for the global environment. The ability of the proposed model is first evaluated by its ability to learn policies when trained by supervised learning in the presence of simple and complex obstacles. Further, the model is evaluated on its ability to generalize to environments and situations not seen in the training set. The ground truth policies required to train the network are generated from a heuristic or an expert. Thus, the key contributions of this paper can be summarized as:

1. A novel memory augmented network architecture that learns to plan in partially observed unknown environments.
2. Extensive experimentation to analyze the ability of the proposed architecture’s ability to learn how to plan and generalize.

2 METHODOLOGY

Section 2.1 outlines notation and formally defines the problem statement. Section 2.2 and 2.3 briefly cover the theory behind value iteration networks and memory augmented networks. Finally, in section 2.4 the intuition and the computation graph is explained for the practical implementation of the model.

2.1 PRELIMINARIES

A partially observable problem can be represented by a Markov Decision Process (MDP) [Sutton & Barto, 1998] over a continuous information space capturing all possible observation-action sequences. We consider a finite horizon discounted MDP denoted by \( \mathcal{M}(S, A, T, R, \gamma) \). In this representation \( S \subseteq \mathbb{R}^n \) is the state space , \( A \subseteq \mathbb{R}^m \) is the action space, \( T : S \times A \times S \rightarrow [0, 1] \) is the transition function, \( R : S \times A \rightarrow \mathbb{R} \) is the reward function, and \( \gamma \in (0, 1] \) is a discount factor. The solution of such an MDP is a policy \( \pi(\mathcal{U}|I) \) that obtains maximum rewards.

Let the agents environment be a MDP \( \mathcal{M} \). The states \( S \) represent the workspace of the agent. Let \( S^{obs} \) and \( S^{goal} \), called the obstacle region and the goal state, respectively, be subsets of \( S \). Denote the obstacle-free portion of the workspace as \( S^{free} := S \setminus S^{obs} \). The dynamics of the agent and environment are modeled by the transition function \( T \). We assume that the control input space \( A \) is a finite discrete set (it is possible to have a continuous control space too). The information available to the agent at time \( t \) to compute the control input \( a_t \) is its current state \( s_t \) and the goal \( X^{goal} \). The problem is then defined as:

Given an initial state \( s_0 \in S^{free} \) and a goal state \( S^{goal} \subseteq S^{free} \), find a policy \( \pi : S \rightarrow A \), if one exists, that maximizes the cumulated reward over time and on applying the control \( a_t := \pi(s_t) \) results in a sequence of states that satisfies \( \{s_0, s_1, \ldots, s_T\} \subseteq S^{free} \) and \( s_T \in S^{goal} \).

An important point to note here is that even though the environment (global space) is formulated as a MDP, at any given instant, the agent only observes a small part of the environment at any given point (locally observed space). However, this locally observed space can also be thought of as a MDP. Our approach computes optimal policies for these locally observed spaces and then uses these to compute a policy optimal in the global space. We explain this in more detail in Section 2.4.

2.2 VALUE ITERATION NETWORKS

The typical algorithm to solve an MDP is Value Iteration (VI) [Sutton & Barto, 1998]. The value of a state (i.e. the expected reward over the time horizon if an optimal policy is followed) is computed iteratively by calculating an action value function \( Q(s, a) \) for each state. The value for state \( s \) can

\[ Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') Q(s', \pi(s')) \]

In this work an agent receives a reward only when it has reached the goal prescribed by the planning task.
then be calculated by $V(s) := \max_a Q(s, a)$. By iterating multiple times over all states and all actions possible in each state, we can get a policy $\pi = \arg\max_a Q(s, a)$. The update rule for value iteration is

$$V_{k+1}(s) = \max_a [R(s, a) + \gamma \sum_{s'} T(s'|s, a)V_k(s)].$$

(1)

A key aspect of our network is the inclusion of this network component that can approximate this Value Iteration algorithm. To this end we use the VI module in Value Iteration Networks (VIN) [Tamar et al. (2016)]. Their insight is that value iteration can be approximated by a convolutional network with max pooling. The standard form for windowed convolution is

$$V(x) = \sum_{k=x-w}^{x+w} V(k)f(k).$$

(2)

[Tamar et al. (2016)] show that the summation in (2) is analogous to $\sum_{s'} T(s'|s, a)V_k(s)$ (1). When (2) is stacked with reward, max pooled and repeated $K$ times, it represents an approximation of the value iteration algorithm over $K$ iterations.

2.3 External Memory Networks

Recent works on deep learning employ neural networks with external memory (Graves et al., 2014a), (Graves et al., 2016), (Kurach et al., 2015), (Parisotto & Salakhutdinov, 2017). Contrary to earlier works that explored the idea of the network learning how to read and access externally fixed memories, these recent works focus on learning to read and write to external memories, and thus forgo the task of designing what to store in the external memory. We are specifically interested in the DNC [Graves et al. (2016)] architecture. This is similar to the work introduced by Oh et al. (Oh et al., 2016) and Chen et al. (Chen et al., 2017). The external memory uses differentiable attention mechanisms to determine the degree to which each location in the external memory $M$ is involved in a read or write operation. The DNC makes use of a controller (a recurrent neural network such as LSTM) to learn to read and write to the memory matrix. A brief overview of the read and write operations follows.

2.3.1 Read and Write Operation

The read operation is defined as a weighted average over the contents of the locations in the memory. This produces a set of vectors defined as the read vectors. At time $t$ the read operation is defined as :

$$r_t = \sum_{i=1}^{K} w_i^R M_t,$$

(3)

where $w_i^R$ are the read weightings, $r_t$ is the read vector, and $M_t$ is the state of the memory at time $t$. These read vectors are appended to the controller input at the next time step which provides it access to the memory. The write operation consists of a write weight $w_i^W$, an erase vector $e_t$ and a write vector $v_t$. The write vector and the erase vector are emitted by the controller. These three components modify the memory at time $t$ as

$$M_t = M_{t-1}(1 - w_t^W e_t^\top) + w_t^W v_t^\top.$$

(4)

Memory addressing is defined separately for writing and reading. A combination of content-based addressing and dynamic memory allocation determines memory write locations, while a combination of content-based addressing and temporal memory linkage is used to determine read locations.

2.4 Memory Augmented Control Model

The key intuition behind designing this architecture is that planning in $M$ can be decomposed into two levels. At a lower level, planning is done in a local space within the boundaries of our locally observed environment space, i.e plan in $M'$ and calculate the optimal policy for this local space, $\pi_*$. It is then possible to use any planning algorithm to calculate the optimal value function $V_*^\pi$ from the

---

2We refer the interested reader to the original paper [Graves et al. (2016)] for a complete description.
optimal policy $\pi_1^* \in \mathcal{M}'$. Let $\Pi = [\pi_1^*, \pi_2^*, \pi_3^*, \ldots, \pi_n^*]$ be the list of optimal policies calculated from the consecutive observation spaces $[\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4, \ldots, \mathcal{M}_n]$. Given these two lists, when a new local space ($\mathcal{M}_{\text{new}}^0$) is observed, a relatively straightforward approach such as nearest neighbor search could be used to find a policy $\pi_{\text{new}}^*$ for this new space. By repeating this procedure over all new observations it could be concluded that one has computed a global policy $\pi_g$ that is optimal in $\mathcal{M}$.

However, this approach fails when there are local minima in the environment. In a 2D/3D world, these local minima could be long narrow tunnels culminating in dead ends (see Fig. 3). In a graph $G = (V, E)$, these minima could be a subgraph, $\bar{G} = (\bar{V}, \bar{E})$ from which no path to goal node $V_{\text{goal}}$ exists. As the environment is only partially observable, the agent has no prior knowledge about the structure of this tunnel and must explore it all the way to the end. Further, when entering and exiting such a structure, the agents observations are the same giving us the exact same MDPs, i.e $\mathcal{M}_1$ and $\mathcal{M}_2$ are the same but the optimal actions at these time steps are not the same, i.e $\pi_1^* \neq \pi_2^*$. To backtrack successfully from these tunnels/nodes, information about previously visited states is required, necessitating memory. In the spirit of learning an end to end algorithm with a differentiable memory network that selectively reads and writes information, it is required of the planning algorithm to be differentiable in nature. We add a VI module [Tamar et al., 2016] that approximates the value iteration algorithm within the framework of a neural network. This VI module when used in conjunction with a differentiable memory scheme, can be trained end to end to produce policies optimal in $\mathcal{M}$. Thus, to summarize, the planning problem is solved by decomposing it into a two level problem. At a lower level a feature rich representation of the environment is used to generate local policies. At the next level, these local policies along with a sparse feature representation of the currently observed environment is used to generate a policy consistent with the global environment. The proposed network architecture is shown in Figure 1.

Computation Graph: To explain the computation graph, consider the case of a 2D grid world with randomly placed obstacles, a start region and a goal region as shown in Fig. 2. The 2D grid world is presented in the form of an image $I$ of size $m \times n$ to the network. Let the goal region be $[m_{\text{goal}}, n_{\text{goal}}]$ and the start position be $[m_{\text{start}}, n_{\text{start}}]$. At any given instant, only a small part of $I$ is observed by the network and the rest of the image $I$ is blacked out. This corresponds to the agent only observing what is visible within the range of its sensor. In addition to this the image is stacked with a reward map $R$, as explained in [Tamar et al., 2016]. The reward map consists of an array of size $m \times n$ where all elements of the array except the one corresponding to index $[m_{\text{goal}}, n_{\text{goal}}]$ are zero. Array element corresponding to $[m_{\text{goal}}, n_{\text{goal}}]$ is set to a high value denoting reward (in our experiments it is set to 10). The input image of dimension $[m \times n \times 2]$ is first convolved with a kernel of size $(3 \times 3)$, 150 channels and stride of 1 everywhere. This is then convolved again with a kernel of size $(1, 1)$, 4 channels and stride of 1. Let this be the reward layer $R$. $R$ is convolved with another filter of size $(3, 3)$ with 4 channels. This is the initial estimate of the action value function or $Q(s, a)$. The initial value of the state $V(s)$ is also calculated by taking max over $Q(s, a)$. The
operations up to this point are summarized by the “Conv” block in Figure 1. Once these initial values have been computed, the model executes a for loop k times (the value of k ranges based on the task). Inside the for loop at every iteration, the R and V are first concatenated. This is then convolved with another filter of size (3,3) and 4 channels to get the updated action value of the state, $Q(s, a)$. We find the value of the state $V(s)$ by taking the max of the action value function. These values of k and the kernel sizes are constant across all three experiments. The updated value maps are then fed into a DNC controller. The DNC controller is a LSTM (hidden units vary according to task) that has access to an external memory. The external memory has 32 slots with word size 8 and we use 4 read heads and 1 write head. The output from the DNC controller and the memory is concatenated through a linear layer to get prediction for the action that the agent should execute. The optimizer used is the RMSProp and we use a learning rate of 0.0001 for our experiments.

This formulation is easy enough to be extended to environments where the state space is larger than two dimensions and the action space is larger. We demonstrate this in our experiments.

3 Experiments

To investigate the performance of MACN, we design our experiments to answer three key questions:

- Can it learn how to plan in partially observable environments with sparse rewards?
- How well does it generalize to new unknown environments?
- Can it be extended to other domains?

We first demonstrate that MACN can learn how to plan in a 2D grid world environment. Without loss of generality, we set the probability of all actions equal. The action space is discrete, $\mathcal{A} := \{\text{down, right, up, left}\}$. This can be easily extended to continuous domains since our networks output is a probability over actions. We show this in experiment 3.2. We then demonstrate that our network can learn how to plan even when the states are not constrained to a two dimensional space and the action space is larger than four actions.

3.1 Navigation in presence of simple obstacles

We first evaluate the ability of our network to successfully navigate a 2D grid world populated with obstacles at random positions. We make the task harder by having random start and goal positions. The full map shown in Fig. 2 is the top down view of the entire environment. The partial map shown in the middle panel of Fig. 2 represents a stitched together version of all states explored by the agent. However, notice that the agent receives information only about its local surroundings as determined by its sensor’s footprint. An instance of this is shown in the rightmost panel of Fig. 2. The input to the network is the sensor map, where the area that lies outside the agent’s sensing abilities is grayed out. VIN: With just the VI module and no memory in place, we test the performance of the value iteration network on this 2D partially observable environment. CNN + Memory: We setup a CNN architecture where the sensor image with the reward map is forward propagated through four convolutional layers to extract features. We test if these features alone are enough for the memory
Table 1: **Performance on 2D grid world with simple obstacles**: All models are tested on maps generated via the same random process, and were not present in the training set. Episodes over 40 (for a $16 \times 16$ wide map), 60 (for $32 \times 32$) and 80 (for $64 \times 64$) time steps were terminated and counted as a failure. Episodes where the agent collided with an obstacle were also counted as failures.

| Model                  | Performance | $16 \times 16$ | $32 \times 32$ | $64 \times 64$ |
|------------------------|-------------|----------------|----------------|----------------|
| VIN                    | Success(%)  | 0              | 0              | 0              |
|                        | Test Error  | 0.63           | 0.78           | 0.81           |
| CNN + Memory           | Success(%)  | 0.12           | 0              | 0              |
|                        | Test Error  | 0.43           | 0.618          | 0.73           |
| MACN (LSTM)            | Success (%) | 88.12          | 73.4           | 64             |
|                        | Test Error  | 0.077          | 0.12           | 0.21           |
| MACN                   | Success(%)  | **96.3**       | **85.91**      | **78.44**      |
|                        | Test Error  | **0.02**       | **0.08**       | **0.13**       |

to navigate the 2D grid world. A natural question to ask at this point is can we achieve planning in partially observable environments with just a planning module and a simple recurrent neural network such as a LSTM. To answer this we also test MACN with a LSTM in place of the memory scheme. We present our results in Table [1]. These results are obtained from testing on a held out test-set consisting of maps with random start, goal and obstacle positions.

Our results show that MACN can learn how to navigate partially observable 2D unknown environments. Note that the VIN does not work by itself since it has no memory to help it remember past actions. We would also like to point out that while the CNN + Memory architecture is similar to Oh et al. (2016), its performance in our experiments is very poor due to the sparse rewards structure. MACN significantly outperforms all other architectures. Furthermore, MACN’s drop in testing accuracy as the grid world scales up is not as large compared to the other architectures. While these results seem promising, in the next section we extend the experiment to determine whether MACN actually learns how to plan or it is overfitting.

### 3.2 Navigation in Presence of Local Minima

The previous experiment shows that MACN can learn to plan in 2D partially observable environments. While the claim that the network can plan on environments it has not seen before stands, this is weak evidence in support of the generalizability of the network. In our previous experiment the test environments have the same dimensions as in the training set, the number of combinations of random obstacles especially in the smaller environments is not very high and during testing some of the wrong actions can still carry the agent to the goal. Thus, our network could be overfitting and may not generalize to new environments. In the following experiment we test our proposed network’s capacity to generalize.

The environment is setup with tunnels. The agent starts off at random positions inside the tunnel. While the orientation of the tunnel is fixed, its position is not. To comment on the the ability of our network to generalize to new environments with the same task, we look to answer the following question: *When trained to reach the goal on tunnels of a fixed length, can the network generalize to longer tunnels in bigger maps not seen in the training set?*

The network is set up the same way as before. The task here highlights the significance of using memory in a planning network. The agent’s observations when exploring the tunnel and exiting the tunnel are the same but the actions mapped to these observations are different. The memory in our network remembers past information and previously executed policies in those states, to output the right action. We report our results in Table [2]. To show that traditional deep reinforcement learning performs poorly on this task, we implement the DQN architecture as introduced in Mnih et al. (2013b). We observe that even after one million iterations, the DQN does not converge to the optimal policy on the training set. This can be attributed to the sparse reward structure of the environment. We report similar findings when tested with A3C as introduced in Mnih et al. (2016). We also observe that the CNN + memory scheme learns to turn around at a fixed length and does not explore the longer tunnels in the test set all the way to the end.
Figure 3: **Grid world environment with local minima.** **Left:** In the full map the blue square represents the current position while the red square represents the goal. **Center-left:** The partial map represents a stitched together version of all states explored by the agent. Since the agent does not know if the tunnel culminates in a dead end, it must explore it all the way to the end. **Center-right:** The sensor input is the information available to the agent. **Right:** The full map that we test our agent on after being trained on smaller maps. The dimensions of the map as well as the tunnel are larger.

Table 2: **Performance on grid world with local minima:** All models are trained on tunnels of length 20 units. The success percentages represent the number of times the robot reaches the goal position in the test set after exploring the tunnel all the way to the end. Maximum generalization length is the length of the longest tunnel that the robot is able to successfully navigate after being trained on tunnels of length 20 units.

| Model         | Success (%) | Maximum generalization length |
|---------------|-------------|-------------------------------|
| DQN           | 0           | 0                             |
| A3C           | 0           | 0                             |
| CNN + Memory  | 12          | 20                            |
| VIN           | 0           | 0                             |
| MACN          | **100**     | **330**                       |

These results offer insight into the ability of MACN to generalize to new environments. Our network is found capable of planning in environments it has not seen in the training set at all. On visualizing the memory (see supplemental material), we observe that there is a big shift in the memory states only when the agent sees the end of the wall and when the agent exits the tunnel. A t-sne [Maaten & Hinton (2008)] visualization over the action spaces (see Fig. 6) clearly shows that the output of our network is separable. We can conclude from this that the network has learned the spatial structure of the tunnel, and it is now able to generalize to tunnels of longer length in larger maps.

Figure 4: **T-sne visualization on 2D grid worlds with tunnels.** **a)** T-sne visualization of the raw images fed into the network. Most of the input images for going into the tunnel and exiting the tunnel are the same but have different action labels. **b)** T-sne visualization from the outputs of the pre planning module. While it has done some separation, it is still not completely separable. **c)** Final output from the MACN. The actions are now clearly separable.
3.3 General Graph Search

In our earlier experiments, the state space was constrained in two dimensions, and only four actions were available. It is nearly impossible to constrain every real world task to a two dimensional space with only four actions. However, it is easier to formulate a lot of partially observable planning problems as a graph.

We define our environment as an undirected graph \( G = (V, E) \) where the connections between the nodes are generated randomly (see Fig. 5). In Fig 5 the blue node is the start state and the red node is the goal state. Each node represents a possible state the agent could be in. The agent can only observe all edges connected to the node it currently is in thus making it partially observable. The action space for this state is then any of the possible nodes that the agent can visit next. As before, the agent only gets a reward when it reaches the goal. We also add in random start and goal positions. In addition, we add a transition probability of 0.8. (For training details and generation of graph see Appendix.) We present our results in Table 3. On graphs with small number of nodes, the reinforcement learning with DQN and A3C sometimes converge to the optimal goal due to the small state size and random actions leading to the goal node in some of the cases. However, as before the MACN outperforms all other models. On map sizes larger than 36 nodes, performance of our network starts to degrade. Further, we observe that even though the agent outputs a wrong action at some times, it still manages to get to the goal in a reasonably small number of attempts. From these results, we can conclude that MACN can learn to plan in more general problems where the state space is not limited to two dimensions and the action space is not limited to four actions.

| Model         | Test Error, Success(%) |
|---------------|-------------------------|
|               | 9 Nodes | 16 Nodes | 25 Nodes | 36 Nodes |
| VIN           | 0.57, 23.39 | 0.61, 14 | 0.68, 0   | 0.71, 0 |
| A3C           | NA, 10    | NA, 7    | NA, 0     | NA, 0    |
| DQN           | NA, 12    | NA, 5.2  | NA, 0     | NA, 0    |
| CNN + Memory  | 0.25, 81.5 | 0.32, 63 | 0.56, 19  | 0.68, 9.7|
| MACN (LSTM)   | 0.14, 98  | 0.19, 96.27 | 0.26, 84.33 | 0.29, 78 |
| MACN          | 0.1, 100  | 0.18, 100 | 0.22, 95.5 | 0.28, 89.4|

Table 3: Performance on General Graph Search. Test error is not applicable for the reinforcement learning models A3C and DQN

3.4 Continuous Control Domain

Learning how to navigate in unknown environments, where only some part of the environment is observable is a problem highly relevant in robotics. Traditional robotics solve this problem by creating and storing a representation of the entire environment. However, this can quickly get memory intensive. In this experiment we extend MACN to a SE2 robot. The SE2 notation implies that the robot is capable of translating in the x-y plane and has orientation. The robot has a differential drive controller that outputs continuous control values. The robot is spawned in the environment shown in Fig (6a). As before, the robot only sees a small part of the environment at any given time. In this case the robot has a laser scanner that is used to perceive the environment.

It is easy to convert this environment to a 2D framework that the MACN needs. We fix the size of the environment to a \( m \times n \) grid. This translates to a \( m \times n \) matrix that is fed into the MACN. The parts of the map that lie within the range of the laser scanner are converted to obstacle free and obstacle occupied regions and added to the matrix. Lastly, an additional reward map denoting a high value for the goal location and zero elsewhere as explained before is appended to the matrix and fed into the MACN. The network output is used to generate waypoints that are sent to the underlying controller. The training set is generated by randomizing the spawn and goal locations and using a suitable heuristic. The performance is tested on a held out test set of start and goal locations. More experimental details are outlined in the appendix.
Figure 6: Navigation in a 3D environment on a continuous control robot. a) The robot is spawned in a 3D simulated environment. b) Only a small portion of the entire map is visible at any given point to the robot. c) The blue circle represents robot start position and the red circle represents goal position. The green line denotes ground truth and blue line indicates the output of MACN.

| Model     | Success (%) |
|-----------|-------------|
| DQN, A3C  | 0           |
| CNN + Memory | 11.60     |
| VIN       | 16.74       |
| MACN      | 91.3        |

Table 4: Performance on robot world

We observe that the proposed architecture is able to find its way to the goal a large number of times and its trajectory is close to the ground truth. This task is a lot harder than the grid world navigation due to the addition of orientation. The lack of explicit planning in the CNN + Memory architecture hampers its ability to get to the goal in this task. In addition to this, as observed before traditional reinforcement learning is unable to converge to the goal.

4 Related Work

Using value iteration networks augmented with memory has been explored before in [Gupta et al., 2017]. In their work a planning module together with a map representation of a robot’s free space is used for navigation in a partially observable environment using image scans. The image scans are projected into a 2D grid world by approximating all possible robot poses. This is in contrast to our work here in which we design a general memory based network that can be used for any partially observed planning problem. Another similar work is that of Oh et al. [Oh et al., 2016] where a network is designed to play Minecraft. The game environment is projected into a 2D grid world and the agent is trained by RL to navigate to the goal. That network architecture uses a CNN to extract high level features followed by a differentiable memory scheme. This is in contrast to our paper where we
approach this planning by splitting the problem into local and global planning. Using differential network schemes with CNNs for feature extraction has also been explored in (Chen et al., 2017).

5  CONCLUSION

Planning in environments that are partially observable and have sparse rewards with deep learning has not received a lot of attention. Also, the ability of policies learned with deep RL to generalize to new environments is often not investigated. In this work we take a step toward designing architectures that compute optimal policies even when the rewards are sparse, and thoroughly investigate the generalization power of the learned policy.

The grid world experiments offer conclusive evidence about the ability of our network to learn how to plan in such environments. We address the concern of oversimplifying our environment to a 2D grid world by experimenting with planning in a graph with no constraint on the state space or the action space. In the future, we intend to extend our policies trained in simulation to a real world platform such as a robot learning to plan in partially observable environments.

REFERENCES

Yevgen Chebotar, Karol Hausman, Zhe Su, Gaurav S Sukhatme, and Stefan Schaal. Self-supervised regrasping using spatio-temporal tactile features and reinforcement learning. In Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on, pp. 1960–1966. IEEE, 2016.

Steven W Chen, Nikolay Atanasov, Arbaaz Khan, Konstantinos Karydis, Daniel D Lee, and Vijay Kumar. Neural network memory architectures for autonomous robot navigation. arXiv preprint arXiv:1705.08049, 2017.

Shreyans Daftry, J Andrew Bagnell, and Martial Hebert. Learning transferable policies for monocular reactive mav control. In International Symposium on Experimental Robotics, pp. 3–11. Springer, 2016.

Alessandro Giusti, Jérôme Guzzi, Dan C Cireşan, Fang-Lin He, Juan P Rodríguez, Flavio Fontana, Matthias Faessler, Christian Forster, Jürgen Schmidhuber, Gianni Di Caro, et al. A machine learning approach to visual perception of forest trails for mobile robots. IEEE Robotics and Automation Letters, 1(2):661–667, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. arXiv preprint arXiv:1410.5401, 2014a.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. arXiv preprint arXiv:1410.5401, 2014b.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. Nature, 538(7626):471–476, 2016.

Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. arXiv preprint arXiv:1702.03920, 2017.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. arXiv preprint arXiv:1511.06392, 2015.

Alex X Lee, Sergey Levine, and Pieter Abbeel. Learning visual servoing with deep features and fitted q-iteration. arXiv preprint arXiv:1703.11000, 2017.

Sergey Levine. Exploring deep and recurrent architectures for optimal control. arXiv preprint arXiv:1311.1761, 2013.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. arXiv preprint arXiv:1504.00702, 2015.
Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013a.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013b.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016.

Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*, 2017.

Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pp. 2154–2162, 2016.

Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. Learning deep neural network policies with continuous memory states. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pp. 520–527. IEEE, 2016a.

Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pp. 528–535. IEEE, 2016b.