Massachusetts Institute of Technology

6.038 Artificial Intelligence in Practice

Problem Set 2 Solutions

Part A

1. Quality of Play

Most of you pointed out that the player with wraparound is quite good at surviving but less good at getting rid of all the robots. Some of you speculated that this had to do with a deliberate weighting. Actually, when you look at the implementation of `evaluate_move` given below, you'll see it is an unintentional consequence of a (perhaps ill-considered) numerical weighting scheme.

When playing with no wraparound, the player seems to pick really stupid moves quite often. This happens because in the lookahead, the player determines that a lot of the moves will end up in a loss, some right away, some later. It has no preference for prolonging the agony so it simply picks the first of these (equally bad in the long run) moves. This leads to the appearance of "suicide."

2. Value of lookahead

Many of you tried to reach strong conclusions on the value of lookahead based on very few (less than 100) games. For example, some of you saw what appeared to be negative effects of more lookahead, which is unintuitive and tried to rationalize this result. However, this "result" is a function of the variation in the scores as a function of the different random seeds in the different game sequences. The default behavior of the code is to use a different random seed, based on starting time each time. This leads to substantial variation in measured performance across any short sequence of games.

We did not necessarily expect you to do enough runs to get reliable numbers but you should be careful of jumping to conclusions based on a few tests on a random process.

The following table records number of wins in 1000 games (unless otherwise specified), always starting from the same random seed so the same sequence of initial boards is used. Even with this relatively large number of games there would still be substantial variance on these measurements if we use other seeds (probably on the order of 20 wins either way). However by fixing the initial random seed we see that the number of wins is (as expected) a monotonically non-decreasing function of lookahead (with rapidly diminishing returns, especially for wraparound where it is so simple to survive).

Lookahead	Wrap	No Wrap
0	731	62
1	787	231
2	802	330
3	807	349
4	807	351
5	807	353
6	*	353

3. Scaling up

As we discussed in class, there are a great number of problems in scaling up this lookahead type of game playing to more complex games (or real life). Notable among these are, the exponential growth of the search as a function of the number of options available and the fact that the real world does not stop to wait on us to pick a move.

4. Machine Learning

We wanted you to think about casting this game playing problem as a machine learning problem. Many of you pointed out the option of trying to learn using stored games. In Part B, we explore this option as compared to learning directly from playing, using reinforcement learning. The advantage of learning methods is that, if successful, they can reduce the performance bottleneck of pure lookahead search.

Part B

A sample program can be found at `/mit/6.038/ps2/solution/robots.c`

1. Search

`evaluate_move` simply performs a depth-first search to find the best move. Many of you got the code right, but forgot a subtle point: you do not need to search any deeper if you are already at a terminating board, i.e., winning or losing, at that step. Without checking this, the code still works, but it is slower.

```
int
  evaluate_move(struct board *b, int move, int depth, int lookahead_depth)
{
  struct board *nb = make_move(b, move, &boardQ[depth+1], FALSE);
  int score = static_score(nb);
  int best_score = -10;
  if (depth >= lookahead_depth  /* end of search */
      || score == 100           /* I wins */
      || score <= 0             /* I lost or made illegal move */
      )
    return(score);
  /* Basically doing a complete search in a depth-first fashion */
  for (move = 0; move < 8; move++) {
    score = evaluate_move(nb, move, depth+1, lookahead_depth);
    if (score > best_score)
      best_score = score;
  }
  return(best_score);
}
```

There are many ways to implement `static_score`. For example, for non-terminal boards, one may use the number of live robots and/or the distance to the robots to estimate the score, or even simply use a constant number between 0 and 100, and they all work reasonably well for this particular problem.

Here is the version that was used in the binary we distributed. This is not in any way "optimal." As many of you noticed, it leads to wasteful play when only one robot is left, probably because of the conversion to int at the end. Probably computing the score based on the single closest robot, rather than a sum based on all robots, would be better. But, hey, perfection would have been boring...

```
int
  static_score(struct board *b)
{
  int i, j, d, d2, ri, live, max, score = 0;
  double s;
  /* Handle the terminal cases */
  if (!b->legal) return(-1);
  if (b->I_eaten) return(0);
  if (b->n_live_R == 0) return(100);
  i = b->I_pos[0];
  j = b->I_pos[1];
  /* Compute a score that is the square of the distance, treats dead
     robots as being very far away */
  for (ri = 0; ri < b->n_R; ri++) {
    live = (b->R_pos[ri][0] >= 0);
    d2 = (live ? i - b->R_pos[ri][0] : BOARD_N);
    score += d2 * d2;
    d2 = (live ? j - b->R_pos[ri][1] : BOARD_N);
    score += d2 * d2;
  }
  /* Normalize by largest possible score and multiply by 100. */
  s = 100. * (double) score / (double)(b->n_R * 2 * BOARD_N * BOARD_N);
  score = (int) s;
  return score;
}
```

2. Supervised Learning

Most of you have noticed that the type of encoding, as well as the learning parameters, affect the training results very much, and that this method gives pitiful performance in the robot game (10% or 20% wins in the wraparound game). In fact, on the larger set of games (1000), we found it hard to get a net (using 40 hidden units) to get better than 90% performance on the training set. This means that 1 out of 10 moves is wrong, which is probably just enough to get the player killed every time in a game lasting 10 moves.

The trickiness of picking good parameters for backprop is the norm; the extremely quick convergence you saw in face learning was a bit unusual, as we pointed out then.

The poor behavior of the neural net learning in this game is surprising. This is probably a function of several things:

- The strategy followed by the lookahead player is very complex, since during the early part of the game it is trying to get the robots to crash. This generates some risky behavior which if not completely learned can lead to immediate failure.
- We need a lot more games to accurately learn this type of complex strategy. Note that the TD learner gets to "make up" its own strategy rather than learn the complex one used by the lookahead player.
- Neither of the encodings we gave you was very good. The whole board encoding has all the information but is related in a very complex way to the strategy. The "relative" encoding seems like it should be better but it is actually worse.

It would take some more experimentation to fully diagnose what is going on. By the way, this is a fairly typical outcome of a machine learning experiment, namely, surprising results (positive or negative) that are difficult to rationalize.

3. Reinforcement Learning

The general approach to solving this problem is, at each step, to check all possible moves and calculate scores for the resulting boards of each move. A score is obtained by running the neural network, except for the special terminating cases: illegal move (score=0), losing (score=0), winning (score=1) or surviving (score=0.5). Pick the highest score and take the corresponding move, while training the neural network using the old board (before the move) as input and the highest score as target.

The code follows:

```
int
  pick_move_by_TD(struct board *b, int iter)
{
  int move, best_move;
  double score, best_score;
  double out_err, hid_err;
  struct board *nb;
  best_score = 0.;
  for (move = 0; move < 8; move++) {
    nb = make_move(b, move, &boardQ[0], FALSE);
    if (!nb->legal || nb->I_eaten) /* we lost */
      score = 0.0;
    else if (nb->n_live_R == 0) /* we won */
      score = 1.0;
    else if (iter == max_iter-1) /* we survived */
      score = 0.5;
    else {
      load_board_into_net(net, nb); /* load new board */
      bpnn_feedforward(net);    /* compute prediction */
      score = net->output_units[1]; /* get prediction */
    }
    if (score > best_score) {   /* keep best prediction */
      best_score = score;
      best_move = move;
    }
  }
  /* The TD(0) update rule -- we should have predicted what we got */
  load_board_into_net(net, b);  /* load old board */
  net->target[1] = best_score;  /* try to get best score */
  bpnn_train(net, alpha, 0.0, &out_err, &hid_err);
  return best_move;
}
```

The results here are also surprising, but good this time. The TD player (using the default representation) reaches around 60% wins very quickly but then sort of stagnates at that level. It is an open question whether many more games or other encodings could boost performance to be competitive with the lookahead player.