Elements of Intelligence: Memory, Communication and Intrinsic Motivation

by

Sainbayar Sukhbaatar

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2018

Professor Rob Fergus

© Sainbayar Sukhbaatar

All Rights Reserved, 2018

Dedication

To my lovely wife Undarmaa and my precious daughter Saraana.

Acknowledgements

First, I would like to thank my advisor Rob Fergus for supporting me throughout my Ph.D. program. Besides being an awesome instructor, he has always been encouraging and understanding, which made the last 5 years a fulfilling and exciting experience for me. Another integral person making all the research in this thesis possible is Arthur Szlam, who have spent countless hours arguing about ideas with me, and making sure they are in the right direction.

Additionally, I would like to thank my committee members, Jason Weston, Kyunghyun Cho and Joan Bruna for their valuable feedback, and also being an inspiration to me as a researcher. Special thanks goes to Kyunghyun for providing detailed remarks and making sure my thesis is in good shape.

I had an opportunity to meet many brilliant people during my time at CILVR lab. I thank Yann LeCun, Ross Goroshin, Pablo Sprechmann, Emily Denton, Michael Mathieu, Mikael Henaff, Xiang Zhang, William Whitney, Alexander Rives, and Jake Zhao for many interesting and fruitful discussions.

During my internships at Facebook AI Research, I've learned a lot from Marc'Aurelio Ranzato, Manohar Paluri, Armand Joulin, Laurens van der Maaten, Tomas Mikolov, Soumith Chintala, Zeming Lin, and Adam Lerer. I also thank Volodymyr Mnih for guiding me during my internship at DeepMind.

I have to thank my daughter Saraana, who grew up along with this thesis, for teaching me more about the development of intelligence than any textbook could have. Finally, all of this would have been impossible without the loving support of my wonderful wife Undarmaa, who was always there for me during the many ups and downs of a Ph.D. program.

Preface

The chapters 3, 4, 5 and 6 of this thesis are appeared in the following publications respectively:

- Sainbayar Sukhbaatar, Arthur Szlam, Gabriel Synnaeve, Soumith Chintala, and Rob Fergus. Mazebase: A sandbox for learning from games. *CoRR*, abs/1511.07401, 2015.

- Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in Neural Information Processing Systems 28*. 2015.

- Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems 29*. 2016.

- Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations*, 2018.

Also, their source code can be downloaded from:

- `http://github.com/facebook/MazeBase`
- `http://github.com/facebook/MemNN`
- `http://cims.nyu.edu/~sainbar/commnet/`
- `http://cims.nyu.edu/~sainbar/selfplay`

Abstract

Building an intelligent agent that can learn and adapt to its environment has always been a challenging task. This is because intelligence consists of many different elements such as recognition, memory, and planning. In recent years, deep learning has shown impressive results in recognition tasks. The aim this thesis is to advance the deep learning techniques to other elements of intelligence.

We start our investigation with memory, an integral part of intelligence that bridges past experience with current decision making. In particular, we focus on the episodic memory, which is responsible for storing our past experiences and recalling them. An agent without such memory will struggle at many tasks such as having a coherent conversation. We show that a neural network with an external memory is better suited to such tasks than traditional recurrent networks with an internal memory.

Another crucial ingredient of intelligence is the capability to communicate with others. In particular, communication is essential for agents participating in a cooperative task, improving their collaboration and division of labor. We investigate whether agents can learn to communicate from scratch without any external supervision. Our finding is that communication through a continuous vector facilitates faster learning by allowing gradients to flow between agents.

Lastly, an intelligent agent must have an intrinsic motivation to learn about its environment on its own without any external supervision or rewards. Our investigation led to one such learning strategy where an agent plays a two-role game with itself. The first role proposes a task, and the second role tries to execute it. Since their goal is to make the other fail, their adversarial interplay pushes them to explore increasingly complex tasks, which leads to a better understanding of the environment.

Table of Contents

Dedication **iii**

Acknowledgements **iv**

Preface **v**

Abstract **vi**

List of Figures **x**

List of Tables **xix**

1 Introduction **1**

2 Background **7**

2.1 Reinforcement Learning 7

2.2 StarCraft environment 10

2.3 RLLab environments 15

3 MazeBase: A Sandbox for Learning from Games **17**

3.1 Introduction 18

3.2 Environment 19

3.3 Tasks 21
3.4 Curriculum 27
3.5 Conclusion 27

4 End-to-end Memory Network 29

4.1 Introduction 30
4.2 Approach 31
4.3 Related Work 36
4.4 Synthetic Question and Answering Experiments 39
4.5 Language Modeling Experiments 46
4.6 Mazebase Experiments 52
4.7 StarCraft Experiments 62
4.8 Writing to the Memory 63
4.9 Conclusions and Future Work 66

5 Learning Multiagent Communication with Backpropagation 68

5.1 Introduction 69
5.2 Communication Model 70
5.3 Related Work 74
5.4 Experiments 76
5.5 Discussion and Future Work 91

6 Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play 92

6.1 Introduction 93
6.2 Approach 94
6.3 Related Work 100

6.4 Experiments 103
6.5 Discussion 115
6.6 Conclusion 118

7 Conclusion **119**

Bibliography **123**

List of Figures

2.1 Different types of unit in the development task. The arrows represent possible actions (excluding movement actions) by an unit, and corresponding numbers shows (blue) amount of minerals and (red) time steps needed to complete. The units under agent's control are outlined by a green border. 12

2.2 A simple combat scenario in StarCraft requiring a kiting tactic. To win the battle, the agent (the larger unit) has to alternate between (left) fleeing the stronger enemies while its weapon recharges, and (right) returning to attack them once it is ready to fire. 14

3.1 Examples of Multigoal (left) and Light Key (right) tasks. Note that the layout and dimensions of the environment varies between different instances of each task (i.e. the location and quantity of walls, water and goals all change). The agent is shown as a red blob and the goals are shown in yellow. For LightKey, the switch is show in cyan and the door in magenta/red (toggling the switch will change the door's color, allowing it to pass through). 21

4.1 A single layer version of our model. 31

4.2 A three layer version of our model. In practice, we can constrain several of the embedding matrices to be the same (see Section 4.2.2). 34
4.3 A recurrent view of MemN2N. 35
4.4 A toy example of an 1-hop MemN2N applied to a bAbI task. 40
4.5 A plot of position encoding (PE) l_{kj}. 41
4.6 Comparison results of different models on bAbI tasks. The best MemN2N model fails (accuracy $< 95\%$) on three tasks, only one more than a MemN2N trained with more strong supervision signals. 44
4.7 Experimental results from an ablation study of MemN2N on bAbI tasks. Left: incremental effects of different techniques. Right: reducing the number of hops. 46
4.8 Example predictions on the QA tasks of [Weston et al., 2016]. We show the labeled supporting facts (support) from the dataset which MemN2N does not use during training, and the probabilities p of each hop used by the model during inference (indicated by values and blue color). MemN2N successfully learns to focus on the correct supporting sentences most of the time. The mistakes made by the model are highlighted by red color. 49
4.9 MemN2N performance on two language modeling tasks compared to an LSTM baseline. We vary the number of memory hops (top row) and the memory size (bottom row). 51
4.10 Average activation weight of memory positions during 6 memory hops. White color indicates where the model is attending during the kth hop. For clarity, each row is normalized to have maximum value of 1. A model is trained on (left) Penn Treebank and (right) Text8 dataset. . . . 53

4.11 In the MazeBase environment, the difficulty of tasks can be varied programmatically. For example, in the **Multigoals** game the maximum map size, fraction of blocks/water and number of goals can all be varied. This affects the difficulty of tasks, as shown by the optimal reward (blue line). It also reveals how robust a given model is to task difficulty. For a 2-layer MLP (red line), the reward achieved degrades much faster than the MemN2N model (green line) and the inherent task difficulty. 56
4.12 Reward for each model jointly trained on the 10 games, with and without the use of a curriculum during training. The y-axis shows relative reward (estimated optimal reward / absolute reward), thus higher is better. The estimated optimal policy corresponds to a value of 1 and a value of 0.5 implies that the agent takes twice as many steps to complete the task as is needed (since most of the reward signal comes from the negative increment at each time step). 57
4.13 A trained MemN2N model playing the **Switches** (top) and **LightKey** (bottom) games. In the former, the goal is to make all the switches the same color. In the latter, the door blocking access to the goal must be opened by toggling the colored switch. The 2nd and 4th rows show the corresponding attention maps. The three different colors each correspond to one of the models 3 attention "hops". 60
4.14 An extension of MemN2N that can write into the memory. 63
4.15 Evolution of the memory content while the model sorts the given input numbers. 65
4.16 The attention map during memory hops while the model sorts given numbers. 66

5.1 An overview of our CommNet model. Left: view of module f^i for a single agent j. Note that the parameters are shared across all agents. Middle: a single communication step, where each agents modules propagate their internal state h, as well as broadcasting a communication vector c on a common channel (shown in red). Right: full model Φ, showing input states s for each agent, two communication steps and the output actions for each agent. 72
5.2 3D PCA plot of hidden states of agents 79
5.3 Left: Traffic junction task where agent-controlled cars (colored circles) have to pass the through the junction without colliding. Right: The combat task, where model controlled agents (red circles) fight against enemy bots (blue circles). In both tasks each agent has limited visibility (orange region), thus is not able to see the location of all other agents. . 80
5.4 An example CommNet architecture for a varying number agents. Each car is controlled by its own LSTM (solid edges), but communication channels (dashed edges) allow them to exchange information. 81
5.5 As visibility in the environment decreases, the importance of communication grows in the traffic junction task. 83
5.6 A harder version of traffic task with four connected junctions. 84

5.7 Left: First two principal components of communication vectors $\tilde{c}$ from multiple runs on the traffic junction task Figure 5.3(left). While the majority are "silent" (i.e. have a small norm), distinct clusters are also present. Right: First two principal components of hidden state vectors h from the same runs as on the left, with corresponding color coding. Note how many of the "silent" communication vectors accompany non-zero hidden state vectors. This shows that the two pathways carry different information. 85

5.8 For three of clusters shows in Figure 5.7(left), we probe the model to understand their meaning (see text for details). 86

5.9 (left) Average norm of communication vectors (right) Brake locations . 86

5.10 Failure rates of different communication approaches on the combat task. 88

5.11 CommNet applied to bAbI tasks. Each sentence of the story is processed by its own agent/stream, while a question is fed through the initial communication vector. A single output is obtained by simply summing the final hidden states. 89

6.1 Illustration of the self-play concept in a gridworld setting. Training consists of two types of episode: self-play and target task. In the former, Alice and Bob take turns moving the agent within the environment. Alice sets tasks by altering the state via interaction with its objects (key, door, light) and then hands control over to Bob. He must return the environment to its original state to receive an internal reward. This task is just one of many devised by Alice, who automatically builds a curriculum of increasingly challenging tasks. In the target task, Bob's policy is used to control the agent, with him receiving an *external* reward if he visits the flag. He is able to learn to do this quickly as he is already familiar with the environment from self-play. 93

6.2 An illustration of two versions of the self-play: (left) Bob starts from Alice's final state in a reversible environment and tries to return her initial state; (right) if an environment can be reset to Alice's initial state, Bob starts there and tries to reach the same state as Alice. 94

6.3 The policy networks of Alice and Bob takes take the initial state s_0 and the target state s^* as an additional input respectively. We set $s^* = s_0$ in a reverse self-play, but set to zero $s^* = \emptyset$ for target task episodes. Note that rewards from the environment are only used during target task episodes. 97

6.4 **Left:** The hallway task from Section 6.4.1. The y axis is fraction of successes on the target task, and the x axis is the total number of training examples seen. Plain REINFORCE (red) learns slowly. Adding an explicit exploration bonus [Strehl and Littman, 2008] (green) helps significantly. Our self-play approach (blue) gives similar performance however. Using a random policy for Alice (magenta) drastically impairs performance, showing the importance of self-play between Alice and Bob. **Right:** Mazebase task, illustrated in Figure 6.1, for p(Light off) = 0.5. Augmenting with the repeat form of self-play enables significantly faster learning than training on the target task alone and random Alice baselines. 106

6.5 Inspection of a Mazebase learning run, using the environment shown in Figure 6.1. (a): rate at which Alice interacts with 1, 2 or 3 objects during an episode, illustrating the automatically generated curriculum. Initially Alice touches no objects, but then starts to interact with one. But this rate drops as Alice devises tasks that involve two and subsequently three objects. (b) by contrast, in the random Alice baseline, she never utilizes more than a single object and even then at a much lower rate. (c) plot of Alice and Bob's reward, which strongly correlates with (a). (d) plot of t_A as self-play progresses. Alice takes an increasing amount of time before handing over to Bob, consistent with tasks of increasing difficulty being set. 109

6.6 Left: The performance of self-play when p(Light off) set to 0.3. Here the reverse form of self-play works well (more details in the text). Right: Reduction in target task episodes relative to training purely on the target-task as the distance between self-play and the target task varies (for runs where the reward goes above -2 on the Mazebase task – unsuccessful runs are given a unity speed-up factor). The y axis is the speedup, and x axis is p(Light off). For reverse self-play, the low p(Light off) corresponds to having self-play and target tasks be similar to one another, while the opposite applies to repeat self-play. For both forms, significant speedups are achieved when self-play is similar to the target tasks, but the effect diminishes when self-play is biased against the target task. 110

6.7 Evaluation on MountainCar (left) and SwimmerGather (right) target tasks, comparing to VIME [Houthooft et al., 2016] and SimHash [Tang et al., 2017] (figures adapted from [Tang et al., 2017]). With reversible self-play we are able to learn faster than the other approaches, although it converges to a comparable reward. Training directly on the target task using REINFORCE without self-play resulted in total failure. Here 1 iteration = 5k (50k) target task steps in Mountain car (SwimmerGather), excluding self-play steps. 112

6.8 A single SwimmerGather training run. (a): Rewards on target task. (b): Rewards from reversible self-play. (c): The number of actions taken by Alice. (d): Distance that Alice travels before switching to Bob. 113

6.9 Plot of Alice's location at time of STOP action for the SwimmerGather training run shown in Figure 6.8, for different stages of training. Note how Alice's distribution changes as Bob learns to solve her tasks. 114

6.10 Plot of reward on the StarCraft sub-task of training marine units vs #target-task episodes (self-play episodes are not included), with and without self-play. A count-based baseline is also shown. Self-play greatly speeds up learning, and also surpasses the count-based approach at convergence. 115

6.11 Plot of reward on the StarCraft sub-task of training where episode length t_{Max} is increased to 300. 116

List of Tables

2.1 Action space of different unit types in the development task. No-op=no operation means no action is sent to the game. 13

4.1 Test error rates (%) on the 20 QA tasks for models using 1k training examples. Key: BoW = bag-of-words representation; PE = position encoding representation; LS = linear start training; RN = random injection of time index noise; LW = RNN-style layer-wise weight tying (if not stated, adjacent weight tying is used); joint = joint training on all tasks (as opposed to per-task training). 47

4.2 Test error rates (%) on the 20 bAbI QA tasks for models using 10k training examples. Key: BoW = bag-of-words representation; PE = position encoding representation; LS = linear start training; RN = random injection of time index noise; LW = RNN-style layer-wise weight tying (if not stated, adjacent weight tying is used); joint = joint training on all tasks (as opposed to per-task training); $*$ = this is a larger model with non-linearity (embedding dimension is $d = 100$ and ReLU applied to the internal state after each hop. This was inspired by [Peng et al., 2015] and crucial for getting better performance on tasks 17 and 19). . 48

4.3 The perplexity on the test sets of Penn Treebank and Text8 corpora. Note that increasing the number of memory hops improves performance. 53

4.4 Reward of the different models on the 10 games, with and without curriculum. Each cell contains 3 numbers: (top) best performing one run (middle) mean of all runs, and (bottom) standard deviation of 10 runs with different random initialization. The estimated-optimal row shows the estimated highest average reward possible for each game. Note that the estimates are based on simple heuristics and are not exactly optimal. 59

4.5 Win rates against StarCraft built-in AI. The 2nd row shows the hand-coded baseline strategy of always attacking the weakest enemy (and not fleeing during cooldown). The 3rd row shows MemN2N trained and tested on StarCraft. The last row shows a MemN2N trained entirely inside MazeBase and tested on StarCraft with no modifications or fine tuning except scaling of the inputs. 62

5.1 Results of lever game (#distinct levers pulled)/(#levers) for our CommNet and independent controller models, using two different training approaches. Allowing the agents to communicate enables them to succeed at the task. 77

5.2 Traffic junction task failure rates (%) for different types of model and module function $f(.)$. CommNet consistently improves performance, over the baseline models. 82

5.3 Traffic junction task variants. In the easy case, discrete communication does help, but still less than CommNet. On the hard version, local communication (see Section 5.2.2) does at least as well as broadcasting to all agents. 83

5.4 Win rates (%) on the combat task for different communication approaches and module choices. Continuous consistently outperforms the other approaches. The fully-connected baseline does worse than the independent model without communication. 87

5.5 Win rates (%) on the combat task for different communication approaches. We explore the effect of varying the number of agents m and agent visibility. Even with 10 agents on each team, communication clearly helps. 88

5.6 Experimental results on bAbI tasks. Only showing some of the task with high errors. 90

6.1 Hyperparameter values used in experiments. TT=target task, SP=self-play 104

Chapter 1

Introduction

Building an intelligent system that can learn and adapt has always been a challenging task. In particular, achieving human-level intelligence is an unsolved problem despite decades of research. Our intelligence comes from our brain, which has almost 100 billion neurons [Azevedo et al., 2009] making trillions of connections. Although we lack full understanding of brain, we know that a brain processes information in a distributed way where each neuron performs a computation. A connection strength between two neurons can change depending on activation patterns of those neurons. We also discovered a hierarchical structure in the visual cortex [Hubel and Wiesel, 1968, Felleman and Essen, 1991] where areas higher in the hierarchy are associated with more abstract representations. Inspired by those findings, artificial neural networks [Fukushima, 1980, Rumelhart et al., 1986] emerged as a machine learning model with successful applications such as recognizing hand-written digits [LeCun et al., 1998].

Deep learning, a recent revival of neural networks fueled by big data and fast computing devices, has shown promising results in many specialized tasks where traditional machine learning approaches have struggled. In particular, deep learning has brought

impressive improvements in image classification tasks [Krizhevsky et al., 2012, Sermanet et al., 2014], even approaching the human performance in some cases [He et al., 2016]. Other successful applications include speech recognition [Hinton et al., 2012], image generation [Radford et al., 2015], machine translation [Wu et al., 2016], and game playing [Mnih et al., 2015b, Silver et al., 2016b]. Despite the vast diversity of those applications, their underlying models are all based on the same multilayer neural network architecture. This universality of deep learning makes it a promising candidate for achieving true artificial intelligence (AI) that can match human intelligence. However, there are many elements in human intelligence where deep learning yet to provide a satisfactory solution. The objective of this thesis to advance deep learning techniques on three such elements: memory, communication, and intrinsic motivation. Next, we will discuss each of them in more details.

Memory

Memory is an essential element of intelligence that allows us to utilize our past experiences in decision making. There are several different types of memory that differ in their decay rate, capacity, and functions. The two main categories are short-term and long-term memories. Short-term memory decays fast and limited in capacity [Miller, 1956]. When we read a sentence, our short-term memory remembers its words, but it forgets them as soon as we understand the meaning of the sentence. In contrast, long-term memory has long-lasting effects and a large capacity. It further divides into implicit and explicit memories. Implicit memory is memories that form when we acquire a skill by repetition such as learning to ride a bicycle. On the contrary, explicit memory contains memories that can be consciously recalled such as our knowledge, or events in past. It helps us remember what we did yesterday, or the plot of a book we read.

There is a loose correspondence between memory types in brain and memories of deep learning models. A hidden layer activation of a recurrent neural network can be thought as a short-term memory since it has a limited capacity and decays fast with time. Model parameters can be considered as an implicit memory because they are long-term and form by repetitive training, but cannot be recalled by the model. However, when it comes to explicit memory, there is no corresponding memory mechanism in deep learning that adapts fast, long-term and can be selectively read. This is a real problem because the importance of such memory in an intelligent agent is apparent. For example, having a meaningful conversation requires an agent to remember its past exchanges.

Although simply recording past events is straightforward with computers, recalling a relevant piece of information from those records is the key challenge. The recently proposed Memory Network model [Weston et al., 2015] provided a partial solution with an external memory module where each memory element can be accessed individually with hard attention. However, training of this hard attention requires extra supervision about which memory should be accessed in what order, which is information not readily available in most cases. Our investigation shows that replacing this hard attention with soft attention makes the model end-to-end differentiable, so it can be trained with the back-propagation algorithm without extra supervision. This allows us to apply the model to a wide range of tasks from question answering to game playing where the memory access pattern is unknown.

Communication

Another fundamental aspect of intelligence is its ability to communicate with others. Communication is particularly useful in cooperative environments, where individuals need to behave as a group and coordinate for better performance. As applications of AI

spread, the number of intelligent agents acting in a common environment is expected to increase, making their coordination more important. One such application is a self-driving car. As more cars on the road become autonomous, it would become more beneficial that they communicate with each other. Informing their location and speed to nearby cars can help avoid accidents, especially in poor visibility. Furthermore, it might be even possible to remove traffic lights and let cars figure out a more efficient way to pass through junctions by coordinating through communication. Other applications that can benefit from communication include elevator control, factory robots and sensor networks.

The traditional approach to multi-agent communication is to supervise its content, or even prespecify the communication protocol. In the case of self-driving cars, for example, we can hard-code cars to communicate their location. However, this approach requires an expert knowledge and unlikely to scale to complex scenarios. In contrast, an emergent communication setting requires agents to invent their own communication protocol beneficial to their end goal. Although humans use highly structured languages for communication, here we focus on communication between AI agents without putting any restriction on the communication medium. This simplification allows us to concentrate on the learning part of communication. In particular, we are interested in whether a group of agents can *learn* to communicate from scratch to increase their performance under a cooperative task without any supervision on the communication protocol.

Our key finding is that using continuous vectors as a communication medium makes it differentiable, so we can use backpropagation to learn the communication protocol. When agents are controlled by individual neural networks, connecting them through continuous vectors essentially merges them into a single big neural network. However, this big neural network is invariant to the permutations of agents and can adapt to a

varying number of agents, which are crucial for processing set inputs. We test our model on diverse tasks from question answering to multi-agent cooperative games.

Intrinsic Motivation

The last component of intelligence that we investigate is its ability to learn about its environment without any external supervision. For example, babies learn to grasp and manipulate objects even if they are not explicitly rewarded for doing so. Such intrinsic motivation for learning would make it easier to train an intelligent agent because it reduces the required amount of external supervision, which is often expensive to obtain.

While there can be many forms of intrinsic motivation, we focus on one where an agent plays a game with itself. In the first half of this game, an agent plays a role of a "task generator" whose goal is to change something in its environment that is hard to replicate, but preferably does not require a lot of efforts. In the second half, the same agent plays a role of a "task executor" whose goal is to make the same change in the environment as the generator. There is another version of the game where the executor is asked to reverse the change made by the generator, which is more suitable for environments where all changes are reversible.

Since both the roles are constantly seeking to maximize their objectives, the task executor eventually learns the currently proposed task. In turn, this forces the task generator to find and propose a new task that the executor has not mastered yet. However, the new task is likely to be just outside the current capability of the executor, since the generator is incentivized to make the tasks simpler. This mechanism creates a curriculum training where the executor is trained on tasks that are not too hard or too easy.

The adversarial interplay between the two roles helps the agent to learn manipulate its environment in increasingly complex ways without any external supervision. We will

experimentally show that such knowledge about the environment facilitates the learning of a new supervised task.

Outline of the thesis

This thesis is organized as follows. In Chapter 2, we briefly review reinforcement learning and REINFORCE algorithm that used for training our models. We also cover several existing environments that are used as a test-bed in our experiments. In Chapter 3, we introduce a new environment that served as a sandbox for developing and testing many of our ideas in this thesis. Chapter 4 is about our memory architecture and its experimental results. Next, Chapter 5 introduces our communication model for multi-agent tasks. In Chapter 6, we discuss our intrinsic motivation method and demonstrate its effectiveness in a diverse set of environments. Note that each chapter has its own related work section. Finally, we conclude this thesis in Chapter 7.

Chapter 2

Background

In this chapter, we briefly review reinforcement learning and simple REINFORCE algorithm for training agents. Then we introduce two environments that we used in our experiments as a test-bed: a multi-agent environment StarCraft that used in the experiments of Chapter 4 and Chapter 6, and a continuous state environment RLLab that used in Chapter 6.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a framework for training an agent acting in an environment so it would receive a high reward. A core part of the agent is a policy π that maps an observation $O(s_t)$ of a state s_t to an action a_t

$$a_t = \pi(O(s_t)),$$

where t denotes time. We omit the observation function and write the policy as $\pi(s_t)$ for brevity. Next, the environment uses this action to update its state and returns a reward r_t

$$s_{t+1}, r_t = \text{Env}(s_t, a_t).$$

This interaction ends after some time $t = T$ when the environment reaches a terminal state, ending the current episode. The training objective in RL is to find a policy that maximizes the total expected reward of an episode

$$\underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_\pi \left[\sum_{t=1}^{T} r_t \right]. \tag{2.1}$$

Although there are a number of ways to solve Equation 2.1, we use a policy gradient approach in our experiments for its simplicity.

REINFORCE [Williams, 1992] is a policy gradient algorithm for training a policy π parameterized θ that outputs a probability distribution over all possible actions, from which the the next action a_t is sampled

$$a_t \sim \pi_\theta(a|s_t).$$

Although (2.1) is not differentiable with respect to θ, there is a workaround to obtain an unbiased estimate of its gradient. For simplicity, let us consider a single step episode

with action a and reward function $r(a)$. Then the gradient can be computed by

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta}\left[r(a)\right] &= \nabla_\theta \sum_a \pi_\theta(a|s) r(a) \\
&= \sum_a \nabla_\theta \pi_\theta(a|s) r(a) \\
&= \sum_a \pi_\theta(a|s) \frac{1}{\pi_\theta(a|s)} \nabla_\theta \pi_\theta(a|s) r(a) \\
&= \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) r(a) \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) r(a)\right]. \qquad (2.2)
\end{aligned}$$

This derivation allows us to sample an unbiased estimate of the gradient by simply following the policy. Then we can perform a stochastic gradient ascent on those samples to maximize the expected reward. If we extend Equation 2.2 to multiple steps, the update rule of parameters θ becomes

$$\Delta\theta = \sum_{t=1}^{T} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{i=t}^{T} r_i\right] \qquad (2.3)$$

$$\theta \leftarrow \theta + \mu \Delta\theta, \qquad (2.4)$$

where μ is a learning rate.

Although (2.3) is an unbiased gradient estimate, its high variance can slow down the training progress. A simple way to reduce this variance is to add a baseline function $b(s_t)$ that predicts the future expected reward of the policy π_θ starting at a state s_t. If the policy π_θ is a neural network, the same network can be used for $b(s_t)$ by adding another

linear layer with a scalar output on top of the last hidden layer. The final update rule is

$$\Delta\theta = \sum_{t=1}^{T} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{i=t}^{T} r_i - b_\theta(s_t) \right) - \alpha \nabla_\theta \left(\sum_{i=t}^{T} r_i - b_\theta(s_t) \right)^2 \right]. \quad (2.5)$$

Here the hyperparameter α is for controlling the relative importance of the second term, which is forces $b(s_t)$ to make an accurate prediction. It is important to note that b_θ in the first term acts like a fixed value, and we do note pass a gradient through it.

2.2 StarCraft environment

StarCraft: Brood War™[1] is a popular real-time strategy game where a player controls multiple units to build various types of structures and units, and eventually use them in combat against an enemy. As an RL platform, it offers rich complexity in both fast-paced precise control and long-term planning. While efficient resource mining and fighting enemy units depend on rapid precise actions, a long-term plan on which technology to develop is essential for winning. For a survey on AI for Real Time Strategy (RTS) games, and especially for StarCraft, see [Ontanón et al., 2013].

Since playing an entire game of StarCraft is out-of-reach for current learning algorithms, we consider here two subtasks that focus on different aspects of the game: 1) an initial development stage where a player collects resources and builds units, and 2) a later stage where units fight against enemy units.

We use TorchCraft [Synnaeve et al., 2016] to connect the game to our Torch framework [Collobert et al., 2011]. This allows us to receive a game state and send orders, enabling us to do a RL loop. A game state is a set of descriptions, each corresponding

[1] StarCraft and Brood War are registered trademarks of Blizzard Entertainment, Inc.

to a single unit present in the game, including buildings and enemy units. A description includes the unit's position and other relevant attributes such as its type, health and cooldown counter. In addition, global game attributes such as the amount of resources are also included in a game state. In return, we can send individual orders to each unit that containing an action name and its relevant arguments. Note that the set of possible actions is not the same for different unit types.

Since there are usually multiple units to control, a policy must output an action for each of them at every time step. A simple solution is to independently control all units with a single policy

$$a_t^i = \pi(s_t^i, \hat{s}_t) \tag{2.6}$$

Here s_t^i is an observation specific to the i-th unit that containing the descriptions of all nearby units. The global observation $\hat{s}$ is the description of global game attributes.

Next, we describe two tasks created in the StarCraft environment.

2.2.1 Development task

The setup of this task is similar to the beginning of a standard StarCraft game of the Terran race, where an agent controls multiple worker units to mine, construct buildings, and train new units, but without enemies to fight. The environment starts with four worker units (SCVs), who can move around, mine nearby minerals and construct new buildings. In addition, the agent controls the command center which can train new workers. Unit types are limited to SCV, command center, supply depot, barracks, and marines. See Figure 2.1 for relations between different units and their actions.

The task objective is to build as many marine units as possible in a given time. To do this, an agent must follow a specific sequence of operations: (i) mine minerals using

Figure 2.1: Different types of unit in the development task. The arrows represent possible actions (excluding movement actions) by an unit, and corresponding numbers shows (blue) amount of minerals and (red) time steps needed to complete. The units under agent's control are outlined by a green border.

the workers; (ii) having accumulated sufficient mineral supply, build a barracks and (iii) once the barracks is complete, train marine units out of it. Optionally, an agent can train a new worker for faster mining, or build a supply depot to accommodate more units. When an episode ends after 200 steps (1 step = 23 frames, so little over 3 minutes), the agent gets rewarded +1 for each marine it has built.

The task is challenging for several reasons. First, the agent has to find an optimal mining pattern (concentrating on a single mineral or mining a far away mineral is inefficient). Then, it has to produce the optimal number of workers and barrack at the right timing for better efficiency. Additionally, a supply depot needs to be built when the number of units reaches the current limit.

The local observation s_t^i covers a 64×64 area surrounding the i-th unit with a resolution of 4, so its spatial dimension is actually 8×8. It contains the type and status (e.g. idle, mining, training, etc.) of every unit in this area, including the i-th unit itself. The global observation $\hat{s}_t$ contains the number of units and accumulated minerals in the

Action ID	SCV	Command center	Barraks
1	move to right	train a SCV	train a marine
2	move to left	no-op	no-op
3	move to top	no-op	no-op
4	move to bottom	no-op	no-op
5	mine minerals	no-op	no-op
6	build a barracks	no-op	no-op
7	build a supply depot	no-op	no-op

Table 2.1: Action space of different unit types in the development task. No-op=no operation means no action is sent to the game.

game

$$\hat{s}_t = \{\lfloor N_{\text{ore}}/25 \rfloor, N_{\text{SCV}}, N_{\text{Barrack}}, N_{\text{SupplyDepot}}, N_{\text{Marines}}\}. \tag{2.7}$$

We divide the ore amount by 25 and round it because it can take very value compared to other numbers.

We refer to units that perform an action as a “active” unit (shown with a green outline in Figure 2.1). This includes worker units (SCVs), the command center, and barrack. Their possible actions are shown in Table 2.1. Although some units have fewer actions, we pad them with “no-op” to have the same number so a single neural network can control each of them. When the no-op action is chosen, nothing is sent to StarCraft for that unit and its previous action is not interrupted.

The move actions simply order the unit to move 32 pixels in the chosen direction. However, this does not guarantee the unit will move 32 pixels by the next step, because the movement can be interrupted by the next action. The more complex actions “mine minerals”, “build a barracks”, “build a supply depot” have the following semantics, respectively: mine the closest mineral (automatically transports to the command center), build a barracks at the current position, build a supply depot on the current position.

Some actions are ignored under certain conditions: “mining” action is ignored if

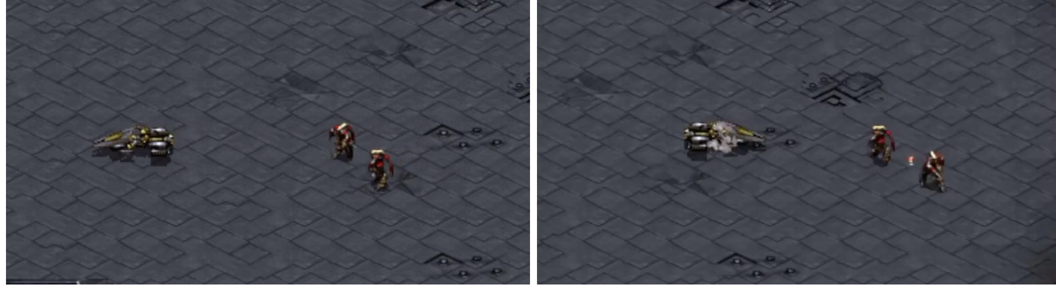

Figure 2.2: A simple combat scenario in StarCraft requiring a kiting tactic. To win the battle, the agent (the larger unit) has to alternate between (left) fleeing the stronger enemies while its weapon recharges, and (right) returning to attack them once it is ready to fire.

the distance to the closest mineral is greater than 12 pixels; “building” and “training” actions are ignored if there are not enough resources; the actions that create a new SCV or a marine will be ignored if the current capacity has reached the limit. Separately, the actions to create an active unit are ignored if the number of active units has reached the limit of 10. Also “build” actions will be ignored if there is not enough room to build at the unit’s location.

2.2.2 Combat task

In this subtask, we focus on “micro-management” scenarios where we only have a limited number of combat units to fight against a similar enemy army. To win in this task, it is important to control the units in a rapid and precise way to increase their effectiveness. Each unit starts with a certain health point, which is reduced every time it is hit by an enemy. If the health point reaches zero, that unit dies. The same is true for enemy units, and a team left with no alive units loses. Note that shooting ranges and hit points are different for different unit types. Also, there is a cooldown period after each shot during which the unit cannot shoot again.

We consider the following three tasks:

- **Kiting** (Terran Vulture vs Protoss Zealot): a match-up where an agent controls a weakly armored fast ranged unit, against a more powerful but melee ranged unit. To win, our unit needs to alternate fleeing the opponents and shooting at them when its weapon has cooled down.

- **Kiting hard** (Terran Vulture vs 2 Protoss Zealots): same as above but with 2 enemies, as shown in Figure 2.2.

- **2 vs 2** (Terran Marines): a symmetric match-up where both teams have 2 ranged units. An useful tactic here is concentrating fire on a single enemy unit.

In all the tasks, the local observation s_t^i covers an area of 256×256, but with multi-resolution encoding (coarser going further from the unit we control) to reduce its dimension. For each unit in this area, the observation contains its type, ID, health, and cooldown counter. If the i-th unit is currently attacking an enemy unit, that enemy unit's ID is also included in s_t^i. There is no global observation $\hat{s}_t$ in this task. All units have the following actions: movements in 4 cardinal directions (see Section 2.2.1 for details), and K attack actions each corresponding to a single enemy unit (i.e. there are K enemy units). We take an action every 8 frames (the atomic time unit @ 24 frames/sec).

2.3 RLLab environments

RLLab [Duan et al., 2016] is a Python package that offers an unified interface to many tasks across several different environments. Box 2D and Mujoco [Todorov et al., 2012] are two such environments where the tasks are in a continuous state space. While Box 2D is a small 2D environment with a simple physics simulator, Mujoco is a 3D environment with a powerful physics engine that can simulate complex dynamics.

Both the environments have a continuous action space, but we discretize them in our experiments. This is done by dividing the action range into K uniformly sized bins, thus turning it into a discrete action with K choices. When an action $k \leq K$ is chosen by a policy, we replace it with the mean value of the corresponding bin before sending it to the environment. When the action space is multi-dimensional, we discretize each dimension independently, resulting in multiple discrete actions. On the policy side, we simply add multiple action heads where each head is a linear layer followed by a softmax.

Chapter 3

MazeBase: A Sandbox for Learning from Games

This chapter introduces MazeBase: an environment for simple 2D games on a grid, designed as a sandbox for reinforcement learning approaches for reasoning and planning. Within it, we create 10 simple games embodying a range of algorithmic tasks (e.g. if-then statements or set negation). Additionally, we also create combat games involving multiple units. A procedural random generation makes those tasks non-trivial, but a flexible curriculum can be enabled to support the learning.

3.1 Introduction

Games have had an important role in artificial intelligence research since the inception of the field. Core problems such as search and planning can be explored naturally in the context of chess or Go [Bouzy and Cazenave, 2001, Silver et al., 2016b]. More recently, they have served as a test-bed for machine learning approaches [Perez et al., 2014]. For example, Atari games [Bellemare et al., 2013] have been played using neural models with reinforcement learning [Mnih et al., 2013, Guo et al., 2014a, Mnih et al., 2015a]. The GVG-AI competition [Perez et al., 2014] uses a suite of 2D arcade games to compare planning and search methods.

In this chapter we introduce the MazeBase game environment, which complements existing frameworks in several important ways.

- The emphasis is on learning to understand the environment, rather than on testing algorithms for search and planning. The framework deliberately lacks any simulation facility, and hence agents cannot use a search method to determine the next action unless they can predict future game states themselves. On the other hand, game components are meant to be reused in different games, giving models the opportunity to comprehend the function of, say, a water tile. Nor are rules of the games provided to the agent, instead they must be learned through exploration of the environment.
- The environment has been designed to allow programmatic control over the game difficulty. This allows the automatic construction of curricula [Bengio et al., 2009a], which we show to be important for training complex models.
- Our games are based around simple algorithmic reasoning, providing a natural path for exploring complex abstract reasoning problems in a language-based

grounded setting. This contrasts with most games that were originally designed for human enjoyment, rather than any specific task. It also differs from the recent surge of work on learning simple algorithms [Zaremba and Sutskever, 2015, Graves et al., 2014, Zaremba et al., 2015], which lack grounding.

- Despite the 2D nature of the environment, we prefer to use a text-based, rather than pixel-based, representation. This provides an efficient but expressive representation without the overhead of solving the perception problem inherent in pixel-based representations. It easily allows for different task specifications and easy generalization of the models to other game settings. We demonstrate this in Chapter 4 by training models in MazeBase and then successfully evaluating them on StarCraft. See [Mikolov et al., 2015] for further discussion.

Using the environment, we introduce a set of 10 simple reasoning games and several combat games. MazeBase is an open-source platform, implemented using Torch and can be downloaded from `http://github.com/facebook/MazeBase`.

3.2 Environment

Each game is played in a 2D rectangular grid. In the specific examples below, the dimensions range from 3 to 10 on each side, but of course these can be set however the user likes. Each location in the grid can be empty, or may contain one or more items. An agent can move in each of the four cardinal directions, assuming no item blocks the agents path. The items in the game are:

- **Block:** an impassible obstacle that does not allow the agent to move to that grid location.

- **Water:** the agent may move to a grid location with water, but incurs an additional cost of (fixed at -0.2 in the games below) for doing so.
- **Switch:** a switch can be in one of m states, which we refer to as colors. The agent can toggle through the states cyclically by a toggle action when it is at the location of the switch.
- **Door:** a door has a color, matched to a particular switch. The agent may only move to the door's grid location if the state of the switch matches the state of the door.
- **PushableBlock:** This block is impassable, but can be moved with a separate "push" actions. The block moves in the direction of the push, and the agent must be located adjacent to the block opposite the direction of the push.
- **Corner:** This item simply marks a corner of the board.
- **Goal:** depending on the task, one or more goals may exist, each named individually.
- **Info:** these items do not have a grid location, but can specify a task or give information necessary for its completion.

The environment is presented to the agent as a list of sentences, each describing an item in the game. For example, an agent might see "Block at [-1,4]. Switch at [+3,0] with blue color. Info: change switch to red." Such representation is compatible with the format of the bAbI tasks, introduced in [Weston et al., 2016]. However, note that we use egocentric spatial coordinates (e.g. the goal G1 in Figure 3.1(left) is at coordinates [+2,0]), meaning that the environment updates the locations of each object

after an action. Furthermore, for tasks involving multiple goals, the environment assist the agent by automatically sets a flag on visited goals.

The environments are generated randomly with some distribution on the various items. For example, we usually specify a uniform distribution over height and width (between 5 and 10 for the experiments reported here), and a percentage of wall blocks and water blocks (each range randomly from 0 to 20%).

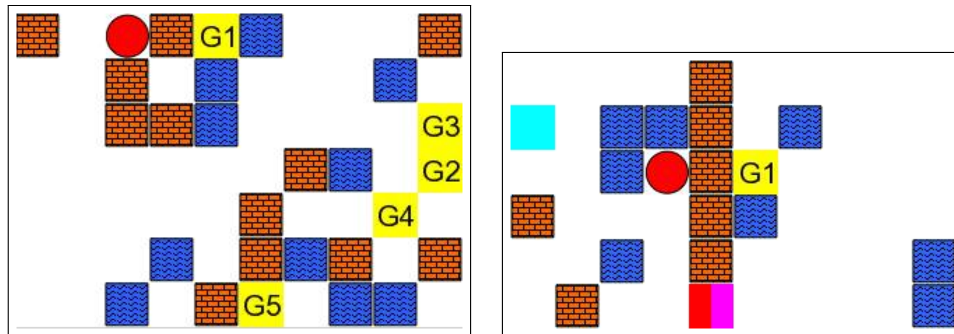

Figure 3.1: Examples of Multigoal (left) and Light Key (right) tasks. Note that the layout and dimensions of the environment varies between different instances of each task (i.e. the location and quantity of walls, water and goals all change). The agent is shown as a red blob and the goals are shown in yellow. For LightKey, the switch is show in cyan and the door in magenta/red (toggling the switch will change the door's color, allowing it to pass through).

3.3 Tasks

Although our game world is simple, it allows for a rich variety of tasks. In this work, we consider two types of tasks: reasoning tasks and combat tasks.

3.3.1 Reasoning Tasks

Here we explore tasks that require different algorithmic components first in isolation and then in combination. These components include:

- **Set operations:** iterating through a list of goals, or negation of a list, i.e. all items *except* those specified in a list.

- **Conditional reasoning:** *if-then* statements, *while* statements.

- **Basic Arithmetic:** comparison of two small numbers.

- **Manipulation:** altering the environment by toggling a **Switch**, or moving a **PushableBlock**.

These were selected as key elements needed for complex reasoning tasks, although we limit ourselves here to only combining a few of them in any given task. We note that most of these have direct parallels to the bAbI tasks [Weston et al., 2016] (except for **Manipulation** which is only possible in our non-static environment). We avoid tasks that require unbounded loops or recursion, as in [Joulin and Mikolov, 2015], and instead view "algorithms" more in the vein of following a recipe from a cookbook. In particular, we want our agent to be able to follow directions; the same game world may host multiple tasks, and the agent must decide what to do based on the "Info" items. As we demonstrate in Section 4.6, standard neural models find this to be challenging.

In all of the tasks, the agent incurs a penalty of 0.1 for each action it makes, this encourages the agent to finish the task promptly. In addition, stepping on a Water block incurs an additional penalty of 0.2. For most games, a maximum of 50 actions are allowed in each episode. The tasks define extra penalties and conditions for the game to end.

- **Multigoals:** In this task, the agent is given an ordered list of goals as "Info", and needs to visit the goals in that order. The number of goals ranges from 2 to 6, and the number of "active" that the agent is required to visit ranges from 1 to 3

goals. The agent is not given any extra penalty for visiting a goal out of order, but visiting a goal before its turn does not count towards visiting all goals. The game ends when all goals are visited. This task involves the algorithmic component of iterating over a list.

- **Exclusion:** The "Info" in this game specifies a list of goals to avoid. The agent should visit all other unmentioned goals. The number of all goals ranges from 2 to 6, but the number of active goals ranges from 1 to 3. As in the Conditional goals game, the agent incurs a 0.5 penalty when it steps on a forbidden goal. This task combines **Multigoals** (iterate over set) with set negation.

- **Conditional Goals:** In this task, the destination goal is conditional on the state of a switch. The "Info" is of the form "go to goal g_i if the switch is colored c_j, else go to goal g_l". The number of colors ranges from 2 to 6 and the number of goals from 2 to 6. Note that there can be more colors than goals or more goals than colors. The task concludes when the agent reaches the specified goal; in addition, the agent incurs a 0.2 penalty for stepping on an incorrect goal, in order to encourage it to read the info (and not just visit all goals). The task requires conditional reasoning in the form of an *if-then* statement.

- **Switches:** In this task, the game has a random number of switches on the board. The agent is told via the "Info" to toggle all the switches to the same color, but the agent has the choice of color. To get the best reward, the agent needs to solve a (very small) traveling salesman problem. The number of switches ranges from 1 to 5 and the number of colors from 1 to 6. The task finishes when the switches are correctly toggled. There are no special penalties in this task. The task instantiates a form of *while* statement.

- **Light Key:** In this game, there is a switch and a door in a wall of blocks. The agent should navigate to a goal which may be on the other side of a wall of blocks. If the goal is on the same side of the wall as the agent, it should go directly there; otherwise, it needs to move to and toggle the switch to open the door before going to the goal. There are no special penalties in this game, and the game ends when the agent reaches the goal. This task combines *if-then* reasoning with environment manipulation.

- **Goto:** In this task, the agent is given an absolute location on the grid as a target. The game ends when the agent visits this location. Solving this task requires the agent to convert from its own egocentric coordinate representation to absolute coordinates. This involves comparison of small numbers.

- **Goto Hidden:** In this task, the agent is given a list of goals with absolute coordinates, and then is told to go to one of the goals by the goal's name. The agent can not see the goal's location as an item on the map, instead it must read this from the info items (and convert from absolute to relative coordinates). The number of goals ranges from 1 to 6. The task also involves very simple comparison operation.

- **Push block:** In this game, the agent needs to push a Pushable block so that it lays on top of a switch. Considering the large number of actions needed to solve this task, the map size is limited between 3 and 7, and the maximum block and water percentage is reduced to 10%. The task requires manipulation of the environment.

- **Push block cardinal:** In this game, the agent needs to push a Pushable block so that it is on a specified edge of the maze, e.g. the left edge. Any location along the edge is acceptable. The same limitation as the Push Block game is applied.

- **Blocked door:** In this task, the agent should navigate to a goal which may lie on the opposite side of a wall of blocks, as in the Light Key game. However, a Pushable block blocks the gap in the wall instead of a door. This requires *if-then* reasoning, as well as environment manipulation.

For each task, we compute offline the optimal solution. For some of the tasks, e.g. **Multigoals**, this involves solving a traveling salesman problem (which for simplicity is done approximately). This provides an upper bound for the reward achievable. This is for comparison purposes only (i.e. it is not to be used for training an agent).

All these are Markovian. Nevertheless, the tasks are not at all simple; although the environment can easily be used to build non-Markovian tasks, we find that the solving these tasks without the agent having to reason about its past actions is already challenging. Note that building a new task is easy in the MazeBase environment, indeed many of those above are implemented in a few hundred lines of code.

3.3.2 Combat tasks

Next, we use MazeBase to implement several simple games similar to the combat tasks of StarCraft introduced in Section 2.2.2. This allows us to train an agent on these games and then test them on Starcraft. Our goal here is not to assess the ability of the agent to generalize, but rather whether we can make the game in our environment close to its counterpart in StarCraft, to show that the environment can be used for prototyping scenarios where training on an actual game may be technically challenging.

In MazeBase, the kiting scenario consists of a standard maze where an agent aims to kill up to two enemy bots. After each shot, the agent or enemy is prevented from firing again for a small time interval (cooldown). We introduce an imbalance by (i) allowing

the agent to shoot farther than the enemy bot(s) and (ii) giving the agent significantly less health than the bot(s); and (iii) by allowing the enemy bot(s) to shoot more frequently than the agent (shorter cooldown). The agent has a shot range of 7 squares, and the bots have a shot range of 4 squares. The enemy bot(s) moves (on average) at .6 the speed of the agent. This is accomplished by rolling a "fumble" each time the bot tries to move with probability .4. The agent has health chosen uniformly between 2 and 4, and the enemy(s) have health uniformly distributed between 4 and 11. The enemy can shoot every 2 turns, and the agent can shoot every 6 turns. The enemy follows a heuristic of attacking the agent when in range and its cooldown is 0, and attempting to move towards the agent when it is closer than 10 squares away, and ignoring when farther than 10 squares.

The 2 vs. 2 scenario is modeled in MazeBase with two agents, each of which have 3 health points, and two bots, which have hitpoints randomly chosen from 3 or 4. Agents and bots have a range of 6 and a cooldown of 3. The bots use a heuristic of attacking the closest agent if they have not attacked an agent before, and continuing to attack and follow that agent until it is killed. In both the kiting and 2 vs. 2 scenarios, we randomly add noise to the agents' inputs to account for the fact that it will encounter a new vocabulary when playing StarCraft. That is, 10% of the time, the numerical value of the enemies' or agents' health, cooldown, etc is taken to be a random value. The reward signal is based on the difference between the armies hit points, and win or loss of the overall battle.

3.4 Curriculum

A key feature of our environment is the ability to programmatically vary all the properties of a given game. We use this ability to construct instances of each game whose difficulty is precisely specified. These instances can then be shaped into a curriculum for training [Bengio et al., 2009a]. As we later demonstrate in Section 4.6.2, this is very important for avoiding local minima and helps to learn superior models.

Each game has many variables that impact the difficulty. Generic ones include: maze dimensions (height/width) and the fraction of blocks & water. For switch-based games (**Switches**, **Light Key**) the number of switches and colors can be varied. For goal based games (**Multigoals**,**Conditional Goals**, **Exclusion**), the variables are the number of goals (and active goals). For the combat game **Kiting**, we vary the number of agents & enemies, as well as their speed and their initial health. The curriculum is specified by upper and lower success thresholds T_u and T_l respectively. If the success rate of the agent falls outside the $[T_l, T_u]$ interval, then the difficulty of the generated games is adjusted accordingly. Each game is generated by uniformly sampling each variable that affects the difficulty over some range. The upper limit of this range is adjusted, depending on which of T_l or T_u is violated. Note that the lower limit remains unaltered, thus the easiest game remains at the same difficulty. For the last third of training, we expose the agent to the full range of difficulties by setting the upper limit to its maximum preset value.

3.5 Conclusion

The MazeBase environment allows easy creation of games and precise control over their behavior. It is designed to be a testbed for developing methods for training agents

that can learn and reason about their world, and so it encourages sharing components across multiple games. We used it to devise a set of 10 simple games embodying algorithmic components. The flexibility of the environment enabled curricula to be created for each game which can aid the learning.

To further show the flexibility of the environment, we showed that with a few hundred lines of code inside MazeBase, we can build simulations of StarCraft micro-combat scenarios. We can use these to help develop model architectures that generalize to StarCraft. We believe that MazeBase will be useful for pushing the boundaries of what our current methods can do. The proposed environment enables a rapid iteration of new models and learning modalities, and building of new games to evaluate these.

Chapter 4

End-to-end Memory Network

In this chapter, we introduce a neural network with a recurrent attention model over a possibly large external memory. The architecture is a form of Memory Network [Weston et al., 2015] but unlike the model in that work, it is trained end-to-end, and hence requires significantly less supervision during training, making it more generally applicable in realistic settings. It can also be seen as an extension of RNNsearch [Bahdanau et al., 2015] to the case where multiple computational steps (hops) are performed per output symbol. The flexibility of the model allows us to apply it to tasks as diverse as (synthetic) question answering [Weston et al., 2016] and to language modeling. For the former our approach is competitive with Memory Networks, but with less supervision. For the latter, on the Penn TreeBank and Text8 datasets our approach demonstrates comparable performance to RNNs and LSTMs. In both cases we show that the key concept of multiple computational hops yields improved results. Additional experiments on reinforcement learning tasks further demonstrate it flexibility.

4.1 Introduction

Two grand challenges in artificial intelligence research have been to build models that can make multiple computational steps in the service of answering a question or completing a task, and models that can describe long term dependencies in sequential data.

Recently there has been a resurgence in models of computation using explicit storage and a notion of attention [Weston et al., 2015, Graves et al., 2014, Bahdanau et al., 2015]; manipulating such a storage offers an approach to both of these challenges. In [Weston et al., 2015, Graves et al., 2014, Bahdanau et al., 2015], the storage is endowed with a continuous representation; reads from and writes to the storage, as well as other processing steps, are modeled by the actions of neural networks.

In this chapter, we present a novel recurrent neural network (RNN) architecture where the recurrence reads from a possibly large external memory multiple times before outputting a symbol. Our model can be considered a continuous form of the Memory Network implemented in [Weston et al., 2015]. The model in that work was not easy to train via backpropagation, and required supervision at each layer of the network. The continuity of the model we present here means that it can be trained end-to-end from input-output pairs, and so is applicable to more tasks, i.e. tasks where such supervision is not available, such as in language modeling or realistically supervised question answering tasks. Our model can also be seen as a version of RNNsearch [Bahdanau et al., 2015] with multiple computational steps (which we term "hops") per output symbol. We will show experimentally that the multiple hops over the long-term memory are crucial to good performance of our model on these tasks, and that training the memory representation can be integrated in a scalable manner into our end-to-end neural network

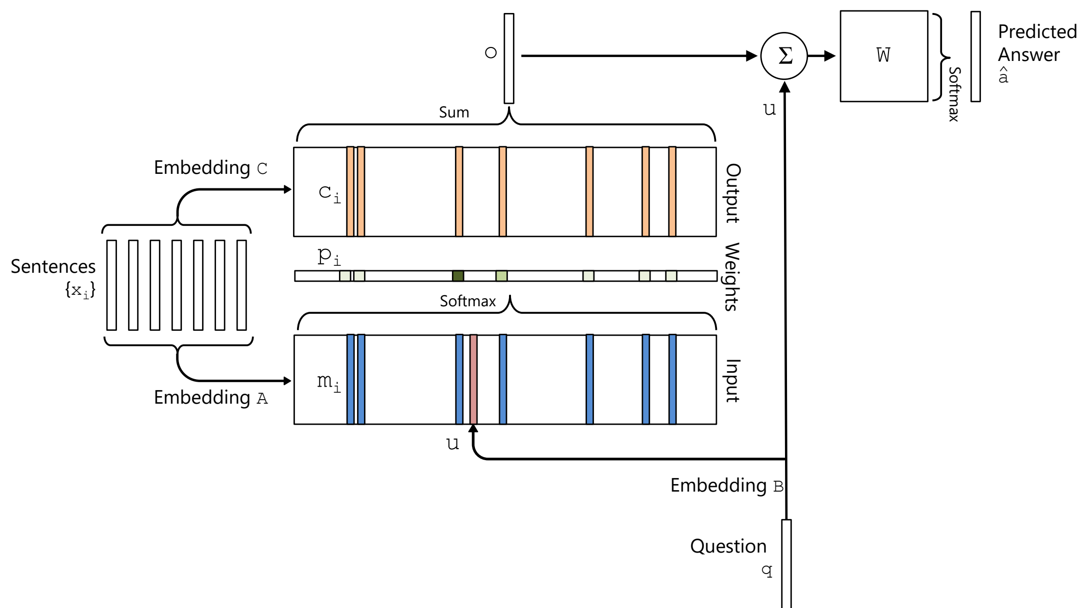

Figure 4.1: A single layer version of our model.

model. Additionally, we will also show that the same model can applied to reinforcement learning tasks in MazeBase and StarCraft environments. Finally, we propose an extension of the model that can write to the memory, which allows us to apply it to set-to-set tasks such as sorting.

4.2 Approach

Our model takes a discrete set of inputs $x_1, ..., x_n$ that are to be stored in the memory, a query q, and outputs an answer a. Each of the x_i, q, and a contains symbols coming from a dictionary with V words. The model writes all x to the memory up to a fixed buffer size, and then finds a continuous representation for the x and q. The continuous representation is then processed via multiple hops to output a. This allows backpropagation of the error signal through multiple memory accesses back to the input during

training.

4.2.1 Single Layer

We start by describing our model in the single layer case, which implements a single memory hop operation. We then show it can be stacked to give multiple hops in memory.

4.2.1.1 Input memory representation

Suppose we are given an input set $x_1, .., x_i$ to be stored in memory. The entire set $\{x_i\}$ is converted into memory vectors $\{m_i\}$ of dimension d computed by embedding each x_i in a continuous space, in the simplest case, using an embedding matrix A (of size $d \times V$). The query q is also embedded (again, in the simplest case via another embedding matrix B with the same dimensions as A) to obtain an internal state u. In the embedding space, we compute the match between u and each memory m_i by taking the inner product followed by a softmax:

$$p_i = \text{Softmax}(u^T m_i), \tag{4.1}$$

where $\text{Softmax}(z_i) = e^{z_i} / \sum_j e^{z_j}$. Defined in this way p is a probability vector over the inputs.

4.2.1.2 Output memory representation

Each x_i has a corresponding output vector c_i (given in the simplest case by another embedding matrix C). The response vector from the memory o is then a sum over the

transformed inputs c_i, weighted by the probability vector from the input:

$$o = \sum_i p_i c_i. \tag{4.2}$$

Because the function from input to output is smooth, we can easily compute gradients and back-propagate through it. Other recently proposed forms of memory or attention take this approach, notably [Bahdanau et al., 2015] and [Graves et al., 2014], see also [Gregor et al., 2015].

4.2.1.3 Generating the final prediction

In the single layer case, the sum of the response vector o and the input embedding u is then passed through a final weight matrix W (of size $V \times d$) and a softmax to produce the predicted label:

$$\hat{a} = \text{Softmax}(W(o + u)) \tag{4.3}$$

The overall model is shown in Figure 4.1. During training, all three embedding matrices A, B and C, as well as W are jointly learned by minimizing a standard cross-entropy loss between $\hat{a}$ and the true label a. Training is performed using stochastic gradient descent (see Section 4.4.2 for more details).

4.2.2 Multiple Layers

We now extend our model to handle K-hop operations. The memory layers are stacked in the following way:

- The input to layers above the first is the sum of the output o^k and the input u^k from

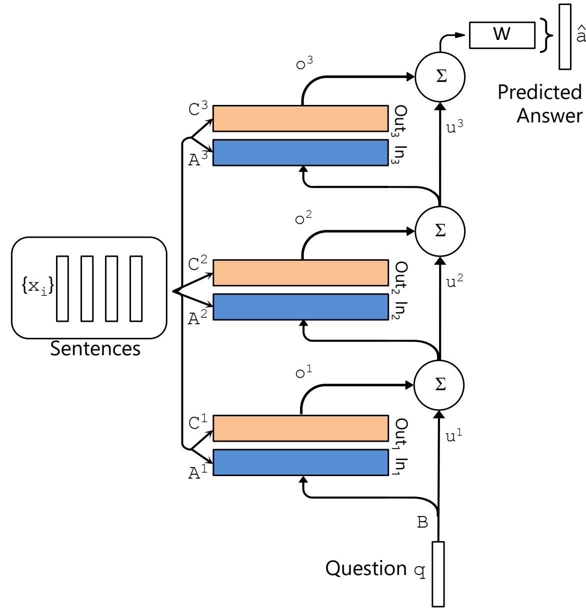

Figure 4.2: A three layer version of our model. In practice, we can constrain several of the embedding matrices to be the same (see Section 4.2.2).

layer k (different ways to combine o^k and u^k are proposed later):

$$u^{k+1} = u^k + o^k. \tag{4.4}$$

- Each layer has its own embedding matrices A^k, C^k, used to embed the inputs $\{x_i\}$. However, as discussed below, they are constrained to ease training and reduce the number of parameters.
- At the top of the network, the input to W also combines the input and the output of the top memory layer: $\hat{a} = \text{Softmax}(Wu^{K+1}) = \text{Softmax}(W(o^K + u^K))$.

We explore two types of weight tying within the model:

1. **Adjacent:** the output embedding for one layer is the input embedding for the one

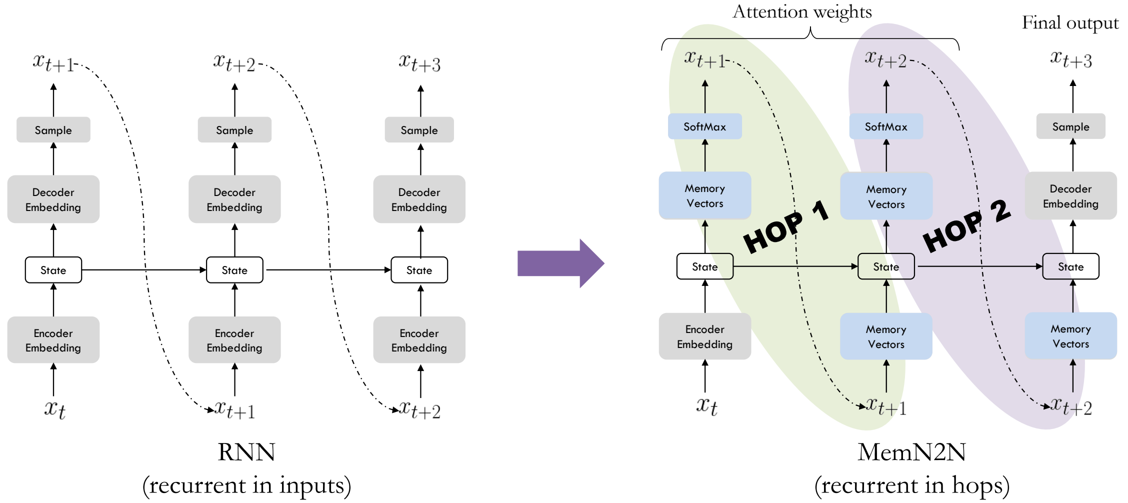

Figure 4.3: A recurrent view of MemN2N.

above, i.e. $A^{k+1} = C^k$. We also constrain (a) the answer prediction matrix to be the same as the final output embedding, i.e $W^T = C^K$, and (b) the question embedding to match the input embedding of the first layer, i.e. $B = A^1$.

2. **Layer-wise (RNN-like):** the input and output embeddings are the same across different layers, i.e. $A^1 = A^2 = ... = A^K$ and $C^1 = C^2 = ... = C^K$. We have found it useful to add a linear mapping H to the update of u between hops; that is, $u^{k+1} = Hu^k + o^k$. This mapping is learnt along with the rest of the parameters and used throughout our experiments for layer-wise weight tying.

A three-layer version of our memory model is shown in Figure 4.2. Overall, it is similar to the Memory Network model in [Weston et al., 2015], except that the hard max operations within each layer have been replaced with a continuous weighting from the softmax.

Note that if we use the layer-wise weight tying scheme, our model can be cast as a traditional RNN where we divide the outputs of the RNN into *internal* (blue boxes)

and *external* (gray boxes) outputs as shown in Figure 4.3. Emitting an internal output corresponds to considering a memory, and emitting an external output corresponds to predicting a label. From the RNN point of view, u in Figure 4.2 and Equation 4.4 is a hidden state, and the model generates an internal output p (attention weights in Figure 4.1 using A. The model then ingests p using C, updates the hidden state, and so on.[1] Here, unlike a standard RNN, we explicitly condition on the outputs stored in memory during the K hops, and we keep these outputs soft, rather than sampling them. Thus our model makes several computational steps before producing an output meant to be seen by the "outside world".

4.3 Related Work

A number of recent efforts have explored ways to capture long-term structure within sequences using RNNs or LSTM-based models [Chung et al., 2014, Graves, 2013, Koutník et al., 2014, Mikolov et al., 2014, Hochreiter and Schmidhuber, 1997, Atkeson and Schaal, 1995]. The memory in these models is the state of the network, which is latent and inherently unstable over long timescales. The LSTM-based models address this through local memory cells which lock in the network state from the past. In practice, the performance gains over carefully trained RNNs are modest (see [Mikolov et al., 2014]). Our model differs from these in that it uses a global memory, with shared read and write functions. However, with layer-wise weight tying our model can be viewed as a form of RNN which only produces an output after a fixed number of time steps (corresponding to the number of hops), with the intermediary steps involving memory

[1] Note that in this view, the terminology of input and output from Figure 4.1 is flipped - when viewed as a traditional RNN with this special conditioning of outputs, A becomes part of the output embedding of the RNN and C becomes the input embedding.

input/output operations that update the internal state.

Some of the very early work on neural networks [Steinbuch and Piske, 1963, Taylor, 1959] considered a memory that performed nearest-neighbor operations on stored input vectors and then fit parametric models to the retrieved sets. This has similarities to a single layer version of our model.

Subsequent work in the 1990's explored other types of memory [Pollack, 1991, Das et al., 1992, Mozer and Das, 1993]. For example, [Das et al., 1992] and [Mozer and Das, 1993] introduced an explicit stack with push and pop operations which has been revisited recently by [Joulin and Mikolov, 2015, Grefenstette et al., 2015] in the context of an RNN model.

Closely related to our model is the Neural Turing Machine [Graves et al., 2014], which also uses a continuous memory representation. The NTM memory uses both content and address-based access, unlike ours which only explicitly allows the former, although the temporal features that we will introduce in Section 4.4.1 allow a kind of address-based access. However, in part because we always write each memory sequentially, our model is somewhat simpler, not requiring operations like sharpening. Furthermore, we apply our memory model to textual reasoning tasks, which qualitatively differ from the more abstract operations of sorting and recall tackled by the NTM.

Our model is also related to [Bahdanau et al., 2015]. In that work, a bidirectional RNN based encoder and gated RNN based decoder were used for machine translation. The decoder uses an attention model that finds which hidden states from the encoding are most useful for outputting the next translated word; the attention model uses a small neural network that takes as input a concatenation of the current hidden state of the decoder and each of the encoders hidden states. A similar attention model is also used in [Xu et al., 2015] for generating image captions. Our "memory" is analogous to their

attention mechanism, although [Bahdanau et al., 2015] is only over a single sentence rather than many, as in our case. Furthermore, our model makes several hops on the memory before making an output; we will see below that this is important for good performance. There are also differences in the architecture of the small network used to score the memories compared to our scoring approach; we use a simple linear layer, whereas they use a more sophisticated gated architecture.

We will apply our model to language modeling, an extensively studied task. [Goodman, 2001] showed simple but effective approaches which combine n-grams with a cache. [Bengio et al., 2003] ignited interest in using neural network based models for the task, with RNNs [Mikolov, 2012] and LSTMs [Hochreiter and Schmidhuber, 1997,Sundermeyer et al., 2012] showing clear performance gains over traditional methods. Indeed, the current state-of-the-art is held by variants of these models, for example very large LSTMs with Dropout [Zaremba et al., 2014] or RNNs with diagonal constraints on the weight matrix [Mikolov et al., 2014]. With appropriate weight tying, our model can be regarded as a modified form of RNN, where the recurrence is indexed by memory lookups to the word sequence rather than indexed by the sequence itself.

After publication of our work, there has been a surge of works on using an external memory with soft-attention for variety of tasks. Extensions of our model have been applied to question answering on children books [Hill et al., 2016], and large-scale knowledge bases [Miller et al., 2016]. A similar memory architecture also used in sentiment analysis, part-of-speech tagging [Kumar et al., 2016], and visual question answering [Xiong et al., 2016a]. In [Vaswani et al., 2017], an attention mechanism is used in machine translation, outperforming traditional sequence-to-sequence approaches. In reinforcement learning, an external memory is proven to be useful as an episodic memory [Oh et al., 2016], and as a spatial memory [Parisotto and Salakhutdinov, 2018]. A

combination of memory and graph neural networks [Pham et al., 2018] has been applied to a molecular structure prediction. An extension of NTM [Graves et al., 2016] has also been proposed for solving graph tasks as well as the toy QA task that we used in our experiments.

4.4 Synthetic Question and Answering Experiments

We perform experiments on the synthetic QA tasks defined in [Weston et al., 2016] (using version 1.1 of the dataset). This QA task consists of a set of statements, followed by a question whose answer is typically a single word (in a few tasks, answers are a set of words). The answer is available to the model at training time, but must be predicted at test time. There are a total of 20 different types of tasks that probe different forms of reasoning and deduction. Here are samples of three of the tasks:

```
Sam walks into the kitchen.
Sam picks up an apple.
Sam walks into the bedroom.
Sam drops the apple.

Q: Where is the apple?

A. Bedroom
```

```
Brian is a lion.
Julius is a lion.
Julius is white.
Bernhard is
green.
Q: What color is
Brian?
A. White
```

```
Mary journeyed to the den.
Mary went back to the kitchen.
John journeyed to the bedroom.
Mary discarded the milk.

Q: Where was the milk before the
den?
A. Hallway
```

Note that for each question, only some subset of the statements contain information needed for the answer, and the others are essentially irrelevant distractors (e.g. the first sentence in the first example). In the Memory Networks of [Weston et al., 2016], this *supporting subset* was explicitly indicated to the model during training and the key difference between that work and this one is that this information is no longer provided. Hence, the model must deduce for itself at training and test time which sentences are relevant and which are not.

Formally, for one of the 20 QA tasks, we are given example problems, each having

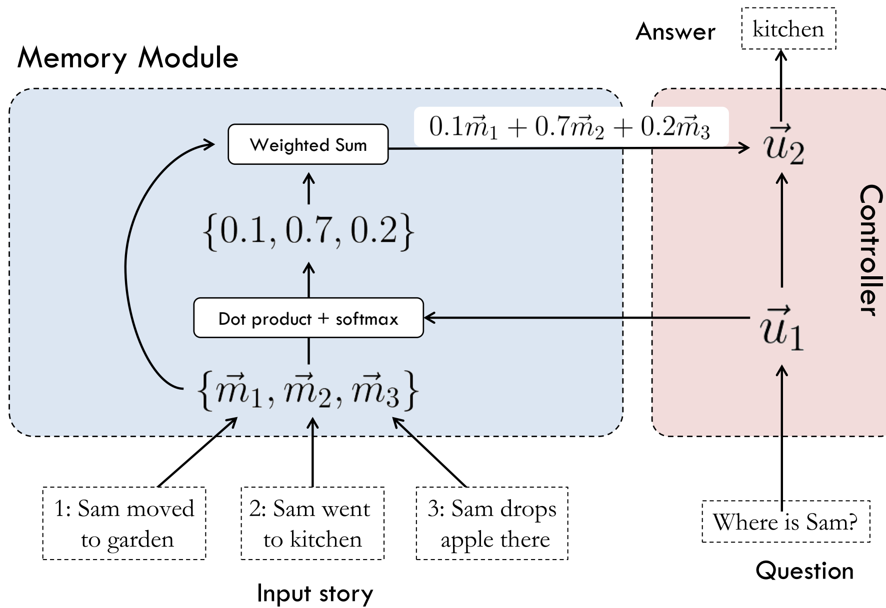

Figure 4.4: A toy example of an 1-hop MemN2N applied to a bAbI task.

a set of I sentences $\{x_i\}$ where $I \leq 320$; a question sentence q and answer a. Let the jth word of sentence i be x_{ij}, represented by a one-hot vector of length V (where the vocabulary is of size $V = 177$, reflecting the simplistic nature of the QA language). The same representation is used for the question q and answer a. Two versions of the data are used, one that has 1000 training problems per task and an another larger one with 10,000 per task.

4.4.1 Model Details

We put each sentence of the story into a single memory slot, and feed the question sentence into the controller module as shown in Figure 4.4. Unless otherwise stated, all experiments used a $K = 3$ hops model with the adjacent weight sharing scheme. For all tasks that output lists (i.e. the answers are multiple words), we take each possible combination of possible outputs and record them as a separate answer vocabulary word.

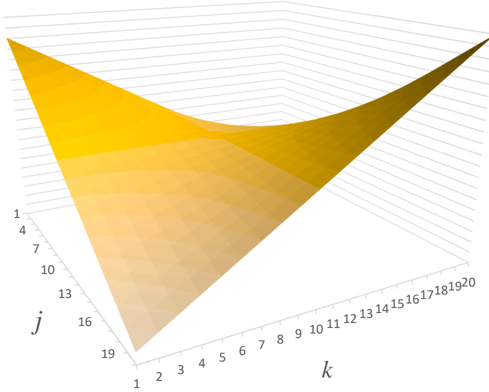

Figure 4.5: A plot of position encoding (PE) l_{kj}.

4.4.1.1 Sentence Representation

In our experiments we explore two different representations for the sentences. The first is the bag-of-words (BoW) representation that takes the sentence

$$x_i = \{x_{i1}, x_{i2}, ..., x_{in}\},$$

embeds each word and sums the resulting vectors: e.g $m_i = \sum_j Ax_{ij}$ and $c_i = \sum_j Cx_{ij}$. The input vector u representing the question is also embedded as a bag of words: $u = \sum_j Bq_j$. This has the drawback that it cannot capture the order of the words in the sentence, which is important for some tasks.

We therefore propose a second representation that encodes the position of words within the sentence. This takes the form: $m_i = \sum_j l_j \cdot Ax_{ij}$, where $\cdot$ is an element-wise multiplication. l_j is a column vector with the structure

$$l_{kj} = 1 - \frac{j}{J} - \frac{k}{d}\left(1 - \frac{2j}{J}\right)$$

(assuming 1-based indexing), with J being the number of words in the sentence, and

d is the dimension of the embedding. Figure 4.5 is a plot of l_{kj} when $J = 20$ and $d = 20$. This sentence representation, which we call position encoding (PE), means that the order of the words now affects m_i. The same representation is used for questions, memory inputs and memory outputs.

4.4.1.2 Temporal Encoding

Many of the QA tasks require some notion of temporal context, i.e. in the first example of Section 4.2, the model needs to understand that Sam is in the bedroom after he is in the kitchen. To enable our model to address them, we modify the memory vector so that $m_i = \sum_j Ax_{ij} + T_A[i]$, where $T_A[i]$ is the i-th row of a special matrix T_A that encodes temporal information. The output embedding is augmented in the same way with a matrix T_c (e.g. $c_i = \sum_j Cx_{ij} + T_C[i]$). Both T_A and T_C are learned during training. They are also subject to the same sharing constraints as A and C. Note that sentences are indexed in reverse order, reflecting their relative distance from the question so that x_1 is the last sentence of the story.

4.4.1.3 Learning time invariance by injecting random noise

We have found it helpful to add "dummy" memories to regularize T_A. That is, at training time we can randomly add 10% of empty memories to the stories. We refer to this approach as random noise (RN).

4.4.2 Training Details

10% of the bAbI training set was held-out to form a validation set, which was used to select the optimal model architecture and hyperparameters. Our models were trained using a learning rate of $\eta = 0.01$, with anneals every 25 epochs by $1/2$ until 100 epochs

were reached. No momentum or weight decay was used. The weights were initialized randomly from $\mathcal{N}(0, 0.1)$. When trained on all the tasks simultaneously with 1k training samples (10k training samples), 60 epochs (20 epochs) were used with a learning rate that anneals by $1/2$ every 15 epochs (5 epochs). All training runs use a batch size of 32 (but cost is not averaged over a batch), and gradients with an ℓ_2 norm larger than 40 are divided by a scalar to have norm 40. In some of our experiments, we explored commencing training with the softmax in each memory layer removed, making the model entirely linear except for the final softmax for answer prediction. When the validation loss stopped decreasing, the softmax layers were re-inserted and training recommenced. We refer to this as linear start (LS) training. In LS training, the initial learning rate is set to $\eta = 0.005$. The capacity of memory is restricted to the most recent 50 sentences. Since the number of sentences and the number of words per sentence varied between problems, a null symbol was used to pad them all to a fixed size. The embedding of the null symbol was constrained to be zero.

On some tasks, we observed a large variance in the performance of our model (i.e. sometimes failing badly, other times not, depending on the initialization). To remedy this, we repeated each training 10 times with different random initializations, and picked the one with the lowest training error. We released our source code at `https://github.com/facebook/MemNN`.

4.4.3 Baselines

We compare our approach (abbreviated to MemN2N) to a range of alternate models:

- **MemNN:** The strongly supervised AM+NG+NL Memory Networks approach, proposed in [Weston et al., 2016]. This is the best reported approach in that paper. It uses a max operation (rather than softmax) at each layer which is trained

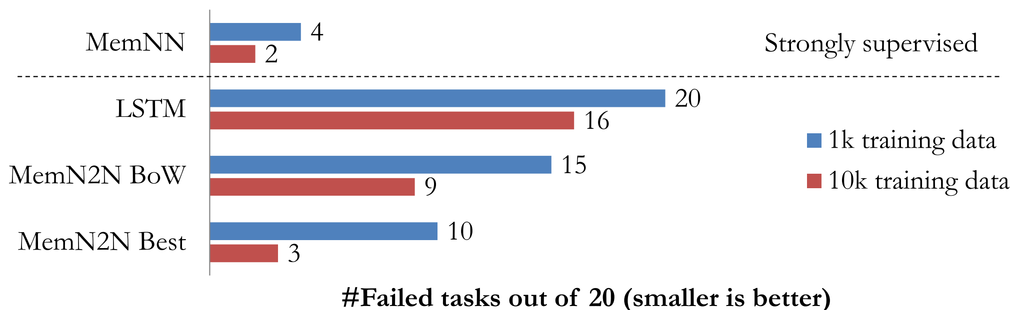

Figure 4.6: Comparison results of different models on bAbI tasks. The best MemN2N model fails (accuracy $< 95\%$) on three tasks, only one more than a MemN2N trained with more strong supervision signals.

directly with supporting facts (strong supervision). It employs n-gram modeling, nonlinear layers and an adaptive number of hops per query.

- **MemNN-WSH:** A weakly supervised heuristic version of MemNN where the supporting sentence labels are not used in training. Since we are unable to backpropagate through the max operations in each layer, we enforce that the first memory hop should share at least one word with the question, and that the second memory hop should share at least one word with the first hop and at least one word with the answer. All those memories that conform are called valid memories, and the goal during training is to rank them higher than invalid memories using the same ranking criteria as during strongly supervised training.

- **LSTM:** A standard LSTM model, trained using question / answer pairs only (i.e. also weakly supervised). For more detail, see [Weston et al., 2016].

4.4.4 Results

We report on a variety of design choices: (i) BoW vs Position Encoding (PE) sentence representation; (ii) training on all 20 tasks independently vs jointly training (joint

training used an embedding dimension of $d = 50$, while independent training used $d = 20$); (iii) two phase training: linear start (LS) where softmaxes are removed initially vs training with softmaxes from the start; (iv) varying memory hops from 1 to 3.

The results across all 20 tasks are given in Figure 4.6. See Table 4.1 and Table 4.2 for more detailed results on the 1k and 10k training set respectively. They show a number of interesting points:

- The best MemN2N models are reasonably close to the supervised models (e.g. 1k: 6.7% for MemNN vs 12.6% for MemN2N with position encoding + linear start + random noise, jointly trained and 10k: 3.2% for MemNN vs 4.2% for MemN2N with position encoding + linear start + random noise + non-linearity[2], although the supervised models are still superior.

- All variants of our proposed model comfortably beat the weakly supervised baseline methods.

- The position encoding (PE) representation improves over bag-of-words (BoW), as demonstrated by clear improvements on tasks 4, 5, 15 and 18, where word ordering is particularly important.

- The linear start (LS) to training seems to help avoid local minima. See task 16 in Table 4.1, where PE alone gets 53.6% error, while using LS reduces it to 1.6%.

- Jittering the time index with random empty memories (RN) as described in Section 4.4.1 gives a small but consistent boost in performance, especially for the smaller 1k training set.

- Joint training on all tasks helps.

[2]Following [Peng et al., 2015] we found adding more non-linearity solves tasks 17 and 19, see Table 4.2.

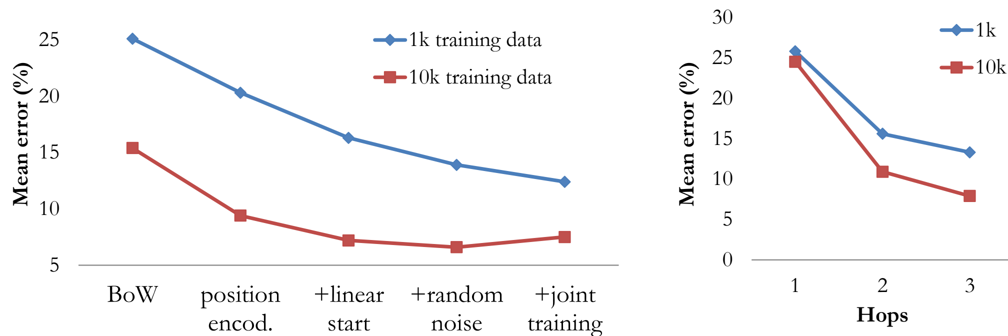

Figure 4.7: Experimental results from an ablation study of MemN2N on bAbI tasks. Left: incremental effects of different techniques. Right: reducing the number of hops.

- Importantly, more computational hops give improved performance as show in Figure 4.7. We give examples of the hops performed (via the values of Equation 4.1) over some illustrative examples in Figure 4.8.

4.5 Language Modeling Experiments

The goal in language modeling is to predict the next word in a text sequence given the previous words x. We now explain how our model can easily be applied to this task.

We now operate on word level, as opposed to the sentence level. Thus the previous N words in the sequence (including the current) are embedded into memory separately. Each memory cell holds only a single word, so there is no need for the BoW or linear mapping representations used in the QA tasks. We employ the temporal embedding approach of Section 4.4.1.

Since there is no longer any question, q in Figure 4.1 is fixed to a constant vector 0.1 (without embedding). The output softmax predicts which word in the vocabulary (of size V) is next in the sequence. A cross-entropy loss is used to train model by backpropagating the error through multiple memory layers, in the same manner as the QA tasks.

Task	Baseline			MemN2N								
	Strongly Supervised MemNN	LSTM	MemNN WSH	BoW	PE	PE LS	PE LS RN	1 hop PE LS joint	2 hops PE LS joint	3 hops PE LS joint	PE LS RN joint	PE LS LW joint
1: 1 supporting fact	0.0	50.0	0.1	0.6	0.1	0.2	0.0	0.8	0.0	0.1	0.0	0.1
2: 2 supporting facts	0.0	80.0	42.8	17.6	21.6	12.8	8.3	62.0	15.6	14.0	11.4	18.8
3: 3 supporting facts	0.0	80.0	76.4	71.0	64.2	58.8	40.3	76.9	31.6	33.1	21.9	31.7
4: 2 argument relations	0.0	39.0	40.3	32.0	3.8	11.6	2.8	22.8	2.2	5.7	13.4	17.5
5: 3 argument relations	2.0	30.0	16.3	18.3	14.1	15.7	13.1	11.0	13.4	14.8	14.4	12.9
6: yes/no questions	0.0	52.0	51.0	8.7	7.9	8.7	7.6	7.2	2.3	3.3	2.8	2.0
7: counting	15.0	51.0	36.1	23.5	21.6	20.3	17.3	15.9	25.4	17.9	18.3	10.1
8: lists/sets	9.0	55.0	37.8	11.4	12.6	12.7	10.0	13.2	11.7	10.1	9.3	6.1
9: simple negation	0.0	36.0	35.9	21.1	23.3	17.0	13.2	5.1	2.0	3.1	1.9	1.5
10: indefinite knowledge	2.0	56.0	68.7	22.8	17.4	18.6	15.1	10.6	5.0	6.6	6.5	2.6
11: basic coreference	0.0	38.0	30.0	4.1	4.3	0.0	0.9	8.4	1.2	0.9	0.3	3.3
12: conjunction	0.0	26.0	10.1	0.3	0.3	0.1	0.2	0.4	0.0	0.3	0.1	0.0
13: compound coreference	0.0	6.0	19.7	10.5	9.9	0.3	0.4	6.3	0.2	1.4	0.2	0.5
14: time reasoning	1.0	73.0	18.3	1.3	1.8	2.0	1.7	36.9	8.1	8.2	6.9	2.0
15: basic deduction	0.0	79.0	64.8	24.3	0.0	0.0	0.0	46.4	0.5	0.0	0.0	1.8
16: basic induction	0.0	77.0	50.5	52.0	52.1	1.6	1.3	47.4	51.3	3.5	2.7	51.0
17: positional reasoning	35.0	49.0	50.9	45.4	50.1	49.0	51.0	44.4	41.2	44.5	40.4	42.6
18: size reasoning	5.0	48.0	51.3	48.1	13.6	10.1	11.1	9.6	10.3	9.2	9.4	9.2
19: path finding	64.0	92.0	100.0	89.7	87.4	85.6	82.8	90.7	89.9	90.2	88.0	90.6
20: agent's motivation	0.0	9.0	3.6	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.2
Mean error (%)	6.7	51.3	40.2	25.1	20.3	16.3	13.9	25.8	15.6	13.3	12.4	15.2
Failed tasks (err. $> 5\%$)	4	20	18	15	13	12	11	17	11	11	11	10

Table 4.1: Test error rates (%) on the 20 QA tasks for models using 1k training examples. Key: BoW = bag-of-words representation; PE = position encoding representation; LS = linear start training; RN = random injection of time index noise; LW = RNN-style layer-wise weight tying (if not stated, adjacent weight tying is used); joint = joint training on all tasks (as opposed to per-task training).

Task	Baseline			MemN2N									
	Strongly Supervised MemNN	LSTM	MemNN WSH	BoW	PE	PE LS	PE LS RN	PE LS LW RN*	1 hop PE LS joint	2 hops PE LS joint	3 hops PE LS joint	PE LS RN joint	PE LS LW joint
1: 1 supporting fact	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2: 2 supporting facts	0.0	81.9	39.6	0.6	0.4	0.5	0.3	0.3	62.0	1.3	2.3	1.0	0.8
3: 3 supporting facts	0.0	83.1	79.5	17.8	12.6	15.0	9.3	2.1	80.0	15.8	14.0	6.8	18.3
4: 2 argument relations	0.0	0.2	36.6	31.8	0.0	0.0	0.0	0.0	21.4	0.0	0.0	0.0	0.0
5: 3 argument relations	0.3	1.2	21.1	14.2	0.8	0.6	0.8	0.8	8.7	7.2	7.5	6.1	0.8
6: yes/no questions	0.0	51.8	49.9	0.1	0.2	0.1	0.0	0.1	6.1	0.7	0.2	0.1	0.1
7: counting	3.3	24.9	35.1	10.7	5.7	3.2	3.7	2.0	14.8	10.5	6.1	6.6	8.4
8: lists/sets	1.0	34.1	42.7	1.4	2.4	2.2	0.8	0.9	8.9	4.7	4.0	2.7	1.4
9: simple negation	0.0	20.2	36.4	1.8	1.3	2.0	0.8	0.3	3.7	0.4	0.0	0.0	0.2
10: indefinite knowledge	0.0	30.1	76.0	1.9	1.7	3.3	2.4	0.0	10.3	0.6	0.4	0.5	0.0
11: basic coreference	0.0	10.3	25.3	0.0	0.0	0.0	0.0	0.1	8.3	0.0	0.0	0.0	0.4
12: conjunction	0.0	23.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0
13: compound coreference	0.0	6.1	12.3	0.0	0.1	0.0	0.0	0.0	5.6	0.0	0.0	0.0	0.0
14: time reasoning	0.0	81.0	8.7	0.0	0.2	0.0	0.0	0.1	30.9	0.2	0.2	0.0	1.7
15: basic deduction	0.0	78.7	68.8	12.5	0.0	0.0	0.0	0.0	42.6	0.0	0.0	0.2	0.0
16: basic induction	0.0	51.9	50.9	50.9	48.6	0.1	0.4	51.8	47.3	46.4	0.4	0.2	49.2
17: positional reasoning	24.6	50.1	51.1	47.4	40.3	41.1	40.7	18,6	40.0	39.7	41.7	41.8	40.0
18: size reasoning	2.1	6.8	45.8	41.3	7.4	8.6	6.7	5.3	9.2	10.1	8.6	8.0	8.4
19: path finding	31.9	90.3	100.0	75.4	66.6	66.7	66.5	2.3	91.0	80.8	73.3	75.7	89.5
20: agent's motivation	0.0	2.1	4.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Mean error (%)	3.2	36.4	39.2	15.4	9.4	7.2	6.6	4.2	24.5	10.9	7.9	7.5	11.0
Failed tasks (err. $> 5\%$)	2	16	17	9	6	4	4	3	16	7	6	6	6

Table 4.2: Test error rates (%) on the 20 bAbI QA tasks for models using 10k training examples. Key: BoW = bag-of-words representation; PE = position encoding representation; LS = linear start training; RN = random injection of time index noise; LW = RNN-style layer-wise weight tying (if not stated, adjacent weight tying is used); joint = joint training on all tasks (as opposed to per-task training); $*$ = this is a larger model with non-linearity (embedding dimension is $d = 100$ and ReLU applied to the internal state after each hop. This was inspired by [Peng et al., 2015] and crucial for getting better performance on tasks 17 and 19).

Story (1: 1 supporting fact)	Support	Hop 1	Hop 2	Hop 3
Daniel went to the bathroom.		0.00	0.00	0.03
Mary travelled to the hallway.		0.00	0.00	0.00
John went to the bedroom.		0.37	0.02	0.00
John travelled to the bathroom.	yes	0.60	0.98	0.96
Mary went to the office.		0.01	0.00	0.00
Sandra journeyed to the kitchen.		0.01	0.00	0.00
Where is John? Answer: bathroom Prediction: bathroom				

Story (2: 2 supporting facts)	Support	Hop 1	Hop 2	Hop 3
John dropped the milk.		0.06	0.00	0.00
Daniel travelled to the bedroom.		0.00	0.00	0.00
John took the milk there.	yes	0.88	1.00	0.00
Sandra went back to the bathroom.		0.00	0.00	0.00
John moved to the hallway.	yes	0.00	0.00	1.00
Mary went back to the bedroom.		0.00	0.00	0.00
Where is the milk? Answer: hallway Prediction: hallway				

Story (3: 3 supporting facts)	Support	Hop 1	Hop 2	Hop 3
John moved to the hallway.		0.00	0.00	0.00
John grabbed the football.	yes	0.00	1.00	0.00
John journeyed to the garden.		0.35	0.00	0.00
Sandra moved to the hallway.		0.00	0.00	0.00
John went back to the hallway.	yes	0.00	0.00	1.00
John journeyed to the garden.	yes	0.62	0.00	0.00
Where was the football before the garden? A: hallway P: hallway				

Story (4: 2 argument relations)	Support	Hop 1	Hop 2	Hop 3
The garden is north of the kitchen.	yes	0.84	1.00	0.92
The kitchen is north of the bedroom.		0.16	0.00	0.08
What is north of the kitchen? Answer: garden Prediction: garden				

Story (5: 3 argument relations)	Support	Hop 1	Hop 2	Hop 3
Jeff travelled to the bedroom.		0.00	0.00	0.00
Jeff journeyed to the garden.		0.00	0.00	0.00
Fred handed the apple to Jeff.	yes	1.00	1.00	0.98
Mary went to the garden.		0.00	0.00	0.00
Fred went back to the bathroom.		0.00	0.00	0.00
Fred got the milk there.		0.00	0.00	0.00
Mary journeyed to the kitchen.		0.00	0.00	0.00
Who gave the apple to Jeff? Answer: Fred Prediction: Fred				

Story (6: yes/no questions)	Support	Hop 1	Hop 2	Hop 3
Sandra travelled to the bedroom.		0.06	0.00	0.01
John took the football there.		0.00	0.00	0.00
Sandra travelled to the office.		0.00	0.45	0.16
Sandra went to the bedroom.	yes	0.89	0.39	0.04
Daniel went back to the kitchen.		0.00	0.16	0.00
John took the apple there.		0.00	0.00	0.00
Mary got the milk there.		0.00	0.00	0.00
Is Sandra in the bedroom? Answer: yes Prediction: Yes				

Story (7: counting)	Support	Hop 1	Hop 2	Hop 3
Daniel moved to the office.		0.00	0.00	0.00
Mary moved to the office.		0.00	0.00	0.00
Sandra picked up the apple there.	yes	0.14	0.00	0.92
Sandra dropped the apple.	yes	0.12	0.00	0.00
Sandra took the apple there.	yes	0.73	1.00	0.08
John went to the bedroom.		0.00	0.00	0.00
How many objects is Sandra carrying? Answer: one Prediction: one				

Story (8: lists/sets)	Support	Hop 1	Hop 2	Hop 3
John moved to the hallway.		0.00	0.00	0.00
John journeyed to the garden.		0.00	0.00	0.00
Daniel moved to the garden.		0.00	0.01	0.00
Daniel grabbed the apple there.	yes	0.03	0.00	0.98
Daniel got the milk there.	yes	0.97	0.02	0.00
John went back to the hallway.		0.00	0.00	0.00
What is Daniel carrying? Answer: apple,milk Prediction: apple,milk				

Story (9: simple negation)	Support	Hop 1	Hop 2	Hop 3
Sandra is in the garden.		0.60	0.99	0.00
Sandra is not in the garden.	yes	0.37	0.01	1.00
John went to the office.		0.00	0.00	0.00
John is in the bedroom.		0.00	0.00	0.00
Daniel moved to the garden.		0.00	0.00	0.00
Is Sandra in the garden? Answer: no Prediction: no				

Story (10: indefinite knowledge)	Support	Hop 1	Hop 2	Hop 3
Julie is either in the school or the bedroom.		0.00	0.00	0.00
Julie is either in the cinema or the park.		0.00	0.00	0.00
Bill is in the park.		0.00	0.00	0.00
Bill is either in the office or the office.	yes	1.00	1.00	1.00
Is Bill in the office? Answer: maybe Prediction: maybe				

Story (11: basic coherence)	Support	Hop 1	Hop 2	Hop 3
Mary journeyed to the hallway.		0.00	0.01	0.00
After that she journeyed to the bathroom.		0.00	0.00	0.00
Mary journeyed to the garden.		0.00	0.00	0.00
Then she went to the office.		0.01	0.06	0.00
Sandra journeyed to the garden.	yes	0.97	0.42	0.00
Then she went to the hallway.	yes	0.00	0.50	1.00
Where is Sandra? Answer: hallway Prediction: hallway				

Story (12: conjunction)	Support	Hop 1	Hop 2	Hop 3
John and Sandra went back to the kitchen.		0.08	0.00	0.00
Sandra and Mary travelled to the garden.		0.05	0.00	0.00
Mary and Daniel travelled to the office.		0.00	0.00	0.00
Mary and John went to the bathroom.		0.01	0.00	0.00
Daniel and Sandra went to the kitchen.	yes	0.74	1.00	1.00
Daniel and Mary journeyed to the office.		0.06	0.00	0.00
Where is Sandra? Answer: kitchen Prediction: kitchen				

Story (13: compound coherence)	Support	Hop 1	Hop 2	Hop 3
Sandra and Daniel travelled to the bathroom.		0.13	0.00	0.00
Afterwards they went back to the office.		0.01	0.00	0.00
Daniel and Mary travelled to the hallway.		0.01	0.00	0.00
Following that they went back to the office.		0.06	0.04	0.00
Mary and Sandra moved to the hallway.	yes	0.59	0.02	0.00
Then they went to the kitchen.	yes	0.02	0.94	1.00
Where is Sandra? Answer: kitchen Prediction: kitchen				

Story (14: time reasoning)	Support	Hop 1	Hop 2	Hop 3
This morning Julie went to the cinema.		0.00	0.03	0.00
Julie journeyed to the kitchen yesterday.		0.00	0.04	0.01
Fred travelled to the cinema yesterday.		0.00	0.05	0.01
Bill travelled to the office yesterday.		0.00	0.07	0.01
This morning Mary travelled to the bedroom.	yes	0.97	0.27	0.01
Yesterday Mary journeyed to the cinema.	yes	0.01	0.33	0.96
Where was Mary before the bedroom? Answer: cinema Prediction: cinema				

Story (15: basic deduction)	Support	Hop 1	Hop 2	Hop 3
Cats are afraid of wolves.	yes	0.00	0.99	0.62
Sheep are afraid of wolves.		0.00	0.00	0.31
Winona is a sheep.		0.00	0.00	0.00
Emily is a sheep.		0.00	0.00	0.00
Gertrude is a cat.	yes	0.99	0.00	0.00
Wolves are afraid of mice.		0.00	0.00	0.00
Mice are afraid of wolves.		0.00	0.00	0.07
Jessica is a mouse.		0.00	0.00	0.00
What is gertrude afraid of? Answer: wolf Prediction: wolf				

Story (16: basic induction)	Support	Hop 1	Hop 2	Hop 3
Lily is a swan.		0.00	0.00	0.00
Brian is a frog.	yes	0.00	0.98	0.00
Lily is gray.		0.07	0.00	0.00
Brian is yellow.	yes	0.07	0.00	1.00
Julius is a swan.		0.00	0.00	0.00
Bernhard is yellow.		0.04	0.00	0.00
Julius is green.		0.06	0.00	0.00
Greg is a frog.	yes	0.76	0.02	0.00
What color is Greg? Answer: yellow Prediction: yellow				

Story (17: positional reasoning)	Support	Hop 1	Hop 2	Hop 3
The red square is below the red sphere.	yes	0.37	0.95	0.58
The red sphere is below the triangle.	yes	0.63	0.05	0.43
Is the triangle above the red square? Answer: yes Prediction: no				

Story (18: size reasoning)	Support	Hop 1	Hop 2	Hop 3
The suitcase is bigger than the chest.	yes	0.00	0.88	0.00
The box is bigger than the chocolate.		0.04	0.05	0.10
The chest is bigger than the chocolate.	yes	0.17	0.07	0.90
The chest fits inside the container.		0.00	0.00	0.00
The chest fits inside the box.		0.00	0.00	0.00
Does the suitcase fit in the chocolate? Answer: no Prediction: no				

Story (19: path finding)	Support	Hop 1	Hop 2	Hop 3
The hallway is north of the kitchen.		1.00	1.00	1.00
The garden is south of the kitchen.	yes	0.00	0.00	0.00
The garden is east of the bedroom.	yes	0.00	0.00	0.00
The bathroom is south of the bedroom.		0.00	0.00	0.00
The office is east of the garden.		0.00	0.00	0.00
How do you go from the kitchen to the bedroom? Answer: s,w Prediction: n,n				

Story (20: agent's motivation)	Support	Hop 1	Hop 2	Hop 3
Yann journeyed to the kitchen.		0.00	0.00	0.00
Yann grabbed the apple there.		0.00	0.00	0.00
Antoine is thirsty.	yes	0.17	0.00	0.98
Jason picked up the milk there.		0.01	0.00	0.00
Antoine travelled to the kitchen.		0.77	1.00	0.00
Why did antoine go to the kitchen? Answer: thirsty Prediction: thirsty				

Figure 4.8: Example predictions on the QA tasks of [Weston et al., 2016]. We show the labeled supporting facts (support) from the dataset which MemN2N does not use during training, and the probabilities p of each hop used by the model during inference (indicated by values and blue color). MemN2N successfully learns to focus on the correct supporting sentences most of the time. The mistakes made by the model are highlighted by red color.

To aid training, we apply ReLU operations to half of the units in each layer. We use layer-wise (RNN-like) weight sharing, i.e. the query weights of each layer are the same; the output weights of each layer are the same. As noted in Section 4.2.2, this makes our architecture closely related to an RNN which is traditionally used for language modeling tasks; however here the "sequence" over which the network is recurrent is not in the text, but in the memory hops. Furthermore, the weight tying restricts the number of parameters in the model, helping generalization for the deeper models which we find to be effective for this task. We use two different datasets:

- **Penn Tree Bank** [Marcus et al., 1993]: This consists of 929k training, 73k validation and 82k test words, distributed over a vocabulary of 10k words. The same preprocessing as [Zaremba et al., 2014] was used.
- **Text8** [Mikolov et al., 2014]: This is a pre-processed version of the first 100M million characters, dumped from Wikipedia. This is split into three sets of 93.3M training, 5.7M validation and 1M test characters. All word occurring less than 5 times are replaced with the <UNK> token, resulting in a vocabulary size of ∼44k.

4.5.1 Training Details

The training procedure we use is the same as the QA tasks, except for the following. For each mini-batch update, the ℓ_2 norm of the whole gradient of all parameters is measured[3] and if larger than $L = 50$, then it is scaled down to have norm L. This was crucial for good performance. We use the learning rate annealing schedule from [Mikolov et al., 2014], namely, if the validation cost has not decreased after one epoch, then the learning

[3]In the QA tasks, the gradient of each weight matrix is measured separately.

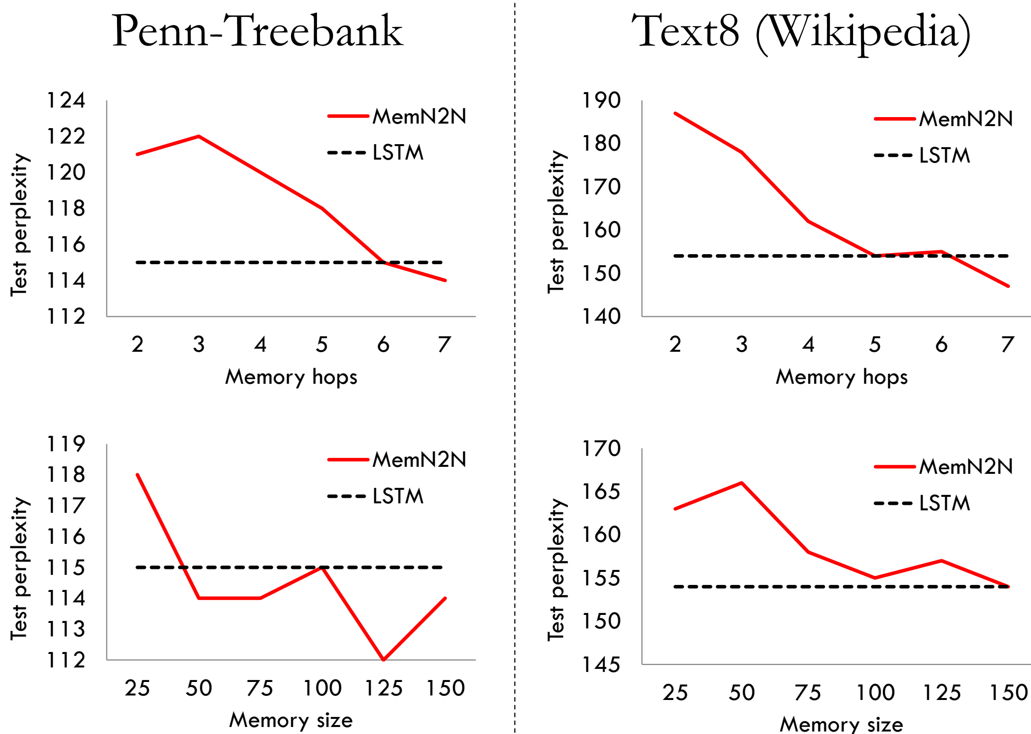

Figure 4.9: MemN2N performance on two language modeling tasks compared to an LSTM baseline. We vary the number of memory hops (top row) and the memory size (bottom row).

rate is scaled down by a factor 1.5. Training terminates when the learning rate drops below 10^{-5}, i.e. after 50 epochs or so. Weights are initialized using a normal distribution $\mathcal{N}(0, 0.05)$ and batch size is set to 128. On the Penn tree dataset, we repeat each training 10 times with different random initializations and pick the one with smallest validation cost. However, we have done only a single training run on Text8 dataset due to limited time constraints.

4.5.2 Results

Figure 4.9 compares variations of our model to LSTM baseline on the two benchmark datasets. More detailed Table 4.3 also include RNN and Structurally Constrained Recurrent Nets (SCRN) [Mikolov et al., 2014] baselines. Note that the baseline architectures were tuned in [Mikolov et al., 2014] to give optimal perplexity.[4] Our MemN2N approach achieves lower perplexity on both datasets (111 vs 115 for RNN/SCRN on Penn and 147 vs 154 for LSTM on Text8). Note that MemN2N has $\sim$1.5x more parameters than RNNs with the same number of hidden units, while LSTM has $\sim$4x more parameters. We also vary the number of hops and memory size of our MemN2N, showing the contribution of both to performance; note in particular that increasing the number of hops helps. In Figure 4.10, we show how MemN2N operates on memory with multiple hops. It shows the average weight of the activation of each memory position over the test set. We can see that some hops concentrate only on recent words, while other hops have more broad attention over all memory locations, which is consistent with the idea that succesful language models consist of a smoothed n-gram model and a cache [Mikolov et al., 2014]. Interestingly, it seems that those two types of hops tend to alternate. Also note that unlike a traditional RNN, the cache does not decay exponentially: it has roughly the same average activation across the entire memory. This may be the source of the observed improvement in language modeling.

4.6 Mazebase Experiments

In this section, we apply our model to the reasoning games implemented in MazeBase from Chapter 3. Unlike the previous experiments, this time we will use reinforce-

[4] They tuned the hyper-parameters on Penn Treebank and used them on Text8 without additional tuning, except for the number of hidden units. See [Mikolov et al., 2014] for more detail.

Model	Penn Treebank # of hidden	# of hops	memory size	Valid. perp.	Test perp.	Text8 # of hidden	# of hops	memory size	Valid. perp.	Test perp.
RNN	300	-	-	133	129	500	-	-	-	184
LSTM	100	-	-	120	115	500	-	-	122	154
SCRN	100	-	-	120	115	500	-	-	-	161
MemN2N	150	2	100	128	121	500	2	100	152	187
	150	3	100	129	122	500	3	100	142	178
	150	4	100	127	120	500	4	100	129	162
	150	5	100	127	118	500	5	100	123	154
	150	6	100	122	115	500	6	100	124	155
	150	7	100	120	114	500	7	100	118	**147**
	150	6	25	125	118	500	6	25	131	163
	150	6	50	121	114	500	6	50	132	166
	150	6	75	122	114	500	6	75	126	158
	150	6	100	122	115	500	6	100	124	155
	150	6	125	120	112	500	6	125	125	157
	150	6	150	121	114	500	6	150	123	154
	150	7	200	118	**111**	-	-	-	-	-

Table 4.3: The perplexity on the test sets of Penn Treebank and Text8 corpora. Note that increasing the number of memory hops improves performance.

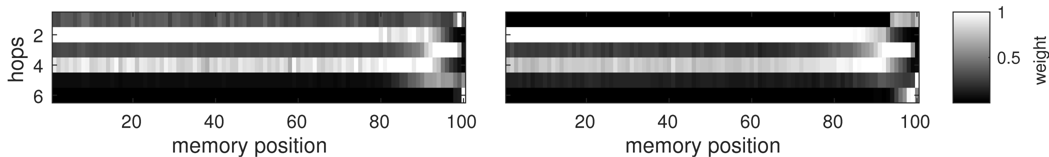

Figure 4.10: Average activation weight of memory positions during 6 memory hops. White color indicates where the model is attending during the kth hop. For clarity, each row is normalized to have maximum value of 1. A model is trained on (left) Penn Treebank and (right) Text8 dataset.

ment learning. This means an output from a MemN2N will be interpreted as an action and will be sent to the environment. However, the only change to the MemN2N architecture is the addition of a linear layer to the final hidden layer for predicting a baseline value.

We do not use our model as memory in those games since they are Markovian. Instead, the memory holds the descriptions of all items currently exist in a game. That

means every memory vector corresponds to a single item. Each item in the game (both physical items as well as "info") is represented as bag-of-words vectors. The spatial location of each item is also represented as a word within the bag. E.g. a red door at [+3,-2] becomes the vector {red_door} + {x=+3,y=-2}, where {red_door} and {x=+3,y=-2} are word vectors of dimension 50. These embedding vectors will be learned at training time. After 3 hops on the memory, the final softmax layer outputs a probability distribution over the set of discrete actions {N,S,E,W,toggle-switch,push-N,push-S,push-E,push-W}.

We use the REINFORCE algorithm from Section 2.1 for training, where α in Equation 2.5 is set to 0.03 in all experiments. The actual parameter update is done by RMSProp [Tieleman and Hinton, 2012] with learning rates optimized for each model type. For better parallelism, the model plays and learns from 512 games simultaneously, which are spread across multiple CPU threads. Training is continued for 20 thousand such parallel episodes, which amounts to approximately 10M game plays. Depending on the model type, the whole training process took from a few hours to a few days using a single machine with 18 cores.

4.6.1 Baselines

We investigate several different types of model as a baseline: (i) simple linear, (ii) multi-layer neural nets, and (iii) convolutional nets. While the input format is quite different for each approach (detailed below), the outputs are same for all the models. We do not consider models that are recurrent in the state-action sequence such as RNNs or LSTMs, because as discussed above, these tasks are Markovian.

- **Linear:** For a simple baseline we take the existence of each possible word-location pair on the largest grid we consider (10×10) and each "Info" item as

a separate feature, and train a linear classifier to the action space from these features. To construct the input, we take a bag-of-words (excluding location words) representation of all items at the same location. Then, we concatenate all those features from the every possible locations and info items. For example, if we had n different words and $w \times h$ possible locations with k additional info items, then the input dimension would be $(w \times h + k) \times n$.

- **Multi-layer Net:** Neural network with multiple fully connected layers separated by tanh non-linearity. The input representation is the same as the linear model.

- **Convolutional Net:** First, we represent each location by a bag-of-words in the same way as the linear model. Hence the environment is presented as a 3D cube of size $w \times h \times n$, which is then fed to four layers of convolution (the first layer has a 1×1 kernel, which essentially makes it an embedding of words). Items without spatial location (e.g. "Info" items) are each represented as a bag of words, and then combined via a fully connected layer to the outputs of the convolutional layers; these are then passed through two fully connected layers to output the actions (and a baseline for reinforcement).

4.6.2 Experimental Results

Before running experiments on all the games, we investigated the effect of various parameters of MultiGoals game on learning performances. The result in Figure 4.11 shows that we can make it harder for learning by adjusting those parameters. In curriculum training, we actually gradually increases them.

Figure 4.12 shows the performance of different models on the games. Each model is trained *jointly* on all 10 games. Given the stochastic nature of reinforcement learning,

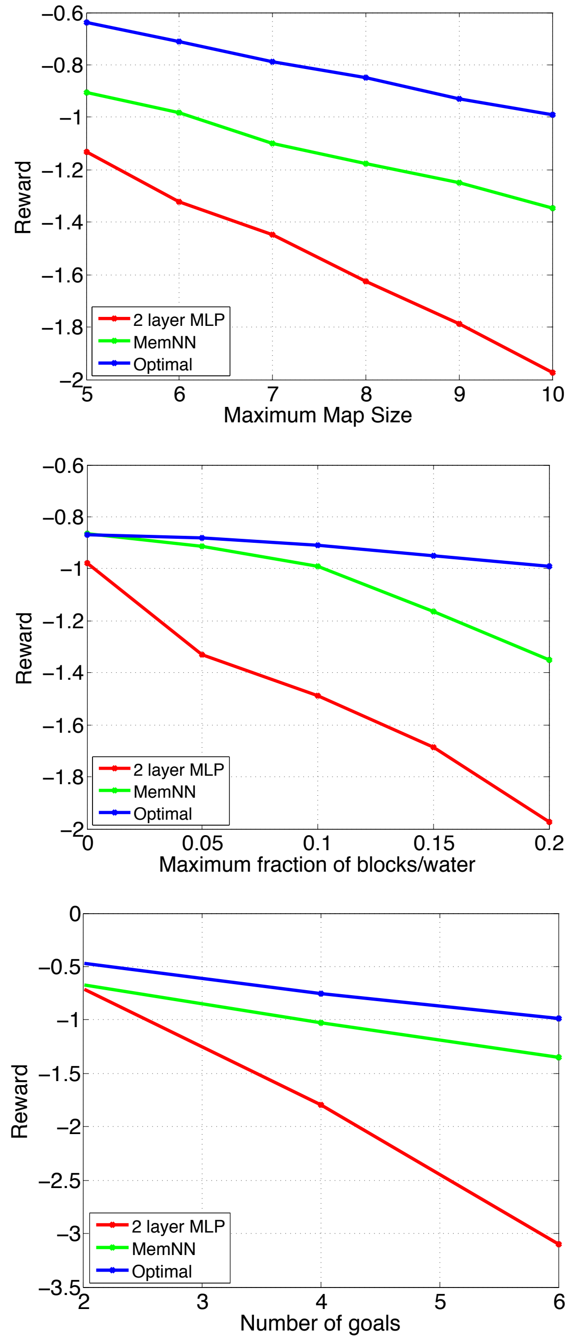

Figure 4.11: In the MazeBase environment, the difficulty of tasks can be varied programmatically. For example, in the **Multigoals** game the maximum map size, fraction of blocks/water and number of goals can all be varied. This affects the difficulty of tasks, as shown by the optimal reward (blue line). It also reveals how robust a given model is to task difficulty. For a 2-layer MLP (red line), the reward achieved degrades much faster than the MemN2N model (green line) and the inherent task difficulty.

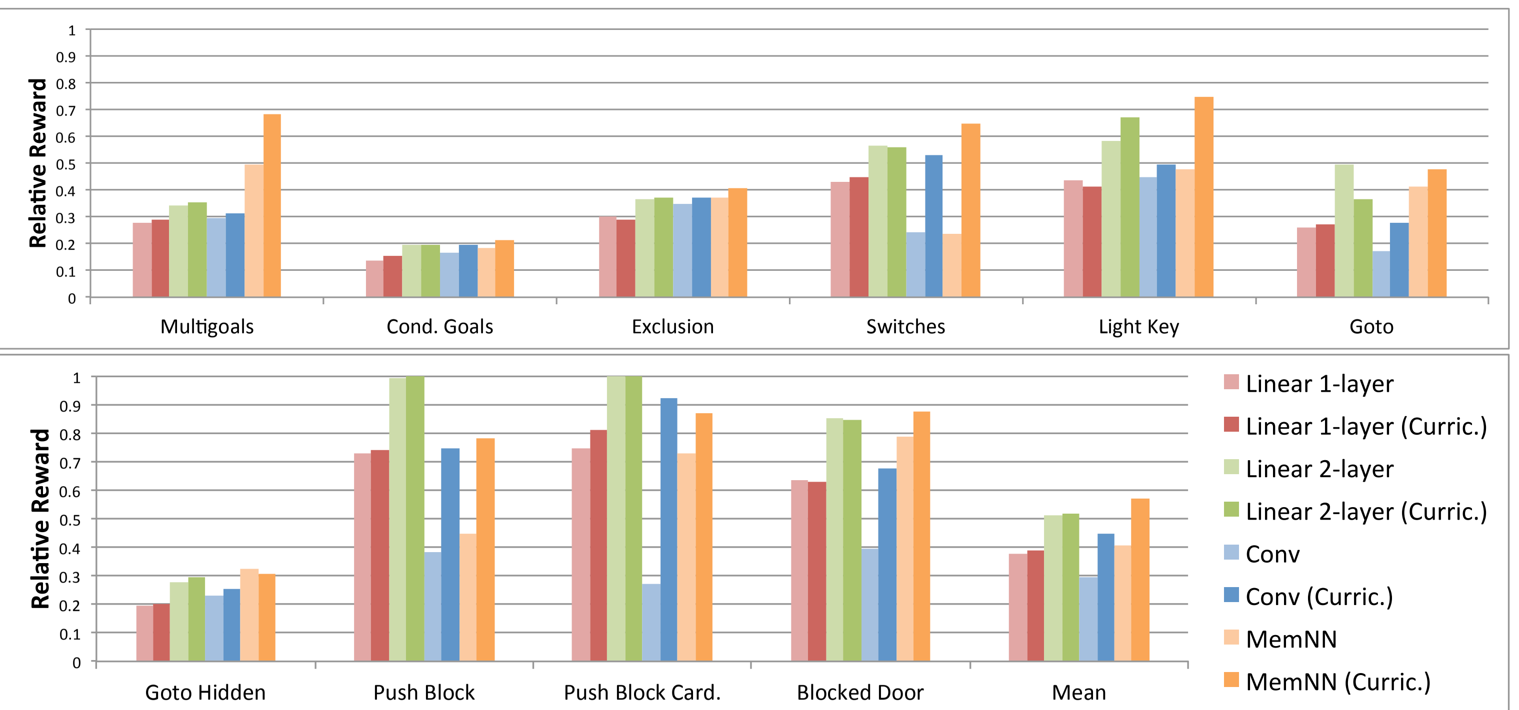

Figure 4.12: Reward for each model jointly trained on the 10 games, with and without the use of a curriculum during training. The y-axis shows relative reward (estimated optimal reward / absolute reward), thus higher is better. The estimated optimal policy corresponds to a value of 1 and a value of 0.5 implies that the agent takes twice as many steps to complete the task as is needed (since most of the reward signal comes from the negative increment at each time step).

we trained each model 10 times and picked the single instance that had the highest mean reward across all tasks (i.e. the same model is evaluated on all 10 tasks). Table 4.4 gives the max,mean and standard deviation of rewards for each task and method. Figure 4.13 shows a trained MemN2N model playing the **Switches** and **LightKey** games. A video showing all games can be found at `https://youtu.be/PVS5kIgAxZc`.

The results revealed a number of interesting points. On many of the games at least some of the models were able to learn a reasonable strategy. The models were all able to learn to convert between egocentric and absolute coordinates by using the corner blocks. They could respond appropriately to the different arrangements of the **Light Key** game, and make decent decisions on whether to try to go directly to the goal, or to first open the door. The 2-layer networks were able to completely solve the the tasks with pushable blocks. That said, despite the simplicity of the games, and the number of trials allowed, the models were not able to completely solve (i.e. discover optimal policy for) most of the games: On **Conditional Goals** and **Exclusion**, all the models did poorly. On inspection, they adopted the strategy of blindly visiting all goals, rather than visiting the correct one.

With some of the models, we were able to train jointly, but made a few of the game types artificially small; then at test time successfully run those games on a larger map. The models were able to learn the notion of the locations independently from the task. (for locations they had seen in training). On the other hand, we tried to test the models above on unseen tasks that were never shown at train time, but used the same vocabulary (for example: "go to the left", instead of "push the block to the left"). None of our models were able to succeed, highlighting how far we are from operating at a "human level", even in this extremely restricted setting.

The MemN2N did best out of all the models on average. However, on the games with

		Multigoals	Cond. Goals	Exclusion	Switches	Light Key	Goto	Goto Hidden	Push Block	Push Block Cardinal	Blocked Door	Mean
No Curriculum	Linear	-3.59 -3.67 ±0.06	-3.54 -3.93 ±0.13	-2.76 -2.62 ±0.06	-1.66 -1.65 ±0.03	-1.94 -1.89 ±0.04	-1.82 -1.78 ±0.02	-2.39 -2.48 ±0.02	-2.50 -2.54 ±0.02	-1.64 -1.65 ±0.02	-1.66 -1.63 ±0.03	-2.35 -2.37 ±0.01
	2 layer NN	-2.90 -3.16 ±0.63	-2.50 -2.88 ±0.77	-2.27 -2.84 ±1.14	-1.25 -1.46 ±0.51	-1.46 -1.76 ±1.06	-0.95 -1.38 ±1.21	-1.68 -2.00 ±1.00	-1.84 -2.33 ±1.00	-1.18 -1.67 ±1.15	-1.24 -1.64 ±1.12	-1.73 -2.11 ±0.91
	ConvNet	-3.36 -4.35 ±0.83	-2.90 -4.17 ±1.06	-2.38 -3.97 ±1.32	-2.96 -2.98 ±0.05	-1.90 -3.78 ±1.57	-2.70 -4.18 ±1.06	-2.06 -3.85 ±1.47	-4.80 -4.95 ±0.05	-4.50 -4.86 ±0.18	-2.70 -4.09 ±1.16	-3.03 -4.12 ±0.85
	MemN2N	-2.02 -3.68 ±0.99	-2.70 -3.51 ±1.04	-2.22 -3.06 ±1.33	-2.97 -2.98 ±0.03	-1.78 -2.72 ±1.56	-1.14 -2.25 ±1.89	-1.44 -3.42 ±1.19	-4.06 -4.47 ±0.68	-1.68 -2.71 ±1.54	-1.34 -2.43 ±1.77	-2.19 -3.13 ±1.14
Curriculum	Linear	-3.42 -3.42 ±0.02	-3.21 -3.17 ±0.03	-2.85 -2.89 ±0.05	-1.58 -1.59 ±0.03	-2.07 -2.03 ±0.03	-1.74 -1.72 ±0.02	-2.31 -2.33 ±0.03	-2.47 -2.45 ±0.02	-1.52 -1.52 ±0.02	-1.68 -1.67 ±0.02	-2.29 -2.28 ±0.00
	2 layer NN	-2.82 -2.84 ±0.02	-2.49 -2.49 ±0.04	-2.25 -2.30 ±0.03	-1.27 -1.29 ±0.02	-1.27 -1.42 ±0.08	-1.29 -1.37 ±0.12	-1.59 -1.67 ±0.04	-1.81 -1.85 ±0.03	-1.13 -1.20 ±0.03	-1.25 -1.24 ±0.02	-1.72 -1.77 ±0.02
	ConvNet	-3.17 -3.16 ±0.06	-2.52 -2.65 ±0.04	-2.20 -2.21 ±0.03	-1.34 -1.67 ±0.59	-1.72 -1.75 ±0.07	-1.70 -1.70 ±0.09	-1.85 -1.90 ±0.04	-2.45 -3.03 ±0.74	-1.33 -1.93 ±0.83	-1.56 -1.74 ±0.29	-1.99 -2.17 ±0.19
	MemN2N	-1.46 -1.98 ±0.73	-2.30 -2.45 ±0.13	-2.03 -2.06 ±0.05	-1.10 -1.57 ±0.76	-1.14 -1.49 ±0.28	-0.98 -1.07 ±0.10	-1.52 -1.42 ±0.48	-2.33 -2.67 ±0.47	-1.41 -1.50 ±0.15	-1.21 -1.57 ±0.58	-1.55 -1.78 ±0.15
Estimated Optimal		-1.00	-0.49	-0.83	-0.71	-0.85	-0.47	-0.47	-1.83	-1.23	-1.06	-0.89

Table 4.4: Reward of the different models on the 10 games, with and without curriculum. Each cell contains 3 numbers: (top) best performing one run (middle) mean of all runs, and (bottom) standard deviation of 10 runs with different random initialization. The estimated-optimal row shows the estimated highest average reward possible for each game. Note that the estimates are based on simple heuristics and are not exactly optimal.

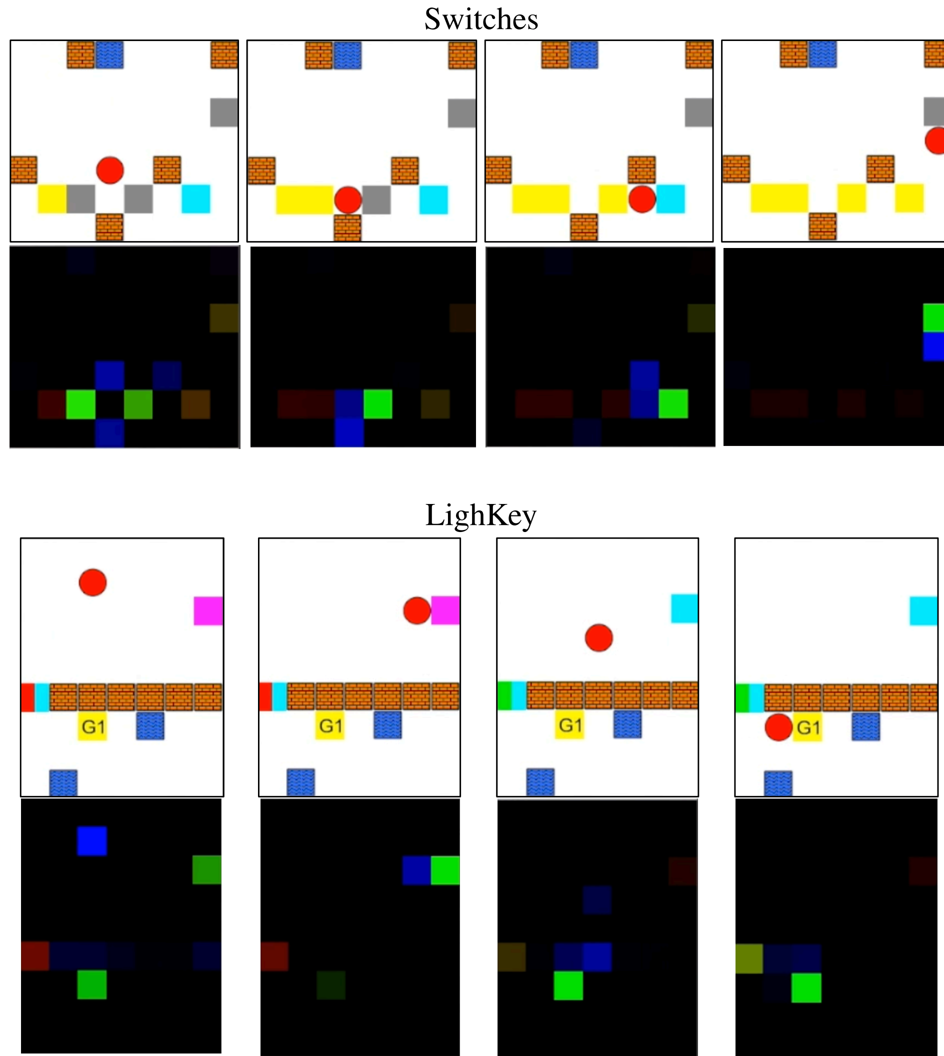

Figure 4.13: A trained MemN2N model playing the **Switches** (top) and **LightKey** (bottom) games. In the former, the goal is to make all the switches the same color. In the latter, the door blocking access to the goal must be opened by toggling the colored switch. The 2nd and 4th rows show the corresponding attention maps. The three different colors each correspond to one of the models 3 attention “hops”.

pushable blocks e.g. **Exclusion** and **Push Block**, the 2 layer neural nets were superior. Although we also trained 3 layer neural net, the results are not included here because it was very similar to the rewards of 2 layer neural net. The linear model did better than might be expected, and surprisingly, the convolutional nets were the worst of the four models. However, the fully connected models had significantly more parameters than either the convolutional network or the MemN2N. For example, the 2 layer neural net had a hidden layer of size 50, and a separate input for the outer product of each location and word combination. Because of the size of the games, this is 2 orders of magnitude more parameters than the convolutional network or MemN2N. Nevertheless, even with a very large number of trials, this architecture could not learn many of the tasks.

The MemN2N seems superior on games involving decisions using information in the **info** items (e.g. **Multigoals**) whereas the 2-layer neural net was better on the games with a pushable block (**Push Block**, **Push Block Cardinal**, and Blocked Door). Note that because we use egocentric coordinates, for **Push Block Cardinal**, and to a lesser extent **Push Block**, the models can memorize all the local configurations of the block and agent.

All methods had significant variance in performance over 10 instances, except for the linear model. However, the curriculum decreased the variance for all the methods, especially for the MemN2N. With respect to different training modalities: the curriculum generally helped all the approaches, but gave the biggest assistance to the MemN2N, particularly for **Push Block** and related games. We also tried supervised training (essentially imitation learning), but the results were more or less the same as reinforcement learning.

	2 vs 2	Kiting	Kiting hard
Attack weakest	85%	0%	0%
MemN2N	80%	100%	100 %
MemN2N (transfer)	65%	96%	72%

Table 4.5: Win rates against StarCraft built-in AI. The 2nd row shows the hand-coded baseline strategy of always attacking the weakest enemy (and not fleeing during cooldown). The 3rd row shows MemN2N trained and tested on StarCraft. The last row shows a MemN2N trained entirely inside MazeBase and tested on StarCraft with no modifications or fine tuning except scaling of the inputs.

4.7 StarCraft Experiments

Finally, we train our model on the combat tasks in the StarCraft environment from Section 2.2.2. In those tasks, the model have to control multiple units to fight against enemy bots. However depending on the unit types, it has to employ different tactics to succeed. The model architecture and training settings are the same as Section 4.6, except that a multi-resolution feature map is used here to make the observations cover large areas.

We find that the MemN2N is able to learn basic tactics such as focusing their fire on weaker opponents to kill them first (thus reducing the total amount of incoming damage over the game). This results in a win rate of 80% over the built-in StarCraft AI on **2 vs 2**, and nearly perfect results on **Kiting** (see Table 4.5). The attacking weakest is a hard-coded strategy where all units simply attack the enemy unit with least health points. It is a very effective strategy in **2 vs 2** scenario, but completely fails in **Kiting**. The video `https://youtu.be/mv-uVU5Rx1g` shows example gameplay of our MemN2N model for the **StarCraft Kiting hard** scenario.

Additionally, we perform a transfer learning experiment where we train a MemN2N on MazeBase and then test it on StarCraft. The MazeBase tasks are designed to closely

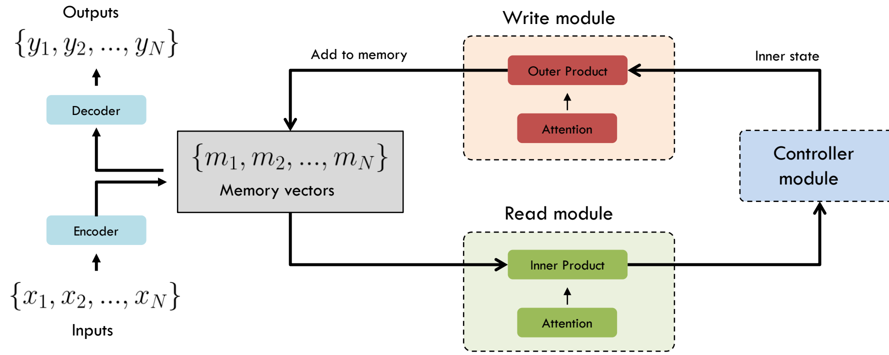

Figure 4.14: An extension of MemN2N that can write into the memory.

match the combat tasks from StarCraft, but many simplifications are made due to the limitations of MazeBase. See Section 3.3.2 for more details. We make no modifications to the model after it is trained on MazeBase, and minimal changes to the interface (we scale the health by a factor of 10, and x,y and cooldown values by a factor of 4). The success rate of the model (the last row of Table 4.5) on StarCraft is comparable to training directly in StarCraft, showing the robustness of MemN2N and that MazeBase can be effectively used as a sandbox for exploring model architectures and hyper-parameter optimization.

4.8 Writing to the Memory

In this section, we introduce an extension of MemN2N that can write into the memory. This allows the memory to be used as an output interface as well as an input interface. This makes it possible to output an set. As with the memory reading, the writing operation to the memory is done in a permutation invariant way, so the output elements have the same property, which is vital for handling sets.

The only modification to a MemN2N model is an addition of a write module as

shown in Figure 4.14. The write module has a similar architecture as the read module, except it uses an outer product instead of an inner product for computing its output. First, it computes an attention over the memory vectors just like the read module

$$p_i = \text{Softmax}(u^T m_i),$$

where u is the internal state vector of the controller, and m_i is the i-th memory vector. Unlike the read module, however, the output from the write module is computed by the outer product of the attention vector $p = [p_1, p_2, \ldots, p_N]^T$ and u

$$O = u \otimes p = up^T.$$

Finally, we add matrix O to the current memory, which ends the write operation. The update rule for i-th memory is then

$$m_i^{k+1} = m_i^k + p_i u^{k+1},$$

where k denotes the current hop. Note that the write module is conditioned on the internal state that already updated by the k-th read operation. Overall, the k-th hop consists of

$$u^{k+1} = \text{Read}(m_i^k, u^k)$$
$$m_i^{k+1} = \text{Write}(m_i^k, u^{k+1}).$$

After K hops, an output set $\hat{a} = \{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_N\}$ is given by applying a classification

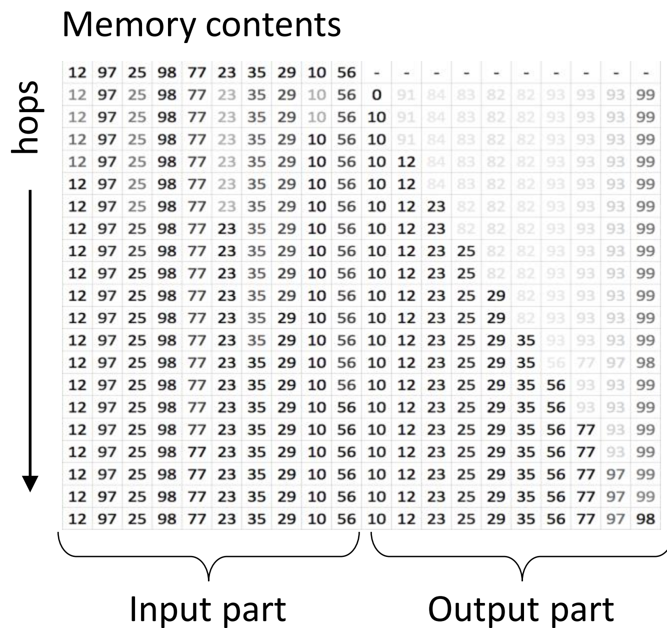

Figure 4.15: Evolution of the memory content while the model sorts the given input numbers.

to the each memory vector

$$\hat{a}_i = \text{Softmax}(W m_i^{K+1} + b),$$

where W and b are the weight and bias of the classification layer.

We test the model a toy task of sorting numbers. The task is to sort 10 given integers that are randomly chosen between 0 and 99. We use the memory of size 20, so we can use the first 10 for placing inputs and the remaining 10 for writing outputs. The numbers as are given as a word to the model, so it has to learn their ordering from scratch. In addition, we have words l_i for encoding memory locations so the model can put its output at the right memory locations. Therefore, the input to the model is

$$\{l_1 + x_1, \ldots, l_{10} + x_{10}, l_{11}, \ldots, l_{20}\},$$

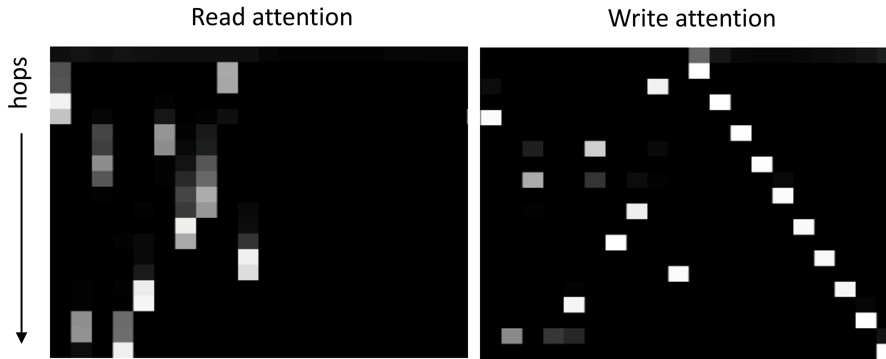

Figure 4.16: The attention map during memory hops while the model sorts given numbers.

and its target output is

$$\{x_1, \ldots, x_{10}, x'_1, \ldots, x'_{10}\}.$$

Here $\{x_i\}$ are the input numbers, and $\{x'_i\}$ are their sorted version.

The final loss function is the sum of all the classification losses at every memory locations, and it is minimized by the backpropagation algorithm. Figure 4.15 shows the memory content of a trained model while it is performing its hops. The model has discovered a simple greedy algorithm to find and write the smallest input element that yet to be sorted. Interestingly, the corresponding attention map shown in Figure 4.16 reveals that when a number is sorted, the model also writes to its input location. Such marking make sense as it can help the model distinguish already sorted numbers from the remaining numbers.

4.9 Conclusions and Future Work

In this work we showed that a neural network with an explicit memory and a recurrent attention mechanism for reading the memory can be successfully trained via backpropagation on diverse tasks from question answering, language modeling to re-

inforcement learning. Compared to the Memory Network implementation of [Weston et al., 2015] there is no supervision of supporting facts and so our model can be used in a wider range of settings. Our model approaches the same performance of that model, and is significantly better than other baselines with the same level of supervision on question answering tasks. On language modeling tasks, it slightly outperforms tuned RNNs and LSTMs of comparable complexity. On both tasks we can see that increasing the number of memory hops improves performance.

On reinforcement learning tasks, we can see some differences in the aptitudes between our model and the baselines, even though their average performance was similar. Our model was superior on tasks where it was necessary to combine information from **info** items with perceptual data, and the fully-connected networks were superior on tasks that were mostly spatial, and especially those where it was possible to memorize all the relevant configurations. Furthermore, our model trained on the simulations inside MazeBase performs well when tested directly on StarCraft, with no fine tuning.

However, there is still much to do. Our model is still unable to exactly match the performance of the memory networks trained with strong supervision, and both fail on several of the 1k QA tasks. Furthermore, smooth lookups may not scale well to the case where a larger memory is required. For these settings, we plan to explore multiscale notions of attention or hashing, as proposed in [Weston et al., 2015]. On MazeBase tasks, we are still far from the optimal performance, which suggests a further exploration of model architectures and training paradigms. Although we introduced a version of the model that can write to the memory, more empirical investigation is required to understand its cababilities and limitations.

Chapter 5

Learning Multiagent Communication with Backpropagation

Many tasks in AI require the collaboration of multiple agents. Typically, the communication protocol between agents is manually specified and not altered during training. In this chapter we explore a simple neural model, called CommNet, that uses continuous communication for fully cooperative tasks. The model consists of multiple agents and the communication between them is learned alongside their policy. We apply this model to a diverse set of tasks, demonstrating the ability of the agents to learn to communicate amongst themselves, yielding improved performance over non-communicative agents and baselines. In some cases, it is possible to interpret the language devised by the agents, revealing simple but effective strategies for solving the task at hand.

5.1 Introduction

Communication is a fundamental aspect of intelligence, enabling agents to behave as a group, rather than a collection of individuals. It is vital for performing complex tasks in real-world environments where each actor has limited capabilities and/or visibility of the world. Practical examples include elevator control [Crites and Barto, 1998] and sensor networks [Fox et al., 2000]; communication is also important for success in robot soccer [Stone and Veloso, 1998]. In any partially observed environment, the communication between agents is vital to coordinate the behavior of each individual. While the model controlling each agent is typically learned via reinforcement learning [Busoniu et al., 2008, Sutton and Barto, 1998], the specification and format of the communication is usually pre-determined. For example, in robot soccer, the bots are designed to communicate at each time step their position and proximity to the ball.

In this chapter, we propose a model where cooperating agents learn to communicate amongst themselves before taking actions. Each agent is controlled by a deep feed-forward network, which additionally has access to a communication channel carrying a continuous vector. Through this channel, they receive the summed transmissions of other agents. However, what each agent transmits on the channel is not specified a-priori, being learned instead. Because the communication is continuous, the model can be trained via back-propagation, and thus can be combined with standard single agent RL algorithms or supervised learning. The model is simple and versatile. This allows it to be applied to a wide range of problems involving partial visibility of the environment, where the agents learn a task-specific communication protocol that aids performance. In addition, the model allows dynamic variation at run time in both the number and type of agents, which is important in applications such as communication between moving

cars.

We consider the setting where we have J agents, all cooperating to maximize reward R in some environment. We make the simplifying assumption of full cooperation between agents, thus each agent receives R independent of their contribution. In this setting, there is no difference between each agent having its own controller, or viewing them as pieces of a larger model controlling all agents. Taking the latter perspective, our controller is a large feed-forward neural network that maps inputs for all agents to their actions, each agent occupying a subset of units. A specific connectivity structure between layers (a) instantiates the broadcast communication channel between agents and (b) propagates the agent state.

We explore this model on a range of tasks. In some, supervision is provided for each action while for others it is given sporadically. In the former case, the controller for each agent is trained by backpropagating the error signal through the connectivity structure of the model, enabling the agents to learn how to communicate amongst themselves to maximize the objective. In the latter case, reinforcement learning must be used as an additional outer loop to provide a training signal at each time step.

5.2 Communication Model

We now describe the controller model Φ used as a policy for multi-agent environments. Let $s_{t,j}$ be the j-th agent's observation of the environment at time t. The input to the controller is the concatenation of all state-views $\mathbf{s}_t = \{s_{t,1}, ..., s_{t,J}\}$, and the controller is a mapping $\mathbf{a}_t = \Phi(\mathbf{s}_t)$, where the output $\mathbf{a}_t$ is a concatenation of discrete actions $\mathbf{a}_t = \{a_{t,1}, ..., a_{t,J}\}$ for each agent. Note that this single controller Φ encompasses the individual controllers for each agents, as well as the communication between agents.

5.2.1 Controller Structure

We now detail our architecture for Φ that is built from modules f^i, which take the form of multilayer neural networks. Here $i \in \{0, .., K\}$, where K is the number of communication steps in the network. We omit the time index t for brevity.

Each f^i takes two input vectors for each agent j: the hidden state h_j^i and the communication c_j^i, and outputs a vector h_j^{i+1}. The main body of the model then takes as input the concatenated vectors $\mathbf{h}^0 = [h_1^0, h_2^0, ..., h_J^0]$, and computes:

$$h_j^{i+1} = f^i(h_j^i, c_j^i) \tag{5.1}$$

$$c_j^{i+1} = \frac{1}{J-1} \sum_{j' \neq j} h_{j'}^{i+1}. \tag{5.2}$$

In the case that f^i is a single linear layer followed by a non-linearity σ, we have: $h_j^{i+1} = \sigma(H^i h_j^i + C^i c_j^i)$ and the model can be viewed as a feedforward network with layers $\mathbf{h}^{i+1} = \sigma(T^i \mathbf{h}^i)$ where $\mathbf{h}^i$ is the concatenation of all h_j^i and T^i takes the block form (where $\bar{C}^i = C^i/(J-1)$):

$$T^i = \begin{pmatrix} H^i & \bar{C}^i & \bar{C}^i & ... & \bar{C}^i \\ \bar{C}^i & H^i & \bar{C}^i & ... & \bar{C}^i \\ \bar{C}^i & \bar{C}^i & H^i & ... & \bar{C}^i \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{C}^i & \bar{C}^i & \bar{C}^i & ... & H^i \end{pmatrix}.$$

A key point is that T is *dynamically sized* since the number of agents may vary. This motivates the the normalizing factor $J-1$ in Equation 5.2, which rescales the communication vector by the number of communicating agents. Note also that T^i is permutation invariant, thus the order of the agents does not matter.

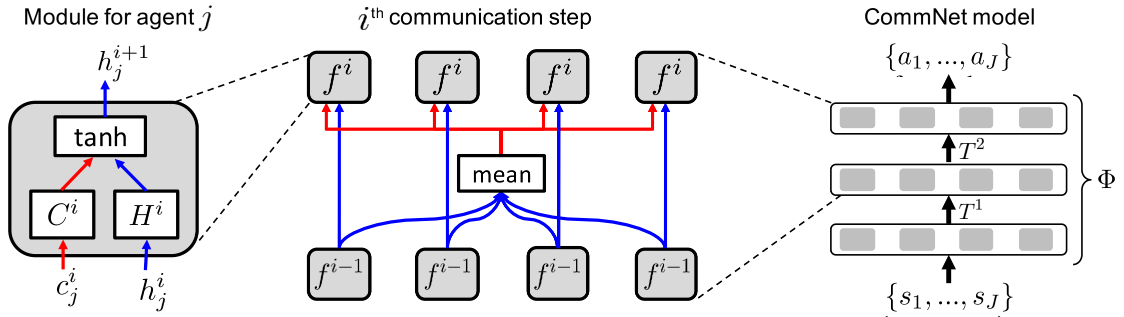

Figure 5.1: An overview of our CommNet model. Left: view of module f^i for a single agent j. Note that the parameters are shared across all agents. Middle: a single communication step, where each agents modules propagate their internal state h, as well as broadcasting a communication vector c on a common channel (shown in red). Right: full model Φ, showing input states s for each agent, two communication steps and the output actions for each agent.

At the first layer of the model an encoder function $h_j^0 = r(s_j)$ is used. This takes as input a state-view s_j and outputs a feature vector h_j^0 (in $\mathbb{R}^{d_0}$ for some d_0). The form of the encoder is problem dependent, but for most of our tasks it is a single layer neural network. Unless otherwise noted, $c_j^0 = 0$ for all j. At the output of the model, a decoder function $q(h_j^K)$ is used to output a distribution over the space of actions. $q(.)$ takes the form of a single layer network, followed by a softmax. To produce a discrete action, we sample from this distribution: $a_j \sim q(h_j^K)$.

Thus the entire model (shown in Figure 5.1), which we call a Communication Neural Net (CommNet), (i) takes the state-view of all agents s, passes it through the encoder $\mathbf{h}^0 = r(\mathbf{s})$, (ii) iterates $\mathbf{h}$ and $\mathbf{c}$ in Equations 5.1 and 5.2 to obtain $\mathbf{h}^K$, (iii) samples actions $\mathbf{a}$ for all agents, according to $q(\mathbf{h}^K)$.

5.2.2 Model Extensions

Local Connectivity: An alternative to the broadcast framework described above is to allow agents to communicate to others within a certain range. Let $N(j)$ be the set of

agents present within communication range of agent j. Then Equation 5.2 becomes:

$$c_j^{i+1} = \frac{1}{|N(j)|} \sum_{j' \in N(j)} h_{j'}^{i+1}. \tag{5.3}$$

As the agents move, enter and exit the environment, $N(j)$ will change over time. In this setting, our model has a natural interpretation as a dynamic graph, with $N(j)$ being the set of vertices connected to vertex j at the current time. The edges within the graph represent the communication channel between agents, with Equation 5.3 being equivalent to belief propagation [Pearl, 1982]. Furthermore, the use of multi-layer nets at each vertex makes our model similar to an instantiation of the GGSNN work of [Li et al., 2015].

Skip Connections: For some tasks, it is useful to have the input encoding h_j^0 present as an input for communication steps beyond the first layer. Thus for agent j at step i, we have:

$$h_j^{i+1} = f^i(h_j^i, c_j^i, h_j^0). \tag{5.4}$$

Temporal Recurrence: We also explore having the network be a recurrent neural network (RNN). This is achieved by simply replacing the communication step i in Equations 5.1 and 5.2 by a time step t, and using the same module f^t for all t. At every time step, actions will be sampled from $q(h_j^t)$. Note that agents can leave or join the swarm at any time step. If f^t is a single layer network, we obtain plain RNNs that communicate with each other. In later experiments, we also use an LSTM as an f^t module.

5.3 Related Work

Our model combines a deep network with reinforcement learning [Guo et al., 2014b, Mnih et al., 2015b, Levine et al., 2016]. Several recent works have applied these methods to multi-agent domains, such as Go [Maddison et al., 2015, Silver et al., 2016b] and Atari games [Tampuu et al., 2015], but they assume full visibility of the environment and lack communication. There is a rich literature on multi-agent reinforcement learning (MARL) [Busoniu et al., 2008], particularly in the robotics domain [Matari, 1997, Stone and Veloso, 1998, Fox et al., 2000, Olfati-Saber et al., 2007, Cao et al., 2013]. Amongst fully cooperative algorithms, many approaches [Lauer and Riedmiller, 2000, Littman, 2001, Wang and Sandholm, 2002] avoid the need for communication by making strong assumptions about visibility of other agents and the environment. Others use communication, but with a pre-determined protocol [Tan, 1993, Melo et al., 2011, Zhang and Lesser, 2013, Maravall et al., 2013].

A few notable approaches involve learning to communicate between agents under partial visibility: [Kasai et al., 2008] and [Varshavskaya et al., 2009], both use distributed tabular-RL approaches for simulated tasks. [Giles and Jim, 2002] use an evolutionary algorithm, rather than reinforcement learning. [Guestrin et al., 2001] use a single large MDP to control a collection of agents, via a factored message passing framework where the messages are learned. In contrast to these approaches, our model uses a deep network for both agent control and communication.

From a MARL perspective, the closest approach to ours is the concurrent work of [Foerster et al., 2016]. This also uses a deep reinforcement learning in multi-agent partially observable tasks, specifically two riddle problems (similar in spirit to our *levers* task) which necessitate multi-agent communication. Like our approach, the communi-

cation is learned rather than being pre-determined. However, the agents communicate in a discrete manner through their actions. This contrasts with our model where multiple continuous communication cycles are used at each time step to decide the actions of all agents. Furthermore, our approach is amenable to dynamic variation in the number of agents.

The Neural GPU [Kaiser and Sutskever, 2016] has similarities to our model but differs in that a 1-D ordering on the input is assumed and it employs convolution, as opposed to the global pooling in our approach (thus permitting unstructured inputs).

Graph Neural Networks: Our model can be regarded as an instantiation of the Graph Neural Network construction of [Scarselli et al., 2009], as expanded in [Li et al., 2015]. In particular, in [Scarselli et al., 2009], the output of the model is the fixed point of iterating Equations 5.3 and 5.1 to convergence, using recurrent models. In [Li et al., 2015], these recurrence equations are unrolled a fixed number of steps and the model trained via backprop through time. In this work, we do not require the model to be recurrent, neither do we aim to reach steady state. Additionally, we regard Equation 5.3 as a pooling operation, conceptually making our model a single feed-forward network with local connections.

Set processing: Although we presented our model as a communication protocol for multi-agents, it is actually a generic model for set inputs. It is permutation invariant and allow variable size inputs, two properties essential for set handling. Another model with such properties is MemN2N from Chapter 4, where set elements are put into a memory and processed through a *central* controller. In contrast, CommNet handles sets in a more *distributed* way by processing each element by its own module. We compare them experimentally in Section 5.4.4. After our publication, [Zaheer et al., 2017] tested a model similar ours on variety of set tasks.

5.4 Experiments

5.4.1 Baselines

We describe three baselines models for Φ to compare against our model.

- **Independent controller:** A simple baseline is where agents are controlled independently without any communication between them. We can write Φ as $\mathbf{a} = \{\phi(s_1), ..., \phi(s_J)\}$, where ϕ is a per-agent controller applied independently. The advantages of this communication-free model is modularity and flexibility.[1] Thus it can deal well with agents joining and leaving the group, but it is not able to coordinate agents' actions.

- **Fully-connected:** Another obvious choice is to make Φ a fully-connected multi-layer neural network, that takes concatenation of h_j^0 as an input and outputs actions $\{a_1, ..., a_J\}$ using multiple output softmax heads. It is equivalent to allowing T to be an arbitrary matrix with fixed size. This model would allow agents to communicate with each other and share views of the environment. Unlike our model, however, it is not modular, inflexible with respect to the composition and number of agents it controls, and even the order of the agents must be fixed.

- **Discrete communication:** An alternate way for agents to communicate is via discrete symbols, with the meaning of these symbols being learned during training. Since Φ now contains discrete operations and is not differentiable, reinforcement learning is used to train in this setting. However, unlike actions in the environment, an agent has to output a discrete symbol at every communication step. But if these are viewed as *internal* time steps of the agent, then the communication

[1] Assuming s_j includes the identity of agent j.

Model Φ	Training method Supervised	Reinforcement
Independent	0.59	0.59
CommNet	**0.99**	**0.94**

Table 5.1: Results of lever game (#distinct levers pulled)/(#levers) for our CommNet and independent controller models, using two different training approaches. Allowing the agents to communicate enables them to succeed at the task.

output can be treated as an action of the agent at a given (internal) time step and we can directly employ REINFORCE from Section 2.1.

At communication step i, agent j will output the index w_j^i corresponding to a particular symbol, sampled according to:

$$w_j^i \sim \text{Softmax}(Dh_j^i) \tag{5.5}$$

where matrix D is the model parameter. Let $\hat{w}$ be a 1-hot binary vector representation of w. In our broadcast framework, at the next step the agent receives a bag of vectors from all the other agents (where $\wedge$ is the element-wise OR operation):

$$c_j^{i+1} = \bigwedge_{j' \neq j} \hat{w}_{j'}^i \tag{5.6}$$

5.4.2 Simple Demonstration with a Lever Pulling Task

We start with a very simple game that requires the agents to communicate in order to win. This consists of m levers and a pool of N agents. At each round, m agents are drawn at random from the total pool of N agents and they must each choose a lever to pull, simultaneously with the other $m-1$ agents, after which the round ends. The goal is for each of them to pull a *different* lever. Correspondingly, all agents receive reward

proportional to the number of distinct levers pulled. Each agent can see its own identity, and nothing else, thus $s_j = j$.

We implement the game with $m = 5$ and $N = 500$. We use a CommNet with two communication steps ($K = 2$) and skip connections from Equation 5.4. The encoder r is a lookup-table with N entries in $\mathbb{R}^{128}$. Each f^i is a two layer neural net with ReLU non-linearities that takes in the concatenation of (h^i, c^i, h^0), and outputs a vector in $\mathbb{R}^{128}$. The decoder is a linear layer plus softmax, producing a distribution over the m levers, from which we sample to determine the lever to be pulled. We compare it against the independent controller, which has the same architecture as our model except that communication c is zeroed. The results are shown in Table 5.1. The metric is the number of distinct levers pulled divided by $m = 5$, averaged over 500 trials, after seeing 50000 batches of size 64 during training. We explore both reinforcement (see Section 2.1) and direct supervision (using the solution given by sorting the agent IDs, and having each agent pull the lever according to its relative order in the current m agents). In both cases, the CommNet performs significantly better than the independent controller.

We analyze a CommNet model trained with supervision on the lever pulling task. The supervision uses the sorted ordering of agent IDs to assign target actions. For each agent, we concatenate its hidden layer activations during game playing. Figure 5.2 shows 3D PCA plot of those vectors, where color intensity represents agent's ID. The smooth ordering suggests that agents are communicating their IDs, enabling them to solve the task.

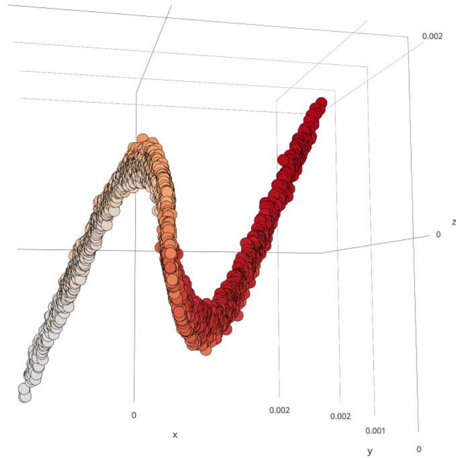

Figure 5.2: 3D PCA plot of hidden states of agents

5.4.3 Multi-turn Games

In this section, we consider two multi-agent tasks using the MazeBase environment from Chapter 3 that use reward as their training signal. The first task is to control cars passing through a traffic junction to maximize the flow while minimizing collisions. The second task is to control multiple agents in combat against enemy bots.

We experimented with several module types. With a feedforward MLP, the module f^i is a single layer network and $K = 2$ communication steps are used. For an RNN module, we also used a single layer network for f^t, but shared parameters across time steps. Finally, we used an LSTM for f^t. In all modules, the hidden layer size is set to 50. MLP modules use skip-connections.

We use REINFORCE from Section 2.1 for training, where α in Equation 2.5 is set to 0.03. Both tasks are trained for 300 epochs, each epoch being 100 weight updates with RMSProp [Tieleman and Hinton, 2012] on mini-batch of 288 game episodes (distributed over multiple CPU cores). In total, the models experience $\sim$8.6M episodes

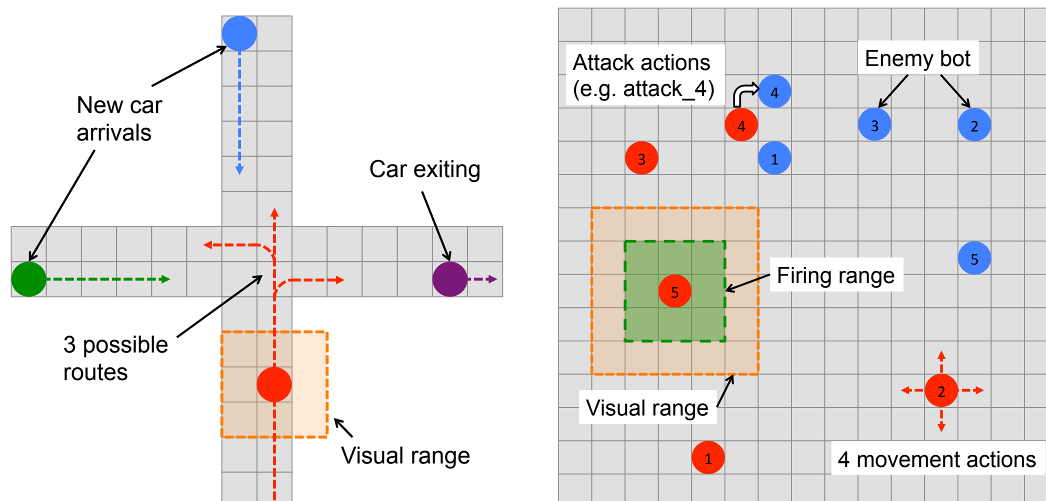

Figure 5.3: Left: Traffic junction task where agent-controlled cars (colored circles) have to pass the through the junction without colliding. Right: The combat task, where model controlled agents (red circles) fight against enemy bots (blue circles). In both tasks each agent has limited visibility (orange region), thus is not able to see the location of all other agents.

during training. We repeat all experiments 5 times with different random initializations, and report mean value along with standard deviation. The training time varies from a few hours to a few days depending on task and module types.

5.4.3.1 Traffic Junction

This consists of a 4-way junction on a 14×14 grid as shown in Figure 5.3(left). At each time step, new cars enter the grid with probability p_{arrive} from each of the four directions. However, the total number of cars at any given time is limited to $N_{\text{max}} = 10$. Each car occupies a single cell at any given time and is randomly assigned to one of three possible routes (keeping to the right-hand side of the road). At every time step, a car has two possible actions: *gas* which advances it by one cell on its route or *brake* to stay at its current location. A car will be removed once it reaches its destination at the edge of the grid.

The number of cars dynamically change during an episode as new cars arrive and

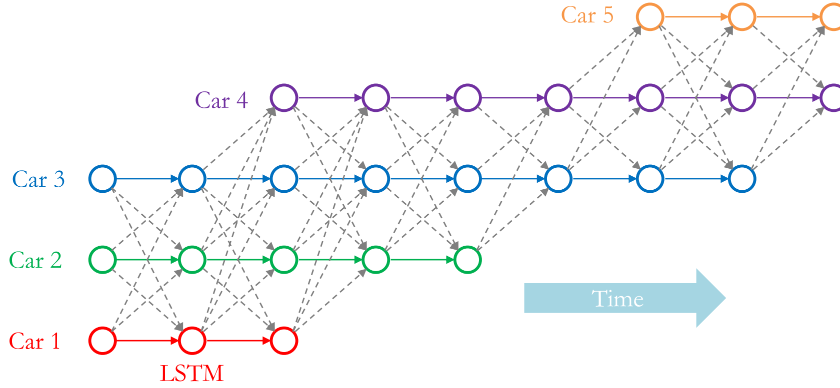

Figure 5.4: An example CommNet architecture for a varying number agents. Each car is controlled by its own LSTM (solid edges), but communication channels (dashed edges) allow them to exchange information.

others leave the grid. Therefore it is an ideal task for our model which can adjust to an arbitrary number of agents. Figure 5.4 shows this when a LSTM is used as f^t module. The computation graph of a CommNet dynamically changes at each time step to accommodate a varying number of agents. Note this differs from previous Graph NN formulations where a graph was fixed during computation iterations.

Two cars *collide* if their locations overlap. A collision incurs a reward $r_{coll} = -10$, but does not affect the simulation in any other way. To discourage a traffic jam, each car gets reward of $\tau r_{time} = -0.01\tau$ at every time step, where τ is the number time steps passed since the car arrived. Therefore, the total reward at time t is:

$$r_t = C^t r_{coll} + \sum_{i=1}^{N^t} \tau_i r_{time},$$

where C^t is the number of collisions occurring at time t, and N^t is number of cars present. The simulation is terminated after 40 steps and is classified as a failure if one or more collisions have occurred.

Each car is represented by one-hot binary vector set $\{n, l, r\}$, that encodes its unique

Model Φ	Module $f()$ type MLP	RNN	LSTM
Independent	20.6$\pm$ 14.1	19.5$\pm$ 4.5	9.4$\pm$ 5.6
Fully-connected	12.5$\pm$ 4.4	34.8$\pm$ 19.7	4.8$\pm$ 2.4
Discrete comm.	15.8$\pm$ 9.3	15.2$\pm$ 2.1	8.4$\pm$ 3.4
CommNet	**2.2**$\pm$ **0.6**	**7.6**$\pm$ **1.4**	**1.6**$\pm$ **1.0**

Table 5.2: Traffic junction task failure rates (%) for different types of model and module function $f(.)$. CommNet consistently improves performance, over the baseline models.

ID, current location and assigned route number respectively. Each agent controlling a car can only observe other cars in its vision range (a surrounding 3×3 neighborhood), but it can communicate to all other cars. The state vector s_j for each agent is thus a concatenation of all these vectors, having dimension $3^2 \times |n| \times |l| \times |r|$.

We use curriculum learning [Bengio et al., 2009a] to make the training easier. In first 100 epochs of training, we set $p_{\text{arrive}} = 0.05$, but linearly increased it to 0.2 during next 100 epochs. Finally, training continues for another 100 epochs. The learning rate is fixed at 0.003 throughout.

In Table 5.2, we show the probability of failure of a variety of different model Φ and module f pairs. Compared to the baseline models, CommNet significantly reduces the failure rate for all module types, achieving the best performance with LSTM module (a video showing this model before and after training can be found at `http://cims.nyu.edu/~sainbar/commnet`).

We also explored how partial visibility within the environment effects the advantage given by communication. As the vision range of each agent decreases, the advantage of communication increases as shown in Figure 5.5. Impressively, with zero visibility (the cars are driving blind) the CommNet model is still able to succeed 90% of the time.

Table 5.3 shows the results on easy and hard versions of the game. The easy version is a junction of two one-way roads on a 7×7 grid. There are two arrival points, each

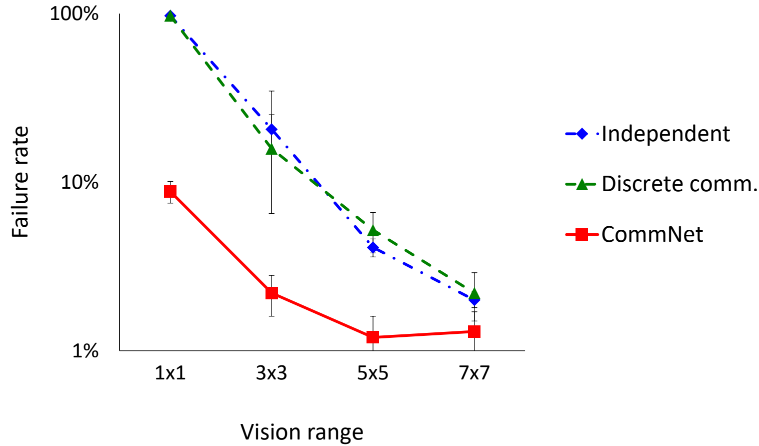

Figure 5.5: As visibility in the environment decreases, the importance of communication grows in the traffic junction task.

Model Φ	Other game versions Easy (MLP)	Hard (RNN)
Independent	15.8± 12.5	26.9± 6.0
Discrete comm.	1.1± 2.4	28.2± 5.7
CommNet	**0.3**± **0.1**	22.5± 6.1
CommNet local	-	**21.1**± **3.4**

Table 5.3: Traffic junction task variants. In the easy case, discrete communication does help, but still less than CommNet. On the hard version, local communication (see Section 5.2.2) does at least as well as broadcasting to all agents.

with two possible routes. During curriculum, we increase N_{total} from 3 to 5, and p_{arrive} from 0.1 to 0.3. The harder version consists from four connected junctions of two-way roads in 18×18 as shown in Figure 5.6. There are 8 arrival points and 7 different routes for each arrival point. We set $N_{\text{total}} = 20$, and increased p_{arrive} from 0.02 to 0.05 during curriculum. Discrete communication works well on the easy version, but the CommNet with local connectivity gives the best performance on the hard case.

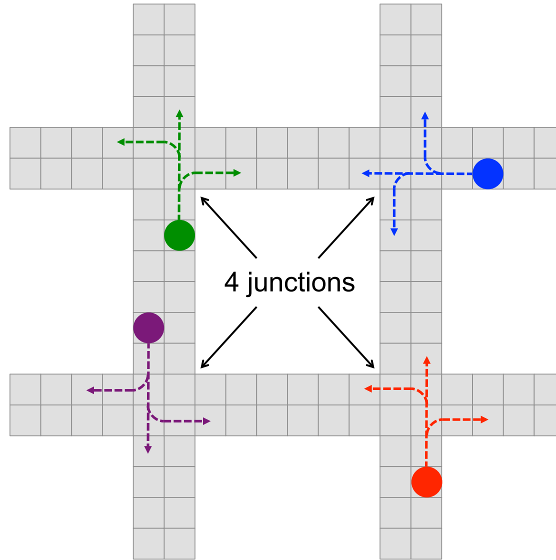

Figure 5.6: A harder version of traffic task with four connected junctions.

5.4.3.2 Analysis of Communication

We now attempt to understand what the agents communicate when performing the junction task. We start by recording the hidden state h_j^i of each agent and the corresponding *communication vectors* $\tilde{c}_j^{i+1} = C^{i+1} h_j^i$ (the contribution agent j at step $i+1$ makes to the hidden state of other agents). Figure 5.7(left) and Figure 5.7(right) show the 2D PCA projections of the communication and hidden state vectors respectively. These plots show a diverse range of hidden states but far more clustered communication vectors, many of which are close to zero. This suggests that while the hidden state carries information, the agent often prefers not to communicate it to the others unless necessary. This is a possible consequence of the broadcast channel: if everyone talks at the same time, no-one can understand.

To better understand the meaning behind the communication vectors, we ran the simulation with only two cars and recorded their communication vectors and locations whenever one of them braked. Vectors belonging to the clusters A, B & C in Fig-

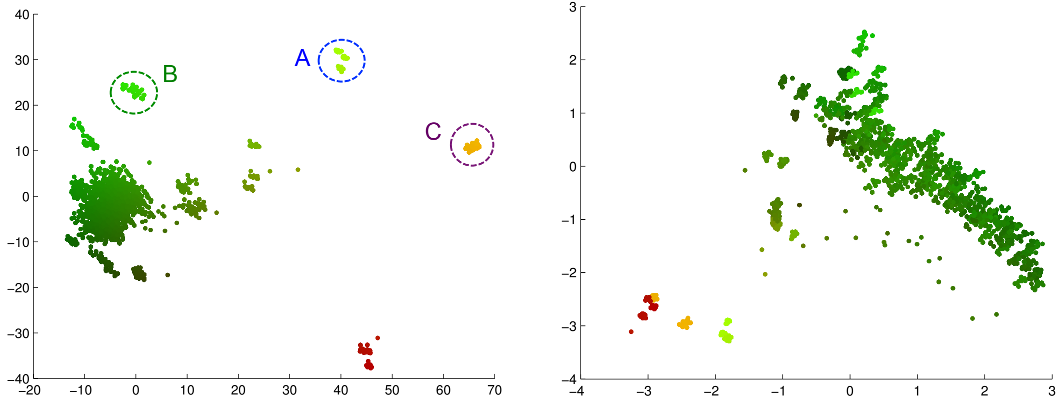

Figure 5.7: Left: First two principal components of communication vectors $\tilde{c}$ from multiple runs on the traffic junction task Figure 5.3(left). While the majority are "silent" (i.e. have a small norm), distinct clusters are also present. Right: First two principal components of hidden state vectors h from the same runs as on the left, with corresponding color coding. Note how many of the "silent" communication vectors accompany non-zero hidden state vectors. This shows that the two pathways carry different information.

ure 5.7(left) were consistently emitted when one of the cars was in a specific location, shown by the colored circles in Figure 5.8 (or pair of locations for cluster C). They also strongly correlated with the other car braking at the locations indicated in red, which happen to be relevant to avoiding collision.

In addition, we also visualize the average norm of the communication vectors in Figure 5.9(left) and brake locations over the 14×14 spatial grid in Figure 5.9(right). In each of the four incoming directions, there is one location where communication signal is stronger. The brake pattern shows that cars coming from left never yield to other directions.

5.4.3.3 Combat Task

We simulate a simple battle involving two opposing teams in a 15×15 grid as shown in Figure 5.3(right). Each team consists of $m = 5$ agents and their initial positions are sampled uniformly in a 5×5 square around the team center, which is picked uniformly

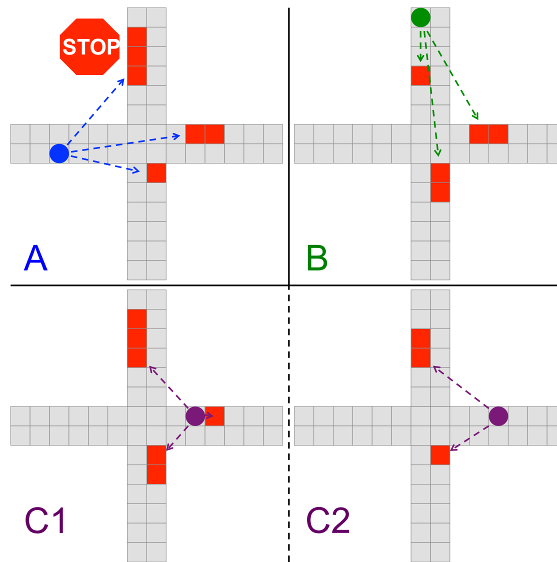

Figure 5.8: For three of clusters shows in Figure 5.7(left), we probe the model to understand their meaning (see text for details).

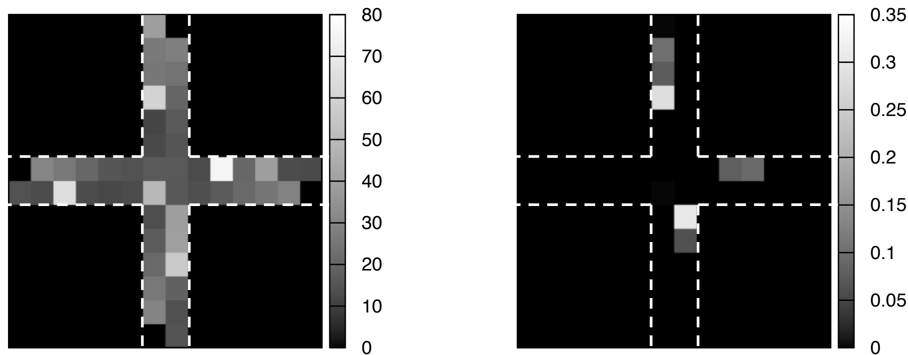

Figure 5.9: (left) Average norm of communication vectors (right) Brake locations

in the grid. At each time step, an agent can perform one of the following actions: move one cell in one of four directions; attack another agent by specifying its ID j (there are m attack actions, each corresponding to one enemy agent); or do nothing. If agent A attacks agent B, then B's health point will be reduced by 1, but only if B is inside the firing range of A (its surrounding 3×3 area). Agents need one time step of cooling down after an attack, during which they cannot attack. All agents start with 3 health points, and die when their health reaches 0. A team will win if all agents in the other

Model Φ	Module $f()$ type		
	MLP	RNN	LSTM
Independent	34.2± 1.3	37.3± 4.6	44.3± 0.4
Fully-connected	17.7± 7.1	2.9± 1.8	19.6± 4.2
Discrete comm.	29.1± 6.7	33.4± 9.4	46.4± 0.7
CommNet	**44.5**± **13.4**	**44.4**± **11.9**	**49.5**± **12.6**

Table 5.4: Win rates (%) on the combat task for different communication approaches and module choices. Continuous consistently outperforms the other approaches. The fully-connected baseline does worse than the independent model without communication.

team die. The simulation ends when one team wins, or neither of teams win within 40 time steps (a draw).

The model controls one team during training, and the other team consist of bots that follow a hard-coded policy. The bot policy is to attack the nearest enemy agent if it is within its firing range. If not, it approaches the nearest visible enemy agent within visual range. An agent is visible to all bots if it is inside the visual range of any individual bot. This shared vision gives an advantage to the bot team. When input to a model, each agent is represented by a set of one-hot binary vectors $\{i, t, l, h, c\}$ encoding its unique ID, team ID, location, health points and cooldown. A model controlling an agent also sees other agents in its visual range (3×3 surrounding area). The model gets reward of -1 if the team loses or draws at the end of the game. In addition, it also get reward of -0.1 times the total health points of the enemy team, which encourages it to attack enemy bots.

Table 5.4 and Figure 5.10 shows the win rate of different module choices with various types of model. Among different modules, the LSTM achieved the best performance. Continuous communication with CommNet improved all module types. Relative to the independent controller, the fully-connected model degraded performance, but the discrete communication improved LSTM module type. We also explored several

Model Φ	Other game variations (MLP) $m = 3$	$m = 10$	5×5 vision
Independent	29.2$\pm$ 5.9	30.5$\pm$ 8.7	60.5$\pm$ 2.1
CommNet	**51.0**$\pm$ **14.1**	**45.4**$\pm$ **12.4**	**73.0**$\pm$ **0.7**

Table 5.5: Win rates (%) on the combat task for different communication approaches. We explore the effect of varying the number of agents m and agent visibility. Even with 10 agents on each team, communication clearly helps.

variations of the task: varying the number of agents in each team by setting $m = 3, 10$, and increasing visual range of agents to 5×5 area. The result on those tasks are shown on Table 5.5. Using CommNet model consistently improves the win rate, even with the greater environment observability of the $5{\times}5$ vision case.

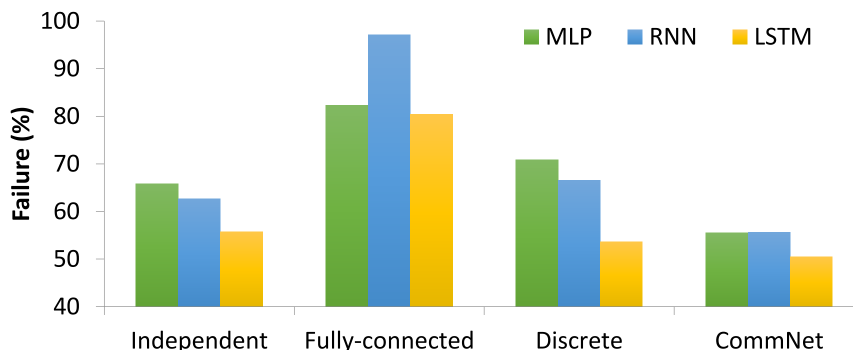

Figure 5.10: Failure rates of different communication approaches on the combat task.

5.4.4 bAbI Tasks

We apply our model to the bAbI [Weston et al., 2016] toy Q & A dataset from Section 4.4, and compare against MemN2N from Chapter 4 and other baselines. The goal of the task is to answer a question after reading a short story. We can formulate this as a multi-agent task by giving each sentence of the story its own agent. Communication among agents allows them to exchange useful information necessary to answer the

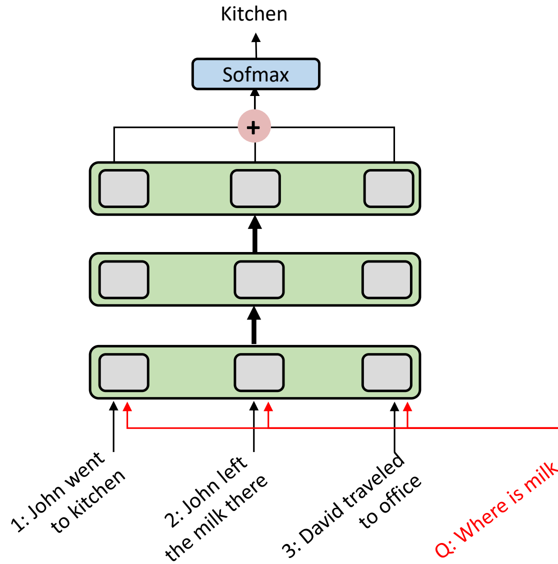

Figure 5.11: CommNet applied to bAbI tasks. Each sentence of the story is processed by its own agent/stream, while a question is fed through the initial communication vector. A single output is obtained by simply summing the final hidden states.

question.

Let the task be $\{s_1, s_2, ..., s_J, q, y^*\}$, where s_j is j-th sentence of story, q is the question sentence and y^* is the correct answer word (when answer is multiple words, we simply concatenate them into single word). Then the input to the model is

$$h_j^0 = r(s_j, \theta_0), \quad c_j^0 = r(q, \theta_q).$$

Here, we use simple position encoding from Section 4.4.1.1 as r to convert sentences into fixed size vectors. Also, the initial communication is used to broadcast the question to all agents. Since the temporal ordering of sentences is relevant in some tasks, we add special temporal word "$t = J - j$" to s_j for all j as done in Section 4.4.1.2.

The $f(.)$ module consists of a two-layer MLP with a skip connection

$$h_j^{i+1} = \sigma(W_i \sigma(H^i h_j^i + C^i c_j^i + h_j^0)),$$

where σ is ReLU non-linearity (bias terms are omitted for clarity). After $K = 2$ communication steps, we add the final hidden states together and pass it through a softmax decoder layer to sample an output word y as shown in Figure 5.11

$$y = \text{Softmax}(D \sum_{j=1}^{J} h_j^K).$$

The model is trained in a supervised fashion using a cross-entropy loss between y and the correct answer y^*. The hidden layer size is set to 100 and weights are initialized from $\mathcal{N}(0, 0.2)$. We train the model for 100 epochs with learning rate 0.003 and mini-batch size 32 with Adam optimizer [Kingma and Ba, 2015] ($\beta_1 = 0.9, \beta_2 = 0.99, \epsilon = 10^{-6}$). We used 10% of training data as validation set to find optimal hyper-parameters for the model.

Results on the 10K version of the bAbI task are shown in Table 5.6, along with other baselines: LSTM , MemN2N, DMN+ [Xiong et al., 2016b] and Neural Reasoner+ [Peng et al., 2015]. Our model outperforms the LSTM baseline, but is worse than the MemN2N model, which is specifically designed to solve reasoning over long stories. However, it successfully solves most of the tasks, including ones that require information sharing between two or more agents through communication.

	Error on tasks (%)							Mean error	Failed tasks
	2	3	15	16	17	18	19	(%)	(err. $>$ 5%)
LSTM	81.9	83.1	78.7	51.9	50.1	6.8	90.3	36.4	16
MemN2N	**0.3**	2.1	**0.0**	51.8	18.6	5.3	2.3	4.2	3
DMN+	**0.3**	**1.1**	**0.0**	**45.3**	4.2	2.1	**0.0**	**2.8**	**1**
Neural Reasoner+	-	-	-	-	**0.9**	-	1.6	-	-
Independent (MLP)	69.0	69.5	29.4	47.4	4.0	**0.6**	45.8	15.2	9
CommNet (MLP)	3.2	68.3	**0.0**	51.3	15.1	1.4	**0.0**	7.1	3

Table 5.6: Experimental results on bAbI tasks. Only showing some of the task with high errors.

5.5 Discussion and Future Work

We have introduced CommNet, a simple controller for MARL that is able to learn continuous communication between a dynamically changing set of agents. Evaluations on four diverse tasks clearly show the model outperforms models without communication, fully-connected models, and models using discrete communication. Despite the simplicity of the broadcast channel, examination of the traffic task reveals the model to have learned a sparse communication protocol that conveys meaningful information between agents. Code for our model (and baselines) can be found at `http://cims.nyu.edu/~sainbar/commnet/`.

One aspect of our model that we did not fully exploit is its ability to handle heterogenous agent types and we hope to explore this in future work. Furthermore, we believe the model will scale gracefully to large numbers of agents, perhaps requiring more sophisticated connectivity structures; we also leave this to future work.

Chapter 6

Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play

We describe a simple scheme that allows an agent to learn about its environment in an unsupervised manner. Our scheme pits two versions of the same agent, Alice and Bob, against one another. Alice proposes a task for Bob to complete; and then Bob attempts to complete the task. In this chapter, we will focus on two kinds of environments: (nearly) reversible environments and environments that can be reset. Alice will "propose" the task by doing a sequence of actions and then Bob must undo or repeat them, respectively. Via an appropriate reward structure, Alice and Bob automatically generate a curriculum of exploration, enabling unsupervised training of the agent. When Bob is deployed on an RL task within the environment, this unsupervised training reduces the number of supervised episodes needed to learn, and in some cases converges to a higher reward.

6.1 Introduction

Model-free approaches to reinforcement learning are sample inefficient, typically requiring a huge number of episodes to learn a satisfactory policy. The lack of an explicit environment model means the agent must learn the rules of the environment from scratch at the same time as it tries to understand which trajectories lead to rewards. In environments where reward is sparse, only a small fraction of the agents' experience is directly used to update the policy, contributing to the inefficiency.

In this chapter we introduce a novel form of unsupervised training for an agent that enables exploration and learning about the environment without any external reward that incentivizes the agents to learn how to transition between states as efficiently as possible. We demonstrate that this unsupervised training allows the agent to learn new tasks within the environment quickly.

Figure 6.1: Illustration of the self-play concept in a gridworld setting. Training consists of two types of episode: self-play and target task. In the former, Alice and Bob take turns moving the agent within the environment. Alice sets tasks by altering the state via interaction with its objects (key, door, light) and then hands control over to Bob. He must return the environment to its original state to receive an internal reward. This task is just one of many devised by Alice, who automatically builds a curriculum of increasingly challenging tasks. In the target task, Bob's policy is used to control the agent, with him receiving an *external* reward if he visits the flag. He is able to learn to do this quickly as he is already familiar with the environment from self-play.

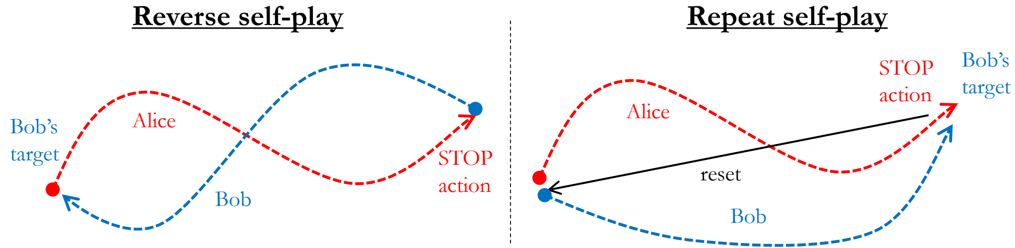

Figure 6.2: An illustration of two versions of the self-play: (left) Bob starts from Alice's final state in a reversible environment and tries to return her initial state; (right) if an environment can be reset to Alice's initial state, Bob starts there and tries to reach the same state as Alice.

6.2 Approach

We consider environments with a single physical agent (or multiple physical units controlled by a single agent), but we allow it to have two separate "minds": Alice and Bob, each with its own objective and parameters. During self-play episodes, Alice's job is to propose a task for Bob to complete, and Bob's job is to complete the task. When presented with a target task episode, Bob is then used to perform it (Alice plays no role). The key idea is that the Bob's play with Alice should help him understand how the environment works and enable him to learn the target task more quickly.

Our approach is restricted to two classes of environment: (i) those that are (nearly) reversible, or (ii) ones that can be reset to their initial state (at least once). These restrictions allow us to sidestep complications around how to communicate the task and determine its difficulty (see Section 6.5.2 for further discussion). In these two scenarios, Alice starts at some initial state s_0 and proposes a task by *doing* it, i.e. executing a sequence of actions that takes the agent to a state s_t. She then outputs a STOP action, which hands control over to Bob as shown in Figure 6.2. In reversible environments, Bob's goal is to return the agent back to state s_0 (or within some margin of it, if the state is continuous), to receive reward. In partially observable environments, the objective is

Algorithm 6.1 Pseudo code for training an agent on a self-play episode

```
function SELFPLAYEPISODE(REVERSE/REPEAT, t_Max, θ_A, θ_B)
    t_A ← 0
    s_0 ← env.observe()
    s* ← s_0
    while True do
        # Alice's turn
        t_A ← t_A + 1
        s ← env.observe()
        a ← π_A(s, s_0) = f(s, s_0, θ_A)
        if a = STOP or t_A ≥ t_Max then
            s* ← s
            env.reset()
            break
        env.act(a)
    t_B ← 0
    while True do
        # Bob's turn
        s ← env.observe()
        if s = s* or t_A + t_B ≥ t_Max then
            break
        t_B ← t_B + 1
        a ← π_B(s, s*) = f(s, s*, θ_B)
        env.act(a)
    R_A ← γ max(0, t_B − t_A)
    R_B ← −γ t_B
    policy.update(R_A, θ_A)
    policy.update(R_B, θ_B)
    return
```

relaxed to Bob finding a state that returns the same observation as Alice's initial state. In environments where resets are permissible, Alice's STOP action also reinitializes the environment, thus Bob starts at s_0 and now must reach s_t to be rewarded, thus repeating Alice's task instead of reversing it. See Figure 6.1 for an example, and Algorithm 6.1 for the pseudo code.

In both cases, this self-play between Alice and Bob only involves internal reward (detailed below), thus the agent can be trained without needing any supervisory signal

Algorithm 6.2 Pseudo code for training an agent on a target task episode

```
function TargetTaskEpisode(t_Max, θ_B)
    t ← 0
    R ← 0
    while True do
        t ← t + 1
        s ← env.observe()
        a ← π_B(s, ∅) = f(s, ∅, θ_B)
        if env.done() or t ≥ t_Max then
            break
        env.act(a)
        R = R+ env.reward()
    policy.update(R, θ_B)
    return
```

from the environment. As such, it comprises a form of unsupervised training where Alice and Bob explore the environment and learn how it operates. This exploration can be leveraged for some target task by training Bob on target task episodes in parallel. The idea is that Bob's experience from self-play will help him learn the target task in fewer episodes. The reason behind choosing Bob for the target task is because he learns to transfer from one state to another efficiently from self-play. See Algorithm 6.2 for the pseudo code of target task training.

For self-play, we choose the reward structure for Alice and Bob to encourage Alice to push Bob past his comfort zone, but not give him impossible tasks. Denoting Bob's total reward by R_B (given at the end of episodes) and Alice's total reward by R_A, we use

$$R_B = -\gamma t_B \tag{6.1}$$

where t_B is the time taken by Bob to complete his task and

$$R_A = \gamma \max(0, t_B - t_A) \tag{6.2}$$

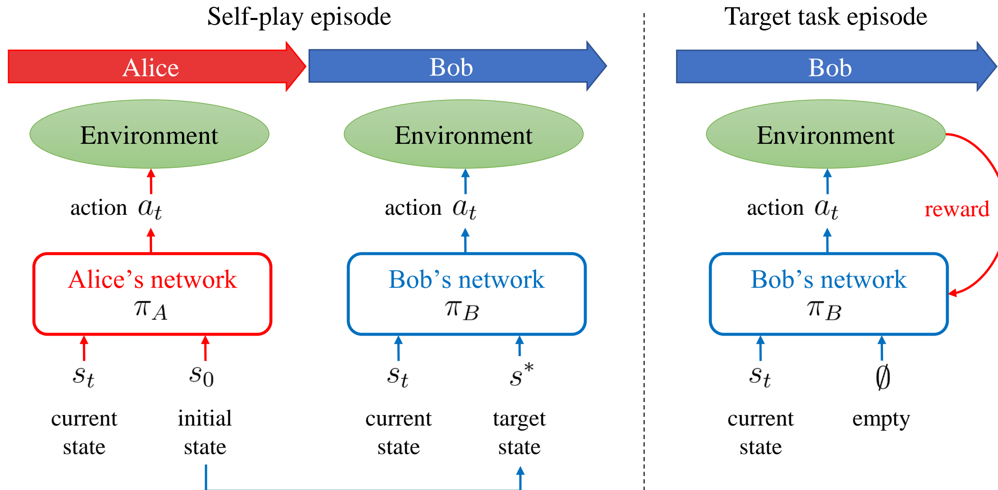

Figure 6.3: The policy networks of Alice and Bob takes take the initial state s_0 and the target state s^* as an additional input respectively. We set $s^* = s_0$ in a reverse self-play, but set to zero $s^* = \emptyset$ for target task episodes. Note that rewards from the environment are only used during target task episodes.

where t_A is the time until Alice performs the STOP action, and γ is a scaling coefficient that balances this internal reward to be of the same scale as external rewards from the target task. The total length of an episode is limited to t_{Max}, so if Bob fails to complete the task in time we set $t_B = t_{\text{Max}} - t_A$.

Thus Alice is rewarded if Bob takes more time, but the negative term on her own time will encourage Alice not to take too many steps when Bob is failing. For both reversible and resettable environments, Alice must limit her steps to make Bob's task easier, thus Alice's optimal behavior is to the find simplest tasks that Bob cannot complete. This eases learning for Bob since the new task will be only just beyond his current capabilities. The self-regulating feedback between Alice and Bob allows them to automatically construct a curriculum for exploration, a key contribution of our approach.

6.2.1 Parameterizing Alice and Bob's actions

Alice and Bob each have policy functions which take as input two observations of state variables, and output a distribution over actions as showns in Figure 6.3. In Alice's case, the function will be of the form

$$a_t = \pi_A(s_t, s_0),$$

where s_0 is the observation of the initial state of the environment and s_t is the observation of the current state. In Bob's case, the function will be

$$a_t = \pi_B(s_t, s^*),$$

where s^* is the target state that Bob has to reach, and set to s_0 when we have a reversible environment. In a resettable environment s^* is the state where Alice executed the STOP action.

When a target task is presented, the agent's policy function is $a_t = \pi_B(s_t, \emptyset)$, where the second argument of Bob's policy is simply set to zero.[1] If s^* is always non-zero, then this is enough to let Bob know whether the current episode is self-play or target task. In some experiments where s^* can be zero, we give third argument $z \in \{0, 1\}$ that explicitly indicates the episode kind.

In the experiments below, we demonstrate our approach in settings where π_A and π_B are tabular; where it is a neural network taking discrete inputs, and where it is a neural network taking in continuous inputs. When using a neural network, we use the same

[1] Note that Bob can be used in multi-task learning by feeding the task description into π_B

network architecture for both Alice and Bob, except they have different parameters

$$\pi_A(s_t, s_0) = f(s_t, s_0, \theta_A), \quad \pi_B(s_t, s^*) = f(s_t, s^*, \theta_B),$$

where f is an multi-layered neural network with parameters θ_A or θ_B.

6.2.2 Universal Bob in the tabular setting

We now present a theoretical argument that shows for environments with finite states, tabular policies, and deterministic, Markovian transitions, we can interpret the self-play as training Bob to find a policy that can get from any state to any other in the least expected number of steps.

Preliminaries: Note that, as discussed above, the policy table for Bob is indexed by (s_t, s^*), not just by s_t. In particular, with the assumptions above, this means that there is a *fast policy* π_{fast} such that $\pi_{\text{fast}}(s_t, s^*)$ has the smallest expected number of steps to transition from s_t to s^*. It is clear that π_{fast} is a universal policy for Bob, such that $\pi_B = \pi_{\text{fast}}$ is optimal with respect to any Alice's policy π_A. In a reset game, π_{fast} nets Alice a return of 0, and in the reverse game, the return of π_{fast} against an optimal Alice can be considered a measure of the reversibility of the environment. However, in what follows let us assume that either the reset game or the reverse game in a perfectly reversible environment is used. Also, let assume the initial states are randomized and its distribution covers the entire state space.

Claim: If π_A and π_B are policies of Alice and Bob that are in equilibrium (i.e., Alice cannot be made better without changing Bob, and vice-versa), then π_B is a fast policy.

Argument: Let us first show that Alice will always get zero reward in equilibrium. If Alice is getting positive reward on some challenge, that means Bob is taking longer

than Alice on that challenge. Then Bob can be improved to use π_{fast} at that challenge, which contradicts the equilibrium assumption.

Now let us prove π_B is a fast policy by contradiction. If π_B is not fast, then there must exist a challenge (s_t, s^*) where π_B will take longer than π_{fast}. Therefore Bob can get more reward by using π_{fast} if Alice does propose that challenge with non-zero probability. Since we assumed equilibrium and π_B cannot be improved while π_A fixed, the only possibility is that Alice is never proposing that challenge. If that is true, Alice can get positive reward by proposing that task using the same actions as π_{fast}, so taking fewer steps than π_B. However this contradicts with the proof that Alice always gets zero reward, making our initial assumption "π_B is not fast" wrong.

6.3 Related Work

Self-play arises naturally in reinforcement learning, and has been well studied. For example, for playing checkers [Samuel, 1959], backgammon [Tesauro, 1995], and Go, [Silver et al., 2016a], and in multi-agent games such as RoboSoccer [Riedmiller et al., 2009]. Here, the agents or teams of agents compete for external reward. This differs from our scheme where the reward is purely internal and the self-play is a way of motivating an agent to learn about its environment to augment sparse rewards from separate target tasks.

Our approach has some relationships with generative adversarial networks (GANs) [Goodfellow et al., 2014], which train a generative neural net by having it try to fool a discriminator network which tries to differentiate samples from the training examples. [Li et al., 2017] introduce an adversarial approach to dialogue generation, where a generator model is subjected to a form of "Turing test" by a discriminator network.

[Mescheder et al., 2017] demonstrate how adversarial loss terms can be combined with variational auto-encoders to permit more accurate density modeling. While GAN's are often thought of as methods for training a generator, the generator can be thought of as a method for generating hard negatives for the discriminator. From this viewpoint, in our approach, Alice acts as a "generator", finding "negatives" for Bob. However, Bob's jobs is to complete the generated challenge, not to discriminate it.

There is a large body of work on intrinsic motivation [Barto, 2013, Singh et al., 2004, Klyubin et al., 2005, Schmidhuber, 1991] for self-supervised learning agents. These works propose methods for training an agent to explore and become proficient at manipulating its environment without necessarily having a specific target task, and without a source of extrinsic supervision. One line in this direction is curiosity-driven exploration [Schmidhuber, 1991]. These techniques can be applied in encouraging exploration in the context of reinforcement learning, for example [Bellemare et al., 2016, Strehl and Littman, 2008, Lopes et al., 2012, Tang et al., 2017, Pathak et al., 2017]; Roughly, these use some notion of the novelty of a state to give a reward. In the simplest setting, novelty can be just the number of times a state has been visited; in more complex scenarios, the agent can build a model of the world, and the novelty is the difficulty in placing the current state into the model. In our work, there is no explicit notion of novelty. Even if Bob has seen a state many times, if he has trouble getting to it, Alice should force him towards that state. Another line of work on intrinsic motivation is a formalization of the notion of empowerment [Klyubin et al., 2005], or how much control the agent has over its environment. Our work is related in the sense that it is in both Alice's and Bob's interests to have more control over the environment; but we do not explicitly measure that control except in relation to the tasks that Alice sets.

Curriculum learning [Bengio et al., 2009b] is widely used in many machine learning

approaches. Typically however, the curriculum requires at least some manual specification. A key point about our work is that Alice and Bob devise their own curriculum entirely automatically. Previous automatic approaches, such as [Kumar et al., 2010], rely on monitoring training error. But since ours is unsupervised, no training labels are required either.

Our basic paradigm of "Alice proposing a task, and Bob doing it" is related to the Horde architecture [Sutton et al., 2011, Schaul et al., 2015]. In those works, instead of using a value function $V = V(s)$ that depends on the current state, a value function that explicitly depends on state and goal $V = V(s, g)$ is used. In our experiments, our models will be parameterized in a similar fashion. The novelty in this work is in how Alice defines the goal for Bob.

The closest work to ours is that of [Baranes and Oudeyer, 2013], who also have one part of the model that proposes tasks, while another part learns to complete them. As in this work, the policies and cost are parameterized as functions of both state and goal. However, our approach differs in the way tasks are proposed and communicated. In particular, in [Baranes and Oudeyer, 2013], the goal space has to be presented in a way that allows explicit partitioning and sampling, whereas in our work, the goals are sampled through Alice's actions. On the other hand, we pay for not having to have such a representation by requiring the environment to be either reversible or resettable.

Several concurrent works are related: [Andrychowicz et al., 2017] form an implicit curriculum by using internal states as a target. [Florensa et al., 2017] automatically generate a series of increasingly distant start states from a goal.[2] [Pinto et al., 2017] use an adversarial framework to perturb the environment, inducing improved robustness of the agent. [Held et al., 2017] propose a scheme related to our "random Alice" strategy.

[2]In their paper they analyzed our approach, suggesting it was inherently unstable. However, the analysis relied on a sudden jump of Bob policy with respect to Alice's, which is unlikely to happen in practice.

6.4 Experiments

The following experiments explore our self-play approach on a variety of tasks, both continuous and discrete, from Mazebase, RLLab [Duan et al., 2016], and StarCraft [Synnaeve et al., 2016] environments. The same protocol is used in all settings: self-play and target task episodes are mixed together and used to train the agent via discrete policy gradient. We evaluate both the reverse and repeat versions of self-play. We demonstrate that the self-play episodes help training, in terms of number of target task episodes needed to learn the task. Note that we assume the self-play episodes to be "free", since they make no use of environmental reward. This is consistent with traditional semi-supervised learning, where evaluations typically are based only on the number of labeled points (not unlabeled ones too). The code we used in our experiments can be found at `http://cims.nyu.edu/~sainbar/selfplay`.

In all the experiments, we use REINFORCE from Section 2.1 for optimizing the policies. In the tabular task below, we use a constant baseline; in all the other tasks we use a neural network for both policy and baseline as in Equation 2.5 (the hyperparameter α is set to 0.1 in all experiments).

For the policy neural networks, we use two-layer fully-connected networks with 50 hidden units in each layer unless otherwise stated. All the network parameters are randomly initialized from $\mathcal{N}(0, 0.2)$, and tanh non-linearity is used. The training uses RMSProp [Tieleman and Hinton, 2012] with hyperparameters 0.97 and $1e-6$. We always do 10 runs with different random initializations and report their mean and standard deviation.

The other hyperparameter values used in the experiments are shown in Table 6.1. In some cases, we used different parameters for self-play and target task episodes. En-

Hyperparameter name	Long Hallway	Mazebase	Mountain Car	Swimmer Gather	StarCraft
Learning rate	0.1	0.003	0.003	0.003	0.003
Batch size	16	256	128	256	32
Max steps of episode (t_{max})	30	80	500	TT: 166 SP: 200	200
Entropy regularization	0	0.003	0.003	TT: 0 SP: 0.003	TT: 0 SP: 0.003
Self-play reward scale (γ)	0.033	0.1	0.01	0.01	0.01
Self-play percentage	-	20%	1%	10%	10%
Self-play mode	Reverse	Both	Repeat	Reverse	Repeat
Frame skip	0	0	0	150 [3]	23

Table 6.1: Hyperparameter values used in experiments. TT=target task, SP=self-play

tropy regularization is implemented as an additional cost maximizing the entropy of the softmax layer. In the StarCraft, skipping 23 frames roughly matches to one action per second.

6.4.1 Long hallway

We first describe a simple toy environment designed to illustrate the function of the asymmetric self-play. The environment consists of M states $\{s^1, ..., s^M\}$ arranged in a chain. Both Alice and Bob have three possible actions, "left", "right", or "stop". If the agent is at s^i with $i \neq 1$, "left" takes it to s^{i-1}; "right" analogously increases the state index, and "stop" transfers control to Bob when Alice runs it and terminates the episode when Bob runs it. We use "return to initial state" as the self-play task (i.e. Reverse in Algorithm 6.1). For the target task, we randomly pick a starting state and target state, and the episode is considered successful if Bob moves to the target state and executes

[3] Experiments in VIME and SimHash papers skip 50 frames, but we matched the total number of frames in an episode by reducing the number of steps.

the stop action before a fixed number of maximum steps.

In this case, the target task is essentially the same as the self-play task, and so running it is not unsupervised learning (and in particular, on this toy example unlike the other examples below, we do not mix self-play training with target task training). However, we see that the curriculum afforded by the self-play is efficient at training the agent to do the target task at the beginning of the training, and is effective at forcing exploration of the state space as Bob gets more competent.

In Figure 6.4 (left) we plot the number of episodes vs rate of success at the target task with four different methods. We set $M = 25$ and the maximum allowed steps for Alice and Bob to be 30. We use fully tabular controllers; the table is of size $M^2 \times 3$, with a distribution over the three actions for each possible (start, end pair).

The red curve corresponds to REINFORCE, with a penalty of -1 given upon failure to complete the task, and a penalty of $-t/t_{\text{Max}}$ for successfully completing the task in t steps. The magenta curve corresponds to taking Alice to have a random policy ($1/2$ probability of moving left or right, and not stopping till the maximum allowed steps). The green curve corresponds to policy gradient with an exploration bonus similar to [Strehl and Littman, 2008]. That is, we keep count of the number of times N_s the agent has been in each state s, and the reward for s is adjusted by exploration bonus $\alpha/\sqrt{N_s}$, where α is a constant balancing the reward from completing the task with the exploration bonus. We choose the weight α to maximize success at 0.2M episodes from the set $\{0, 0.1, 0.2, ..., 1\}$. The blue curve corresponds to the asymmetric self-play training.

We can see that at the very beginning, a random policy for Alice gives some form of curriculum but eventually is harmful, because Bob never gets to see any long treks. On the other hand, policy gradient sees very few successes in the beginning, and so trains

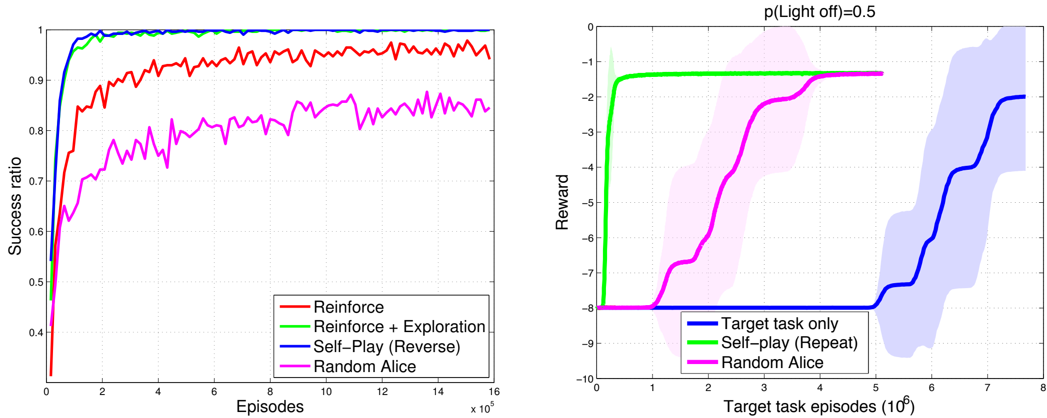

Figure 6.4: Left: The hallway task from Section 6.4.1. The y axis is fraction of successes on the target task, and the x axis is the total number of training examples seen. Plain REINFORCE (red) learns slowly. Adding an explicit exploration bonus [Strehl and Littman, 2008] (green) helps significantly. Our self-play approach (blue) gives similar performance however. Using a random policy for Alice (magenta) drastically impairs performance, showing the importance of self-play between Alice and Bob. **Right:** Mazebase task, illustrated in Figure 6.1, for p(Light off) = 0.5. Augmenting with the repeat form of self-play enables significantly faster learning than training on the target task alone and random Alice baselines.

slowly. Using the self-play method, Alice gives Bob easy problems at first (she starts from random), and then builds harder and harder problems as the training progresses, finally matching the performance boost of the count based exploration. Although not shown, similar patterns are observed for a wide range of learning rates.

6.4.2 Mazebase: Light key

We now describe experiments using the MazeBase environment from Chapter 3. We use an environment where the maze contains a light switch (whose initial state is sampled according to a predefined probability, p(Light off)), a key and a wall with a door (see Figure 6.1). An agent can open or close the door by toggling the key switch, and turn on or off light with the light switch. When the light is off, the agent can only see the (glowing) light switch. In the target task, there is also a goal flag item, and the

objective of the game is reach to that goal flag.

In self-play, the environment is the same except there is no specific objective. An episode starts with Alice in control, who can navigate through the maze and change the switch states until she outputs the STOP action. Then, Bob takes control and tries to return everything to its original state in the reverse self-play. In the repeat version, the maze resets back to its initial state when Bob takes the control, who tries to reach the final state of Alice. However, Bob does not need to worry about things that are invisible to him. For example, if Alice started with light "off" in reverse self-play, Bob does not need to match the state of the door, because it would be invisible to him when the light is off.

In the target task, the agent and the goal are always placed on opposite sides of the wall. Also, the light and key switches are placed on the same side as the agent, but the light is always off and the door is closed initially. Therefore, in order to succeed, the agent has to turn on the light, toggle the key switch to open the door, pass through it, and reach the goal flag.

Both Alice and Bob's policies are modeled by a fully-connected neural network with two hidden layers each with 100 and 50 units respectively. The encoder into each of the networks takes a bag of words over (objects, locations); that is, there is a separate word in the lookup table for each (object, location) pair.

In Figure 6.4 (right), we set p(Light off)=0.5 during self-play[4] and evaluate the repeat form of self-play, alongside two baselines: (i) target task only training (i.e. no self-play) and (ii) self-play with a random policy for Alice. With self-play, the agent succeeds quickly while target task-only training takes much longer.[5] Figure 6.5 shows details of a

[4]Changing p(Light off) adjusts the seperation between the self-play and target tasks. For a systematic evaluation of this, please see Appendix 6.4.2.1.

[5]Training was stopped for all methods except target-only at 5×10^6 episodes.

single training run, demonstrating how Alice and Bob automatically build a curriculum between themselves though self-play.

6.4.2.1 Biasing for or against self-play

The effectiveness of our approach depends in part on the similarity between the self-play and target tasks. One way to explore this in our environment is to vary the probability of the light being off initially during self-play episodes[6]. Note that the light is always off in the target task; if the light is usually on at the start of Alice's turn in reverse, for example, she will learn to turn it off, and then Bob will be biased to turn it back on. On the other hand, if the light is usually off at the start of Alice's turn in reverse, Bob is strongly biased against turning the light on, and so the test task becomes especially hard. Thus changing this probability gives us some way to adjust the similarity between the two tasks.

Figure 6.6 (left) shows what happens when p(Light off)=0.3. Here reverse self-play works well, but repeat self-play does poorly. As discussed above, this flipping, relative to the previous experiment, can be explained as follows: low p(Light off) means that Bob's task in reverse self-play will typically involve returning the light to the on position (irrespective of how Alice left it), the same function that must be performed in the target task. The opposite situation applies for repeat self-play, where Bob needs to encounter the light typically in the off position to help him with the test task.

In Figure 6.6 (right) we systematically vary p(Light off) between 0.1 and 0.9. The y-axis shows the speed-up (reduction in target task episodes) relative to training purely on the target-task for runs where the reward goes above -2. Unsuccessful runs are given a unity speed-up factor. The curves show that when the self-play task is not biased against

[6]The initial state of the light should dramatically change the behavior of the agent: if it is on then agent can directly proceed to the key.

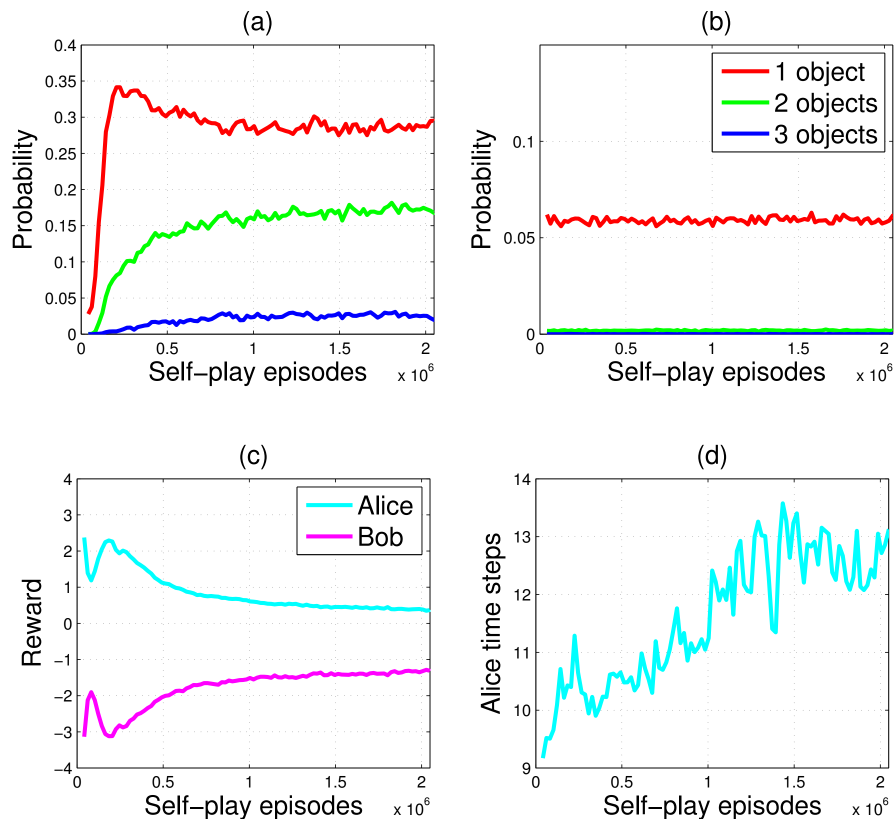

Figure 6.5: Inspection of a Mazebase learning run, using the environment shown in Figure 6.1. (a): rate at which Alice interacts with 1, 2 or 3 objects during an episode, illustrating the automatically generated curriculum. Initially Alice touches no objects, but then starts to interact with one. But this rate drops as Alice devises tasks that involve two and subsequently three objects. (b) by contrast, in the random Alice baseline, she never utilizes more than a single object and even then at a much lower rate. (c) plot of Alice and Bob's reward, which strongly correlates with (a). (d) plot of t_A as self-play progresses. Alice takes an increasing amount of time before handing over to Bob, consistent with tasks of increasing difficulty being set.

the target task it can help significantly.

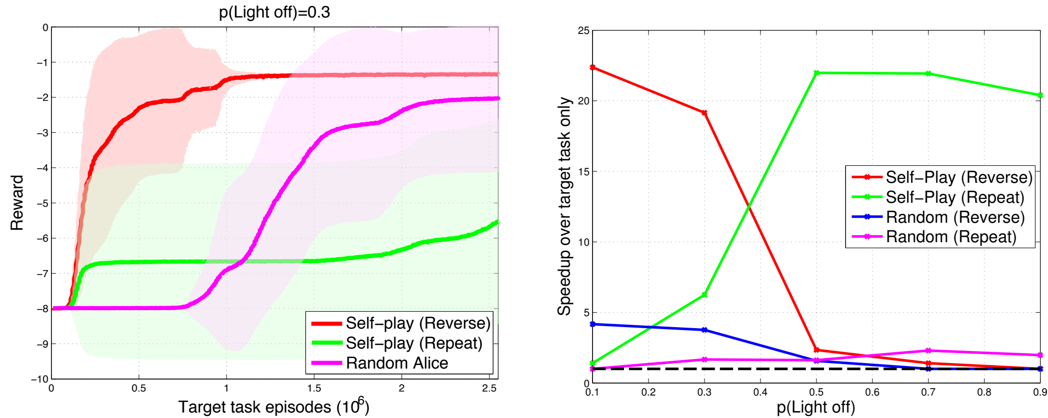

Figure 6.6: Left: The performance of self-play when p(Light off) set to 0.3. Here the reverse form of self-play works well (more details in the text). Right: Reduction in target task episodes relative to training purely on the target-task as the distance between self-play and the target task varies (for runs where the reward goes above -2 on the Mazebase task – unsuccessful runs are given a unity speed-up factor). The y axis is the speedup, and x axis is p(Light off). For reverse self-play, the low p(Light off) corresponds to having self-play and target tasks be similar to one another, while the opposite applies to repeat self-play. For both forms, significant speedups are achieved when self-play is similar to the target tasks, but the effect diminishes when self-play is biased against the target task.

6.4.3 RLLab: Mountain Car

We applied our approach to the Mountain Car task in RLLab from Section 2.3. Here the agent controls a car trapped in a 1-D valley. It must learn to build momentum by alternately moving to the left and right, climbing higher up the valley walls until it is able to escape. Although the problem is presented as continuous, we discretize the 1-D action space into 5 bins (uniformly sized) enabling us to use discrete REINFORCE, as above. We also added a secondary action head with binary actions to be used as STOP action. An observation of state s_t consists of the location and speed of the car.

As in [Houthooft et al., 2016, Tang et al., 2017], a reward of +1 is given only when

the car succeeds in climbing the hill. In self-play, Bob succeeds if $\|s_B - s_A\| < 0.2$, where s_A and s_B are the final states (location and velocity of the car) of Alice and Bob respectively.

The nature of the environment makes it highly asymmetric from Alice and Bob's point of view, since it is far easier to coast down the hill to the starting point that it is to climb up it. Hence we exclusively use the reset form of self-play. In Figure 6.7 (left), we compare this to current state-of-the-art methods, namely VIME [Houthooft et al., 2016] and SimHash [Tang et al., 2017]. Our approach (blue) performs comparably to both of these. We also tried using policy gradient directly on the target task samples, but it was unable to solve the problem.

6.4.4 RLLab: SwimmerGather

We also applied our approach to the SwimmerGather task in RLLab, where the agent controls a worm with two flexible joints, swimming in a 2D viscous fluid. In the target task, the agent gets reward +1 for eating green apples and -1 for touching red bombs, which are not present during self-play. Thus the self-play task and target tasks are different: in the former, the worm just swims around but in the latter it must learn to swim towards green apples and away from the red bombs.

The observation state consists of a 13-dimensional vector describing location and joint angles of the worm, and a 20 dimensional vector for sensing nearby objects. The worm takes two real values as an action, each controlling one joint. We add a secondary action head to our models to handle the 2nd joint, and a third binary action head for STOP action. As in the mountain car, we discretize the output space (each joint is given 9 uniformly sized bins) to allow the use of discrete policy gradients.

In addition to the REINFORCE algorithm, we also used TRPO [Schulman et al.,

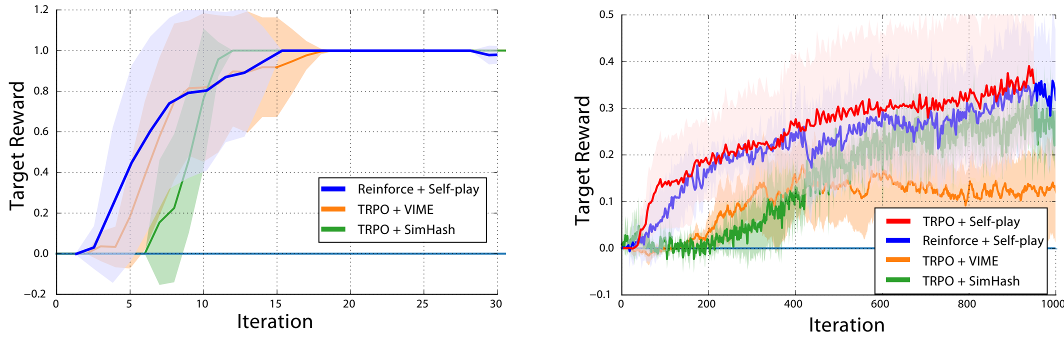

Figure 6.7: Evaluation on MountainCar (left) and SwimmerGather (right) target tasks, comparing to VIME [Houthooft et al., 2016] and SimHash [Tang et al., 2017] (figures adapted from [Tang et al., 2017]). With reversible self-play we are able to learn faster than the other approaches, although it converges to a comparable reward. Training directly on the target task using REINFORCE without self-play resulted in total failure. Here 1 iteration = 5k (50k) target task steps in Mountain car (SwimmerGather), excluding self-play steps.

2015] for training, where we used step size 0.01 and damping coefficient 0.1. The batch consists of 50,000 steps, of which 25% comes from target task episodes, while the remaining 75% is from self-play. The self-play reward scale γ set to 0.005. We used two separate network for the policy and baseline in TRPO, and the baseline network has L2 weight regularization with coefficient of $1e-5$.

Bob succeeds in a self-play episode when $\|l_B - l_A\| < 0.3$ where l_A and l_B are the final locations of Alice and Bob respectively. Figure 6.7 (right) shows the target task reward as a function of training iteration for our approach alongside state-of-the-art exploration methods VIME [Houthooft et al., 2016] and SimHash [Tang et al., 2017]. The self-play models trained with REINFORCE and TRPO performed similarly. In both cases, it enables them to gain reward significantly earlier than other methods, although both converge to a similar final value to SimHash. A video of our worm performing the test task can be found at `https://goo.gl/Vsd8Js`.

In Figure 6.8 shows details of a single training run. The changes in Alice's behavior, observed in Figure 6.8(c) and (d), correlate with Alice and Bob's reward (Figure 6.8(b))

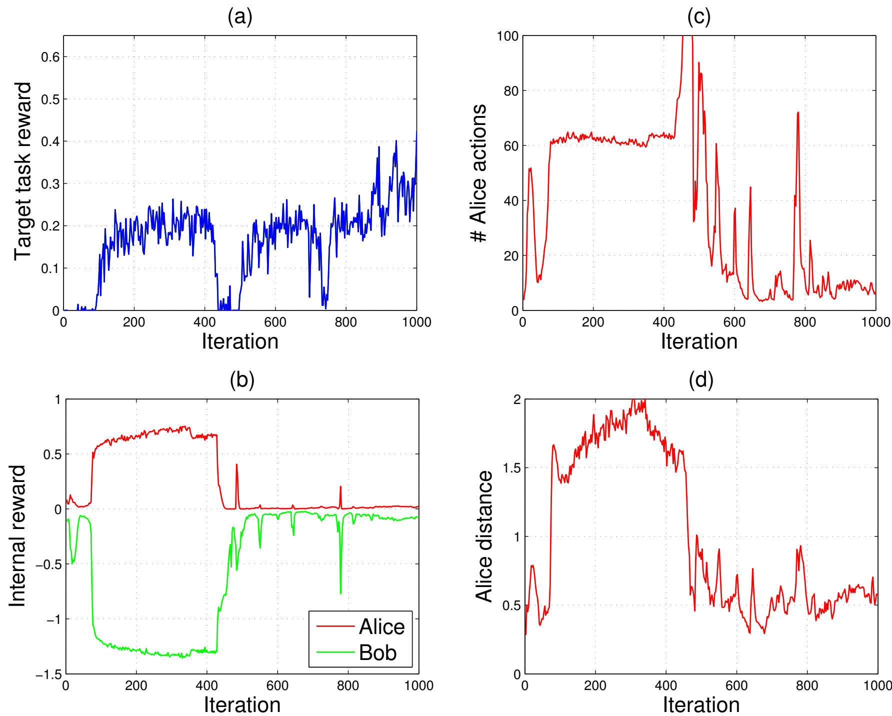

Figure 6.8: A single SwimmerGather training run. (a): Rewards on target task. (b): Rewards from reversible self-play. (c): The number of actions taken by Alice. (d): Distance that Alice travels before switching to Bob.

and, initially at least, to the reward on the test target (Figure 6.8(a)). In Figure 6.9 we visualize for a single training run the locations where Alice hands over to Bob at different stages of training, showing how the distribution varies.

6.4.5 StarCraft: Training marines

Finally, we applied the repeat variant of our self-play approach to the economic development task in StarCraft from Section 2.2.1. The target task is to build as many Marine units as possible in a fixed time. Since the reward is sparse and only given after a marine is built, exploration is important factor for this task.

During self-play episodes, Alice and Bob control the workers and can try any com-

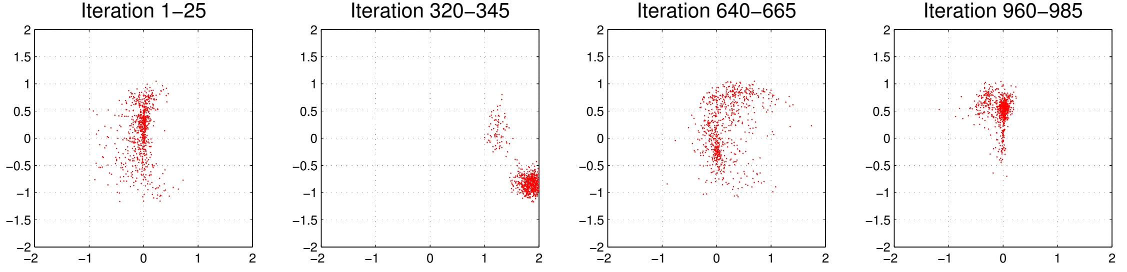

Figure 6.9: Plot of Alice's location at time of STOP action for the SwimmerGather training run shown in Figure 6.8, for different stages of training. Note how Alice's distribution changes as Bob learns to solve her tasks.

bination of actions during the episode. Since exactly matching the game state is almost impossible, Bob's success is only based on the global state $\hat{s}$ of the game, which includes the number of units of each type (including buildings), and accumulated mineral resource (see Equation 2.7 for the exact definition). Thus, Bob's policy for controlling i-th unit takes the form of

$$a_t^i = \pi_B(s_t^i, \hat{s}_t, \hat{s}^*),$$

where s_t^i is the local observation of i-th unit, and $\hat{s}_t$, $\hat{s}^*$ are the global observation of Bob's current state and his target state (i.e. the final state of Alice). Bob will succeed only if

$$\forall j : \quad \hat{s}_t[j] \geq \hat{s}^*[j],$$

where $s[j]$ denote j-th element of vector s. So Bob's objective in self-play is to make as many units and mineral as Alice in shortest possible time.

We compare self-play to a count-based exploration baseline similar to the hallway experiment. An extra reward of $\alpha/\sqrt{N(\hat{s}_t)}$ is given at every step, where N is the visit count function and $\hat{s}_t$ is a global observation. Note that this baseline utilizes the same abstraction of global observations as the self-play. We found $\alpha = 0.1$ to be works the

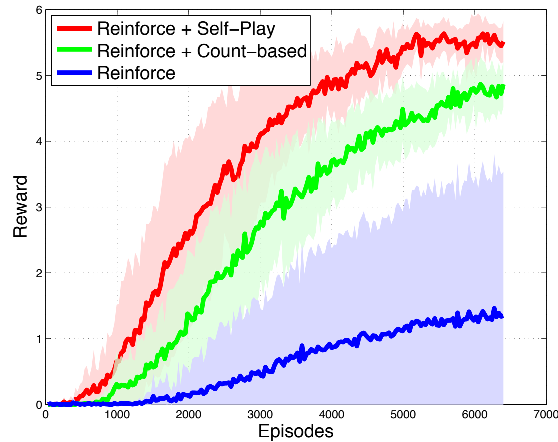

Figure 6.10: Plot of reward on the StarCraft sub-task of training marine units vs #target-task episodes (self-play episodes are not included), with and without self-play. A count-based baseline is also shown. Self-play greatly speeds up learning, and also surpasses the count-based approach at convergence.

best.

Figure 6.10 compares the self-play to REINFORCE baselines, with and without count-based exploration, on the target task. The self-play clearly outperforms the baselines. In Figure 6.11 we show the result of an additional experiment where we extended the length of the episode from 200 to 300, giving more time to the agent for development. The self-play still outperforms baselines methods. Note that to make more than 6 marines, an agent has to build a supply depot as well as a barracks.

6.5 Discussion

6.5.1 Meta-exploration for Alice

We want Alice and Bob to explore the state (or state-action) space, and we would like Bob to be exposed to many different tasks. Because of the form of the standard reinforcement learning objective (expectation over rewards), Alice only wants to find

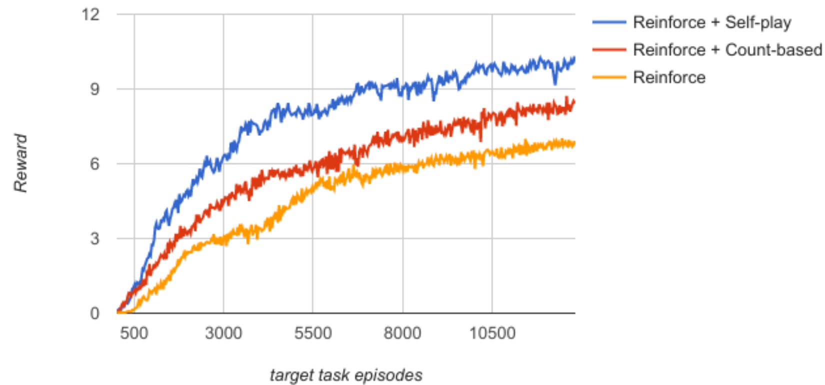

Figure 6.11: Plot of reward on the StarCraft sub-task of training where episode length t_{Max} is increased to 300.

the single hardest thing for Bob, and is not interested in the space of things that are hard for Bob. In the fully tabular setting, with fully reversible dynamics or with resetting, and without the constraints of realistic optimization strategies, we saw in Section 6.2.2 that this ends up forcing Bob and Alice to learn to make any state transition as efficiently as possible. However, with more realistic optimization methods or environments, and with function approximation, Bob and Alice can get stuck in sub-optimal minima.

For example, let us follow the argument in the third paragraph of Section 6.2.2, and assume that Bob and Alice are at an equilibrium (and that we are in the tabular, finite, Markovian setting), but now we can only update Bob's and Alice's policy locally. By this we mean that in our search for a better policy for Bob or Alice, we can only make small perturbations, as in policy gradient algorithms. In this case, we can only guarantee that Bob runs a fast policy on challenges that Alice has non-zero probability of giving; but there is no guarantee that Alice will cover all possible challenges. With function approximation instead of tabular policies, we cannot make any guarantees at all.

Another example with a similar outcome but different mechanism can occur using the reverse game in an environment without fully reversible dynamics. In that case, it could be that the shortest expected number of steps to complete a challenge (s_0, s_T) is longer than the reverse, and indeed, so much longer that Alice should concentrate all her energy on this challenge to maximize her rewards. Thus there could be equilibria with Bob matching the fast policy only for a subset of challenges even if we allow non-local optimization.

The result is that Alice can end up in a policy that is not ideal for our purposes. In Figure 6.9 we show the distributions of where Alice cedes control to Bob in the swimmer task. We can see that Alice has a preferred direction. Ideally, in this environment, Alice would be teaching Bob how to get from any state to any other efficiently; but instead, she is mostly teaching him how to move in one direction.

One possible approach to correcting this is to have multiple Alices, regularized so that they do not implement the same policy. More generally, we can investigate objectives for Alice that encourage her to cover a wider distribution of behaviors.

6.5.2 Communicating via actions

In this work we have limited Alice to propose tasks for Bob by doing them. This limitation is practical and effective in restricted environments that allow resetting or are (nearly) reversible. It allows a solution to three of the key difficulties of implementing the basic idea of "Alice proposes tasks, Bob does them": parameterizing the sampling of tasks, representing and communicating the tasks, and ensuring the appropriate level of difficulty of the tasks. Each of these is interesting in more general contexts. In this work, the tasks have incentivized efficient transitions. One can imagine other reward functions and task representations that incentivize discovering statistics of the states

and state-transitions, for example models of their causality or temporal ordering, cluster structure.

6.6 Conclusion

In this chapter, we described a novel method for intrinsically motivated learning which we call asymmetric self-play. Despite the method's conceptual simplicity, we have seen that it can be effective in both discrete and continuous input settings with function approximation, for encouraging exploration and automatically generating curriculums. On the challenging benchmarks we consider, our approach is at least as good as state-of-the-art RL methods that incorporate an incentive for exploration, despite being based on very different principles. Furthermore, it is possible show theoretically that in simple environments, using asymmetric self-play with reward functions from Equations 6.1 and 6.2, optimal agents can transit between any pair of reachable states as efficiently as possible.

Chapter 7

Conclusion

The objective of this thesis was to explore different ways to advance current deep learning techniques towards achieving an intelligent agent. For this purpose, we have chosen three elements that are essential for intelligence: memory, communication, and intrinsic motivation. In each of them, we made a novel contribution either in model architecture or in learning framework. Then we experimentally verified their capabilities on a multitude of tasks, ranging from language modeling to playing StarCraft.

In case of memory and communication, we proposed new deep learning architectures MemN2N and CommNet. Although their intended applications were considerably different, both the models shared several common properties. First, both of them were a generic model for processing sets in a permutation invariant way, which is not true for existing deep learning models such as ConvNets and RNNs.[1] However, the two models take different approaches when handling a set. While MemN2N processes information in a centralized way through its controller module, CommNet takes a more distributed approach where each element has its own processing power. It is difficult to tell which

[1]Since the publication of our works, several other models have been proposed for sets [Zaheer et al., 2017, Vaswani et al., 2017].

approach is superior because they have different advantages. The centralized processing makes sense in a task where the coordination between elements is more important than processing of individual elements. The controller module can examine each element individually and make a coordinated decision. However, this centralized processing also becomes a computational bottleneck. In contrast, the distributed processing of CommNet has far more computational power since each element has its own controller, but their coordination is weak since all the elements share a single broadcast channel for their communication. This communication bottleneck is particularly harmful if the task requires multiple parallel pairwise communications. The recently proposed models [Hoshen, 2017, Vaswani et al., 2017, Velikovi et al., 2018] alleviate this problem by marrying an attention mechanism like MemN2N to a distributed processing like CommNet. The attention mechanism removes the communication bottleneck by allowing each element to choose an opponent to communicate, making multiple parallel communications possible.

Another common key idea in memory and communication was the relaxation of a discrete operation to a continuous space. In MemN2N, we replaced the previous hard attention with a soft attention mechanism. In CommNet, we allowed agents to communicate via a continuous vector instead of discrete words. Those relaxations made it possible to train the models end-to-end using the back-propagation algorithm. In addition, we also observed that those continuous vectors often behaved like a discrete variable after training, even though there was no explicit mechanism enforcing that. The soft attention of MemN2N usually focuses on one or few memory locations. In CommNet, an analysis of communication vectors showed they often stay near zero or belong to one of the few clusters.

Nevertheless, we are still far from achieving human-level memory and communica-

tion capabilities. The current capacity of MemN2N cannot scale to hold thousands of vectors. Learning where to attend on such a large memory is challenging, although some kind of curriculum training might help. Still, the computational complexity of MemN2N grows linearly with its memory size, which is not desirable for a large-scale memory. Also, there are other memory types that we have not addressed. For example, it is not clear if MemN2N can be used as a semantic memory that can hold a knowledge graph. In case of CommNet, an obvious issue is that its communication medium is a continuous vector. While this might not be a problem if the communication is only between AI agents, it certainly restricts the model from being used as a communication protocol with humans. Another limitation of CommNet is that it assumes full cooperation, so cannot be applied to more general scenarios where agents have different objectives.

The third element we investigated was intrinsic motivation, where our contribution was a novel learning scheme rather than a model architecture. The scheme is based on an agent with two minds that play a game by challenging each other. This interplay between the two minds forces the agent to explore and learn about its environment without any external supervision. Such unsupervised learning in a RL setting is a promising research direction as it offers richer signals compared to more traditional unsupervised learning on static samples. For example, learning to transfer efficiently between two given states is a useful skill that does not require a lot of supervision. In our self-play games, one of the minds generates such two states, eliminating the need for any supervision. However, determining whether an agent has reached a given state is a difficult problem requiring a proper abstraction of a state space, which lies outside the scope of this thesis. Furthermore, an interesting research direction is to let the model imagine an abstract target without actually acting on the environment, which relates to the feature control of [Jaderberg et al., 2018].

While multi-agent RL is a well-established area, controlling a single agent with multiple minds that are optimizing different objectives is an interesting novel direction. Although we only focused on learning to control, the same approach can be employed for learning different aspects of an environment by changing the game rule. One such possibility is to learn causal relations in an environment by letting one mind propose a causal hypothesis while another mind tries to disprove it.

Even though we investigated multiple elements of intelligence in this thesis, there are many other elements that need to be solved before an intelligent agent is conceivable. For example, an intelligent agent needs a life-long learning capability that allows it to learn constantly without forgetting its past knowledge. Even detecting a novel situation is a non-trivial problem that needs a solution. Furthermore, learning a set of abstract actions, and performing exploration and planning via those abstract actions is essential for training an agent on long-term tasks. For tackling those challenges, it is important to advance current deep learning techniques, but also building an environment complex enough to test all those elements of intelligence is crucial.

Bibliography

[Andrychowicz et al., 2017] Andrychowicz, M., Crow, D., Ray, A. K., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems 30*.

[Atkeson and Schaal, 1995] Atkeson, C. G. and Schaal, S. (1995). Memory-based neural networks for robot learning. *Neurocomputing*, 9:243–269.

[Azevedo et al., 2009] Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., Filho, W. J., Lent, R., and Herculano-Houzel, S. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *The Journal of Comparative Neurology*, 513(5):532–541.

[Bahdanau et al., 2015] Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*.

[Baranes and Oudeyer, 2013] Baranes, A. and Oudeyer, P.-Y. (2013). Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73.

[Barto, 2013] Barto, A. G. (2013). Intrinsic motivation and reinforcement learning. In *Intrinsically motivated learning in natural and artificial systems*, pages 17–47. Springer.

[Bellemare et al., 2013] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[Bellemare et al., 2016] Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems 29*, pages 1471–1479.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Bengio et al., 2009a] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009a). Curriculum learning. In *Proceedings of the 26th International Conference on Machine Learning*.

[Bengio et al., 2009b] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009b). Curriculum learning. In *Proceedings of the 26th International Conference on Machine Learning*, pages 41–48.

[Bouzy and Cazenave, 2001] Bouzy, B. and Cazenave, T. (2001). Computer Go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103.

[Busoniu et al., 2008] Busoniu, L., Babuska, R., and De Schutter, B. (2008). A comprehensive survey of multiagent reinforcement learning. *Systems, Man, and Cybernetics, IEEE Transactions on*, 38(2):156–172.

[Cao et al., 2013] Cao, Y., Yu, W., Ren, W., and Chen, G. (2013). An overview of recent progress in the study of distributed multi-agent coordination. *IEEE Transactions on Industrial Informatics*, 1(9):427438.

[Chung et al., 2014] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

[Collobert et al., 2011] Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[Crites and Barto, 1998] Crites, R. H. and Barto, A. G. (1998). Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2):235–262.

[Das et al., 1992] Das, S., Giles, C. L., and Sun, G.-Z. (1992). Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *In Proceedings of The Fourteenth Annual Conference of Cognitive Science Society*.

[Duan et al., 2016] Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33rd International Conference on Machine Learning*.

[Felleman and Essen, 1991] Felleman, D. J. and Essen, D. C. V. (1991). Distributed hierarchical processing in the primate cerebral cortex. *Cerebral cortex*, 1 1:1–47.

[Florensa et al., 2017] Florensa, C., Held, D., Wulfmeier, M., Zhang, M., and Abbeel, P. (2017). Reverse curriculum generation for reinforcement learning. In *1st Conference on Robot Learning (CoRL)*.

[Foerster et al., 2016] Foerster, J. N., Assael, Y. M., de Freitas, N., and Whiteson, S. (2016). Learning to communicate to solve riddles with deep distributed recurrent Q-networks. *arXiv*, abs/1602.02672.

[Fox et al., 2000] Fox, D., Burgard, W., Kruppa, H., and Thrun, S. (2000). Probabilistic approach to collaborative multi-robot localization. *Autonomous Robots*, 8(3):325–344.

[Fukushima, 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202.

[Giles and Jim, 2002] Giles, C. L. and Jim, K. C. (2002). Learning communication for multi-agent systems. In *Innovative Concepts for Agent Based Systems*, pages 377–390. Springer.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, pages 2672–2680.

[Goodman, 2001] Goodman, J. T. (2001). A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434.

[Graves, 2013] Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

[Graves et al., 2014] Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint: 1410.5401*.

[Graves et al., 2016] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwinska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A. P., Hermann, K. M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538 7626:471–476.

[Grefenstette et al., 2015] Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. (2015). Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*.

[Gregor et al., 2015] Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). Draw: A recurrent neural network for image generation. In *Proceedings of the 32nd International Conference on Machine Learning*.

[Guestrin et al., 2001] Guestrin, C., Koller, D., and Parr, R. (2001). Multiagent planning with factored MDPs. In *Advances in Neural Information Processing Systems*.

[Guo et al., 2014a] Guo, X., Singh, S., Lee, H., Lewis, R. L., and Wang, X. (2014a). Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems 27*, pages 3338–3346.

[Guo et al., 2014b] Guo, X., Singh, S., Lee, H., Lewis, R. L., and Wang, X. (2014b). Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems*.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

[Held et al., 2017] Held, D., Geng, X., Florensa, C., and Abbeel, P. (2017). Automatic goal generation for reinforcement learning agents. *CoRR*, abs/1705.06366.

[Hill et al., 2016] Hill, F., Bordes, A., Chopra, S., and Weston, J. (2016). The goldilocks principle: Reading children's books with explicit memory representations. In *International Conference on Learning Representations*.

[Hinton et al., 2012] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hoshen, 2017] Hoshen, Y. (2017). Vain: Attentional multi-agent predictive modeling. In *Advances in Neural Information Processing Systems*, pages 2698–2708.

[Houthooft et al., 2016] Houthooft, R., Chen, X., Duan, Y., Schulman, J., Turck, F. D., and Abbeel, P. (2016). Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks. *arXiv 1605.09674*.

[Hubel and Wiesel, 1968] Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195 1:215–43.

[Jaderberg et al., 2018] Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2018). Reinforcement learning with unsupervised auxiliary tasks. In *International Conference on Learning Representations*.

[Joulin and Mikolov, 2015] Joulin, A. and Mikolov, T. (2015). Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in Neural Information Processing Systems*.

[Kaiser and Sutskever, 2016] Kaiser, L. and Sutskever, I. (2016). Neural gpus learn algorithms. In *International Conference on Learning Representations (ICLR)*.

[Kasai et al., 2008] Kasai, T., Tenmoto, H., and Kamiya, A. (2008). Learning of communication codes in multi-agent reinforcement learning problem. *IEEE Conference on Soft Computing in Industrial Applications*, pages 1–6.

[Kingma and Ba, 2015] Kingma, D. and Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

[Klyubin et al., 2005] Klyubin, A. S., Polani, D., and Nehaniv, C. L. (2005). Empowerment: a universal agent-centric measure of control. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*, pages 128–135.

[Koutník et al., 2014] Koutník, J., Greff, K., Gomez, F. J., and Schmidhuber, J. (2014). A clockwork RNN. In *Proceedings of the 31st International Conference on Machine Learning*.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

[Kumar et al., 2016] Kumar, A., Irsoy, O., Ondruska, P., Iyyer, M., Bradbury, J., Gulrajani, I., Zhong, V., Paulus, R., and Socher, R. (2016). Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of The 33rd International Conference on Machine Learning*.

[Kumar et al., 2010] Kumar, M. P., Packer, B., and Koller, D. (2010). Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems 23*.

[Lauer and Riedmiller, 2000] Lauer, M. and Riedmiller, M. A. (2000). An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proceedings of the 17th International Conference on Machine Learning*.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Levine et al., 2016] Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40.

[Li et al., 2017] Li, J., Monroe, W., Shi, T., Jean, S., Ritter, A., and Jurafsky, D. (2017). Adversarial learning for neural dialogue generation. In *EMNLP*.

[Li et al., 2015] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2015). Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*.

[Littman, 2001] Littman, M. L. (2001). Value-function reinforcement learning in markov games. *Cognitive Systems Research*, 2(1):55–66.

[Lopes et al., 2012] Lopes, M., Lang, T., Toussaint, M., and Oudeyer, P. (2012). Exploration in model-based reinforcement learning by empirically estimating learning progress. In *Advances in Neural Information Processing Systems 25*, pages 206–214.

[Maddison et al., 2015] Maddison, C. J., Huang, A., Sutskever, I., and Silver, D. (2015). Move evaluation in go using deep convolutional neural networks. In *International Conference on Learning Representations (ICLR)*.

[Maravall et al., 2013] Maravall, D., De Lope, J., and Domnguez, R. (2013). Coordination of communication in robot teams by reinforcement learning. *Robotics and Autonomous Systems*, 61(7):661–666.

[Marcus et al., 1993] Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The Penn Treebank. *Comput. Linguist.*, 19(2):313–330.

[Matari, 1997] Matari, M. (1997). Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83.

[Melo et al., 2011] Melo, F. S., Spaan, M., and Witwicki, S. J. (2011). Querypomdp: Pomdp-based communication in multiagent systems. In *Multi-Agent Systems*, pages 189–204.

[Mescheder et al., 2017] Mescheder, L. M., Nowozin, S., and Geiger, A. (2017). Adversarial variational bayes: Unifying variational autoencoders and generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning*.

[Mikolov, 2012] Mikolov, T. (2012). Statistical language models based on neural networks. *Ph. D. thesis, Brno University of Technology*.

[Mikolov et al., 2015] Mikolov, T., Joulin, A., and Baroni, M. (2015). A roadmap towards machine intelligence. *CoRR*, abs/1511.08130.

[Mikolov et al., 2014] Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. (2014). Learning longer memory in recurrent neural networks. *CoRR*, abs/1412.7753.

[Miller et al., 2016] Miller, A. H., Fisch, A., Dodge, J., Karimi, A.-H., Bordes, A., and Weston, J. (2016). Key-value memory networks for directly reading documents. In *EMNLP*.

[Miller, 1956] Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

[Mnih et al., 2015a] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015a). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Mnih et al., 2015b] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Wierstra, D., Legg, S., and Hassabis, D. (2015b).

Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Mozer and Das, 1993] Mozer, M. C. and Das, S. (1993). A connectionist symbol manipulator that discovers the structure of context-free languages. *Advances in Neural Information Processing Systems*, pages 863–863.

[Oh et al., 2016] Oh, J., Chockalingam, V., Satinder, and Lee, H. (2016). Control of memory, active perception, and action in minecraft. In *Proceedings of The 33rd International Conference on Machine Learning*.

[Olfati-Saber et al., 2007] Olfati-Saber, R., Fax, J., and Murray, R. (2007). Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233.

[Ontanón et al., 2013] Ontanón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). A survey of real-time strategy game AI research and competition in starcraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(4):293–311.

[Parisotto and Salakhutdinov, 2018] Parisotto, E. and Salakhutdinov, R. (2018). Neural map: Structured memory for deep reinforcement learning. In *International Conference on Learning Representations*.

[Pathak et al., 2017] Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning*.

[Pearl, 1982] Pearl, J. (1982). Reverend bayes on inference engines: A distributed hierarchical approach. In *AAAI*.

[Peng et al., 2015] Peng, B., Lu, Z., Li, H., and Wong, K. (2015). Towards Neural Network-based Reasoning. *ArXiv preprint: 1508.05508*.

[Perez et al., 2014] Perez, D., Samothrakis, S., Togelius, J., Schaul, T., and Lucas, S. (2014). The GVG-AI competition.

[Pham et al., 2018] Pham, T., Tran, T., and Venkatesh, S. (2018). Graph Memory Networks for Molecular Activity Prediction. *ArXiv e-prints*.

[Pinto et al., 2017] Pinto, L., Davidson, J., Sukthankar, R., and Gupta, A. (2017). Robust adversarial reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*.

[Pollack, 1991] Pollack, J. (1991). The induction of dynamical recognizers. *Machine Learning*, 7(2-3):227–252.

[Radford et al., 2015] Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434.

[Riedmiller et al., 2009] Riedmiller, M., Gabel, T., Hafner, R., and Lange, S. (2009). Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533.

[Samuel, 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

[Scarselli et al., 2009] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80.

[Schaul et al., 2015] Schaul, T., Horgan, D., Gregor, K., and Silver, D. (2015). Universal value function approximators. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1312–1320.

[Schmidhuber, 1991] Schmidhuber, J. (1991). Curious model-building control systems. In *Proc. Int. J. Conf. Neural Networks*, pages 1458–1463. IEEE Press.

[Schulman et al., 2015] Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. (2015). Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning*.

[Sermanet et al., 2014] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. In *International Conference on Learning Representations (ICLR)*.

[Silver et al., 2016a] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016a). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Silver et al., 2016b] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot,

M., et al. (2016b). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Singh et al., 2004] Singh, S. P., Barto, A. G., and Chentanez, N. (2004). Intrinsically motivated reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1281–1288.

[Steinbuch and Piske, 1963] Steinbuch, K. and Piske, U. (1963). Learning matrices and their applications. *IEEE Transactions on Electronic Computers*, 12:846–862.

[Stone and Veloso, 1998] Stone, P. and Veloso, M. (1998). Towards collaborative and adversarial learning: A case study in robotic soccer. *International Journal of Human Computer Studies*, (48).

[Strehl and Littman, 2008] Strehl, A. L. and Littman, M. L. (2008). An analysis of model-based interval estimation for markov decision processes. *J. Comput. Syst. Sci.*, 74(8):1309–1331.

[Sundermeyer et al., 2012] Sundermeyer, M., Schlüter, R., and Ney, H. (2012). LSTM neural networks for language modeling. In *Interspeech*, pages 194–197.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press.

[Sutton et al., 2011] Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., and Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *AAMAS '11*, pages 761–768.

[Synnaeve et al., 2016] Synnaeve, G., Nardelli, N., Auvolat, A., Chintala, S., Lacroix, T., Lin, Z., Richoux, F., and Usunier, N. (2016). Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

[Tampuu et al., 2015] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., and Vicente, R. (2015). Multiagent cooperation and competition with deep reinforcement learning. *arXiv:1511.08779*.

[Tan, 1993] Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the 10th International Conference on Machine Learning*.

[Tang et al., 2017] Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., Schulman, J., Turck, F. D., and Abbeel, P. (2017). #exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems*.

[Taylor, 1959] Taylor, W. K. (1959). Pattern recognition by means of automatic analogue apparatus. *Proceedings of The Institution of Electrical Engineers*, 106:198–209.

[Tesauro, 1995] Tesauro, G. (1995). Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

[Todorov et al., 2012] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *IROS*, pages 5026–5033. IEEE.

[Varshavskaya et al., 2009] Varshavskaya, P., Kaelbling, L. P., and Rus, D. (2009). *Distributed Autonomous Robotic Systems 8*, chapter Efficient Distributed Reinforcement Learning through Agreement, pages 367–378.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*.

[Velikovi et al., 2018] Velikovi, P., Cucurull, G., Casanova, A., Romero, A., Li, P., and Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.

[Wang and Sandholm, 2002] Wang, X. and Sandholm, T. (2002). Reinforcement learning to play an optimal nash equilibrium in team markov games. In *Advances in Neural Information Processing Systems*, pages 1571–1578.

[Weston et al., 2016] Weston, J., Bordes, A., Chopra, S., and Mikolov, T. (2016). Towards ai-complete question answering: A set of prerequisite toy tasks. In *International Conference on Learning Representations (ICLR)*.

[Weston et al., 2015] Weston, J., Chopra, S., and Bordes, A. (2015). Memory networks. In *International Conference on Learning Representations (ICLR)*.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256.

[Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson,

M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G. S., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

[Xiong et al., 2016a] Xiong, C., Merity, S., and Socher, R. (2016a). Dynamic memory networks for visual and textual question answering. In *Proceedings of The 33rd International Conference on Machine Learning*.

[Xiong et al., 2016b] Xiong, C., Merity, S., and Socher, R. (2016b). Dynamic memory networks for visual and textual question answering. *Proceedings of the 33rd International Conference on Machine Learning*.

[Xu et al., 2015] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. (2015). Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *ArXiv preprint: 1502.03044*.

[Zaheer et al., 2017] Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. (2017). Deep sets. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 3391–3401. Curran Associates, Inc.

[Zaremba et al., 2015] Zaremba, W., Mikolov, T., Joulin, A., and Fergus, R. (2015). Learning simple algorithms from examples. *CoRR*, abs/1511.07275.

[Zaremba and Sutskever, 2015] Zaremba, W. and Sutskever, I. (2015). Reinforcement learning neural turing machines. In *arXiv preprint: 1505.00521*.

[Zaremba et al., 2014] Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

[Zhang and Lesser, 2013] Zhang, C. and Lesser, V. (2013). Coordinating multi-agent reinforcement learning with limited communication. In *Proc. AAMAS*, pages 1101–1108.