CS221 PROJECT PROGRESS REPORT

Playing Space Invaders and Q*bert using Deep Reinforcement Learning

Shreyash Pandey (shreyash@stanford.edu)
Vivekkumar Patel (vivek14@stanford.edu)

1 Abstract

In this project, we plan to build an agent using reinforcement learning to play games space-invaders and Q-bert. For the progress report, we have only implemented an agent for Space Invaders and will discuss about it. These games involve a very large number of states and hence, we use Q-learning using a deep neural network to approximate the Q-function.

2 Task Definition and Modelling

We are trying to achieve a task of learning to play Space Invaders. Our agent interacts with the game enviroment through a sequence of observations, actions and rewards. We use the Atari Emulator to emulate the environment for these games. The game is played in discrete time steps. At every time step, the agent chooses an action, based on the current state or randomly, from a possible set of actions. The emulator then simulates this action and brings the game to a new state. It also returns any reward received during this step. Hence, we can model our problem as follows:

- State s: We consider the state at any time instance to be an array of pixel values representing the image frame at that instance.
- Action a: The agent has six possible actions in the game of space invaders. These are: SHOOT, LEFT (move left), RIGHT (move right), SHOOTANDLEFT (shoot and move left), SHOOTNRIGHT (shoot and move right), NOOP (do nothing).
- Reward r: The reward is returned by the environment after the agent performs an action. We clip the reward so that it lies in the range [-1,1].

We want to make the agent learn to play the game in such a way such that it maximises the total score.

3 Concepts

3.1 Reinforcement Learning

Reinforcement learning has been widely used to solve the problem of programming intelligent agents that learn how to play a game with human-level skills or better. The goal is to learn policies for sequential decision problems by maximizing a discounted cumulative future reward. The reinforcement learning agent must learn an optimal policy for the problem, without being explicitly told if its actions are good or bad.

3.2 Convolutional Neural Network

Recent advances in the use of convolutional neural networks (CNNs), has completely transformed the representation learning domain in Computer Vision - making it possible to automatically learn feature representation from high-dimensional and noisy input data such as images. High-level Computer Vision tasks such as recognition (image classification) and detection have benefited greatly by using CNN based features. Because of their success and versatility, there has been considerable work in enhancing CNN components and improving such deep architectures.

4 Approach

4.1 Q-learning

Q-learning is a model free reinforcement learning technique. Due to a large number of states in the game, it is impossible to make and store a Q-table which would contain the Q-values and best actions for each state. Hence, we use a function to approximation for this task. We use a deep neural network to model this function, which would approximate the Q-table and would calculate the Q-values for each action, for a given state.

$Q^*(s,a)$, where a is an action and s is a state, is the expected value (cumulative discounted reward) of taking action a in state s and then following the optimal policy. Q-learning uses temporal differences to estimate the value of Q*(s,a). An experience (s,a,r,s') provides one data point for the value of Q(s,a). The data point corresponds to the event that the agent received the future value of $r + \gamma V(s')$, where $V(s') = \max_{a'} Q(s',a')$; this is the actual current reward plus the discounted estimated future value. This new data point is called a return. The agent can use the following equation called Bellman's equation to update its estimate for $Q(s,a)$:

$$Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$$

or, equivalently,

$$Q[s,a] \leftarrow (1-\alpha)Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'])$$

. Here, α is the learning rate. Thus lesser the alpha, the more we value our experience and more the α the more we update our experience/table with newly observed returns.

4.2 Exploration-Exploitation Tradeoff

We can train our agent to learn the Q-values of state-action pairs using the equation in the previous subsection. However, this gives rise to a problem. At the beginning, our agent does not have any knowledge of the states or actions. Taking the action that maximizes the expected reward currently may not be the optimal action. It may very well happen that a certain action which has a low Q-value currently, may lead to more rewarding states. This happens because only picking the action that maximizes the expected reward does not lead to the agent exploring all the possible states of the game.

To fix this, we make use of the ϵ greedy policy. In this policy, with probability ϵ, the agent chooses its action randomly from the set of actions, and with probability $1-\epsilon$, it chooses the action with the highest Q-value. We keep ϵ high initially and decrease it gradually. This way, the agent not just explores a lot more states in the state-space, but also learns the optimal policy.

4.3 Deep Q Networks

The input to the deep Q network, which is also a state, is an image frame. Hence, we make use of Convolutional Neural Networks for this processing.

CNNs have proved to be effective in learning from high dimensional data. This is exactly the problem we deal with here. From the image frame, the network parameters get trained in such a way so as to figure out the object positions on an image frame and hence, helping the agent take the optimal actions.

4.4 Experience Replay

Deep Q-Networks when trained for long, may forget the information learnt from the past. Or, training them on a situation just once may not be enough. To solve this problem, we use experience replay.

In experience replay, we store agent's experiences, the tuple (s_t, a_t, r_t, s_{t+1}) in a memory cache. To train the network, we then randomly sample a batch of experiences randomly, and train using the above formulation. Meanwhile, the agent also keeps on playing the game, performing actions for various states, and we keep on storing these experiences in the memory cache. This is an efficient way to learn from past experiences. This way, the agent's learning is logically separated from gaining experience. The interleaving of these two processes also ensures that the agent manages to learn and form the optimal policy.

5 Training

For our training update rule, we use the fact that every Q function for some policy obeys the Bellman equation:

$$Q_\pi(s, a) = r + \gamma Q_\pi(s', \pi(s'))$$

The difference between the two sides of the equality is known as the temporal difference error, δ:

$$\delta = Q(s, a) - (r + \gamma \max_a Q(s', a))$$

To minimise this error, we will use the Huber loss. The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy. We calculate this over a batch of transitions, BB, sampled from the replay memory:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r)\ \in\ B} \mathcal{L}(\delta)$$

where

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

6 Progess till now

We decided to implement agent using the Deep Neural Networks using the PyTorch library. The syntax of PyTorch is closer to the numpy syntax and hence we were able to grasp it quickly.

We have implemented our DQN with experienced replay for Space Invaders and are currently iterating over the episodes to train the network. The CNN architecture consists of 3 convolution layers, each with kernel size 5x5 and stride 2, and each followed by batch normalization layers. The final fully connected layer has 6 outputs (one for each action). Our basic Deep Q Learning implementation achieves better performance than the random score of 180. The maximum score observed in 600 episodes of training is 635, and the average score is 320.

7 Planned Work Ahead

We plan to implement an agent using DQN for the Q*Bert game. Later, we plan to implement it using a DRQN. If time permits, we would like to explore more techniques that could be used to improve the performance of our agent.

References

[1] Greg Brockman et al. OpenAI Gym. 2016. eprint *: arXiv:1606.01540.*

[2] Nihit Desai, et al. "Deep Reinforcement Learning to play Space Invaders." *CS221 Project.* Autumn, 2016.

[3] Clare Chen, et al. "Deep Q-Learning With Recurrent Neural Networks." *CS229 Project.* Autumn, 2016.

[4] Adam Paszke, "PyTorch Reinforcement Learning (DQN) tutorial"