Parallel Gauss Seidel Algorithm on Distributed Memory Architecture

Project - High Performance Computing at EPFL

Olagoke Lukman O.
lukman.olagoke@epfl.ch
EPFL

***Abstract*—The aim of this project is to present parallel implementation of the Gauss-seidel(GS) iterative algorithm for the solution of linear equations using MPI as a parallel programming paradigm.**

I. Introduction

A. Reinforcement Learning

Reinforcement Learning is a part of the Machine Learning domain but also of the domain of Artificial Intelligence. The goal of the agents is to maximize its reward in a specific environment by acting correctly. That means an agent has a set of possible actions for a set of states and and it has to choose the best combination in order to have the highest reward at the end.

We will use a neural network with a simple structure of one input layer and one output layer. The input layer is a grid of size 20 by 20 and the neurone activity is equal to

$$r_j(s) = exp(-\frac{(x_j - x)^2}{\sigma_x^2} - \frac{(\psi_j - \dot{x})^2}{\sigma_\psi^2})$$

where $s = (x, \dot{x})$ is the state of the car and x_j and ψ_j the state of the neurone in the grid.

Algorithm 1 $SARSA(\lambda)$ Algorithm

1: **Initialize**
2: **Use policy to select action**

$$P(a^* = a) = \frac{exp(Q(s,a)/\tau)}{\Sigma_{a'} exp(Q(s,a')/\tau)}$$

3: **Observe** R**,** s'**, choose next action** a' **according to policy**
4: **Calculate TD error in Sarsa**

$$\delta_t = R_{t+1} - [Q_a(s,a) - \gamma Q_{a'}(s',a')]$$

5: **Update eligibility trace**

$$e_{aj}(t) = \gamma\lambda e_{aj}(t - \Delta t) + \begin{cases} r_j & if a = \text{action taken} \\ 0 & otherwise \end{cases}$$

6: **Update weights**

$$\Delta w_{aj} = \eta \delta_t e_{aj}$$

7: **Old** a' **becomes** a**, old** s' **becomes** s **and return to 2**

This algorithm is an extension of the basic $SARSA$ that you can get by setting $\lambda = 0$. You can see the usage of an eligibility trace. The policy we are using is a soft-max solution that assigns a probability to the actions and pick one according to this distribution.

B. Eligibility Trace

The principle of the eligibility traces is to give more weight according to the occurrence of a state-action pair. In other words we want to update the state-action pairs that have been visited by the agent. In our case, we add the neurone activity to the eligibility trace only if the action of the activity is the one taken by the agent else the trace remains the same.

C. Implementation

The code is in Python and we use the multiprocessing module to compute in parallel different agents in order to decrease drastically the time require for each experiment as each trial finished only when the agent has reached the goal or the maximum number of steps (20'000 steps). Using a pool of processes we can limit the number of parallel computation to avoid freezing the computer.

The command line arguments allow you to choose which experiment you want. Using 0 will launch the pool of agents and show the learning curve at the end. The option 1 will launch only one agent and show you the vector field after the trials.

II. Experiments

A. Basic Simulations

The Figure 1 show us 20 agents trained over 50 trials. The parameters are $\eta = 0.001$, $\gamma = 0.95$, $\lambda = 0.5$, $\tau = 1$ and the size of the neural network is 20x20. The blue line is the average value of the time needed to reach the hill and the green is a moving average of window size of 7. As expected, we can see that the agents learn how to win with less steps as at the beginning they need around 5500 seconds and after 50 trials they need around 2500 seconds in average.

B. Vector Field

For the next experiment, we computed a vector field that represents the decision with the highest Q-value. The expected behaviour should be to apply the force in the direction of the velocity until the car stops to apply the force in the opposite direction to gain the maximum velocity we can.

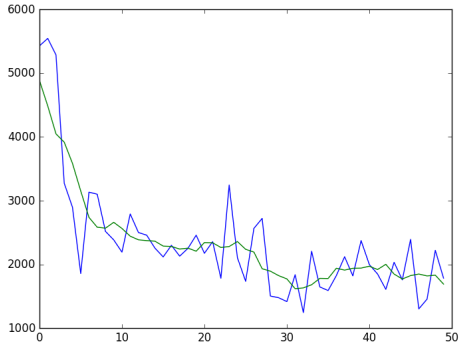
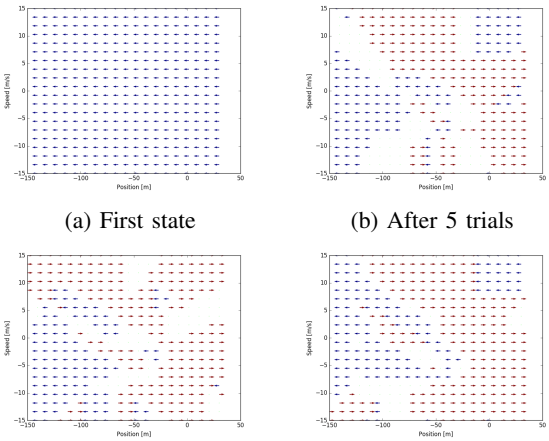


Fig. 1: Basic simulation with 20 agents and 50 trials



(a) First state (b) After 5 trials

(c) After 25 trials (d) After 50 trials

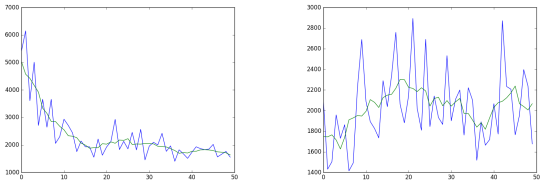
Fig. 2: Evolution of the vector field

The weights have been initialized to 0 to study the evolution of the vector field from a known state. For the other parameters, we have $\eta = 0.001$, $\gamma = 0.95$, $\lambda = 0.5$ and $\tau = 1$. In the Figure 2, you can see the evolution with a negative force in blue and a positive in red. We observe that for most of the regions, the agent would take the right decision to gain in velocity but the area where the car has a negative velocity and near the win point should be in blue to increase the velocity. We can explain that by the fact that the car would try to reach the hill even if it doesn't have enough energy.

C. Exploration Temperature parameter

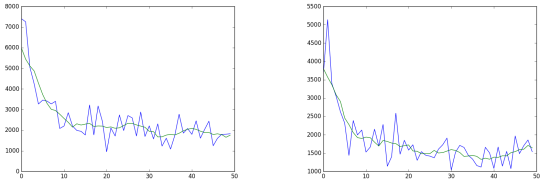
If we set the exploration temperature parameter to 0 or ∞, we get the uniform distribution over a space of size 3 as $Q(s,a)/\tau$ will tend to 0 or ∞ for every action. That means if we have a big value we increase the exploration policy against the exploitation as we will try sub-optimal actions to discover new features.

Keeping this in mind, we want to try to discover at the beginning of the trials and then improve the features we found with a lower exploitation policy.



(a) τ decaying over time (b) $\tau = \infty$

Fig. 3: Experiments with different values for the exploration temperature parameter



(a) $\lambda = 0$ (b) $\lambda = 0.95$

Fig. 4: Experiments with different values for the eligibility trace parameter

The Figure 3 demonstrates that varying the value of the exploration temperature parameter can be a good idea as the learning curve is slightly faster to converge than the basic simulation. On the contrary and as expected, using a very tiny or big value will force the agent to explore at every iterations as the probability distribution to choose an action is uniform. The parameters used are $\eta = 0.001$, $\gamma = 0.95$ and $\lambda = 0.5$.

D. Eligibility Trace Decay Rate

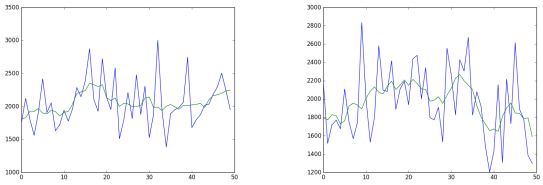
The definition of the algorithm tells us that if we set the λ parameter to zero, we will update only the weights of the chosen action with the δ_t value and the activity of the neural network r_j. Therefor we don't expect a better result but if $\lambda > 0$ that means we will set a short-memory on the trace to keep track of the visited state-action pair to increase those with a high number of visits by the agent.

As you can see in the Figure 4 the learning curve is better with a value near one as expected by the analysis. The experiment used the parameters $\eta = 0.001$, $\gamma = 0.95$ and $\tau = 1$.

E. Weights Initialization

For all the other experiments, we use a random initialization of the weights. In this last, we will try to start with every weight set to one or zero and use the usual parameters $\eta = 0.001$, $\gamma = 0.95$, $\lambda = 0.5$, $\tau = 1$.

In the Figure 5, we can see that the learning curve starts from the beginning around the convergence value. We still have a lot of variations because of the fact that the agents start from a random position and we can see that in both case, the learning curve becomes worst and fall back to a better solution as the neural network needs to be trained correctly. This initialization of the weights means that we start with the



(a) Starting with weights set to 0

(b) Starting with weights set to 1

Fig. 5: Experiments with different values for the weights initialization

same vector field in the Figure 2 (a) and then the car will gain a huge amount of energy in one direction.

III. Conclusion

According the results of the experiments, we can say that the exploration temperature parameter should be used with a decaying function and set to a big number at the beginning. This will allow the agent to explore more different state-action pairs.

We also used an eligibility trace as a short-memory mechanism to keep track of the number of visits by the agent of each state-action pair and then increase the eligibility when this number is higher. We saw that the learning curve is slightly better when setting it to 0.95.

Finally, choosing a weight initialization of one or zero will produce a lower escape latency of the agents at the beginning. It can be useful if you want to reduce the time to train the model.