Name: Hazem Taha ID:400550393

ECE718 Summer-2024

Project Report

Introduction

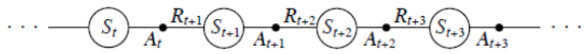
Q-Learning Algorithm:

Q-learning is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for a given finite Markov decision process. The algorithm works by learning a Q-value, which represents the expected accumulative rewards of taking a given action in a given state and following the optimal policy thereafter. The Q-value is updated iteratively using the Bellman equation, which considers the immediate reward from the current action and the estimated optimal future rewards. Over time, Q-learning converges to an optimal policy, enabling an agent to make decisions that maximize cumulative rewards in an environment.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

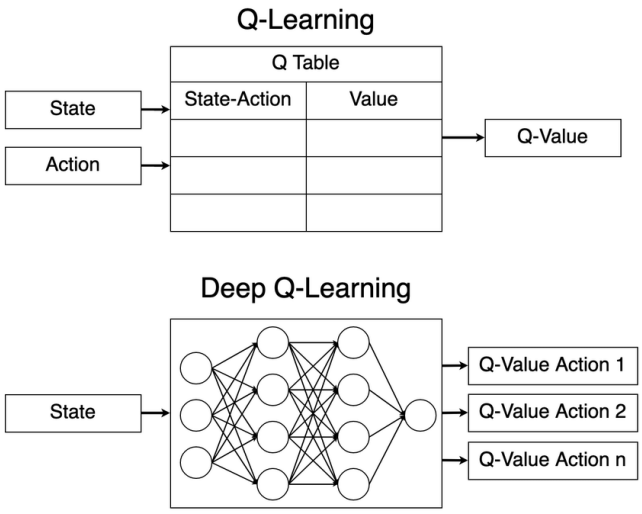
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal



Deep Q-Learning (DQL):

Deep Q-learning is an extension of the Q-learning algorithm that leverages deep neural networks to handle environments with high-dimensional state spaces, where traditional Q-learning would struggle. Instead of maintaining a table of Q-values for every state-action pair, Deep Q-learning uses a neural network to approximate the Q-value function, mapping states to Q-values for each possible action. This approach enables the algorithm to generalize across similar states, making it feasible to learn policies in complex environments like video games or robotics.

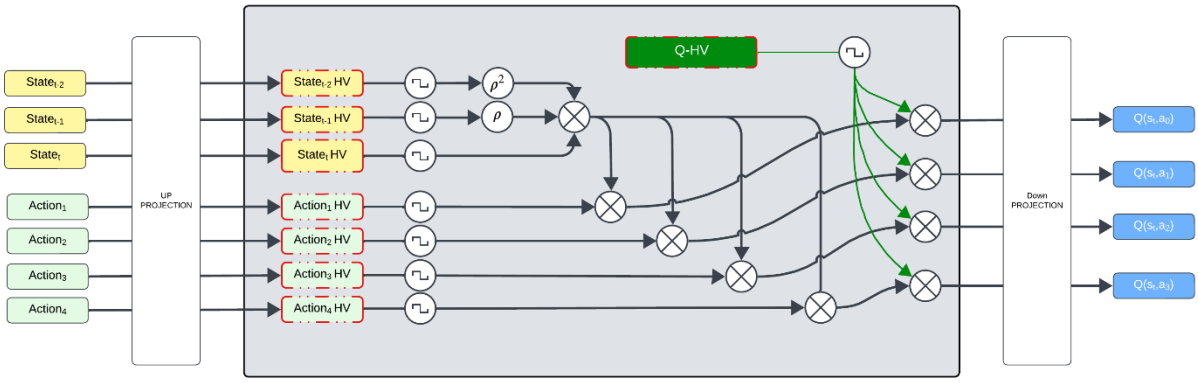
In my implementation, I choose a slightly “decorated” DQL with an additional technique (experience replay) where the agent stores past experiences and randomly samples a batch of timesteps in every training loop to break the correlation between consecutive learning steps. There are also a few other additions that can be don but I currently sticked with that.



HDC based Q-Learning:

First approach:

The original approach I followed is explained in the below figure.

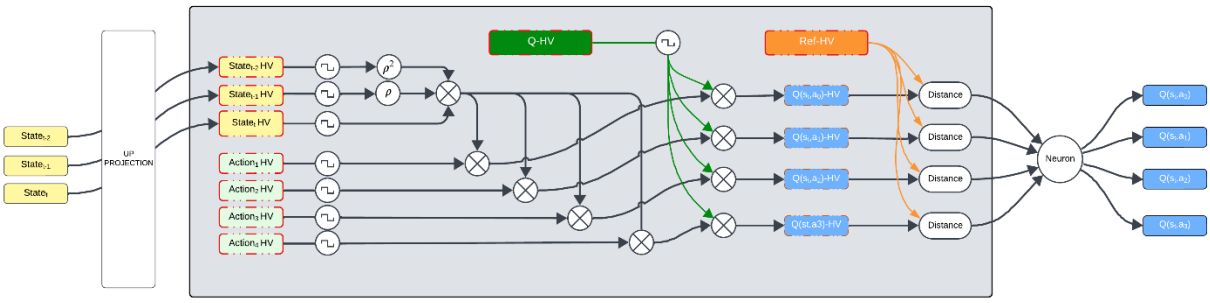


Where the algorithm goes as follows:

- Up-project a collection of states and a set of one-hot encoded actions.
- Stack the states by permuting and binding them.
- Bind each action with the stacked state Hypervector to get a set of (s, a) bounds.
- Unbind each state-action pair from a trainable Q Hypervector (representing the Q-table) to get a set of Q(s,a) hypervectos.
- Down project the hypervectors to scalar values, use them in the loss calculation like as before with the DNN, and back propagate through the Q-Hypervector to train it.

The issue that I encountered in this implementation is that down projection from a HD vector to a single scalar causes a significant loss of information and prevents successful training (although I noticed increasing the hypervector size tries to mitigate this but it's a kill in time and resources).

Therefore, another approach that I thought about was to compose the answer directly through interactions in the HD domain just as we did with calculating the similarities in Hrrformer. I will calculate the distance form my Q(s,a) hypervector to a fixed reference hypervector and scale and shift this distance to be on my original scale. As I don't originally know the original scale relative to the scale of in the HDC domain, I will train a simple neural network to know the scale and shifting needed. The algorithm is shown here:



Where the algorithm goes as follows:

- Up-project a collection of states.
- Randomly generate a set of action hypervectors.
- Stack the states by permuting and binding them.
- Bind each action with the stacked state Hypervector to get a set of (s, a) bounds.
- Unbind each state-action pair from a trainable Q Hypervector (representing the Q-table) to get a set of Q(s,a) hypervectors.
- Calculate the distance between each Q(s,a) hypervector and the chosen reference vector. In my implementation I took an action hypervector as the reference vector because it doesn't change throughout the iterations.
- Pass the distances through the single neuron, use the results in the loss calculation like as before with the DNN, and back propagate through the neuron and the Q-Hypervector to train them.

Software results:

The baseline implementation with 3 feedforward layers of sizes (64, 128, 64) solved the environed in ~ 400 episodes. By solving the environment, I mean it trained well enough to exceed a certain cumulative reward. In the FHRR implementation, the environment was

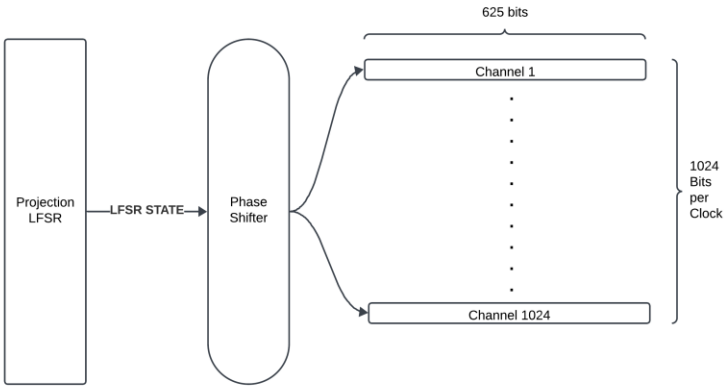
solved in ~ 900 episodes with hypervectors dimensions of 10,000. Although it took more iterations. This approach is more mathematically efficient and scalable than deep neural networks.

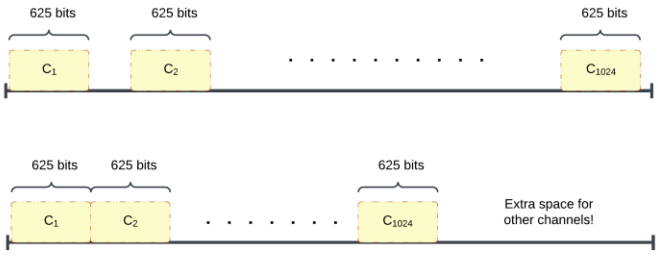
Hardware implementation

Up-projection Matrix

While designing the random number generator for the up-porjction matrix $\boldsymbol{A_{UP}}$. I planned to process a collection of 16 hypervector elements at once to paralyze the computations. Hence, I designed an LFSR of size 20 bits that generates 1024 bits per clock and use them to compose 128x8-bit elements filling 16 columns of the up-projection matrix. Therefore, I can generate 16 elements of the state hypervector per clock. Notice that I need to keep generating for 625 clocks to traverse the full matrix.

$$\boldsymbol{A_{UP}} = \begin{bmatrix} \boldsymbol{A_1^1} & \cdots & \boldsymbol{A_1^{10k}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{A_8^1} & \cdots & \boldsymbol{A_8^{10k}} \end{bmatrix}$$





It can be noted that there is enough space in the spectrum to generate more bits and possibly accommodate the whole action hypervectors matrix generation, but this greatly affect the compilation time and will limit the pipeline flexibility later.

Name: Hazem Taha ID:400550393

Using a simulation script given by the professor, I extracted the transformation matrix that can shift any current LFSR state one channel width forward. I then used it iteratively (in a combinational logic) to project the current state into all the channels, counting on the compiler to optimize the design and give me a more efficient circuit.

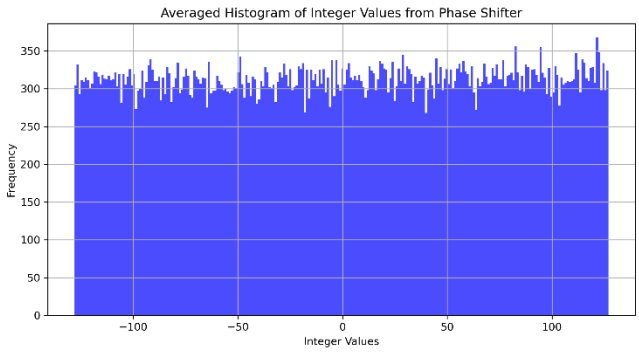
To see the effect of changing the channel separation on the randomness of the assembled 8-bit fixed points numbers, I did a couple of experiments using a slightly altered version of the code you shared with me.

First, I tried to set the channel separation to the minimum (625) and assemble the numbers by taking bits of consecutive channels. The statistical results of the experiments are presented in the below table:

Channel Separation	Assemble Method	Minimum Frequency	Maximum Frequency	Mean Frequency	Median Frequency	STD of Frequencies
625	Consecutive	268.9	364.5	312.50	312.25	17.08
625	Separated	262.3	368.2	312.50	311.35	17.44
1000	Consecutive	268.5	356.9	312.50	312.25	16.98
1000	Separated	261.9	360.1	312.50	312.25	17.25

Note that consecutive means that I assemble the numbers by taking bits of consecutive channels and separated means that I assemble the numbers by taking bits from channels that are 8-channels apart.

It was noted that there isn't a big difference, and all experiments showed slightly uniform distribution of values (like the graph below). Therefore, I adapted the consecutive approach with 625 channel separation (channels are tightly close to each other).



Action Hypervectors Matrix:

I used another LFSR of the same size to generate the actions hypervectors, the second LFSR to generate 512 bits per clock and used them to compose 64x8-bit elements filling 16 columns of the matrix. Therefore, I need to keep generating 625 clocks to traverse the full matrix.

$$\mathbf{Actions} = \begin{bmatrix} \mathbf{Action}_1^1 & \cdots & \mathbf{Action}_1^{10k} \\ \vdots & \ddots & \vdots \\ \mathbf{Action}_4^1 & \cdots & \mathbf{Action}_4^{10k} \end{bmatrix}$$

Both LFSRs are being controlled by two different initiate signals, giving more flexibility in case of pipelining instead of buffering the generated data.

Hardware Results:

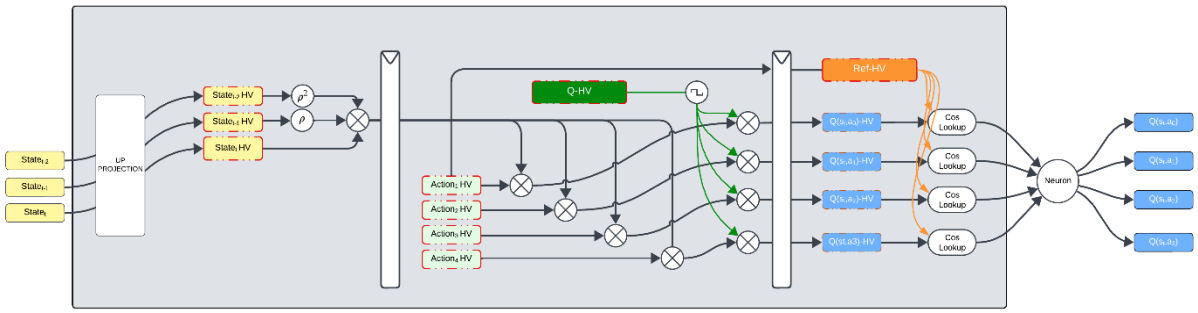
First Compilation (Single Cycle):

Compile time 0:22:04
Total logic elements 22,220 / 114,480 (19 %)
Total registers 133
Total pins 299 / 529 (57 %)
Total memory bits 131,072 / 3,981,312 (3 %)
Embedded Multiplier 9-bit elements 128 / 532 (24 %)
Maximum operating frequency 24.97 MHz

It can be noticed that we are barley using registers and that we use large amounts of logical elements. It's also clear that the system needs to be pipelined to increase the frequency, however, we must be careful in designing the pipeline as every stage might needs to buffer a large amount of data with it.

Compilation of 2-stage Pipelined System:

After applying 2 stages of pipelining to the system as shown in the below figure, the results were as follows:



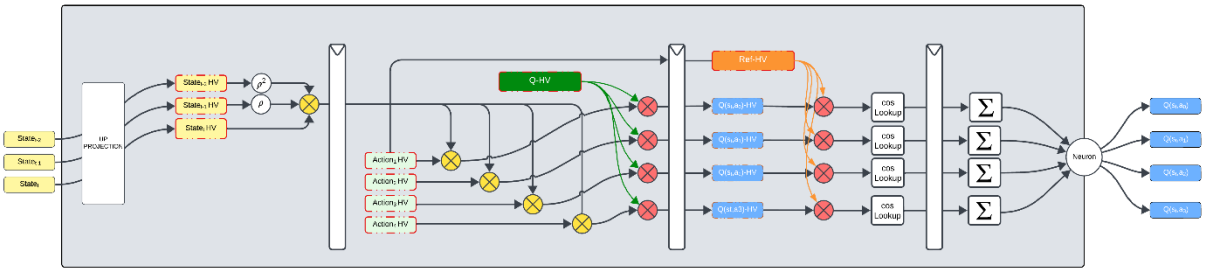
Total logic elements 20,752 / 114,480 (18 %)
Total registers 936
Total pins 299 / 529 (57 %)
Total memory bits 131,072 / 3,981,312 (3 %)

Embedded Multiplier 9-bit elements 128 / 532 (24 %)
Maximum operating frequency 51.06 MHz

The frequency increased but on the account of 9x increase in the used registers. The frequency is not above 50 MHz but there is still a setup time violation at the last stage. This can be explained by the fact that calculating the distance is a multi-step process with one of them being an adder tree that adds 16x8-bit elements together. This process should be broken down into two steps.

Compilation of 3-stage Pipelined System:

Finally, the below figure shows a proposed 3-stage pipelined system that should breakdown the distance calculation.



The results of the implementation were as follows:

Compile time 00:11:44
Total logic elements 21,355 / 114,480 (19 %)
Total registers 1385
Total pins 299 / 529 (57 %)
Total memory bits 131,072 / 3,981,312 (3 %)
Embedded Multiplier 9-bit elements 128 / 532 (24 %)
Maximum operating frequency 76.08 MHz

The use of a large number of registers can be justified due to the large amounts of data that needs to be buffered between different stages in HDC. These number of logic elements is relatively reasonable considering the number of operations we are doing between hypervectors, the number of logic elements used in the LFSRs specifically indicates that the compiler have found a way to optimize and reduce the deep computational path that project the state into 1024 channels.