Game Playing Programs that learn from experience:
Complementary material to Programming Assignment 4

In the programming assignment, you used hill climbing to learn the appropriate weights for a neural net backgammon player. This report describes how to use a special form of reinforcement learning to train the neural network called $TD(\lambda)$.

You can use this report so as to (i) get ideas to improve your solution for the programming assignment or (ii) implement the $TD(\lambda)$ algorithm at some later point so as to satisfy your curiosity how a stronger learning agent operates. Most of the infrastructure to implement $TD(\lambda)$ will be already available once you have worked on the programming assignment. Towards the end of this report you can also find pseudo-code that can guide your implementation.

TD-Gammon Revisited

As mentioned in the programming assignment, Tesauro created a master backgammon player he called TD-Gammon. Here are some additional details derived from:

`http://www.cs.ualberta.ca/~sutton/td-backprop-pseudo-code.text`
`http://www.cs.ualberta.ca/~sutton/book/11/node2.html`

Tesauro first created a neural network player he called *Neurogammon*. Neurogammon had special features encoded into the neural network for representing backgammon playing knowledge and was trained against hand-picked examples. It learned to play reasonably well and was categorized as an "intermediate" player.

Tesauro's next experiment was a backgammon player that learned by playing itself with zero-knowledge of the game, except for required technical information such as legal moves, win-states, etc. Tesauro applied a learning method described as a "gradient-descent form of the TD(λ) algorithm" that we will describe later. This player, called TD-Gammon 0.0 trained by playing itself. After 300,000 rounds, it could play as well as all contemporary computer programs.

Later modifications reincorporated the specialized backgammon features into the neural network (TD-Gammon 1.0) and multi-ply searches (TD-Gammon 2.0 and 3.0). The most recent version plays at the level of the best human champions in the world.

This report describes how you can recreate TD-Gammon 0.0 for your intellectual stimulation and pleasure. Additionally, we will be examining both the theory of the method and the practical minutia that results from the application thereof. It should be noted that training takes time!!

The TD algorithm

Before we discuss the TD algorithm, we should talk briefly about the supervised learning technique known as backpropagation. Backpropagation works for neural networks that are "feed forward", meaning there is no feedback or loops. Provided with an expected output, backpropagation takes the error and applies it starting at the outputs and propagating backwards with the amount of correction dependent on the existing weight.

But for backpropagation to work in a game like backgammon, after every move the network would require the "correct" decision and use that knowledge to train the network. As that information is not available, Tesauro used temporal difference methods. TD methods are somewhat like backpropagation over time to assign credit or blame of some reward to a previous state.

More specifically, when the network wins or loses a game, were all moves equally responsible for that win or loss? Obviously not. So, a temporal difference method is used to identify and reinforce the credit or blame with respect to the outcome.

The actual mechanics work like this. You start with a random network. This network is used to play both sides of a backgammon game. At each point, the network evaluates the next moves to pick the best move. Obviously, when the weights are random, the moves will also be random. However, even though the evaluation means nothing at the start, you still treat the next evaluation as if it were the correct one. So, for instance, if it is white's move, and the evaluation says that white's odds of winning are 0.75, and then on black's move, the evaluation function recalculates the new odds of white wining to be 0.80, the algorithm treats the 0.80 to be the correct value, even though the new number is also an estimate, and trains the network accordingly. This component behaves exactly like backpropagation.

What the temporal difference algorithm adds is to maintain information over time about these trainings. While the information does decay eventually, it does ow forward in time influencing future training. When one player or the other actually wins, and the odds of winning are now concrete (i.e., 0.0 or 1.0), a more correct error is computed and applied to training with influence from the previous trainings!

Even though the data is flowing forward in time, the net result is that the impact of winning or losing is backpropagated to the previous moves played.

The structure of the neural network we are using is shown in the Figure.

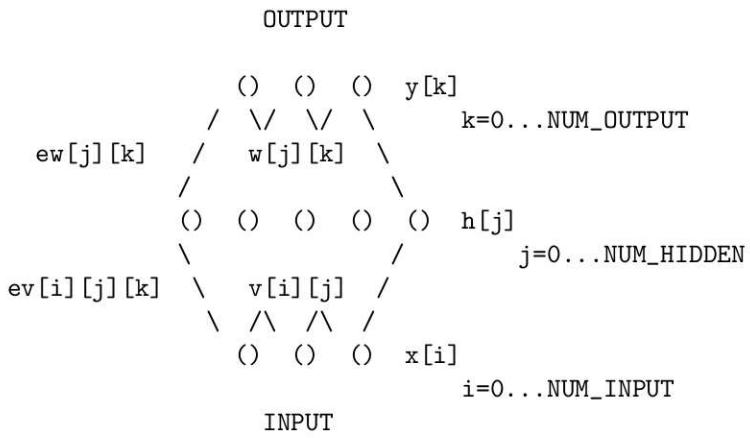

Figure 1: Sample structure of the neural network used.

Mathematically, at each move, the weights are adjusted according to the following formula:

$$w_{jk}^{t+1} = w_{jk}^{t} + \beta \Omega_k e_{jk}^{t}$$

$$u_{ij}^{t+1} = u_{ij}^{t} + \alpha \sum_{k} \Omega_k e_{ijk}^{t}$$

where:

- w_{jk} is the weight from hidden unit j to output unit k
- u_{ij} is the weight from input unit i to hidden unit j

- k is an output unit
- $\Omega_k = y_k^{t+1} - y_k^t$ is the difference of the output of node k between time $t+1$ and t
- α, β are learning rates
- e_{ijk}, e_{jk} are the eligibility traces at time t

Eligibility traces represent the information carried forward from one training to another. They are computed using the following formula:

$$e_{jk}^{t+1} = \lambda e_{jk}^t + (y_k^t \times (1 - y_k^t) \times h_j^t)$$

and

$$e_{ijk}^{t+1} = \lambda e_{ijk}^t + (y_k^t \times (1 - y_k^t) \times w_{jk}^t \times h_j^t \times (1 - h_j^t) \times x_i^t)$$

where

- λ is the decay parameter
- y_k^t is the value of output node k at time t
- h_j^t is the value of hidden node j at time t
- x_j^t is the value of input node i at time t

Yes, it can take some time to understand these equations, but they do have reasonable and logical explanations.

Implementation Details

Despite a plethora of published research about or based on TD gammon, there is actually very little written about how to recreate Tesauro's results. Even Tesauro's paper itself is sketchy on details. Here are the known configuration parameters:

- $\alpha = 0.1$
- $\lambda = 0.7$
- Hidden Unit Count = 40

There are at least two unknowns:

1. **Did Tesauro wipe out his eligibility trace data at the end of each complete game?** Theoretically, an eligibility trace should only apply to a sequence of moves within the same game. However, experimentation on the part of the TA suggests that it works better to not reset the data each round.
2. **How did Tesauro create his evaluation function?** Tesauro had four outputs; two represented the odds of white or black winning, and two represented the odds of white or black getting a gammon. So, if the output suggested that the odds of white winning was 0.8 and the odds of black winning was 0.8, how should it evaluate the board?

Additionally, three additional tweaks are often used in reinforcement learning that Tesauro does not mention.

1. **Alpha Splitting:** The first tweak is splitting α into two parameters (α and β) and applying one learning on the weights from input to hidden units, and the other for learning on the weights from hidden units to output nodes.

2. **Alpha Decay:** The second tweak is to use an α decay. The idea is that at the beginning of training, α should be large leading to large changes. But, as time goes on, α should decay to prevent the accidental loss of good playing knowledge.

3. **Player History:** It is possible for a good player to be "lost" by accident during training. Backgammon involves some chance, and poor behavior might occasionally appear good. One option is to save the player's neural network and restore it if it's performance worsens.

Overall, you should probably experiment with up to three of the following parameters: α, λ, hidden unit count, output count (and representation), evaluation function (only with multiple outputs, eligibility trace lifetime, alpha splitting, alpha decay, or player history. You should try a number of combinations of values for your three parameters to explore the impact. It can take a long time to run all of these experiments (that's why we suggest not to experiment with more than 3), but you can prioritize and automate the procedure.

Here are a few other hints:

- **Neural Network Setup** Use 198 input units for the network as did Tesauro:
  - For each point on the backgammon board, four inputs represent how many pieces of one color are on that point. If there is 0-3 pieces, than 0-3 inputs should be enabled (e.g., 0 pieces = 000, 1 = 100, 2 = 110, 3 = 111) and if there are n pieces, with $n > 4$, then the first three units are set to 1, and the fourth unit is set to $(n - 3) = 2$. This accounts for 192 units.
  - Two additional units encode the number of white and black pieces on the bar where the value of the input is $\frac{n}{2}$.
  - Two more units encode the number of white and black pieces successfully removed from the board. The value of each input is $\frac{n}{15}$.
  - Finally, two units encode whose turn it is to play. For white, 01 and for black 10.
- **Shared Network** When you're playing rounds using the code framework, both players are sharing the same neural network and training parameters. In the code, our `BackPropPlayer` has a method `createLearningPartner` that returns another `BackPropPlayer` that shares the network.
- **Player Evaluation** Your player should be evaluated by its performance against your expectiminimax player from the programming assignment. Every 1000 rounds, you should test your player and report the number of wins out of 100. For fun, you might also test against a random player and your hill-climbing neural network player. If you feel that your expectiminimax player is too weak (or is easily beaten by your td-gammon player) you can attempt to integrate with gnubg.
- **Rounds** You should run your experiments for at least 30,000 rounds. If you are really ambitious, try running them for 300,000 rounds. You should know that 300,000 rounds will probably require a day or two, so if you want to do that, start early.
- **Non-learning Mode** You need to be able to turn learning off for when your player is playing evaluation rounds.
- **Unexpected Results** The provided code has, in various configurations, experienced improvement in the first 50,000 rounds, and then slow worsening as the rounds approach 200,000. This may be a bug in the code, or it may be expected. In any case, it may effect your results.

If you work on the TD-gammon approach (e.g., during the holidays), please contact the instructor with your experience and results.

Important Note: Since converting the mathematical description of the TD algorithm into code is difficult, you are being provided with a pseudo code implementation of this procedure. However, it will probably take some time to debug and get working correctly!

```
/* MOST OF THIS IS ACTUAL JAVA CODE, BUT BEWARE      */
/* IT is STILL PSEUDO-CODE-ish... YOU WILL HAVE      */
/* TO FILL IN THE BLANKS (make real variables, etc.) */

public double gradient(HiddenUnit u) {
  return u.getValue() * (1.0 - u.getValue());
}

/* Ew and Ev must be set up somewhere to the proper size and set to 0 */
public void backprop(double[] in, double[] out, double[] expected) {
  /* compute eligability traces */
  for (int j = 0; j < net.hidden[0].length; j++)
    for (int k = 0; k < out.length; k++) {
      /* ew[j][k] = (lambda * ew[j][k]) + (gradient(k)*hidden_j) */
      Ew[j][k] =
        (LAMBDA * Ew[j][k]) +
        (gradient(net.hidden[1][k]) * net.hidden[0][j].getValue());
      for (int i = 0; i < in.length; i++)
        /* ev[i][j][k] =
           (lambda * ev[i][j][k]) +
           (gradient(k)+w[j][k]+gradient(j)+input_i)*/
        Ev[i][j][k] = ( ( LAMBDA * Ev[i][j][k] ) +
                        ( gradient(net.hidden[1][k]) *
                          net.hidden[1][k].weights[j] *
                          gradient(net.hidden[0][j])     *
                          input[i]
                        )
                      );
    }

  double error[] = new double[out.length];

  for (int k =0; k < out.length; k++)
    error[k] = expected[k] - out[k];

  for (int j = 0; j < net.hidden[0].length; j++)
    for (int k = 0; k < out.length; k++) {
      /* weight from j to k, shown with learning param of BETA */
      net.hidden[1][k].weights[j] += BETA * error[k] * Ew[j][k];
```

```
      for (int i = 0; i < in.length; i ++) {
        net.hidden[0][j].weights[i] += ALPHA * error[k] * Ev[i][j][k];
      }
    }

}

public Move move(Backgammon backgammon) {
  Move bestMove = null;
  double expectedUtility = -1.0;
  double nextoutput [] = null;
  for (Move m: backgammon.getMoves()) {
    /*
     * output is an array of 1-4 depending on configuration
     *
     * evaluate the next board for the OTHER PLAYER; you'll
     * have to adjust your utility accordingly...
     *
     * getCurrentBoard is the board AFTER the move
     */
    double output[] = net.getValue(m.getCurrentBoard(), otherPlayer);
    double utility = computeUtility(output);
    if (utility > expectedUtility) {
      bestMove = m;
      expectedUtility = utility;
      nextoutput = output;
    }
  }
  if (learningMode) {
    /* you'll need to pass the input values to backprop... */
    double currentinput[] = boardToVector(m.getOriginalBoard());

    /* get original board is the board BEFORE the move... i.e., right now */
    /*
     * OH OH!!!! WARNING WARNING WARNING, DANGER WILL ROBINSON!!!
     * you must call the neural net's get value on the original input
     * BEFORE you call backprop. The HiddenUnit class has a "getValue" that
     * is cached; it is not computed every time. If you don't call this
     * before backprop, then the nodes will have the values set from
     * the last call to the whole network's getValue() which may or may not
     * be the one you want
     */
    double currentoutput[] = net.getValue(m.getOriginalBoard(), thisPlayer);
```

```
     * notice, you don't have to monkey with nextoutput even though it is
     * for the other player... this is because it's output is player independent.
     * if one output, it equals odds of white winning, if two outputs than
     * one is odds of white winning, other is odds of black winning
     */
    backprop(currentinput[], currentoutput[], nextoutput[]);
  }
  return bestMove;
}

public void lost() {
  if (learningMode) {
    /* assuming two output with odds for white and black */
    double actual[] = new double[2];
    if (myPlayer == Board.WHITE){
      actual[Board.WHITE]=0.0;
      actual[Board.BLACK]=1.0;
    }
    else{
      actual[Board.WHITE]=1.0;
      actual[Board.BLACK]=0.0;
    }

    /* you'll have to save currentBoard from the move method or something */
    double in[] = boardToVector(currentBoard);
    double out[] = net.getValue(in);

    backprop(in, out, actual);
  }
}

/* figure out boardToVector, win(), etc. by yourself!!! */
```