

# Reinforcement Learning and the Temporal Difference Algorithm

By John Lenz

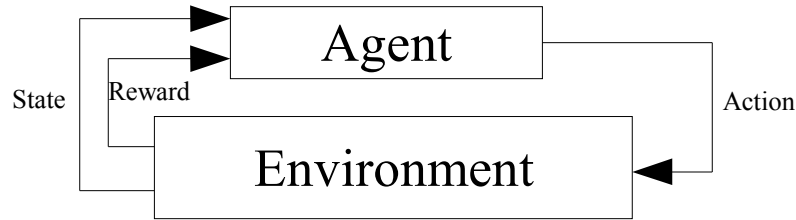
# 1. Introduction to Reinforcement Learning

Reinforcement learning is learning to maximize a reward signal by exploring many possible actions. The agent is not told the correct actions; instead it explores the possible actions and remembers the reward it receives. With supervised learning, an agent takes an action and is then told what was the correct action. For example, the agent will classify a picture as a number “3” and the teacher will explain that the picture is the number “8”. In reinforcement learning, the agent takes an action and then receives a reward based on that action; there is no teacher to give the correct action. In some problems like games such as checkers or chess, the “correct” action isn't even known.

Reinforcement learning can be applied to many control problems where there is no expert knowledge about the task. Reinforcement learning attempts to mimic one of the major ways humans learn. Instead of being told what to do, we learn through experience; in our interaction with the environment, we feel pain and pleasure punishing or rewarding us for our actions. In a similar way, a reinforcement learning agent learns to interact with an unknown and unspecified environment. Reinforcement learning can be applied to any goal directed and decision-making problem; specific knowledge about the environment and expert teaching are not required.

The reinforcement learning problem model is an agent continuously interacting with an environment. The agent and the environment interact in a sequence of time steps. At each time step  $t$ , the agent receives the state of the environment and a scalar numerical reward for the previous action, and then the agent then selects an action. A time sequence  $t=1,2,3\dots$  can either form an episode, where the state is reset to the initial state and  $t$  is reset to 1 after a specific terminal state is reached, or time can continue marching towards infinity to form a continual task. A game like checkers or tic-tac-toe is an example of an episodic task, where the agent plays until

the game is won or lost, while a robot controlled vacuum tasked with keeping a room clean is a continuous task.



Formally, the goal of the agent is to maximize the expected return,  $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$  that is, the sum of all future rewards discounted by a number between zero and one. For episodic tasks like games, all future rewards past the end of the episode are defined to be zero. The goal of the agent is to learn a policy, which how the agent selects actions; it maps states to actions. All reinforcement learning algorithms are based on estimating the value function which is the expected return starting from state  $s_t$  and following policy  $\pi$ .  $V^\pi(s) = E_\pi\{R_t | s_t = s\}$  The value function is an estimate of *how good* it is for an agent choosing actions based the policy  $\pi$  to be in a given state. Similarly, the state-action value function is the expected return starting from state  $s$ , taking action  $a$ , and thereafter following policy  $\pi$ , defined as

$$Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\}$$

The reinforcement learning problem is to find an optimal policy and optimal value function. A policy  $\pi$  is defined to be better than policy  $\pi'$  if the expected return is greater than or equal for all states. That is,  $\pi \geq \pi'$  if and only if  $V^\pi(s) \geq V^{\pi'}(s)$  for all  $s$ . There is always at least one policy that is greater than or equal to all other policies, and these form the set of optimal policies, denoted  $\pi^*$ . They all share the same value function, called the optimal value function, denoted  $V^*$ , and defined as  $V^*(s) = \max_{\pi} V^\pi(s)$ . The optimal state-action value function is defined as  $Q^*(s, a) = \max_{\pi} Q^\pi(s, a) = E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\}$ .

## 2. Temporal Difference Learning

An obvious approach to learning the value function is to update the estimate of the value function when the actual return is known.  $\Delta V(s_t) = \alpha [R_t - V(s_t)]$  This method is called the constant- $\alpha$  Monte Carlo method, where  $\alpha$  is a learning parameter between 0 and 1. Since the actual return is the sum of all future rewards, this algorithm must wait until the end of the episode when the expected return is known before the value function is updated. Richard Sutton proposed to instead estimate the expected return by the next reward plus the value of the next state. The update to the value function takes the difference of successive estimates of the value function, thus the name *temporal difference*. The simplest method, known as TD(0), is given by  $\Delta V_t(s_t) = \alpha [r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)]$

More generally, the expected return can be estimated by the next n rewards,

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \quad \Delta V_t(s_t) = \alpha [R_t^{(n)} - V_t(s_t)]$$

Richard Sutton proposed the TD( $\lambda$ ) algorithm, which mixes TD(0) and Monte Carlo methods and uses a weighted average of n-step returns. The resulting estimation of the expected return, called the  $\lambda$ -return is defined as  $R_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{(n-1)} R_t^{(n)}$   $\Delta V_t(s_t) = \alpha [R_t^\lambda - V_t(s_t)]$  If  $\lambda=0$ , this reduces to  $R_t^{(1)}$  which is the TD(0) algorithm. If  $\lambda=1$ , this reduces to  $R_t^{(\infty)}$  or just  $R_t$  which is the constant- $\alpha$  Monte Carlo method. The parameter  $\lambda$  can vary between 0 and 1, and provides a trade off between updating based on the final result and updating based only on the next estimate.

The definition given above is called the forward view of TD( $\lambda$ ), since the weight update requires knowledge about the future. Richard Sutton also proved an equivalent view of TD( $\lambda$ ), called the backward view. In the backward view of TD( $\lambda$ ), every state has an eligibility trace at time t, denoted  $e_t(s)$ . On each step, the eligibility traces for all states are decayed by  $\gamma\lambda$  and the

current state gets incremented by one. The TD( $\lambda$ ) algorithm becomes

$$e_t(s) = \gamma \lambda e_{t-1}(s) \text{ if } s \neq s_t \text{ and } e_t(s) = \gamma \lambda e_{t-1}(s) + 1 \text{ if } s = s_t$$

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t). \quad \Delta V_t(s) = \alpha \delta_t e_t(s), \text{ for all } s$$

In the backward view, the value function of all recently visited states is updated by  $\delta_t$ , where “recently visited” is defined in terms of the states eligibility trace.

Temporal difference learning can be easily extended to the control problem, that is, learning the optimal policy  $\pi^*$ . One policy learning method is to use an on-line  $\epsilon$ -greedy policy while training: instead of learning  $V(s)$  learn  $Q(s,a)$ , then every time-step select the action with the largest value of  $Q(s,a)$   $1-\epsilon$  percent of the time and select a random action  $\epsilon$  percent of the time. The random actions cause the agent to explore the state space. The TD( $\lambda$ ) is extended to the Sarsa( $\lambda$ ) algorithm for on-line control by replacing  $V(s)$  with  $Q(s,a)$  and  $e_t(s)$  with  $e_t(s,a)$ .

```
Repeat for each episode and each step in episode
  Take action  $a$ , observe reward  $r$  and next state  $s'$ 
  Choose  $a'$  from  $s'$  using some policy involving  $Q$ 
   $\delta = r + \gamma Q(s', a') - Q(s, a)$ 
   $e(s, a) = e(s, a) + 1$ 
  For all  $s, a$ :
     $Q(s, a) = Q(s, a) + \alpha \delta e(s, a)$ 
     $e(s, a) = \gamma \lambda e(s, a)$ 
   $s = s'$ ;  $a = a'$ 
```

### 3. Function Approximations to the Value Function

The TD( $\lambda$ ) and Sarsa( $\lambda$ ) algorithms given above assume the value function and the state-action value function are represented by a large table, with one entry per state or state-action pair. The agent is unable to generalize to states that have not been experienced and as the number of states grows, the space and time needed to store these tables grows very large. To solve these problems, the value function can be approximated by any of the function approximation supervised learning methods: artificial neural networks, radial basis networks, support vector machines, etc.

One popular method is to use the gradient-descent search method. Assuming  $f(W, x)$  is a continuous differentiable function of  $W$  for  $V(s)$ , the forward view of TD( $\lambda$ ) becomes

$\Delta \vec{w} = \alpha (R_t - f(\vec{w}, s_t)) \nabla_{\vec{w}} f(\vec{w}, s_t)$  In the backward view, every entry in the weight vector has an eligibility trace  $\vec{e}_t = \gamma \lambda \vec{e}_{t-1} + \nabla_{\vec{w}} f(\vec{w}, s_t)$   $\Delta \vec{w} = \alpha \delta_t \vec{e}_t$  The Sarsa( $\lambda$ ) algorithm is identical, with  $f(W, x)$  a continuous differentiable function of  $W$  for  $Q(s, a)$ .

## 4. Implementation

Using C++, I implemented a two-layer feed-forward artificial neural network with bias terms to approximate the value function and the state-action value function. The hidden layer uses the hyperbolic tangent activation function, while the single output neuron uses the sigmoid activation function. Both the value of  $\alpha$  and  $\lambda$  can be set to arbitrary values, while  $\gamma$  is set equal to 1.

An agent is built on top of the neural network, and implements two modes of learning. In the first mode, the agent is given the current state, the current reward, and the list of all possible actions. The agent then evaluates the state-action value function using the neural network and then selects the best action  $1-\epsilon$  percent of the time and a random action  $\epsilon$  percent of the time. The second mode uses an after-state value function to pick the current action. The agent is given the current state, the current reward, and a list of all possible next states. The agent then evaluates  $V(s)$  for all possible next states and again selects the best  $1-\epsilon$  percent of the time and a random action  $\epsilon$  percent of the time. All states and actions are represented by vectors of real numbers, so the agent and the neural network are independent of whatever environment they interact with.

The agent is used by the environment code, and is implemented as a C++ class in the files named `agent.h` and `agent.cc` and has the following public functions:

- `Agent(float lambda, float alpha, int numInputNodes, int numHiddenNodes, int random)`

The constructor takes the learning parameters `lambda` and `alpha`, the number of input nodes and the number of hidden nodes in the neural network, and the random probability  $\epsilon$  of selecting a random action.

- `void load(string filename)`
- `void save(string filename)`

These functions load and save the weights of the neural network to the file given as filename.

- `void markEndEpoch()`
- `void markEndEpoch(int reward, State state)`

These functions mark the end of an epoch, with the second function carrying out one more weight update before the end of the epoch.

- `void setTraining(boolean train)`

The value of `setTraining` determines if any weight updates occur.

- `Vector makeStateDecision(TimeStep ts, float reward, Vector state)`

This function iterates one time-step, selecting the action based on the after-state value function. The first parameter, `TimeStep ts`, is a C++ class representing the list of all possible next states. The other two parameters are the reward for the previous action and the current state. This function then returns the next state that it selected.

- `Vector makeControlDecision(TimeStep ts, float reward, Vector state)`

This function iterates one time-step, selecting the action based on the state-action value function  $Q(s,a)$ . Again, `TimeStep` is a C++ class representing the list of all possible actions, while the other two parameters are the reward and the current state. This function returns the action that it selected.

The `Vector` type is an arrays of floats, while the `TimeStep` class is implemented in the file `timestep.h` as a simple queue. These functions provide the entire interface the environment uses to interact with the agent. Many different types of environments can be implemented without needing to change any code in the agent.

The artificial neural network is implemented in the files `neural.h` and `neural.cc` and has the public functions, many of which are obvious:

- `NeuralNet(float lambda, float alpha, int numInputLayer, int numHiddenLayer)`
- `void load(string filename)`
- `void save(string filename)`
- `float calculate(Vector input)`
- `float calculate(Vector input1, Vector input2)`
- `void saveStateForUpdate()`
- `void updateWeights(float reward)`
- `void markEndEpoch()`

The calculate functions return the value of the neural network given the input. The calculate function with two inputs just concatenates the two input vectors to form the input vector for the neural network. The `saveStateForUpdate()` function uses the last call to `calculate()` as the current input and updates the eligibility traces. The gradient is calculated using the back propagation algorithm. Recall, the update trace update equation is  $\vec{e}_t = \gamma \lambda \vec{e}_{t-1} + \nabla_{\vec{w}} f(\vec{w}, s_t)$

```

let  $\phi'$  = the derivivite of the sigmoid function
let  $\tanh'$  = the derivitive of the hyperbolic tangent

oldOutput=output
// backpropagate to hidden-output weights
for i=0 to numHidden-1
    outputTracei= $\lambda$ *outputTracei + hiddenOutputi* $\phi'(u)$ 
outputTracebias= $\lambda$ *outputTracebias+ $\phi'(u)$ 

// backpropagate to input-hidden weights
for j=0 to numHidden-1
     $\delta$ = $\tanh'(hiddenActivation_j)$ * $\phi'(u)$ *outputWeightj
    for i=0 to numInput-1
        hiddenTracej,i= $\lambda$ *hiddenTracej,i+inputValuei* $\delta$ 
    hiddenTracej,bias= $\lambda$ *hiddenTracej,bias+ $\delta$ 

```

The `updateWeights(float reward)` function uses the last call to `calculate()` as the current input and updates the weights based on the reward, the final output, and the eligibility traces. Recall, the weight update is given by  $\Delta \vec{W} = \alpha \delta_t \vec{e}_t$

```

error =  $\alpha$  * (reward + output - oldOutput)
for i=0 to numHidden-1
    outputWeighti = outputWeighti + error * outputTracei
for i=0 to numHidden
    for j=0 to numInput-1
        hiddenWeighti,j = hiddenWeighti,j + error * hiddenTracei,j

```

During one time-step, the agent calls `calculate()` on the action it selected and then calls `saveStateForUpdate()`. In the next time-step the agent calls `calculate()` on the current state and then calls `updateWeights(reward)`. Separating the weight update into two stages like this does not require the neural network and the agent to store previous values of the state and previous values of the hidden outputs.

The algorithm for `Agent::makeStateDecision` is

```

Given the reward and the current state,
if training
    if not start of an epoch and not a random move
        nnet.calculate(currentState)
        nnet.updateWeights(reward)

    // mark the current state for future use of weights
    nnet.calculate(currentState)
    nnet.saveStateForUpdate()

for each state in list of possible next states
    nnet.calculate(state)
return best action 1- $\epsilon$  percent and random action  $\epsilon$  percent

```

while the algorithm for `Agent::makeControlDecision` is given by

```

Given the reward and the current state,
for each action in list of possible actions
  nnet.calculate(curState, action)
curAction=best action 1-ε percent and random action ε percent

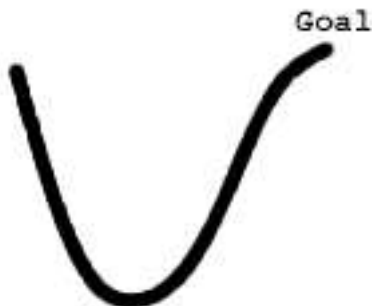
if training
  if not start of an epoch and not a random move
    nnet.calculate(currentState, curAction)
    nnet.updateWeights(reward)

  // mark the current state for future use of weights
  nnet.calculate(currentState, curAction)
  nnet.saveStateForUpdate()

```

## 5. Tests

The first test I implemented was Example 8.2 in section 8.7 in Sutton's book. The goal of the problem is to drive a car to the top of a mountain. The controller has three possible actions, full throttle forward, full throttle backwards, and no throttle. The problem is, even at full throttle, the car is unable to accelerate up the steep slope. The car must first move away from the goal up the other slope to gain enough momentum to make it to the top. There are two state variables,



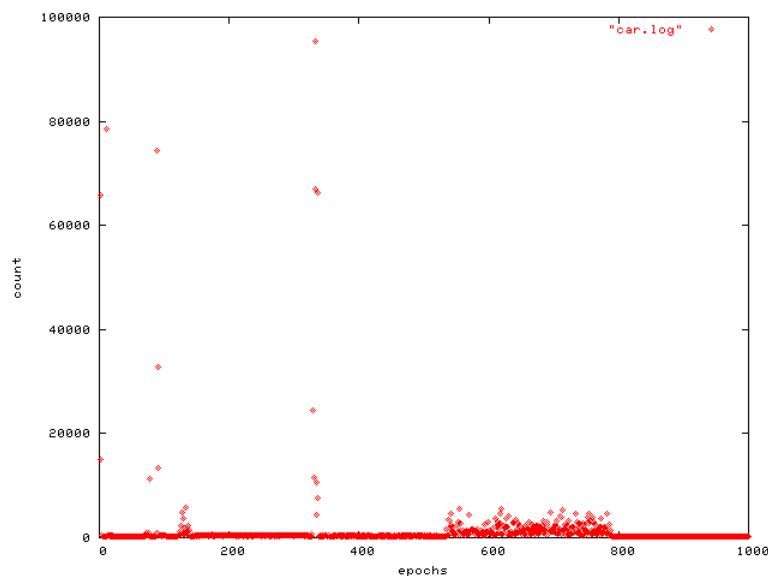
the position and the velocity. The action taken by the agent can be either  $a = +1$ ,  $a = -1$ , or  $a = 0$ . The environment is implemented in the file called `car.cc`. The state update is given by the equations:

$$\begin{aligned}
 x_{t+1} &= x_t + v \\
 v_{t+1} &= v_t + 0.001 a_t - 0.0025 \cos(3 * x_t)
 \end{aligned}$$

The position is kept between -1.2 and 0.5, and when the agent has reached position 0.5 the goal

has been reached. The velocity is bound inside  $-0.07$  and  $0.07$ . If the car reaches the top on the right, it just hits a wall; its velocity is set to zero and the position is kept at  $-1.2$ .

The agent was trained to learn the state-action value function with a reward of  $-1$  for every time-step in which it has not reached the goal and a reward of  $+1$  when it reached the goal. The agent was presented with the current state and the list of three actions. After trying many state representations, the fastest to converge was to break the position up into 9 sections and the velocity up into 9 sections. The state was then represented by a binary vector of length 18, with the first 9 inputs a one-of-n encoding of the position (ex. 001000000 for the third section) and the second set of 9 inputs a one-of-n encoding of the velocity. The actions were also represented as a one-of-n encoding, with 100 full throttle forward, 010 no throttle, and 001 full throttle backwards. Other encodings might work as well; it would be interesting to try a fuzzy encoding of the input vectors.

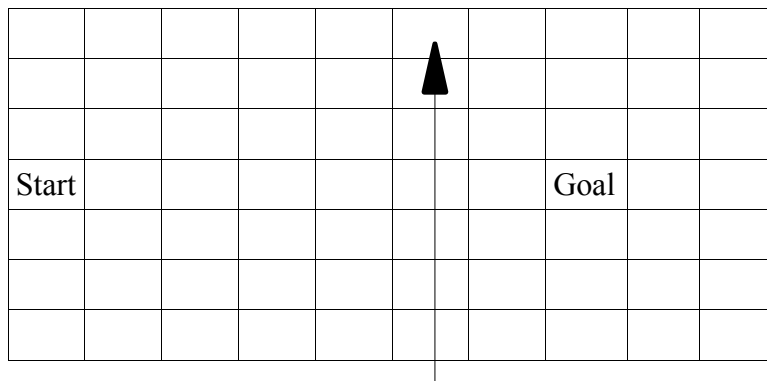


This graph shows the number of steps the agent took to get to the top, versus the epoch number. The initial run of the agent with random weights took 65,873 steps to get to the top, but by the ninth epoch, the agent had converged to 186 steps to get to the top. Notice that in later

epochs the agent would randomly take an action into a part of the state space it had not explored yet causing the spikes in the count, but the agent eventually converged back to around 200 steps to the top.

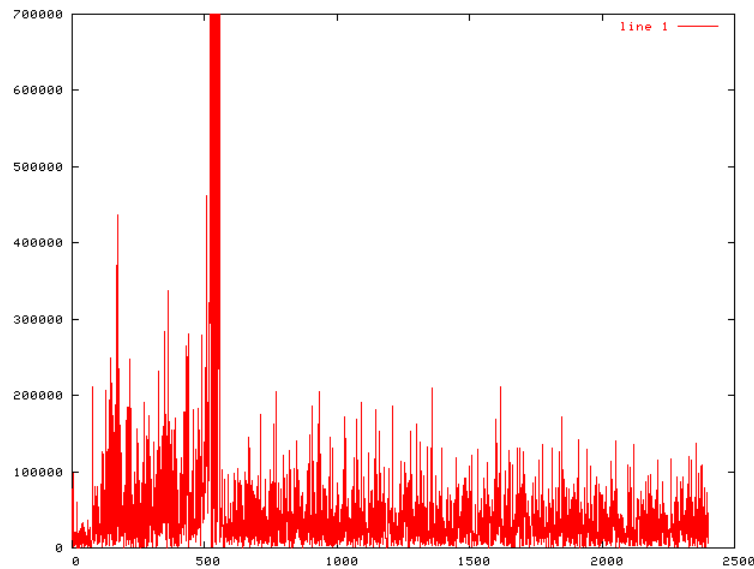
The time to converge was dependent on the initial random values. About 25% of the time, the agent would converge very fast (within 15-20 epochs) to around 200 steps, while other trials the agent would never improve past on average 20,000 steps. I only ran the car program for the first 1000 epochs, and eventually all agents should converge to the solution.

The second trial I ran was Example 6.5 from section 6.4 of Sutton's book. In this problem, the agent is placed on a 10 by 7 grid and must walk from the start square to the goal square. The difficulty is a strong cross wind across the middle of the board. On columns with a wind, the agent is pushed either one or two squares upward by the wind. The agent could move in the four cardinal directions: left, right, up, and down.



Wind strength: 0 0 0 1 1 1 2 2 1 0

The agent learned the state-action value function with a reward of -1 for each time step the agent did not reach the goal and +1 when the agent reached the goal. The state was a binary vector of size 70, with each input representing one square. The actions were encoded with a one-of-4 encoding.



This is a graph of count vs. epochs. Running for 2400 epochs, which took over 20 hours, the average number of counts to reach the top is decreasing. The agent takes a random action 10% of the time, thus causing a large exploration of the state space.

## 6. Games

Temporal difference learning is also applicable to n-in-a-row type games like tic-tac-toe, checkers, connect four, backgammon, and chess. These type of games use the after-state value function approximation instead of the state-action value function. This allows the network to exploit symmetries in the state space, where different starting states and different actions can lead to the same state. In such cases, different state-action pairs lead to the same final state and thus must have the same value of  $Q(s,a)$  since the expected reward is the same. But an agent learning  $Q(s,a)$  would need to learn each case separately while if the agent used a value function, the agent would immediately evaluate both state-action pairs equally.

Perhaps the most famous application of temporal difference learning to games is Gerald

Tesauro's TD-Gammon program. Tesauro used a neural network and a gradient descent after-state value function  $TD(\lambda)$  learning algorithm. Using this as a base and adding deeper searches and some expert knowledge, Tesauro's backgammon program plays at the level of the best human players in the world. Other work is being done attempting to mimic the success of TD-Gammon and the  $TD(\lambda)$  algorithm in Go, Chess, and Othello.

During training, the agent has an unlimited supply of games through self play. Two separate agents are created and play against each other, only one of which is training and updating its weights. Every set number of epochs (I tried 1000) the weights are copied from the training agent to the other agent. In this way, the agent trains for 1000 games against a static, unchanging opponent. After this learning session, the agent starts a new session with its current policy as the static opponent.

I implemented both tic-tac-toe and checkers with varying degrees of success. The state space for both these games is very large. Discounting symmetry and illegal board positions, tic-tac-toe has  $3^9$  or almost 20,000 states (since each board position can have an "X", an "O" or be free) and checkers has  $5^{32}$  or over  $10^{22}$  states (each board position can have a white king, a white man, a red king, a red man, or be free). The success is largely dependent on the state representation and the values of  $\epsilon$ ,  $\lambda$ , and  $\alpha$ .

Tic-tac-toe is implemented in the file called tictactoe.cc. The agent uses self play to learn. The reward is +1 at the end of any game the agent won, -1 at the end of any game the agent lost, and 0 at the end of a tie and all intermediate board positions. I experimented with many state representations, first trying a raw representation. The input vector is an array of length 9 with a value of +1 if the square is the player's, -1 if the square is the opponent's, and 0 if the square is free. Even after many self training games, I was able to consistently beat the

computer.

Levkovich, Hildeshaim, and Levkovich-Frank found a better representation of the board. In their tic-tac-toe game, the board input was a vector of 6 elements. The first input was the number of rows and columns with only one token of the player and no tokens of the opponent. The second input was the number of rows and columns with two tokens of the player and no tokens of the opponent. The third input was the same, but for three tokens of the player. The fourth, fifth, and sixth inputs were the same, but for the number opponent pieces. After around 30,000 games of self play, the agent had learned to play a descent game of tic-tac-toe. It learned how to force a win upon a mistake and how to force a draw otherwise.

The second game I implemented was checkers, and the code is located in the checkers directory. A traditional alpha-beta tree search AI and checkers engine written by Martin Fierz is included. My code uses Fierz's checkers engine to generate lists of legal moves, execute moves, and implement the rules of checkers. The agent can train against itself, but it can also train against Fierz's alpha-beta AI.

Again, the reward was +1 for a win, -1 for a loss, and 0 for all intermediate board positions. The only state representation I tried was raw input. The input vector was of length 32, and each input was -1 for an opponent king, -1/3 for an opponent man, 0 for a free square, 1/3 for a player's man, and +1 for a player's king.

After around 400,000 games of self play, the agent could play a decent game of checkers; I was unable to consistently win against the agent, although the agent was unable to beat the traditional alpha-beta AI. But starting again from random weights and training against the traditional AI for over 1,000,000 games, the agent performed much worse. During the training against itself each agent won on average half the games but while playing the traditional AI it lost

every game. Since the agent never won a single game, it was unable to learn “good” actions; it could only learn “bad” actions. Theoretically, the agent will converge to the good actions after it has exhausted all the bad actions and it starts winning games by random, but I was unable to take that much time.

A lot of interesting questions remain unanswered. Would the agent perform better if it first was trained through self play so it could get an idea about how to win and then played against the traditional AI? Would perhaps first learning against a weaker traditional AI player (say one that spent less time searching) help in the training? Is there a better representation for the state? What are the optimal values of  $\epsilon$ ,  $\lambda$ , and  $\alpha$ ? Since the training of the 400,000 games took over 18 hours, I was unable to explore and test many possible state representations and learning parameters. None of the trained agents were able to beat the traditional AI.

## References

- Levkovich, Chen and Hildeshaim, Tali and Levkovich-Frank, Orly. Temporal Difference Learning Project. [http://www.geocities.com/chen\\_levkovich/tdlearningproject.html](http://www.geocities.com/chen_levkovich/tdlearningproject.html)
- Olson, Daniel Kenneth (1993). "Learning to Play Games From Experience: An Application of Artificial Neural Networks and Temporal Difference Learning".  
<http://citeseer.nj.nec.com/olson93learning.html>
- Sutton, Richard and Barto, Andrew. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 1998. Available on line at  
<http://www-anw.cs.umass.edu/~rich/book/the-book.html>
- Tesauro, Gerald (1995). "Temporal Difference Learning and TD-Gammon". *Communications of the ACM*, March 1995 / Vol. 38, No. 3. Available on line at  
<http://www.research.ibm.com/massive/tdl.html>