

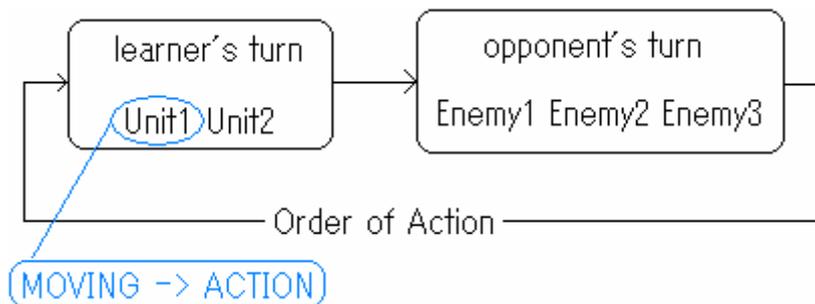
# Trying out machine learning tricks in a tactical game

Hanhua Yin

## Summary

This is my attempt of applying machine learning to replace pre-defined AI of a tactical game I created. Instead of using reinforcement learning, I used simple linear regression (with some extra tweaks), which works reasonably well.

## Game



It is a turn-by-turn tactical game (SRPG) on a 2D grid. 2 sides take turns to try to neutralize units of the other side. A unit undergoes 2 phases during its turn: In MOVING phase, the unit can move to a location within range. If a unit moves to a position occupied by a friendly unit, it will mount to the top of that unit. A mounted unit cannot be attacked by others. In ACTION phase, the unit can either choose to ATTACK, use ITEM, or REST. See game manual for detailed explanation.

## Basic Idea

Let the learner (the entity controlled by machine learning algorithm; “player” and “learner” are used interchangeably) try to learn to play better against some opponent.<sup>1</sup>

## Control Using Machine Learning

To play better against the opponent, the learner can make predictions about the opponent’s response to learner’s action. Instead of directly predicting how the opponent will react, the learner predicts the value of a reward function *if* it is evaluated at the beginning of learner’s next turn (that is, after the opponent’s response). The reward function measures how well the learner is doing under a game state (big reward when the learner is winning etc). By predicting the future reward, the learner indirectly predicts the result of the opponent’s reaction. By seeking maximum predicted reward, the learner seeks an optimal action.

---

<sup>1</sup> Although it is trained against a deterministic AI, the learner does not know how the AI would respond, since the point is to learn from some unknown opponent.

## Example

To be more concrete, here is the example problem I use throughout the discussion and the one I actually implemented and tested. “Player” units are controlled by machine learning algorithm. “Enemy” units are controlled by a scripted AI.

Example units used to train the machine learning model (upper limit used)

Side	Name	HP/MHP	BL/MBL	MOV	RNGs	POWs	ATK	DEF	STR	Items	(X,Y,Z)
Player	Yaruo	10/10	3/3	5	1-2	5-3	3	3	2	OXX	(2,4,0)
Player	Yarumi	10/10	3/3	3	1-2	5-3	3	3	1	OXX	(6,1,0)
Enemy	Wandering Soul 1	10/10	3/3	2	1-2	5-3	2	3	0	OXX	(2,13,0)
Enemy	Wandering Soul 2	10/10	3/3	2	1-2	5-3	2	3	0	OXX	(6,11,0)
Enemy	Wandering Soul 3	10/10	3/3	2	1-2	5-3	2	3	0	OXX	(9,12,0)

Simplifying assumptions used in implementation (to decrease coding complexity) are:

(1) MHP, MBL, and max # of items are the same across all units. This is used to simplify reward function so that I can cancel a few common terms.

(2) MOV, RNGs, POWs, ATK, DEF, and STR are the same across all enemy units. This is used to decrease number of linear regression models to consider.

### A Close-Up on Reward Function

$$\omega_1 \frac{\#\_of\_player\_units}{\#\_of\_enemy\_units} + \omega_2 \frac{\frac{1}{\sum_{player} \frac{MHP}{HP+1}}}{\frac{1}{\sum_{enemy} \frac{MHP}{HP+1}}} + \omega_3 \frac{\frac{1}{\sum_{player} \frac{MBL}{BL+1}}}{\frac{1}{\sum_{enemy} \frac{MBL}{BL+1}}} + \omega_4 \frac{\frac{1}{\sum_{player} \frac{maximum\_#\_of\_items}{\#\_of\_items+1}}}{\frac{1}{\sum_{enemy} \frac{maximum\_#\_of\_items}{\#\_of\_items+1}}}$$

With positive weights  $\omega_1 \gg \omega_2 > \omega_3 > \omega_4$  - using assumption (1), this is just

$$\omega_1 \frac{\#\_of\_player\_units}{\#\_of\_enemy\_units} + \omega_2 \frac{\frac{1}{\sum_{player} \frac{1}{HP+1}}}{\frac{1}{\sum_{enemy} \frac{1}{HP+1}}} + \omega_3 \frac{\frac{1}{\sum_{player} \frac{1}{BL+1}}}{\frac{1}{\sum_{enemy} \frac{1}{BL+1}}} + \omega_4 \frac{\frac{1}{\sum_{player} \frac{1}{\#\_of\_items+1}}}{\frac{1}{\sum_{enemy} \frac{1}{\#\_of\_items+1}}}$$

This reward function captures, roughly, how I make decisions while playing this game:

Unit count is the most important factor to consider;  $\sum_{\text{player}} \frac{1}{\text{HP}+1}$  grows quickly as HP of

any player unit approaches zero, decreasing  $\frac{1}{\sum_{\text{player}} \frac{1}{\text{HP}+1}}$  and thus total reward

(denominator term is the opposite); other terms are similar.<sup>2</sup>

### A Close-Up on Training Vector

Since units come and go, and a unit with 0 HP is still active (so that a 0 HP unit is not the same as a neutralized unit which is removed from the field), it seems natural to use variable-length training vectors. To handle this, I simply let the program switch among different models automatically. Using reward function specified in previous section, reward scales properly among models. Using assumption (2), there are 9 linear regression models to fit for the example used, producing 9 sets of parameters, which are loaded to the game.

Units on the field	Model name	Dimension	Notes
	12E3	5x6+1	1P: First Player Unit 2P: Second Player Unit E: Enemy Unit  Each  represents a 6-element array containing the unit's HP, BL, number of items, and (x, y, z) position on the field.
	12E2	4x6+1	
	12E1	3x6+1	
	1E3	4x6+1	
	1E2	3x6+1	
	1E1	2x6+1	
	2E3	4x6+1	
	2E2	3x6+1	
	2E1	2x6+1	

### Outline of Prediction Process

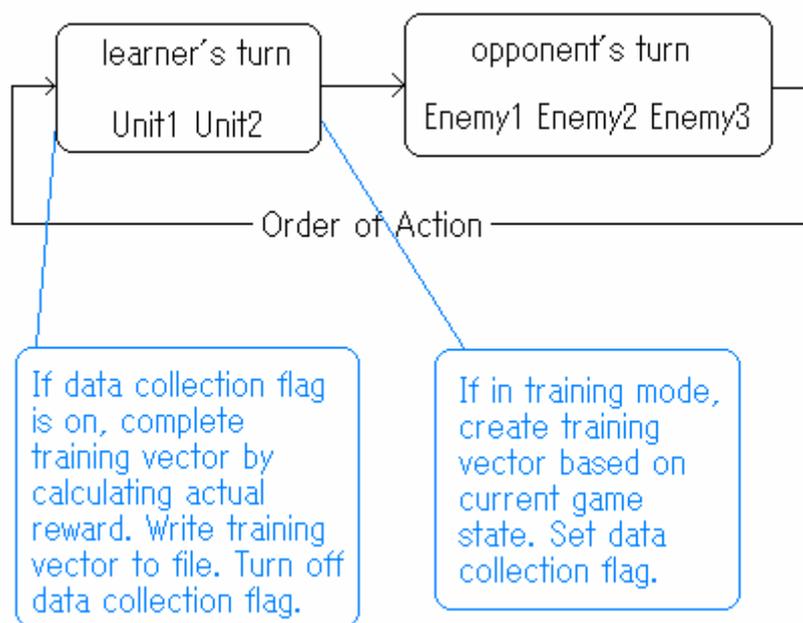
For a unit under learner control, decide the course of action as follows:

- 1) Backup game state
- 2) Act according to AI, but write resulting game state to BestState; predict reward using parameters obtained from linear regression and write to BestReward

<sup>2</sup> +1 is used to avoid division by zero. When number of enemy units is zero, maximum reward is outputted.

- 3) Restore game state from backup
- 4) Loop over all possible actions:
  - 4a) for each action, get resulting state and predict reward
  - 4b) compare this reward with BestReward: if this reward is bigger, write current state to BestState and current reward to BestReward
  - 4c) restore game state from backup
- 5) Copy BestState to current game state, and go to next unit<sup>3</sup>

#### Outline of Data Collection Process



#### Actual Implementation Issue and Extra Tweaks

Actual training data have been obtained by: 1) manually controlling player's units; 2) letting (scripted) AI control player's units; 3) after enough data are obtained, letting machine learning algorithm take control of player's units.

Since prediction can be noisy, I initially used a tolerance factor, TOL, so that course of action is altered only if there's some action with predicted reward TOL bigger than the action initially suggested by AI. As more data are obtained, I realized that it is actually

---

<sup>3</sup> This is a sequential algorithm, meaning each unit under learner control tries sequentially to minimize accumulated relative damage (relative in a sense that this also takes into account damage learner's unit inflicts to opponent's unit during current turn) the opponent could inflict to the learner in next turn, as if the opponent would act immediately. A sequential algorithm is used since otherwise number of cases to consider can easily grow out of bound. (Probably OK for current example, but for situation where each side has a lot of units, this is infeasible.)

hampering the performance, so I removed it (set it to 0). This means ML algorithm is controlling the units without the help of AI at all.<sup>4</sup>

To avoid getting stuck in a loop forever, unwinnable for both sides, a small positive bonus is added to predictions resulting from aggressive actions.<sup>5</sup>

The following tweaks are added to get around of an intrinsic problem of linear regression (at least in current formulation) – it cannot predict potential reward several steps in the future:

I added a small bonus for predictions resulting from mounting. Mounting is usually a good approach to take, despite the fact that it may not yield immediate reward.

I also added a small penalty term proportional to distance between player's units. So, if predicted reward does not say much, the units prefer to stick together.

### **Why It Works**

1) from actual performance

As stated in previous section, by setting TOL to zero after some training, ML algorithm alone takes control of player's units. It improves AI since there are many situations in which AI-controlled player cannot beat AI-controlled enemy, while ML-controlled player beats AI-controlled enemy nicely. 4 set of stage data are attached for reference: stagea.txt, stage1a.txt, stage2a.txt, and stage3a.txt are AI versus AI versions, while stage.txt, stage1.txt, stage2.txt, and stage3.txt are the same stages except that ML algorithm takes control of the player side, winning the game unwinnable by AI-controlled player previously.<sup>6</sup>

Of course, improvement is expected only for matches between ML-controlled player and AI-controlled enemy in a situation similar to the example (i.e., 2 player units, 3 or fewer enemy units) since training data are based on these assumptions, but it can be trained to adapt to more general situations.

2) from reasoning

Unlike chess, one rarely needs to think a lot of steps ahead in this game. Most of the time one step ahead is sufficient. For situations requiring more steps of thinking to play well, tweaks mentioned in previous section, which provide ML algorithm with some "general principles", partially fix this problem, achieving reasonable performance.

---

<sup>4</sup> It still uses AI-suggested action as an "initial condition", but any initial condition will produce the same result since in practice virtually no predictions are identical after some training.

<sup>5</sup> By the way, a variable bonus, increasing with number of turns, turned out to make units insufficiently aggressive at the beginning and overly aggressive towards the end.

<sup>6</sup> To load a stage into a game, simple run the game (dream.exe) with the name of the stage file as parameter. For example, to load stage2a.txt, start the game using "dream stage2a.txt". Once in the game, press any key to enter the field, and press Q to fast-forward if you wish. Press F10 anytime to reset the game.