HARSH SINGHAL
hsinghal@cs.wisc.edu

Independent Study Report CS799 Fall 16

Introduction and Problem Description

The Agent World is a test-bed for artificial intelligence techniques in a simplified universe. The universe consists of four types of entities: animals (also known as, 'agents' or 'players'), minerals, vegetables, and walls. One or more agents will be in a universe at a time. For each action an agent takes, it receives a reward. Negative rewards include running into walls (-1), pushing minerals (-2), bumping into other agents (-3), and running into sliding minerals (-25). Positive rewards include eating vegetables (+5), and pushing minerals into other agents (+25). Note that although an agent receives a slight negative reward for pushing a mineral, it receives a large positive reward if a mineral it has pushed slides into another agent. If an agent does not run into any other object, it receives a reward of zero.

Q-learning

Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It learns an action-value function that ultimately gives the expected utility (it is like extended look-ahead reward of going to some terminal or non-terminal state) of taking a given action in a given state. Agents simply select the action with the highest Q-value in each state. For any finite MDP, Q-learning eventually finds an optimal policy.

Q-function is defined as below:

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

Methodology:

The main task of training is to adjust the weight values corresponding to each of the inputs to maximize the total reward. Using 36 Perceptrons (=number of sensors) to predict the optimal Q-value in each sensor direction, the agent proceeds in agent world in the direction of the perceptron predicting maximum Q-value.

The number of input units in each perceptron was 144 (number of sensors * 4).
Initially the inputs and all the weights are assigned in range (0,1).
Inputs are initialised as per below, while the weights are assigned by random function.
Output is initially 0 (the forward direction).

For first set of 36 inputs:
inputs [i] = 1/ distance, if the sensor ‘i’ sees a wall in the direction provided by sensor using the provided functions, else 0. The inverse is done to keep the inputs in range (0,1)
And so on for the next 36x3 inputs corresponding to each of the 3 other elements.

Now ANNs predict the Q-value using feed forwarding technique. Then after taking a step in the maximum Q-value output direction, again the inputs were computed and new Q-values were calculated and so on.

Using the maximum of these new Q-values, loss is calculated and then backpropogated. The idea is that the reward which a particular weight causes, must affect that particular weight value only(positively or negatively) and so it is necessary to backpropogate the reward backwards along the path from which it came, rather than feeding it from the front to all the ANN weights.

Steps:
1) measure current sensors - call them s0
2) predict ForAll actions a, Q(s0, a) through forwardprop
3) Apply actionToDo in the agent world. Remeasure sensors, call these S1. Measure immediate reward, call it R.
4) Use your neural network to compute the max Q(S1, a) . Let Qbest(S1) = max Q(S1, a)
5) Train with the I/O pair: input: S0 output: Q(s0, actionToDo) + alpha(r + gamma * Qbest(S1) - Q(s0, actionToDo)

Alpha (learning rate) and gamma (discount factor while calculating utility of a state) can be set as per choice.

The learning rate or *step size* determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information.

The discount factor determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the action values may diverge. Even with a discount factor only slightly lower than 1, the *Q*-function learning leads to propagation of errors and instabilities when the value function is approximated with an artificial neural network. In that case, it is known that starting with a lower discount factor and increasing it towards its final value yields accelerated learning.

Exploration: When a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs crude “exploration”. As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is “greedy”, it settles with the first effective strategy it finds.

A simple and effective fix for the above problem is **ε**-*greedy exploration* – with probability *ε* choose a random action, otherwise go with the “greedy” action with the highest Q-value. In their system DeepMind actually decreases *ε* over time from 1 to 0.1 – in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate. Exploration technique could be used to initialise the weights properly initially.

While training, I was facing a problem adjusting weights correctly.
When I used linear activation function, the weight values were reaching extremes
To restrain the weight values, i am using Rectified Linear Unit (ReLU).

In case of simple perceptron, there is no hidden layer between input units and output unit . So, the inputs to the output unit are w1 * i1, w2 * i2.. (Currently I am not using a bias node)

Results and Discussion:

I have run the player from the start, i.e., without any initial training, up-to 100 games of 5000 steps each. Learning rate(alpha) and Discount Factor(gamma) are both 0.2.
There are 4 players : Random Player , Assassin, Smart Player and HarshPlayer and 100-100 vegetables and minerals.
Random Player – Does random movements
ShavlikPlayer - walk towards Vegetables; if none, move 180 degrees away from closest Player
Assassin - if this player sees another player, runs right toward it. Otherwise does a random walk.
HarshPlayer – Movement based on reinforcement learning

The agent is able to take a decision for all the basic scenarios (not hitting a wall, avoiding the moving and the still minerals, moving towards vegetables, moving in all directions and not just standing still) Though it yields a good score but the score is not competitive. One of the reasons is that sometimes the player gets stuck around the local maxima. I could not find how to correct this behaviour altering parameters alpha or gamma. Also it does not try to gain points by hitting other players with minerals.

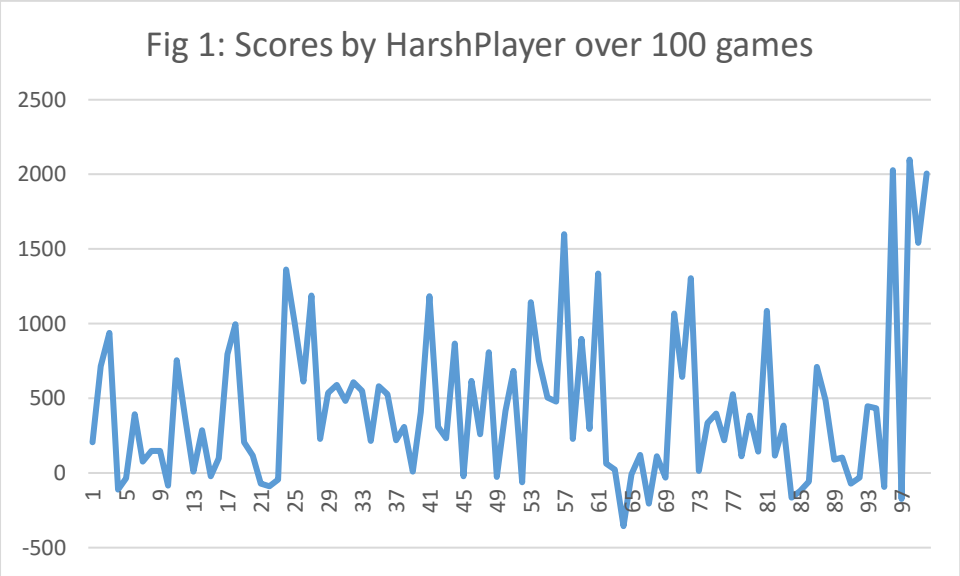
Hidden layer could not be added as the coding and implementation would have been much more complex. Also, with 144 inputs already, the program speed would have taken a hit if hidden layer calculations would have also been performed, and as we can see, there is a time limit bound for the total time of each step size given to a player to do the computations.

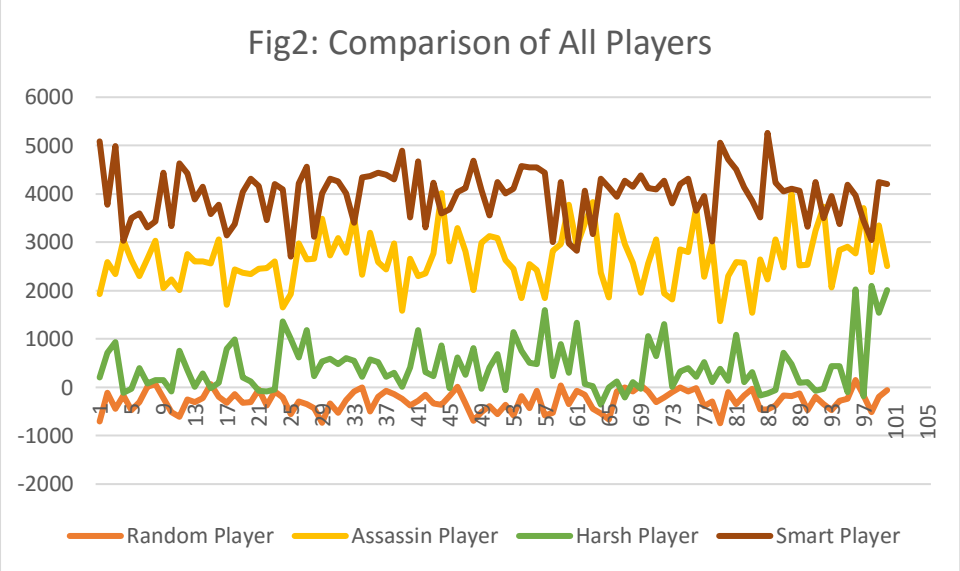
I faced a number of challenges for training the agent, like how to decide how many inputs/outputs to have, whether to add a hidden layer or not, whether to add bias vector or not, what range must the weight values and input values to be kept in, what kind of activation function to use and how to calculate expected value and loss and how to backpropogate that loss.

In the beginning, I was providing very few number of features which were not sufficient to learn playing in the agent world (16 for the four directions times number of elements). On the other hand, adding more features posed problem that player was giving preference to a particular direction, maybe due to some normalization. By trial and error, I finalized number of features to be given as an input to the ANN which was the inverse of distances of all the objects in all the directions of the sensors.
After this, I faced problem adjusting the weights properly as they were approaching infinity which I solved using the rectified linear unit.

Once trained, the trained ANNs were used to predict the next step to be taken in the agent world. I tried to add hidden units to the implementation but it turns out, coding the back-propogation function becomes complex.
Also, adding a hidden layer made it difficult to perceive the changes which each of the specific weights was actually having on the final output, so debugging was becoming complex.

The scores of learning curve of HarshPlayer are as shown below in Fig.1
In Fig2., we compare the scores of HarshPlayer with SmartPlayer, Random Player and Assassin Player.





Conclusion and Possible Improvements

Numerous improvements are possible over the current results.
Player oscillates around the local maxima sometimes (in left bottom corner of field). This could be improved with more exploration (**ε**-*greedy exploration as discussed above*) or adding bias. This time I kept constant learning rate and discount factor, I could vary it across games, depending upon the learning of the model at that instant.

I would also like to add the optimisations that Smart Player is doing in order to improve score for HarshPlayer. For eg., SmartPlayer tries to move forward initially despite the minerals, since there is more vegetation on the right side of the field after crossing the minerals in centre of field. HarshPlayer could be trained to do that.
The player could learn to be offensive than just defensive as there are many points for hitting other players with minerals.

Also, for adding hidden layer, I could use Google's Tensor-flow to speed up the calculations when the calculations grow longer and model becomes huge. In this case the model - number of total inputs (144) at input layer and hidden layer could be distributed across 4 workers so as to converge faster and asynchronously. In this manner, the inputs, hidden layer and the outputs could all be increased to a much larger model for faster convergence, accurate predictions and higher scores. The model W_i could also take previous runs - W_{i-1} and W_{i-2} as inputs for better results.

References

CS540 and CS760 Slides
I would like to thank Professor Shavlik for making available the AgentWorld TestBed and for his guidance. It was a useful learning experience.