ECE 277 Final Project FA 22

Kwok Hung Ho A15151703

Parallelization of Q(λ) Learning

Introduction

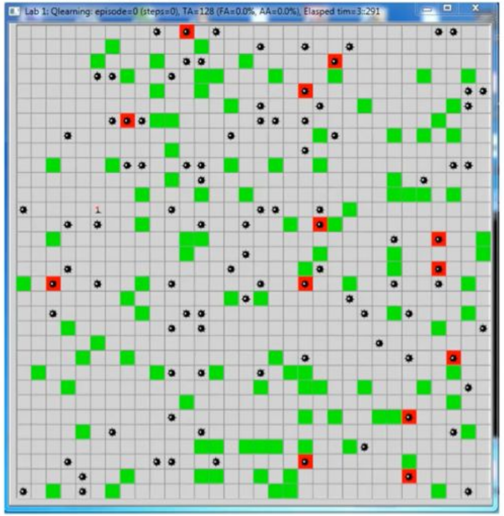
Motivation:

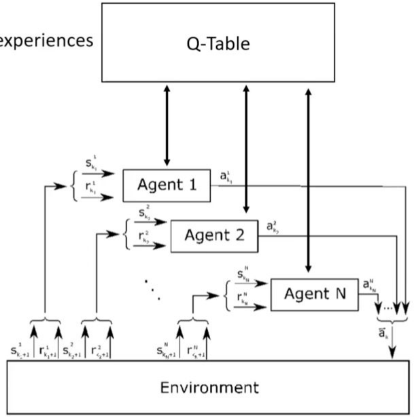
- Q(λ) is generally believed to outperform simple one-step Q-learning, since it uses single experiences to update evaluations of multiple state/action pairs (SAPs) that have occurred in the past. (Wiering, 1998)
- Goal is to implement **Asynchronous Multi-Agent Q(λ)-Learning**
- Compare performance with **Asynchronous Multi-Agent Q-Learning**

Background

- Lab 3 Extension

The number of agents = 128
Environment size = 32x32 (1024)
The number of mines = 96
The number of a flag = 1



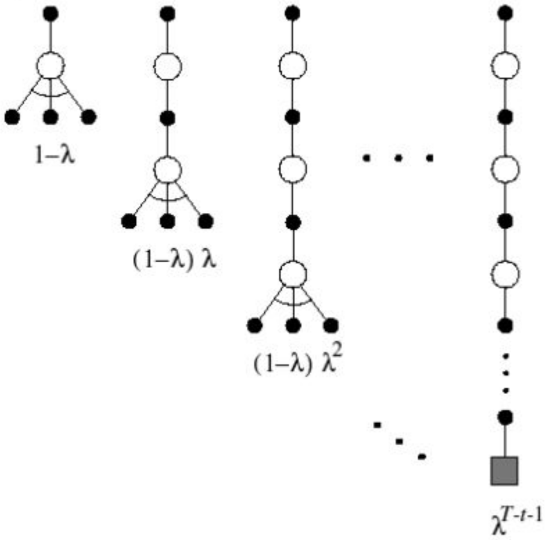



```
if (m_episode == 0 && m_steps==0) {// only for first episode
    env.reset(m_sid);
    agent_init(); // initQ table + self initialization
}else {
    active_agent = checkstatus(board, env.m_state, flag_agent);

    if (m_newepisode) {
        env.reset(m_sid);
        agent_init_episode(); // set all agents in active status
        float epsilon = agent_adjustepsilon(); // adjust epsilon
        m_steps = 0;
        printf("EP=%4d, eps=%4.3f\n", m_episode, epsilon);
        m_episode++;
    }else {
        short* action = agent_action(env.d_state[m_sid]);
        env.step(m_sid, action);
        agent_update(env.d_state[m_sid], env.d_state[m_sid ^ 1], env.d_reward);

        m_sid ^= 1;
        episode = m_episode;
        steps = m_steps;
    }}
m_steps++; env.render(board, m_sid); return m_newepisode;
```

Background: Types of Q(λ)

- Watson's Q(λ)
  - Early in learning, eligibility traces will be "cut"
  - Not much advantage of eligibility traces
- Peng's Q(λ)
  - Backup max action except at the end
  - Never cut traces
  - Difficult to implement
- Naive Q(λ)
  - Backup max at current action

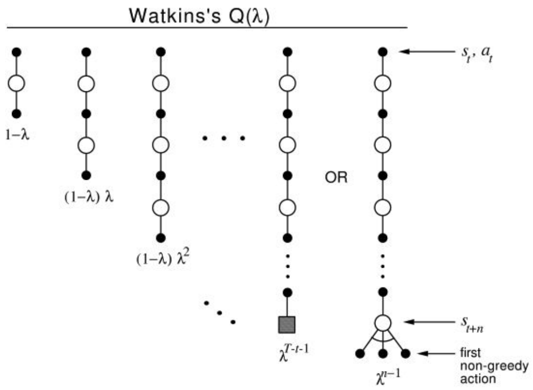
Peng's Q(λ)



Tabular version of Peng's Q(λ) algorithm

1. $Q(s,a) = 0$ and $e(s,a) = 0$ for all s, a
2. Do Forever:
   (a) $s_t \leftarrow$ the current state
   (b) Choose an action a_t according to current exploration policy
   (c) Carry out action a_t in the world. Let the short-term reward be r_t, and the new state s_{t+1}
   (d) $\delta'_t = r_t + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)$
   (e) $\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t)$
   (f) For each state-action pair (s,a) do
   $e(s,a) = \gamma\lambda e(s,a)$
   $Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e(s,a)$
   (g) $Q_{t+1}(s_t,a_t) = Q_{t+1}(s_t,a_t) + \alpha\delta'_t$
   $e(s_t,a_t) \leftarrow e(s_t,a_t) + 1$

Q-Learning VS Q(λ)-Learning



Q-Learning

- model-free, off-policy
- Single step update
- Special case of λ=0
- Parameters
  - α: learning rate
  - γ: discount factor

Q(λ)-Learning

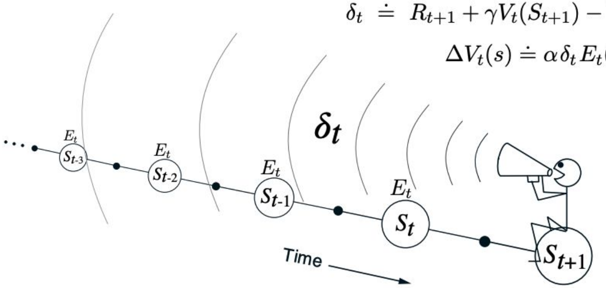
- model-free, off-policy
- Updates entire experience (with e-trace)
- Parameters
  - α: learning rate
  - γ: discount factor
  - λ: lambda

Background: Eligibility Traces

- When a TD error or terminal state occurs, eligibility traces keep responsible the states and actions that are for blame of the error. (Update the entire path)
- Different types of traces:
    - Accumulating, replacing, dutch, etc.
- ε-Greedy balances exploration and exploitation by choosing between exploration and exploitation randomly.

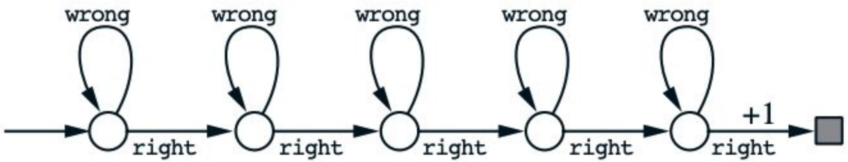
$$\delta_t \doteq R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)$$

$$\Delta V_t(s) \doteq \alpha \delta_t E_t(s)$$



- Shout δ_t backwards over time
- The strength of your voice decreases with temporal distance by $\gamma\lambda$

- Accumulating traces can do poorly on certain types of tasks



Q-Learning VS Q(λ)-Learning

Algorithm view

Initialize $Q(s,a) = 0, \forall s \in S, a \in A(s)$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode:)
    Choose A from current state S using policy derived from Q (e.g. ϵ-greedy)
    Take action A
    Observe next state S' and R
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$
  Until S is terminal

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $A^* \leftarrow \arg\max_a Q(S',a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
    $\delta \leftarrow R + \gamma Q(S',A^*) - Q(S,A)$
    $E(S,A) \leftarrow E(S,A) + 1$ (accumulating traces)
    or $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$ (dutch traces)
    or $E(S,A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
        else $E(s,a) \leftarrow 0$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Problem Statement

Critical Questions:

- Can we parallelize Q(λ)?
- How can we parallelize Eligibility Traces?
- What optimizations can we introduce?

Chosen Problem Statement:

- Implement Watson’s Q(λ)
- Use Replacing Traces

Strategy

Strategy: Single Agent

Key observations:

- We have to update Q with E using matrix addition
- Then we update the trace using current state
- Each time a non-greedy action happens, E <-0
- Otherwise, E is decayed by γλ

Matrix Addition

Matrix Set

    Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
    Repeat (for each episode):
        $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
        Initialize S, A
        Repeat (for each step of episode):
            Take action A, observe R, S'
            Choose A' from S' using policy derived from Q (e.g., ε-greedy)
            $A^* \leftarrow \arg\max_a Q(S',a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
            $\delta \leftarrow R + \gamma Q(S',A^*) - Q(S,A)$
            $E(S,A) \leftarrow E(S,A) + 1$ (accumulating traces)
            or $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$ (dutch traces)
            or $E(S,A) \leftarrow 1$ (replacing traces)
            For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
                $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
                If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
                                else $E(s,a) \leftarrow 0$
            $S \leftarrow S'$; $A \leftarrow A'$
        until S is terminal

Matrix Scaling

Strategy: Async Multi Agent

Key observations:

- We cannot simply have dim E = dim Q
- Each agent needs its own trace

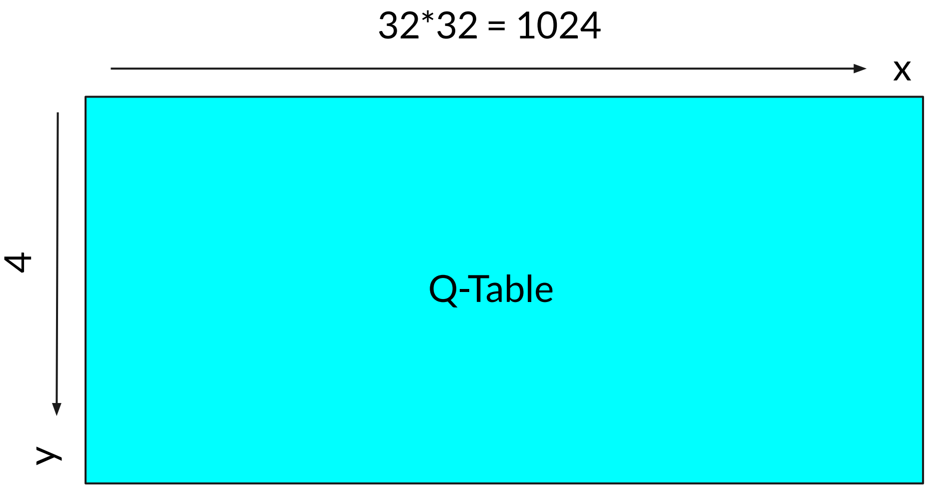
Matrix Addition

Matrix Set

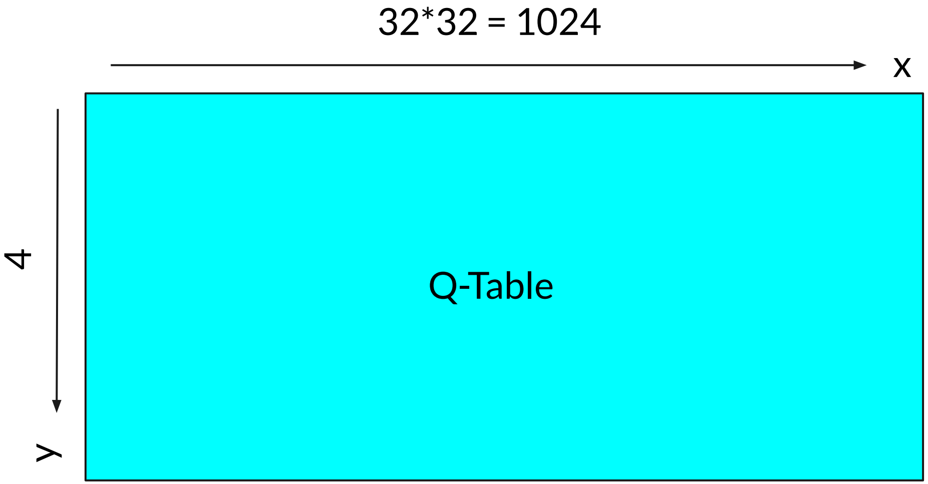
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
 $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
 Initialize S, A
 Repeat (for each step of episode):
  Take action A, observe R, S'
  Choose A' from S' using policy derived from Q (e.g., ε-greedy)
  $A^* \leftarrow \arg\max_a Q(S',a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
  $\delta \leftarrow R + \gamma Q(S', A^*) - Q(S,A)$
  $E(S,A) \leftarrow E(S,A) + 1$ (accumulating traces)
  or $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$ (dutch traces)
  or $E(S,A) \leftarrow 1$ (replacing traces)
  For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
   $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
   If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
     else $E(s,a) \leftarrow 0$
  $S \leftarrow S'$; $A \leftarrow A'$
 until S is terminal

Matrix Scaling

Strategy: Q-Table



Strategy: Q-Table

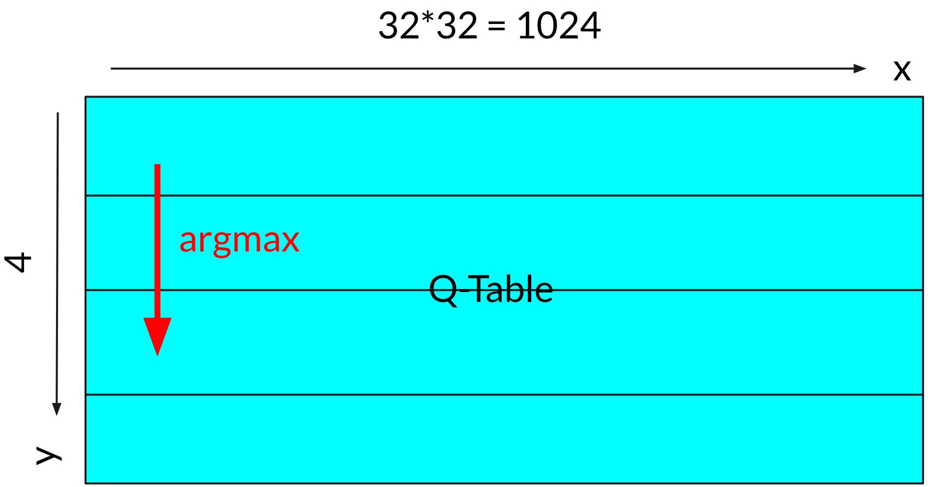



```
// Function to initialize Q parallelly
__global__ void initQTable(float *Q_Table, int nx,int ny)
{
    unsigned int ix = threadIdx.x;
    unsigned int iy = blockIdx.x;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
        Q_Table[idx] = 0;
}
```

```
initQTable <<< 4, 1024 >>>(Q_Table,1024,4);
```

Using 8 Warps (1024/32 = 8)

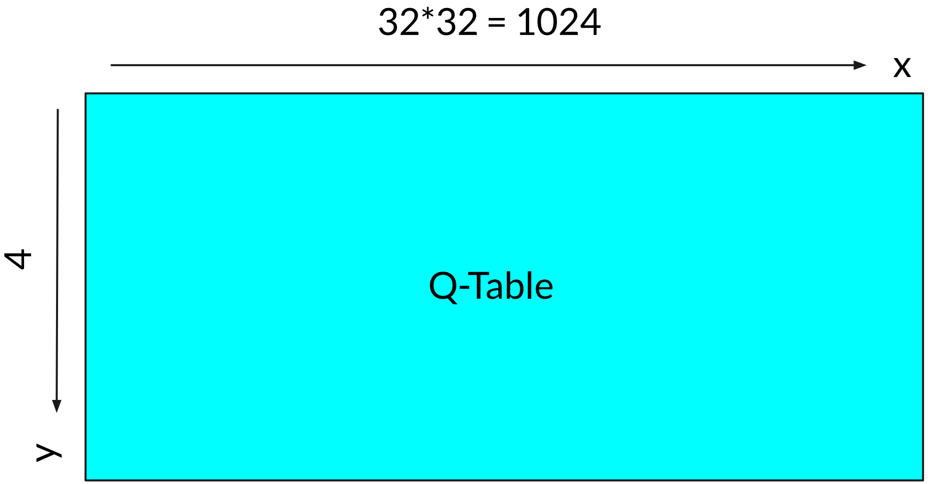
Strategy: Q-Table

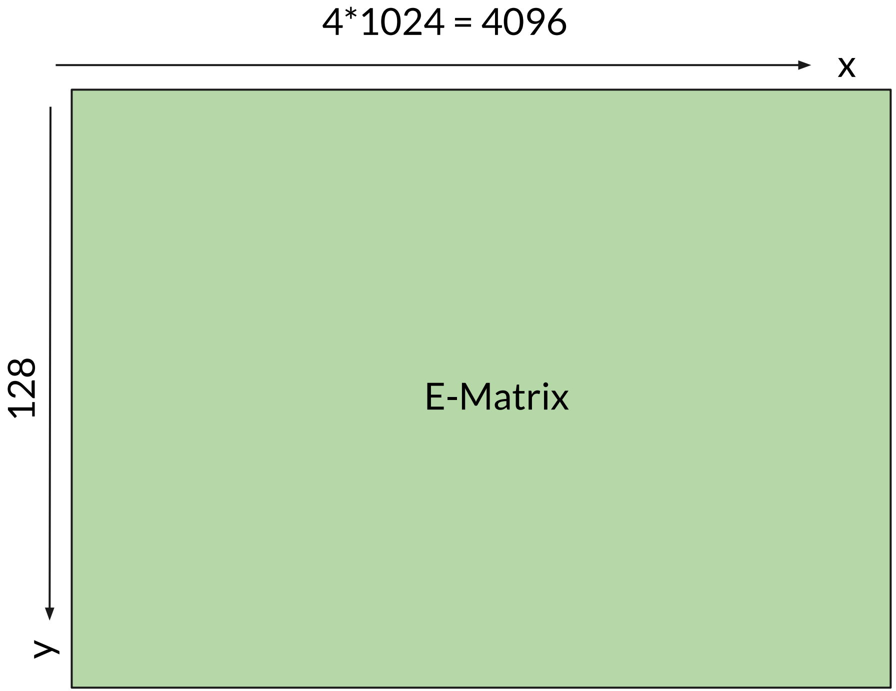


$$A^* \leftarrow \arg\max_a Q(S', a) \text{ (if } A' \text{ ties for the max, then } A^* \leftarrow A')$$

```
// getting argmax & max
float mx = Q[xn];
short argmax = 0;
for (short i = 0; i < 4; i++) {
    int nidx = xn + i*(32*32); // argmax action for a given next State
    if (Q[nidx]>mx) {
        mx = Q[nidx];
        argmax = i;
    }
}
```

Strategy: E-Matrix

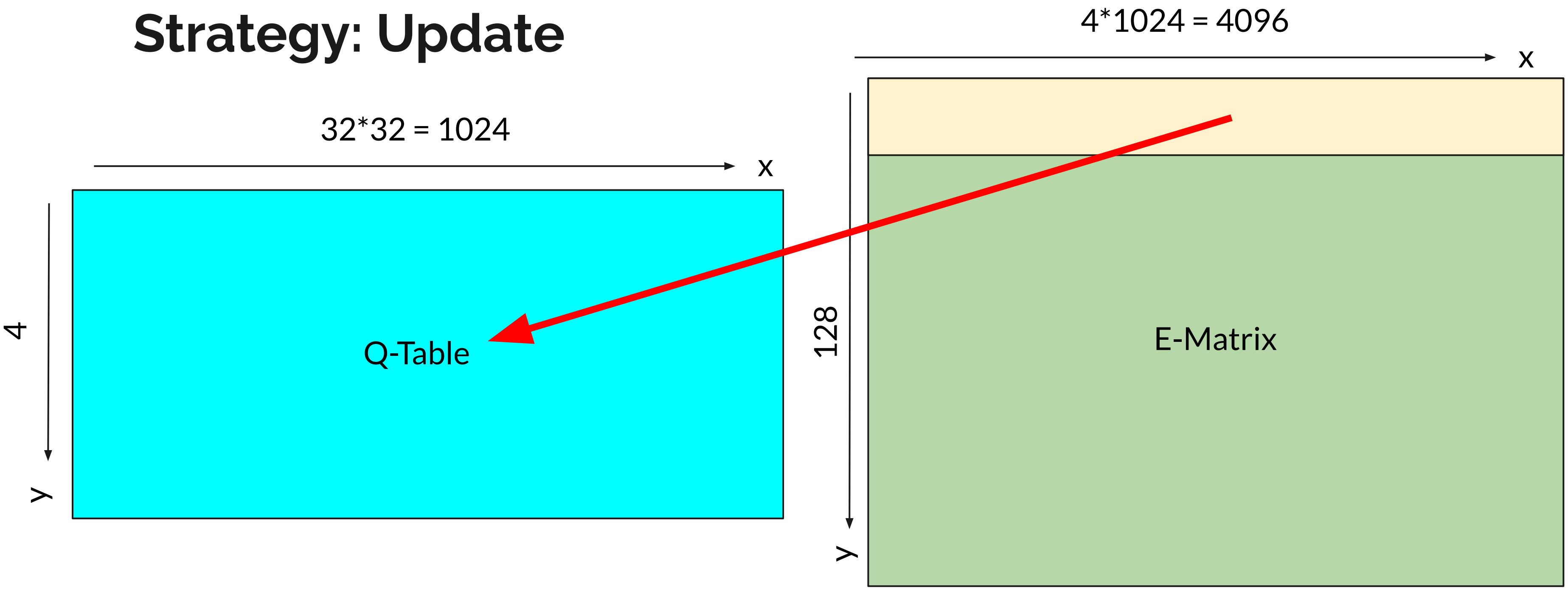




Strategy: Update

For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$$



Problem

- E Matrix ends up being large: 128*4096 = 524,288
- E is sparse, so most of the threads are adding 0

Strategy: E-trace Struct

- We can use an Array of Structure (AoS)
- Can update only traces that are non-zero
  - By storing trace length, we reduce time complexity from O(N) to O(M)
  - Where N = stateAction space * numAgents, and M = traceLength*numAgents

```
struct eligibilityTraces {
    int trace_length = 0; // length of trace
    int x[COLS * ROWS]; // history of state.x
    int y[COLS * ROWS]; // history of state.y
    int past_actions[COLS * ROWS]; // history of actions
    float E[COLS * ROWS]; // e-Trace value
};
```

Strategy: E-trace Struct

```
struct eligibilityTraces {
    int trace_length = 0; // length of trace
    int x[COLS * ROWS]; // history of state.x
    int y[COLS * ROWS]; // history of state.y
    int past_actions[COLS * ROWS]; // history of actions
    float E[COLS * ROWS]; // e-Trace value
};
```

- Create AoS etrace[128]
- For an agent: etrace[agent_id]
  - Using the variable **trace_length**, we loop backward from i = **trace_length->0**
    - We can find all our past state/actions using **x,y, past_actions**
    - **qid <- past_actions*state_space + y*num_columns + x**
    - Then we can update **Q** at each qid
    - Then update traces with $\gamma\lambda$ (shift array)
  - If exploratory action was taken, trace_length=0

Strategy: Reward update variants

- Mine (-1)
  - Backup/Not backup
- Flag (+1)
  - Always backup
- Nothing (+0)
  - Never backup

Strategy: Optimization

- Reduce ALU operations in Kernel
  - Avoid (+,-,*,/) when possible (hardcode values)
- Use golden rule: 2-8 warps per thread block
- Coalesced access with SMEM
- syncthreads() to avoid race conditions
- Use SMEM and shuffle operations for argmax
- #pragma unroll
- Declare variable type to avoid type conversion overhead
- Code structure in if statements

```
for ( int i = 0; i < 5; i++ )
    b[i] = i;
```

```
b[0] = 0;
b[1] = 1;
b[2] = 2;
b[3] = 3;
b[4] = 4;
```

Strategy: Optimization

- **Reduce ALU operations in Kernel**
  - **Avoid (+,-,*,/) when possible (hardcode values)**
- **Use golden rule: 2-8 warps per thread block**
- Coalesced access with SMEM
- syncthreads() to avoid race conditions
- Use SMEM and shuffle operations for argmax
- **#pragma unroll**
- **Declare variable type to avoid type conversion overhead**
- **Code structure in if statements**

Bold ones are implemented

Code Walkthrough

Demo

Results & Analysis

Performance Metrics

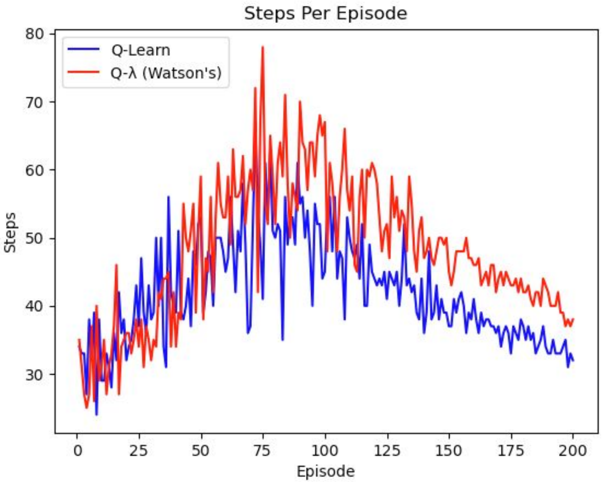
- CUDA Occupancy Nsight Compute
  - Cannot attach Nsight Compute Profiler (No .exe built)
- CUDA Occupancy Calculator
  - Profiler in VS deprecated
- Kernel execution time
- Steps per episode

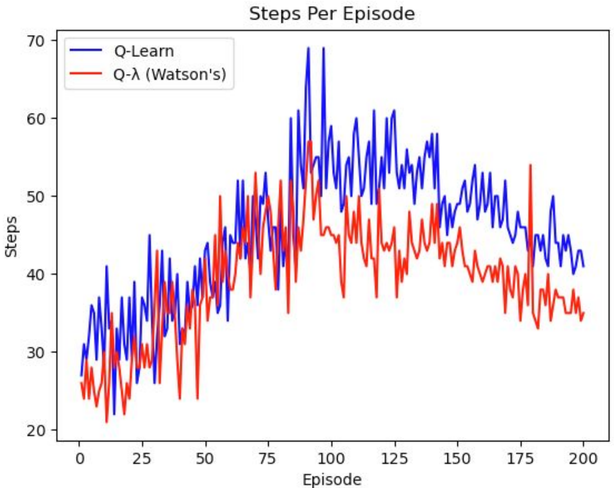
Attribute	NVIDIA RTX A2000 Compute Capability: 8.6 Driver Model: WDDM
MEMPOOL_SUPPORTED_HANDLE_TYPES	0
MULTI_GPU_BOARD	0
MULTI_GPU_BOARD_GROUP_ID	0
MULTIPROCESSOR_COUNT	26
PAGEABLE_MEMORY_ACCESS	0
PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES	0
PCI_BUS_ID	1
PCI_DEVICE_ID	0
PCI_DOMAIN_ID	0
RAM_LOCATION	1
RAM_TYPE	17
RESERVED_SHARED_MEMORY_PER_BLOCK	1024
SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO	32
SPARSE_CUDA_ARRAY_SUPPORTED	1
STREAM_PRIORITIES_SUPPORTED	1
SURFACE_ALIGNMENT	512
TCC_DRIVER	0
TEXTURE_ALIGNMENT	512
TEXTURE_PITCH_ALIGNMENT	32
TOTAL_CONSTANT_MEMORY	65536
TOTAL_MEMORY	6435635200
UNIFIED_ADDRESSING	1
VIRTUAL_ADDRESS_MANAGEMENT_SUPPORTED	1
WARP_SIZE	32

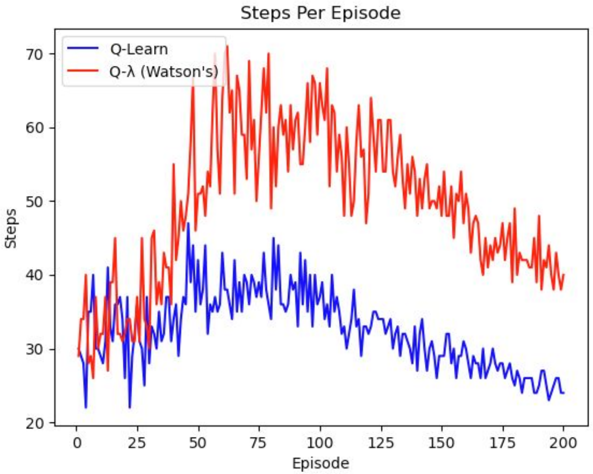
Performance: Steps/Episode

- With Δε=0.005, we have 200 episodes
- steps/episode
- Saved #steps into array[200]

	Test 1	Test 2	Test 3
Q	8424	9062	6518
Q(λ)	9589	7857	9889

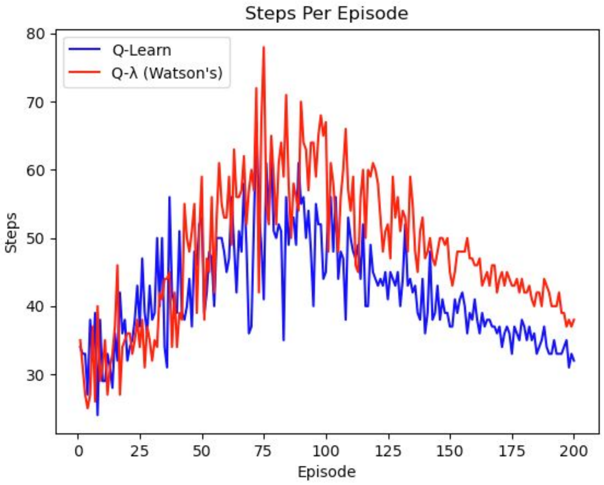


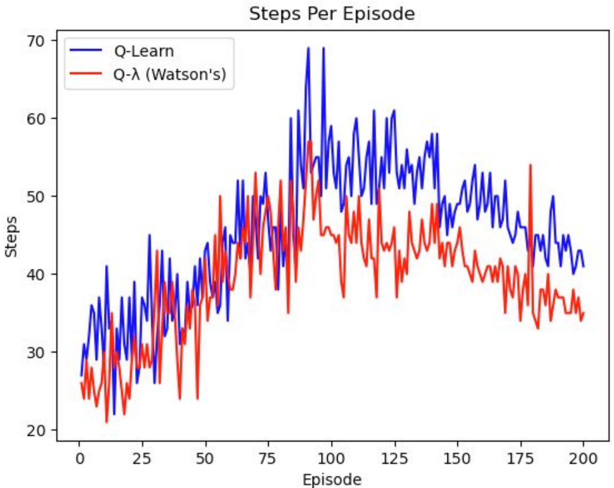


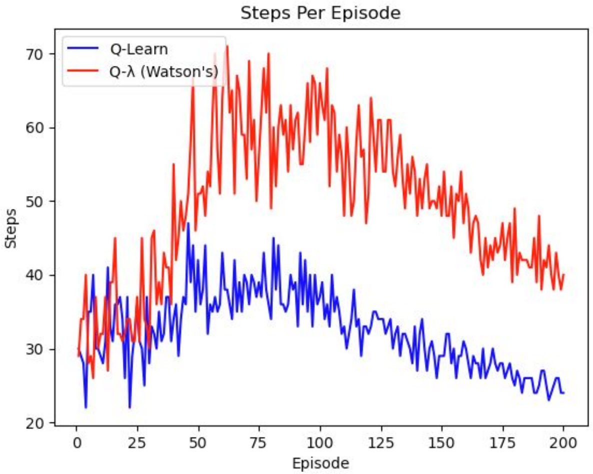


Analysis: Steps/Episode

- Due to random board generation, fluctuations are too large (spawn dependent)
- No conclusions can be made







Performance: Kernel Execution Time

- CPU function: **agent_update**
- Kernel function tested: **updateQ**
- Use cuda events
- A running average is kept:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
updateQ <<< 1, 128 >>>(cstate, nstate, rewards, d_action, Q_Tabl
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float kernel_time;
cudaEventElapsedTime(&kernel_time, start, stop);

runningsum += kernel_time;
total_steps++;
avg_kernel_time = runningsum / total_steps;
printf("kernel exec time \t\t\t: %f ms\n", kernel_time);
printf("kernel average time \t\t\t: %f ms\n", avg_kernel_time);
```

```
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.005120 ms
ernel average time                 : 0.006481 ms
Episode=   73, epsilon=0.630                    ;
ernel exec time                    : 0.007168 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006480 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006480 ms
ernel exec time                    : 0.006208 ms
ernel average time                 : 0.006480 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006480 ms
ernel exec time                    : 0.006144 ms
ernel average time                 : 0.006480 ms
ernel exec time                    : 0.008192 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.007168 ms
ernel average time                 : 0.006481 ms
ernel exec time                    : 0.007168 ms
ernel average time                 : 0.006481 ms
```

Analysis: Kernel Execution Time

- These values are all the converged values after >200 episodes
- Significant difference in Kernel times ~0.001ms
    - Expected behavior
- Also somewhat board dependent
    - (Fluctuation of 0.0003)
    - Cannot tell effect of optimizations

```
kernel exec time                                  : 0.004192 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
Episode=  73, epsilon=0.630
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005248 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005248 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
kernel exec time                                  : 0.005120 ms
kernel average time                               : 0.005155 ms
```

Algorithm	Kernel avg exec time
Q(λ)	0.006718 ms
Q(λ) Optimized	0.006447 ms
Q-Learning	0.005112 ms

Space For Improvement

- Parameter adjusting
  - Adjust lambda for faster convergence
  - Adjust learning rate, discount factor
  - Dynamic epsilon update based on episode ($\varepsilon = 1/i$) where i is the ith episode
- SMEM caching for argmax
  - Use a SMEM cache of size 4: **__shared__ float cache[4]**
- Create own environment
- Implement Peng's Q(λ)

Results & Conclusion

- Watson’s Q(λ) successfully implemented
- Watson’s Q(λ) does not indicate improvement in steps/episode
    - Probably due to eligibility traces zeroing a lot early during exploration (Sutton & Barto)
- Watson’s Q(λ)-learning has higher kernel execution time than Q-learning
- Although Watson’s Q(λ) was did not show performance boost, eligibility traces can be extended to algorithms like SARSA(λ) and TD(λ), which can benefit much more from eligibility traces by being on-policy