AI XPrize Machine Learning Plan

Remy Bubulka, Coleman Gibson, Kieran Groble, Lewis Kelley

December 11, 2017

Introduction

In general, there are two proposed methods for training the robot AI, one with a human in the loop and one with only AI's bouncing data off of each other. The game has two roles, the "human" and the "robot."

Due to the potential confusion of having the "human" be controlled by either an actual person or an AI, for this document we shall refer to the role giving instructions as the Commander and the role receiving commands as the Worker.

From a high level, whatever system is put in for the Worker, it should operate as a function that inputs:

- a list of blocks with their locations and visual information
- a command string from the Commander
- a 2D point the Commander pointed to

and outputs:

- a 2D point representing the block to pick up
- a 2D point representing where to drop the block

Notable Assumptions

Essentially, this AI that is being produced should act as a function that looks like this:

```
make_move(List<Blocks> world,
          String command,
          Point pointed)
    -> (Point start, Point end, bool should_flip)
```

Note that there is precisely *one* point action. We assume that the user will only point to *at most one* location for each command. This is due to the added complexity of handling an arbitrary number of pointed-to locations and correlating the times of their occurrence with the times of each individual word in the spoken command ("Move this <point> block there <point>").

Additionally, this assumes that each command will ask for only one block to be moved. Phrases such as, "Move all the blocks over here," will not be possible.

Full Q-Learning

System Design

The map will be broken down into a discrete number of cells. Each square block will take up an odd number of cells, as will the representation of the arm. The pointer will point to exactly one cell. To pick up a block, the arm must contain the centermost cell of a block. If there are multiple blocks underneath the arm, select the one closest to the center.

Worker Design

The Worker will be implemented using a Q-learning algorithm.

State

The state of the system will be made of:

- the current locations and visual details of all the blocks
- the current location of the Worker's arm
- the syntax tree of the last command from the Commander
- the location pointed to by the Commander
- whether a block is being currently held by the arm and the visual details of that block

Actions

The possible actions at any state (assuming that the action will not take the arm out of bounds) will be:

- move north by one cell
- move east by one cell
- move south by one cell
- move west by one cell
- move north by ten cells
- move east by ten cells
- move south by ten cells
- move west by ten cells
- toggle arm (to grab / drop the block)
- flip arm (to flip blocks)
- stop

There are 10 possible actions from each state, leading to a rather large explosion of possible states. The 10 cell movements were chosen to give the algorithm a better chance of finding the exit each

Training Phases

In order to get a basically trained AI to perform the job of the Worker, an automated Commander will be created using data gathered from the previous "Human as Worker" stage.

This has the great advantage of being extremely fast; without a human in the loop, thousands of trials may be executed in the same time that a human could issue a single instruction. Unfortunately, the only system that can effectively mimic instructions from any potential user is humans. If we only train with an AI Commander, the Worker may learn only the subset of interactions that the AI Commander knows about, and will be unable to handle more diverse and unexpected input.

To counter this, once the Worker is trained to a "passable" point, fully automated learning will be switched off and the game will be made available for humans to play as the Commander. Since the Worker has been previously trained, the humans will produce much more relevant data from their interactions with the Worker, as opposed to if they were trying to talk to a non-responsive, untrained Worker.

Completely Automated

- Outline

Stage	Number of Instructions	Number of Blocks	Sequence of Instructions
(1) Proof of Concept	1	1	Fixed
(2) World Size	1	5	Fixed
(3) Instruction Count	5	5	Fixed
(4) Randomize Order	5	5	Randomized
(5) Randomize Phrases	5	5	Randomized
(6) Randomize Accuracy	5	5	Randomized
(7) Randomize Configuration	5	5	Randomized

Stage	Phrases Used	"Human" Accuracy	World Configuration
(1)	Fixed	Perfect	Fixed
(2)	Fixed	Perfect	Fixed
(3)	Fixed	Perfect	Fixed
(4)	Fixed	Perfect	Fixed
(5)	Semi-Randomized	Perfect	Fixed
(6)	Semi-Randomized	Randomized	Fixed
(7)	Semi-Randomized	Randomized	Randomized

- Implementation of the AI Commander During the previous "Human as Worker" phase, the system gathered lots of data about how humans act as the Commander. Using these interactions, we will assemble an AI Commander by patching together different phrases (at random, when necessary).

Human in the Loop

Once the Worker AI reaches a certain acceptable baseline, the AI Commander will be swapped out with a wide array of human Commanders to provide a much more diverse set of interactions than with a single partner.

This then essentially gets turned into single-player Blocksworld, allowing for a much more playable and accessible game.

Reduced Q-Learning

System Design

This proposal follows the same layout as Full Q-Learning, but it attempts to handle the most significant hurdle to that method: scale. With all of the possible statements that could be given by the commander and how many different arrangements the board can have, Q-Learning would have too many states to remember and would rarely ever find itself in the same state.

Our improvements to this is twofold: reduce the language into structure and reduce the precision of location of blocks.

Structural Language

In order to reduce the number of possible permuations that arise as the result of natural language, and given the fairly restrictive nature of the domain of many commands, the structure of the command and its direct object are likely the most relevant pieces of information about that command. Take these sentences with their direct objects underlined, for example.

Move that *red block* over here. Place that *red block* around there. Set the *red block* near there.

To us, all of these sentences are saying the same thing (assuming the Commander is pointing to the same spot). If we only think about the structure and the direct object, we're able to group these sentences together into one shared state as well as distinguish between sentences like the following.

Don't move the *red block* over here. Move the *blue block* over here. Move the *red block* into that corner.

There are some obvious drawbacks to such a simplistic solution. For starters, this won't be able to handle more complex sentences such as:

Move the *red block* next to that yellow block. Move *it* a bit to the right.

These are significant, and a more useful way of reducing a sentence may be useful.

Location of Blocks

In order to collapse the different possibilities of the locations of the blocks, we reduce the state of the world into a k-d tree. The state will only have a precise coordinate for the block it is closest to, and after that it will have less and less detailed knowledge of the precise locations of more distant blocks.

The reasoning behind this is that moving a single block a tiny bit, especially when that block is irrelevant to the given command, would completely

ignore any gained knowledge of an otherwise identical problem. If the locations of distant blocks are fuzzed, then small changes in their location will not affect the remembered state nearly as much.

Neural Network

System Design

Another option for mapping inputs to instructions is to use a neural network. This method would allow for increased flexibility in the text input, as we would not need to constrain the format in any way. It would, however, require us to have a sizable data set before we are able to get meaningful results. This will be difficult to do without either a significant time commitment or a large number of people to help.

Worker Design

Inputs

There are several possible ways we could provide inputs to our neural network, with our most significant choices being in how we provide the blocks and point locations from the Commander. One option is to fix the number of possible blocks or to give a maximum possible number. This would be most likely to give good results, but would provide constraints on the way the Worker views the outside world. The other option is to use an LSTM or similar recurrent layer to allow for variable length inputs. This would allow for an increase in flexibility, but may decrease the performance of our network. Neither the set of input blocks nor the points has any temporal locality, which is the usual use case of recurrent networks. A "window" of blocks is not really something that makes sense.

Outputs

We also have a number of options for our neural network output. One option is to set a discrete set of possible locations. The output to the neural network would then be the index of the block to move, followed by the location to place the block. Another option is to output two points. The first point would again refer to a block, but instead of being an index we would consider the block closest to the given point. The second point would then be the exact location to place it. We could also consider the output to be a set of instructions for the final robot. As an example, it could be the starting and

final configurations for the arm. However, this method would be much more difficult to gather data for, and to test outside of the lab.

Training Phases

Using neural networks would provide a similar set of trade offs as the other methods. We will either need to spend a significant amount of time gathering data or use generated data, which may not be varied enough to allow us to eventually consider the entire English language. Similar to above, we would likely proceed by training on generated data until our Worker is passable, then allow access to human commanders who would be able to provide higher quality data.

Conclusion

In brief, this is a very hard problem to solve, mostly due to the continuous and consequentially enormous input space. Natural language is in itself a huge task, but when combined with an additionally enormous space of the placement of a various number of blocks placed at non-discrete positions on a board, the problem gets substantially more difficult.

Reinforcement learning works best when there is a manageable number of states to track and to score, but this is entirely infeasible without significantly reducing the search space, and even then those reductions may be corruptive or insufficient to handle to problem.

We believe our best option is to utilize a neural network, train it to a "passable" level using the automated Commander, and finally turn it loose onto human players to gather a more diverse set of interactions.