MAZEBASE: A SANDBOX FOR LEARNING FROM GAMES

Sainbayar Sukhbaatar
Department of Computer Science
Courant Institute
New York University
sainbar@cs.nyu.edu

Arthur Szlam, Gabriel Synnaeve, Soumith Chintala & Rob Fergus
Facebook AI Research
New York
{aszlam,gab,soumith,robfergus}@fb.com

ABSTRACT

This paper introduces an environment for simple 2D maze games, designed as a sandbox for machine learning approaches to reasoning and planning. Within it, we create 10 simple games based on algorithmic tasks (e.g. embodying simple if-then statements). We deploy a range of neural models (fully connected, convolutional network, memory network) on these games, with and without a procedurally generated curriculum. We show that these architectures can be trained with reinforcement to respectable performance on these tasks, but are still far from optimal, despite their simplicity. We also apply these models to games involving combat, including StarCraft, demonstrating their ability to learn non-trivial tactics which enable them to consistently beat the in-game AI.

1 INTRODUCTION

The past few years have seen a resurgence of interest in neural models for game playing (Mnih et al., 2013; Guo et al., 2014; Mnih et al., 2015). These works apply deep reinforcement learning techniques to a suite of Atari arcade games with impressive results. For some of the simpler games which are Markovian in nature, the models achieve super-human performance. But on games that require planning or reasoning they score worse than humans. However, for the purposes of advancing AI, the Atari game collection is limited by the fixed nature of the games: they cannot be altered and new ones cannot be added easily. It thus serves as a benchmark for models, rather than a sandbox which would facilitate learning as well as aiding the development of new models.

To address this, our paper introduces a simple 2D game environment over which we have total control. To demonstrate the use of the environment, we devise a series of tasks that carefully explore a range of elementary algorithmic elements. The software framework, which we consider to be the main contribution of this work, is designed to be compact but flexible, enabling us to implement games as diverse as simple role-playing game (RPG) puzzles and StarCraft-like combat, as well as facilitating the creation of new ones. Furthermore, it offers precise tuning of the task difficulty, facilitating the construction of curricula to aid training.

Using these games, we explore various architectures, ranging from linear models to memory networks (Sukhbaatar et al., 2015). These models can be trained either using supervised imitation learning or reinforcement learning, although here we will focus on reinforcement. We show that although the models can learn to complete non-trivial tasks, they also miss many interesting aspects of the games. The fine control offered by our game environment allows us to tease apart individual contributions of the model, learning algorithm and data, enabling useful conclusions to be drawn.

One game designed in our environment involves StarCraft™*-like combat between agents and enemies, complete with health and cooldown periods (when shooting is prohibited). Insights gained from this game enable us to successfully apply our models to real StarCraft combat, where they consistently outperform several baseline bots.

*StarCraft and Brood War are registered trademarks of Blizzard Entertainment, Inc.

1.1 CONTRIBUTIONS

Our main contribution is the game environment and accompanying set of basic games. Written in Torch[†], it offers rapid prototyping of games and is easy to connect to models that control the agent's behavior. Moreover, the data format is interoperable with the tasks in Weston et al. (2015a). In this paper we will give some baseline results on a specific set of games, but we hope the environment will be used for more than just these.

A secondary contribution of this work is to show that memory networks (Sukhbaatar et al., 2015) trained with reinforcement learning are able to perform nontrivial algorithmic tasks, including simple combat within StarCraft.

1.2 RELATED WORK

The environment we document in this work can be thought of as a small practical step towards an implementation of some of the ideas discussed at length in Mikolov et al. (2015). In particular, interfacing the agent and the environment in (quasi-)natural language was inspired by discussions with the authors of that work. However, here, our ambitions are more local, and we try to focus more finely on the border where current models fail (but nearly succeed), rather than aim for a global view of a path towards AI. For example, we specifically avoid algorithmic tasks that require unbounded recursions or loops, as we find that there is plenty of difficulty in learning simple if-then statements. Furthermore, for the example games we describe below, we allow large numbers of training runs, as the noise from reinforcement with discrete actions is still challenging even with many samples.

In non-game environments, there has been recent work on learning simple algorithms. (Graves et al., 2014; Vinyals et al., 2015; Joulin & Mikolov, 2015; Zaremba & Sutskever, 2015) demonstrate tasks such as sorting and reversal of inputs. The algorithms instantiated in our games are even simpler, e.g. conditional statements or navigation to a location, but involve interaction with an environment. In some of these approaches (Mnih et al., 2013; Guo et al., 2014; Mnih et al., 2015; Joulin & Mikolov, 2015; Zaremba & Sutskever, 2015) the models were trained with reinforcement learning or using discrete search, allowing possibly delayed rewards with discrete action spaces. Our games also involve discrete actions, and these works inform our choice of the reinforcement learning techniques. Several works have also demonstrated the ability of neural models to learn to answer questions in simple natural language within a restricted environment (Weston et al., 2015b; Sukhbaatar et al., 2015). The tasks we present here share many features with those in Weston et al. (2015a), and indeed, the input-output format our games use is inter-operable with their stories. However, during training and testing, the environment in Weston et al. (2015a) is static, unlike the game worlds we consider.

Developing AI for game agents has an extensive literature. Our work is similar to Mnih et al. (2013); Guo et al. (2014); Mnih et al. (2015) in that we use reinforcement and neural models when training on games. However, our goal is carefully understand the limits of these models and how to improve them. Our sandbox is complementary to the Atari games benchmark- it is meant to aid in design and training of models as well as evaluating them. In this work, we also discuss AI for micro-combat in StarCraft. For a survey on AI for Real Time Strategy (RTS) games, and especially for StarCraft, see Ontanón et al. (2013). Real-time strategy games research has been conducted mainly on open-source clones of existing games, like Wargus (2002-) and Spring (2004-), or on the more research oriented Open RTS (2003). More recently, research has been conducted on micro-RTS (Ontanón, 2013), and mainly on StarCraft (through BWAPI (2008-)), with a focus on the annual StarCraft AI competition (2010-).

In RTS games players control units to grow an economy, build factories, and construct more units to combat the enemy. They often have multiple units, simultaneous moves, durative actions, and can have an extremely large combinatorial complexity. Here, in this work, the focus on an environment that allows building and training on simple scenarios. While we can build small parts of RTS games in our framework, it is not intended to be a replacement for existing frameworks, but rather is meant to easily allows the construction of scenarios to understand ML models. In this work we will demonstrate such simple scenarios and test models on them; like Synnaeve & Bessiere (2011); Wender & Watson (2012); Churchill et al. (2012)(for StarCraft), we will focus on micromanagement, but with only a few active objects, and eschew all search methods. On the other hand, we will work

[†]http://torch.ch

without feature engineering nor heuristics, and the models we apply are the same for puzzle games as they are for the combat scenarios.

2 ENVIRONMENT AND TASKS

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. The 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. (2015a). However, note that we use egocentric spatial coordinates (e.g. the goal G1 in Fig. 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, we have two versions of the game. In one, the environment automatically sets a flag on visited goals. In the harder versions, this mechanism is absent but the agent has a special action that allows it to release a "breadcrumb" into the environment, enabling it to record locations it has visited. In the experiments below, unless otherwise specified, we report results on games with the explicit flag.

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%).

2.1 TASKS

Although our game world is simple, it allows for a rich variety of tasks. In this work, we explore tasks that require different algorithmic components (such as conditional reasoning or planning short routes) first in isolation and then in combination. Our goal is to build tasks that may require a few stages of action, and where each stage is relatively basic. We avoid tasks that require unbounded loops or recursion, as in Joulin & 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. We note that this can already be challenging for standard models.

In all of the tasks, the agent incurs a fixed penalty for each action it makes. In the experiments below, this is set to 0.1. In addition, stepping on a Water block incurs an additional penalty of 0.2. For most games, a maximum of 50 actions are allowed. The tasks define extra penalties and conditions for the game to end.

[‡]This is consistent with Sukhbaatar et al. (2015), where the "agent" answering the questions was also given them in egocentric temporal coordinates.

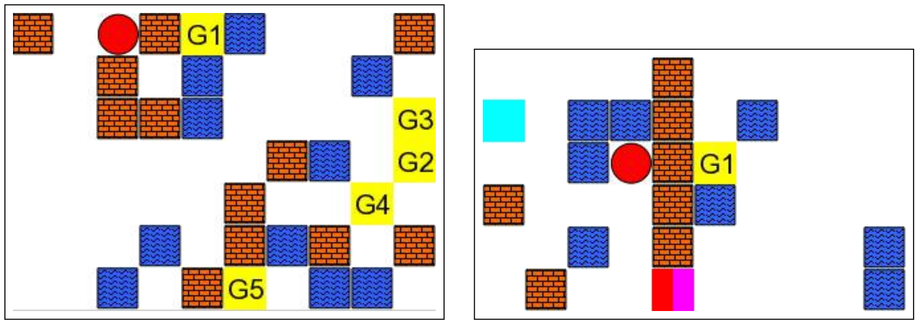

Figure 1: Examples of Multigoal (left) and Light Key (right) tasks. Note that the layout and dimensions of the environment varies between different instances of each task (i.e. the location and quantity of walls, water and goals all change). The agent is shown as a red blob and the goals are shown in 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).

- **Multigoals:** In this task, the agent is given an ordered list of goals as "Info", and needs to visit the goals in that order. In the experiments below, 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. 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.
- **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." In the experiments below, the number of the number of colors range 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).
- **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 form 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.
- **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 switches to the same color, and the agent has the choice of color; to get the best reward, the agent needs to solve a (very small) traveling salesman problem. In the experiments below, 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.
- **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 wrong 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 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.
- **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.
- **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 is not directly given the goal's location, it must read this from the list of goal locations. The number of goals ranges from 1 to 6.
- **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%.

- **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 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 PushableBlock blocks the gap in the wall instead of a door.

Finally, note that with the exception of the **Multigoals** task, all these are Markovian; and **Multigoals** is Markovian with the explicit "visited" flag, which we use in the experiments below. 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. Examples of each game are shown at `https://youtu.be/kwnp8jFRi5E`

3 MODELS

We investigate several different types of model: (i) simple linear, (ii) multi-layer neural nets, (iii) convolutional nets and (iv) end-to-end memory networks (Weston et al., 2015b; Sukhbaatar et al., 2015). While the input format is quite different for each approach (detailed below), the outputs are the same: a probability distribution over set of discrete actions {N,S,E,W,toggle switch,push-N,push-S,push-E,push-W}; and a continuous baseline value predicting the expected reward. 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 bag-of-words (excluding location words) representation of all items at the same location. Then, we concatanate 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 bag-of-words in the same way as linear model. Hence the environment is presented as a 3D cube of size $w \times h \times n$, which is then feed to four layers of convolution (the first layer has 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).

Memory Network: 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. As a consequence, the memory network has to learn the spatial arrangement of the grid, unlike the convolutional network. Otherwise, we use the architecture from (Sukhbaatar et al., 2015) with 3 hops and tanh nonlinearities.

4 TRAINING PROCEDURES

We use policy gradient (Williams, 1992) for training, which maximizes the expected reward using its unbiased gradient estimates. First, we play the game by feeding the current state x_t to the model, and sampling next action a_t from its output. After finishing the game, we update the model parameters θ by

$$\Delta\theta = \sum_{t=1}^{T} \left[\frac{\partial \log p(a_t|x_t, \theta)}{\partial \theta} \left(\sum_{i=t}^{T} r_i - b \right) \right],$$

where r_t is reward given at time t, and T is the length of the game.

Instead of using a single baseline b value for every state, we let the model output a baseline value specific to the current state. This is accomplished by adding an extra head to models for outputting the baseline value. Beside maximizing the expected reward with policy gradient, the models are also trained to minimize the distance between the baseline value and actual reward. The final update rule is

$$\Delta\theta = \sum_{t=1}^{T} \left[\frac{\partial \log p(a_t|x_t,\theta)}{\partial\theta} \left(\sum_{i=t}^{T} r_i - b(x_t,\theta) \right) - \alpha \frac{\partial}{\partial\theta} \left(\sum_{i=t}^{T} r_i - b(x_t,\theta) \right)^2 \right].$$

Here hyperparameter α is for balancing the two objectives, which is set to 0.03 in all experiments. The actual parameter update is done by RMSProp (Tieleman & Hinton, 2012) with learning rates optimized for each model type.

For better parallelism, the model plays and learns from 512 games simultaneously, which spread on multiple CPU threads. Training is continued for 20 thousand such parallel episodes, which amounts to 10M game plays. Depending on the model type, the whole training process took from few hours to few day on 18 CPUs.

4.1 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. As we demonstrate, 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** (see Section 6), we vary the number of agents & enemies, as well as their speed and their initial health.

The curriculum is specified by an upper and lower success thresholds T_u and T_l respectively. If the success rate of the model 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 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 model to the full range of difficulties by setting the upper limit to its maximum preset value.

5 Experimental Results

Table 1 and Fig. 2 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, we trained each model 10 times and picked the single instance that had the highest mean reward. A video showing a trained MemNN model playing each of the games can be found at `https://youtu.be/kwnp8jFRi5E`. 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) most of the games:

- On **Conditional Goals** and **Exclusion**, all models did poorly. On inspection, it appears 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 make 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

		Multigoals	Cond. Goals	Exclusion	Switches	Light Key	Goto	Goto Hid.	Push Block	Push Block Card.	Blocked Door	Mean
No Curric.	Linear	-3.59	-3.54	-2.76	-1.66	-1.94	-1.82	-2.39	-2.50	-1.64	-1.66	-2.35
	2 layer NN	-3.14	-2.61	-2.42	-1.32	-1.54	-1.07	-1.83	-1.86	-1.21	-1.25	-1.82
	ConvNet	-3.36	-2.90	-2.38	-2.96	-1.90	-2.70	-2.06	-4.80	-4.50	-2.70	-3.03
	MemNN	-2.02	-2.70	-2.22	-2.97	-1.78	-1.14	-1.44	-4.06	-1.68	-1.34	-2.19
Curriculum	Linear	-3.42	-3.21	-2.85	-1.58	-2.07	-1.74	-2.31	-2.47	-1.52	-1.68	-2.29
	2 layer NN	-2.82	-2.49	-2.25	-1.27	-1.27	-1.29	-1.59	-1.81	-1.13	-1.25	-1.72
	ConvNet	-3.17	-2.52	-2.20	-1.34	-1.72	-1.70	-1.85	-2.45	-1.33	-1.56	-1.99
	MemNN	-1.46	-2.30	-2.03	-1.10	-1.14	-.98	-1.52	-2.33	-1.41	-1.21	-1.55
	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 1: Reward of the best performing model (from 10 runs) on the 9 games, with and without curriculum. 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.

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.

With respect to comparisons between the models:

- On average, the memory network did best out of the methods. However, on the games with pushable blocks, the 2 layer neural nets were superior e.g. **Exclusion** and **Push Block** or the the same **Blocked Door**. Although we also trained 3 layer neural net, the result 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 memory network. 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 memory network. Nevertheless, even with very large number of trials, this architecture could not learn many of the tasks.
- The memory network 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.
- The memory network had a significant variance in performance over its 10 instances, whereas the variance for the 2-layer neural net was much smaller (and the variance was negligible for the linear model). However, the curriculum significantly decreased the variance for all methods.

With respect to different training modalities:

- The curriculum generally helped all approaches, but gave the biggest assistance to the memory networks, 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 for reinforcement. The one exception was learning to use the Breadcrumb action for the **Multigoals** game. None of our models were able to learn to use the breadcrumb to mark visited locations without supervision. Note that in the tables we show results with the explicit “visited” flag given by the environment.

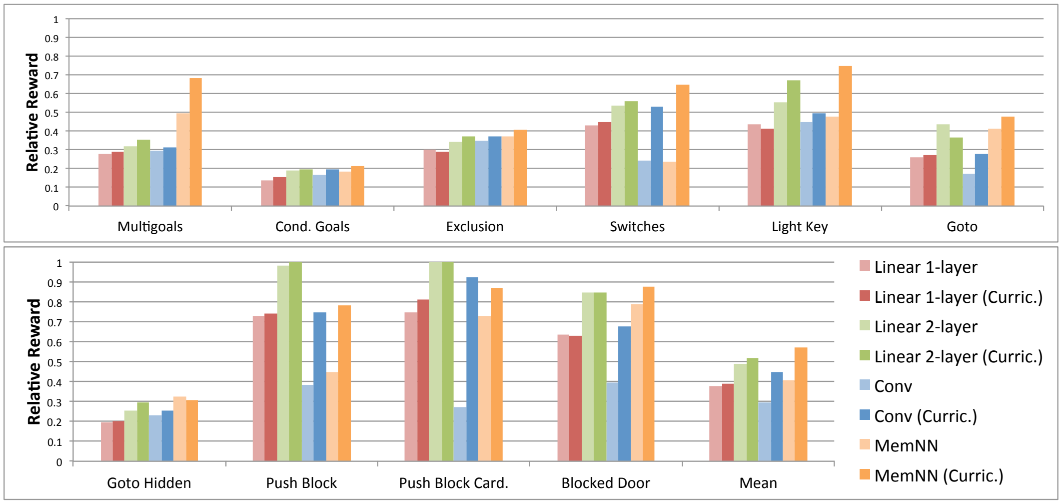

Figure 2: Reward for each model jointly trained on the 10 games, with and without the use of a curriculum during training. The y-axis shows relative reward (estimated optimal reward / absolute reward), thus higher is better. The estimated optimal policy corresponds to a value of 1 and a value of 0.5 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).

6 Combat Games

We also use our framework to design a very simple combat game, **Kiting**[§]. This consists of a standard maze with blocks (but no water) where an agent aims to kill up to two enemy bots. The shooter is prevented from firing again for a small time interval (cooldown). A large negative reward is received if this agent is killed. Otherwise, a small negative reward is imposed each time step, to encourage the agent to fight the enemy. We deliberately introduce an imbalance by (i) allowing the agent to move faster and shoot farther than the bot and (ii) giving the agent significantly less health than the bot(s). The agent has a shot range of 4 squares, and the bots have a shot range of 2 squares; the enemy moves half as often as the agent. The agent has 3 health points, and the enemy(s) have health uniformly distributed between 1 and 10. Training uses 2 layer NN and MemNN with a curriculum where we increase the health of the enemy(s), their move rate, and the possibility of fighting two at once. After training, they are able to win 83% and 85% of the time respectively. They successfully learn to shoot and then run away, slowly wearing down the bots as they chase after the agent. See video `https://youtu.be/Xy3wDXL4mBs` for a demonstration of this behavior.

6.1 StarCraft

To demonstrate the relevance of our environment to real video games, we investigate the performance of our models on StarCraft: Brood War. We used BWAPI (2008-) to connect the game to our Torch framework. We can receive the game state and send orders, enabling us to do a reinforcement learning loop.

We train 2 layer neural network and MemNN models using the difference between the armies hit points, and win or loss of the overall battle, as the only reward signals. The features used are all categorical (as for MazeBase) and represent the hit points (health), the weapon cooldown, and the x and y positions of the unit. We used a multi-resolution encoding (coarser going further from the unit we control) of the position on 256×256 map to reduce the number of parameters. In the multiple units case, each unit is controlled independently. We take an action every 8 frames (the atomic time unit @ 24 frames/sec). The architectures and hyper-parameter settings are the same as used in the **Kiting** game. The results can be found in Table 2, where we compare to an "attack

[§]For explanation of the name, see `http://www.urbandictionary.com/define.php?term=Kiting`

Figure 3: Left: The **Kiting** game where the agent (red circle) is shooting at an enemy bot from a distance. It learns to shoot and then flee from the less mobile bots. Right: StarCraft 2-vs-2 combat. Our models learn to concentrate fire on the weaker of the two enemy bots.

	2 vs 2	Kiting	Kiting hard
Attack weakest	85%	0%	0%
2 layer NN	80% (38k)	89% (190k)	30% (275k)
MemNN	80% (83k)	92% (120k)	41% (360k)

Table 2: Win rates against StarCraft built-in AI. The numbers in parenthesis shows how many games were necessary for training.

weakest" heuristic. We considered simplified scenarios of the game ("micro-management"), that only involves combat with a limited number of troops:

- **2 vs 2** (Terran Marines): a symmetric match-up where both teams have 2 ranged units.
- **StarCraft Kiting** (Terran Vulture vs Protoss Zealot): a match-up where we control 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.
- **StarCraft Kiting hard** (Terran Vulture vs 2 Protoss Zealots): same as above but with 2 enemies.

We find that the models are 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**. The video `https://youtu.be/Hn0SRa_Uark` shows example gameplay of our MemNN model for the **StarCraft Kiting hard** scenario.

Moreover, the success rate in **StarCraft Kiting** is comparable to that achieved in MazeBase **Kiting**, showing that tasks in our game environment are comparably challenging to real world gameplay.

7 CONCLUSIONS

We have introduced a new environment for games designed to test AI models. We evaluated several standard models on them and shown that they solve most tasks, but it a way that is still far from optimal. The memory networks were able to solve some tasks that the fully-connected models and convnets could not, although overall the performance was similar.

We were able to take the same models are directly apply them to games involving combat. When applied to StarCraft micro-games, the models were better than baseline bots. Their effective performance on this new domain validates our 2D maze games as being non-trivial, despite their simplicity. Source code for MazeBase and the games can be found at `http:\comingsoon`.

REFERENCES

BWAPI. Brood War API, 2008-. URL https://github.com/bwapi/bwapi.

Churchill, David, Saffidine, Abdallah, and Buro, Michael. Fast heuristic search for rts game combat scenarios. In *AIIDE*, 2012.

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

Guo, Xiaoxiao, Singh, Satinder, Lee, Honglak, Lewis, Richard L, and Wang, Xiaoshi. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems 27*, pp. 3338–3346. 2014.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. *NIPS*, 2015.

Mikolov, Tomas, Joulin, Armand, and Baroni, Marco. A roadmap towards machine intelligence. *CoRR*, abs/1511.xxxx, 2015. URL http://arxiv.org/abs/1511.xxxx.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

Ontanón, Santiago. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013. URL "https://github.com/santiontanon/microrts".

Ontanón, Santiago, Synnaeve, Gabriel, Uriarte, Alberto, Richoux, Florian, Churchill, David, and Preuss, Mike. 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, 2013.

Open RTS. ORTS - A Free Software RTS Game Engine, 2003. URL https://skatgame.net/mburo/orts/index.html.

Spring. Spring Engine, a 3D RTS game engine, 2004-. URL https://springrts.com/.

StarCraft AI competition, 2010-. URL http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/.

Sukhbaatar, Sainbayar, Szlam, Arthur, Weston, Jason, and Fergus, Rob. End-to-end memory networks. *NIPS*, 2015.

Synnaeve, Gabriel and Bessiere, Pierre. A bayesian model for rts units control applied to starcraft. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pp. 190–196. IEEE, 2011.

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

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. *NIPS*, 2015.

Wargus. Wargus, a WarCraft 2 mod in Stratagus, 2002-. URL http://wargus.sourceforge.net.

Wender, Stefan and Watson, Ian. Applying reinforcement learning to small scale combat in the real-time strategy game starcraft: broodwar. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pp. 402–408. IEEE, 2012.

Weston, J., Bordes, A., Chopra, S., and Mikolov, T. Towards ai-complete question answering: A set of prerequisite toy tasks. In *arXiv preprint: 1502.05698*, 2015a.

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

Williams, Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pp. 229–256, 1992.

Zaremba, Wojciech and Sutskever, Ilya. Reinforcement learning neural turing machines. In *arXiv preprint: 1505.00521*, 2015.