Optimizing adaptive profiling in Alleria using Reinforcement Learning

Shreshth Tuli

Department of CSE, IIT Delhi

Abstract

Alleria is a new framework for generating instruction and detailed memory traces. It can be used by researchers to collect interesting information about one or more target applications. The earlier proposed heuristic based adaptive profiling mechanism has great performance improvement with respect to it's counterparts. This document throws light on one improvement that is made to Alleria using Reinforcement Learning algorithms. It makes Alleria intelligent in the sense that it can figure out by itself how many processing threads and buffers are required and dynamically adjust these parameters. This intelligence is provided using Q-Learning algorithm with ϵ-greedy approach.

1 Introduction : Q-Learning

Q-learning is a reinforcement learning algorithm that tries to find optimal actions by learning a state-action value function. The state-action value function, or simply $Q(s, a)$, is a table having rows as states, actions as columns, and values as entries. Values quantify the reward that the agent is expected to collect if it selects the corresponding action from the corresponding state and proceed from-there-on-out by following the optimal policy. Thus if the value function is known, then the optimal policy is simply to select the action having the highest value for the current state. The issue is that the Q-values are initially unknown, so they have to be learned in some way. Q-learning learns values by exploring the state-action space (literally just trying different action) and updating Q based on rewards that are measured after each action.

Thus, the two important components of Q-learning are:

1. Exploring state-action space.
2. Updating Q

Many flavors of algorithms have been devised to address these points; well focus on one of the simpler and more widely used ones.

1.1 Exploring state-action space

Exploring state-action space requires us to visit several state-actions and measure their instantaneous rewards, i.e., the reward obtained by performing action a from state s. Generally state-action space is huge, even for simple problems, so its necessary to focus exploration in promising regions. We use an ϵ-greedy approach where the optimal action $(arg\ max_a Q_t(s,a))$ is selected with probability $1-\epsilon$, while an arbitrary action is selected with probability ϵ. In addition, we apply a decay that decreases ϵ over time. This causes the agent to exploit information its already collected, instead of continuing to explore un-visited state-actions.

1.2 Updating Q

Q is updated according to the following equation:

$$Q_{t+1}(s,a) \ = \ Q_t(s,a) \ + \ \alpha(r \ + \ \gamma(\max_a Q_t(s',a) \ - \ Q_t(s,a))$$

s is the agents current state, a is the action it performs, r is the instantaneous reward, s' is the state the agent finds itself in after the action is performed, and γ and α are parameters.

The update equation is identical to an on-the-fly average $\bar{x_{t+1}} = \bar{x_t} + 1/t(x_t - \bar{x_t})$ where the sample is $r + \gamma \max_a Q_t(s',a)$. This sample combines the current reward r with the future reward $\max_a Q_t(s',a)$ to let information leak backward from s' to s. This combination allows strong instantaneous rewards to be lessened if the future state is bad, which is intuitively appropriate. The factor γ determines the influence of future values on current values, and α plays the role of a learning rate.

2 Implementation

The implementation of the model with ϵ-greedy algorithm has been done on C++.

The states include the possible pairs (θ_1, θ_2), where θ_1 denotes the number of thread counts and θ_2 denotes the number of buffers. The performance parameter denoted by P is used directly as reward.

The procedural approach can be translated into plain English steps as follows:

1. Initialize the Q-values table, Q(s, a).
2. Observe the current state, s.
3. Choose an action, a, for that state based on one of the action selection policies explained here on the previous page
4. Take the action, and observe the reward, r, as well as the new state, s'.
5. Update the Q-value for the state using the observed reward and the maximum reward possible for the next state. The updating is done according to the forumla and parameters described above.
6. Set the state to the new state, and repeat the process until a terminal state is reached.

Appendices

C++ Code

```
/*
Author: Shreshth Tuli
Date:   26/02/2018

This program functions as the "brain" of a reinforcement learning
agent whose goal is to provide the optimal values of number of buffers and
number of threads for efficient profiling.
For use in Alleria profiler code
*/

#include <iostream>
#include <random.h>

///////////////////////////////////////////////////////////////////////////////

//Computational parameters
float gamma = 0.75;      //look-ahead weight
float alpha = 0.2;       //"Forgetfulness" weight.  The closer this is to 1 the more weight is given to recent samples.
                         //A high value is kept because of a highly dynamic situation, we cannot keep it very high as then the system might not converge

//Parameters for getAction()
float epsilon;           //epsilon is the probability of choosing an action randomly.  1-epsilon is the probability of choosing the optimal action

//Variable 1 Parameters - Number of threads
const int numTheta1States = 10;
float theta1InitialCount = 1;
float theta1Max = 10;
float theta1Min = 1;
float deltaTheta1 = 1;
int s1 = int((theta1InitialCount - theta1Min)/deltaTheta1);
                 //This is an integer between zero and numTheta1States-1 used to index the state number of servo1

```

```
//Variable 2 Parameters - Number of Buffers
const int numTheta2States = 10;
float theta2InitialCount = 1;
float theta2Max = 10;
float theta2Min = 1;
float deltaTheta2 = 1;
int s2 = int((theta2InitialCount - theta2Min)/deltaTheta2);          //This is an integer between zero and numTheta2States-1 used to index the state number of servo2

//Initialize Q to zeros
const int numStates = numTheta1States*numTheta2States;
const int numActions = 5;
float Q[numStates][numActions];

//Initialize the state number. The state number is calculated using the theta1 state number and
//the theta2 state number.  This is the row index of the state in the matrix Q. Starts indexing at 0.
int s = int(s1*numTheta2States + s2);
int sPrime = s;

//Initialize vars for getDeltaDistance()
float distanceNew = 0.0;
float distanceOld = 0.0;
float deltaDistance = 0.0;

//These get used in the main loop
float r = 0.0;
float lookAheadValue = 0.0;
float sample = 0.0;
int a = 0;

//Returns an action 0, 1, ... 4 : NONE, theta1++, theta1--, theta2++, theta2--
int getAction(){
  int action;
  float valMax = -10000000.0;
  float val;
  int aMax;
  float randVal;
  int allowedActions[5] = {-1, -1, -1, -1, -1};  //-1 if action of the index takes you outside the state space.  +1 otherwise
  boolean randomActionFound = false;

```

```
//find the optimal action.  Exclude actions that take you
  outside the allowed-state space.
if((s1 + 1) != numTheta1States){
  allowedActions[0] = 1;
  val = Q[s][0];
  if(val > valMax){
    valMax = val;
    aMax = 0;
  }
}
if(s1 != 0){
  allowedActions[1] = 1;
  val = Q[s][1];
  if(val > valMax){
    valMax = val;
    aMax = 1;
  }
}
if((s2 + 1) != numTheta2States){
  allowedActions[2] = 1;
  val = Q[s][2];
  if(val > valMax){
    valMax = val;
    aMax = 2;
  }
}
if(s2 != 0){
  allowedActions[3] = 1;
  val = Q[s][3];
  if(val > valMax){
    valMax = val;
    aMax = 3;
  }
}

//implement epsilon greedy
randVal = float(random(0,101));
if(randVal < (1.0-epsilon)*100.0){    //choose the optimal
  action with probability 1-epsilon
  action = aMax;
}else{
  while(!randomActionFound){
    action = int(random(0,5));        //otherwise pick an
  action between 0 and 4 randomly (inclusive), but don't use
  actions that take you outside the state-space
```

```
      if(allowedActions[action] == 1){
        randomActionFound = true;
      }
    }
  }
  return(action);
}

//Given a and the global(s) find the next state.  Also keep
    track of the individual joint indexes s1 and s2.
void setSPrime(int action){
  if(action ==0){
    //NONE
    sPrime = s
  }
  else if (action == 1){
    //theta1++
    sPrime = s + numTheta2States;
    s1++;
  }else if (action == 2){
    //theta1--
    sPrime = s - numTheta2States;
    s1--;
  }else if (action == 3){
    //theta2++
    sPrime = s + 1;
    s2++;
  }else{
    //theta2--
    sPrime = s - 1;
    s2--;
  }
}


//Update the number of threads and buffers (this is the physical
     state transition command)
void setPhysicalState(int action){
  float currentCount;
  float finalCount;
  if (action == 1){
    currentCount = //theta 1 read
    finalCount = currentCount + deltaTheta1;
    //theta1 write finalCount
  }else if (action == 2){
```

```
    currentCount = //theta 1 read
    finalCount = currentCount - deltaTheta1;
    //theta1 write finalCount
  }else if (action == 3){
    currentCount = //theta2 read
    finalCount = currentCount + deltaTheta2;
    //theta2 write finalCount
  }else if(action == 4){
    currentCount = //theta2 read
    finalCount = currentCount - deltaTheta2;
    //theta2 write finalCount
  }
}


//Get the reward using the increase in performance since the
    last call
float getDeltaDistance(){
  //get current performance
  distanceNew = // read current performance
  deltaDistance = distanceNew - distanceOld;
  //if (abs(deltaDistance) < 57.0 || abs(deltaDistance) > 230.0)
    {          //don't count noise
  //  deltaDistance = 0.0;
  //}
  distanceOld = distanceNew;
  return deltaDistance;
}


//Get max over a' of Q(s',a'), but be careful not to look at
    actions which take the agent outside of the allowed state
    space
float getLookAhead(){
  float valMax = -100000.0;
  float val;
  if((s1 + 1) != numTheta1States){
    val = Q[sPrime][0];
    if(val > valMax){
      valMax = val;
    }
  }
  if(s1 != 0){
    val = Q[sPrime][1];
    if(val > valMax){
```

```
      valMax = val;
    }
  }
  if((s2 + 1) != numTheta2States){
    val = Q[sPrime][2];
    if(val > valMax){
      valMax = val;
    }
  }
  if(s2 != 0){
    val = Q[sPrime][3];
    if(val > valMax){
      valMax = val;
    }
  }
  return valMax;
}

void initializeQ(){
  for(int i=0; i<numStates; i++){
    for(int j=0; j<numActions; j++){
      Q[i][j] = 10.0;               //Initialize to a positive number to represent optimism over all state-actions
    }
  }
}

/////////////////////////////////////////////////////////////////

const int readDelay = 200;                      //allow time for the agent to execute after it sets its physical state
const float explorationMinutes = 1.0;         //the desired exploration time in minutes
const float explorationConst = (explorationMinutes*60.0)/((float)(readDelay))/1000.0);  //this is the approximate exploration time in units of number of times through the loop

int t = 0;
void main(){
  while (true){
    t++;
    epsilon = exp(-float(t)/explorationConst);
    a = getAction();           //a is beween 0 and 4
    setSPrime(a);              //this also updates s1 and s2.
    setPhysicalState(a);
```

```
    delay(readDelay);                              //put a delay after the physical action occurs so the agent has some delay between two performance reads
    r = getDeltaDistance();
    lookAheadValue = getLookAhead();
    sample = r + gamma*lookAheadValue;
    Q[s][a] = Q[s][a] + alpha*(sample - Q[s][a]);
    s = sPrime;

    if(t == 2){                   //need to reset Q at the beginning since a spurious value may arise at the first initialization
      initializeQ();
    }
  }
}
```

Listing 1: C++ code

References

[1] http://andraugust.com/rl/

[2] https://www.cse.unsw.edu.au/ cs9417ml/RL1/algorithms.html

[3] https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

[4] https://sites.ualberta.ca/ szepesva/papers/RLAlgsInMDPs.pdf

[5] https://en.wikipedia.org/wiki/Reinforcement_learning

[6] https://github.com/samindaa/RLLib