CS 221 Reinforcement Learning Techniques for Atari Breakout

Vishakh Hegde, Rahul Makhijani, and Chiraag Sumanth
SUNet ID: vishakh, rahulmj, csumanth

1 Introduction

1(a) Motivation

As we move towards making machines understand human sensory input forms such as auditory speech and vision, systems must be designed to process and effectively learn from these very high-dimensional inputs. This ability to learn directly from high-dimensional environmental input is crucial to take the next leap in Reinforcement Learning. With the advent of Deep Learning techniques for computer vision, it is now possible to directly train RL agents to play video games by directly giving game screen images as input for training, allowing the neural network to automatically learn features from its hidden states, and not utilize any hand-crafted feature representations.

In the past, people have used many different techniques to successfully play Breakout (along with many other Atari games). They include traditional Q-learning algorithms on some features extracted from the game screen (things like ball position, velocity and direction) to more sophisticated algorithms like Deep Q learning [1]. After the demonstration of the effectiveness of DQN on Atari games, people have improved it with using double DQN [4] and using DQN with AC-3 [5] and have performed a thorough analysis of the respective algorithms. We have open sourced our code on Github and can be accessed here: https://github.com/vishakhhegde/DeepRLAtariBreakout

1(b) Breakout

Atari Breakout is a class single player game in which the player has control of the paddle and the goal is to bounce the ball off the paddle and hit as many tiles as possible. Each tile is worth 1 point and the player has multiple lives.

The goal is to maximize the score in the Atari 2600 game Breakout. At any point in the game, the possible physical paddle actions one can take are the following: {**right, left, stay there**}.

The evaluation metric used is the total points scored in the Breakout game, by each algorithm, after playing 100 episodes after the same amount of training.

2 Infrastructure

We use the *OpenAI Gym* environment for this project. The API of the environment allows us to input an action (a number between 0 and 5). Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from 2, 3, 4 and get the following as the output:

- **Observation:** RGB image of the game screen, an array of shape (210, 160, 3)
- **Reward:** For each action taken, we get a reward $\in \{0, 1\}$.
- **isEnd:** Whether or not the game is over

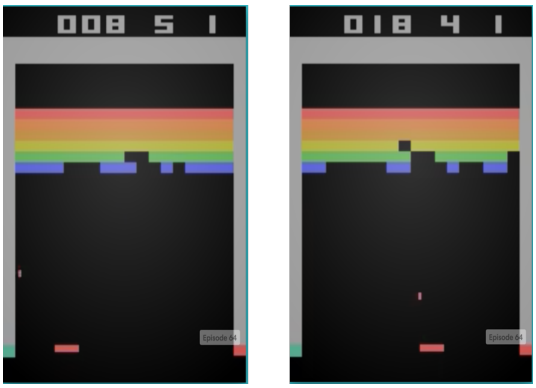

Figure 1: The Game Screen provided by Open-AI Gym

Interestingly, this environment provides us with no prior knowledge about the mapping between the six possible actions it takes in and the actions are actually relevant to playing the game, which has only three physical actions associated with it, as previously described. We implement our algorithm using python and use `TensorFlow` to train our neural networks.

3 Approach

3(a) Feature Extraction using Neural Networks

Since images are high dimensional inputs, we would like to get a low dimensional representation which encodes the core essence of the state of the game. While people have tried using hand built features and have also got good performance, we would like to follow DeepMind's approach of using a Q Neural Network (we experiment with multiple neural network architectures, including a 1 hidden layer network and a three layer convolutional neural network). We build neural networks to directly output a 6 dimensional Q-value, given the state and input the action corresponding to the maximum Q-value in the OpenAI environment to play the game.In order to play the game effectively, we humans probably use the following information about the ball: { **Direction, Velocity, Acceleration**.} The reason for this is that (we know from physics that) external force, current position and velocity is enough to completely specify the trajectory of an object.

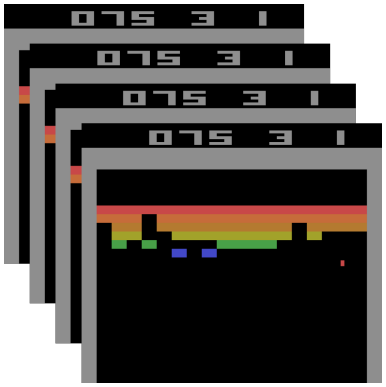

Figure 2: Successive Frames stored to capture notion of relative velocity and direction

In order to mimic humans, the agent has to figure out these three attributes of the ball before taking the action. Therefore, with 2 frames, humans can generally figure out the direction of the ball and the velocity. However, in order to figure out acceleration, we need at least three consecutive frames. Intuitively, the more frames we have, the better is the action we can potentially take. Following [1], we chose to use 4 consecutive frames to take an action.

3(b) State

The state here is an $80 \times 80 \times 4$ vector. i.e., we use four 80×80 greyscale images. These are resized from their original (210, 160, 3) and converted to grey-scale. While one of these is the current image from the game environment, the remaining three are the images before the last three steps were taken. Resizing to 80×80 is to reduce the number of parameters. Colors do not really matter (humans can play well even if the game-screen is converted to greyscale). Hence, using grey-scale is probably not going to make much of a difference for the neural network and can reduce the number of parameters.

3(c) Experience Replay

It is common to use stochastic gradient descent to update weights in a neural network. The reason for this is that very accurate gradients are not really necessary; an approximation of it is sufficient. While this is true for tasks like classification, it is not intuitive that we can use stochastic gradient descent to update weights of the network, since at every stage, only the current game state is to take an action. While it is possible to update the weights based on the cost produced by the current game state, it will be highly noisy and consecutive updates will be highly correlated. This will lead to parameters getting stuck in a bad local minima or an oscillating behavior.

One way to eliminate noisy updates from a single state is to use the average loss produced by multiple states to make the update. However, for stochastic gradient descent to work well, we need to have states that are independent and identically distributed. In order to satisfy such an assumption (or at least get an approximation of that), we can store multiple experiences and randomly sample a minibatch from the set of experiences in order to update the weights. This will smooth our learning and avoids catastrophic

oscillations in the parameters space. This method is known as experience replay and we use this in our implementation. Each experience is a tuple of (state, action, reward, next state, isEnd).

3(d) Gibbs Sampling

We also use Gibbs Sampling (apart from uniform sampling) to sample experiences from experience buffer. We maintain two lists, namely:

- Experience
- Sampling Probability Distribution (unnormalized)
- Sampling Probability Distribution (normalized)

Both the lists attain a maximum size of 100000, which is the size of our experience buffer. We add to the experience buffer the state of the game every time we take an action and get a new state. However, we modify the unnormalized `Sampling Probability Distribution` in the following manner:

- If the action was taken based on Q-values from the neural network, we initialize the value corresponding to input state to 1.
- If the action was random as a result of following the ϵ greedy exploration strategy, we initialize the value corresponding to input state to 0.5.
- Add the cost from a minibatch forward-pass to all the values corresponding to states in the minibatch.
- Use the normalized `Sampling Probability Distribution` whenever we need to sample from the minibatch.

We sample from the normalized `Sampling Probability Distribution` whenever we choose states from a mini-batch to train the network on. Following are the consequences of using Gibbs Sampling:

- Place less emphasis on learning from random actions and more emphasis on learning from actions given by the neural network.
- Place more emphasis on learning high-loss examples. High-loss examples are states which are hard to learn from. The intuition is that placing emphasis on learning from tough examples will help us learn faster.

3(e) ϵ - greedy exploration strategy

It is necessary to have a good exploration strategy to explore actions (and hence go to unexplored states) that is not given by the current policy. In our project, we use the ϵ greedy strategy, where, with probability ϵ, the agent takes a random action and with probability $1 - \epsilon$, we take the action given by the current policy (choose greedily most optimal action).

We have three stages to our strategy:

- **Large Exploration:** We first observe the environment by taking completely random actions.
- **Decreasing Exploration with Increasing Exploitation:** In this stage, we linearly reduce epsilon from 1 to a small value (0.05).
- **Only Exploit:** Concentrate on using the output produced by the policy to improve the policy.

3(f) Our Algorithms

Algorithm 1 Deep Q Learning with Experience Replay

Initialize replay memory D to size N
Initialize action-value function Q with random weights
for episode = 1, M **do**
  Initialize state s_1
  for t = 1, T **do**
    With probability ϵ select random action a_t
    Otherwise select $a_t = argmax_a Q(s_t, a; \theta_i)$
    Execute action at in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = \{s_t, a_t, x_{t+1}\}$ and preprocess $\Phi_{t+1} = \Phi(s_t + 1)$
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
     Set y_j:= r_j for terminal s_{j+1}
     Set y_j:= $rj + \gamma \max a0Q(\Phi_{j+1}, a0; \theta)$ for non terminal ϕ_{j+1}
    Sample a minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Perform a gradient step on $(y_j - Q(s_j, a_j; \theta_i))^2$ with respect to θ
  return
return

We provide the pseudo code of the Q-learning algorithm with experience replay, in Algorithm 1.

3(g) Sampling Strategies

We use two sampling strategies for minibatch sampling. Sampling can not only affect the speed of training, it can also push the policy towards a better local optima (or otherwise) depending on how good the strategy is. Following are the two sampling strategies:

- Uniform Sampling
- Gibbs Sampling

3(h) Manipulating Rewards

Intuitively, we will learn more from actions suggested by the neural network and the state obtained by taking that actions, than from random actions. Intuitively, it makes sense to use a high reward for non-random actions and a low reward for random actions. We use a reward of 5 for non-random actions and a reward of 1 for random actions.

4 Neural Network Architectures

4(a) Baseline: 1 Layer Neural Network

We use a one layer neural network as our baseline. This one layer is a fully connected layer and connects directly to the input. Here is the network architecture:

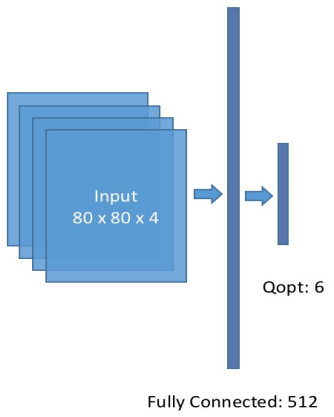

Figure 3: 1 Layer Neural Network Architecture

4(b) 3 Layer CNN

Convolutions are introduced in neural networks to mainly achieve spatial invariance and to reduce the number of parameters of the network. While spatial invariance is not desirable, we use convolutions mainly to reduce the number of parameters in the network. Convolutional neural networks learn a hierarchical representation of the input, starting from basic building blocks like edge detectors and Gabor like filters. We believe that such a deep network will learn a good representation that can be used to output Q-values.

Justification for the choice of the deep network architecture:

- We reduce the stride of the network as we go deeper into the network since deeper layers have larger receptive fields, which is done routinely when designing CNNs.
- We use a fully connected layer with 512 neurons so that the network learns a compact representation of the game state.

Here is the network architecture we have currently implemented:

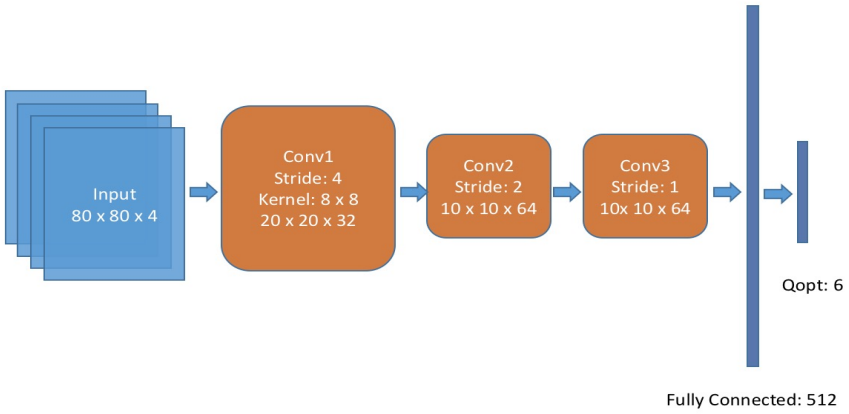

Figure 4: 3 Layer CNN Architecture

4(c) Implementation and Training

We use python to implement our Q-learning algorithm and TensorFlow to implement and train our neural networks.

5 Experiments and Results

We perform the following experiments to compare our methods:

- **Comparing Policies Learned Using Different Architectures:** We fix the sampling strategy to uniform sampling. We compare the performance of 1 layer neural network and 3 layer CNN. We have two comparison metrics: average score over 100 games and maximum score over 100 games. We do this comparison at various stages (number of actions taken) of learning.
- **Comparing Sampling Strategies:** We fix the neural network used to the 3 layer neural network since we see from the previous experiment that it performs better than 1 layer network. «Write more»

5(a) Comparing Network Architectures

5(a).1 Human performance:

Before we started implementing our algorithms, we wanted to see how a human performs. One of us on this project team, played a total of 100 episodes of Atari Breakout and achieved an average score of 29.50 over about 100 games. The performance reduced towards the end because of fatigue and boredom (computers do not get bored, fortunately). We aim to beat this performance.

5(a).2 Dumb Baseline:

The most dumb baseline possible is a random policy that takes random action for each state. This gets an average score of 1.38 and a max score of 6 among 100 games. Any learning algorithm must necessarily perform better than this even if it learns very little.

5(a).3 Baseline:

We train a 1 layer neural network over many steps. We find that the policy given by this network is only slightly better than a random policy. The average score is 2.96 and the max score is 9. This is especially surprising since it has a large number of parameters and therefore the number of functions this network can potentially learn is very large. However, it might be the case that the family of functions it represents are all highly biased for the problem at hand.

5(a).4 Deep Q Learning with uniform sampling:

We expect it to perform much better than our baseline. We hoped that we could train it to perform better than average human accuracy. We trained this for about $15M$ iterations and took about 10 days to train it on a GPU. We had to babysit the learning process initially for about 100000 iterations before we were confident that it was learning effectively. Training a Deep Q Network is hard because the algorithm is highly sensitive to the following hyper-parameters:

- **Learning rate:** In many of our initial runs, we found that a high learning rate leads to exploding and oscillating Q-values since it tries to learn very aggressively. Therefore it was import to reduce to learning rate to the range of 10^{-6}. We also found that low learning rates are also detrimental. The reason is that the updates are so slow that the experience buffer loses all states from the exploration stage even before the network starts learning something.
- **Experience Buffer Size:** In practice, the higher the size of the experience, the better it is. However, due to hardware restrictions (RAM), we were forced to keep the buffer size at 100000. A small buffer size will lead to losing valuable exploration information after a certain number of steps.
- **Number of explorations:** It is critical to have the right amount of exploration states stored in the buffer. If we have a lot of such states, there is a high chance that we will choose bad mini-batches in the early learning stage which will add bad outputs to the experience buffer. This will therefore affect further learning efficiency. If we have very few exploration states, the parameters might get stuck in a local minima and we might be forced to believe that we cannot do any better. The number of exploration states can be controlled by fixing the initial ϵ, final ϵ and the rate at which it changes to go from initial ϵ to final ϵ.

Table of Scores

Model	Max Score	Average Score(for 100 games)
Random	6	1.38
1 Layer CNN	9	2.96
3 Layer CNN	52	24.45

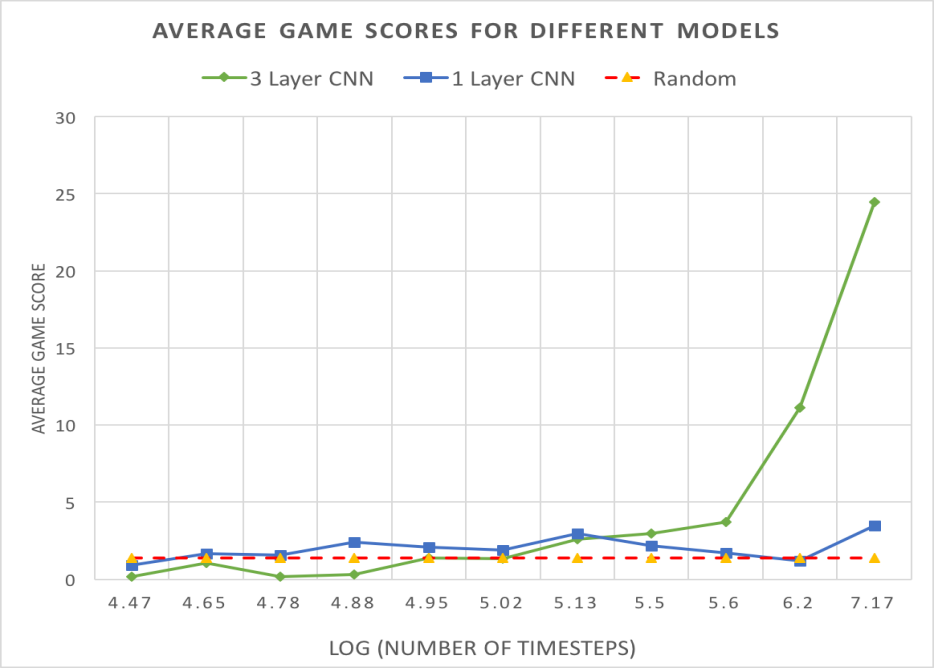

Figure 5: Plot of the average scores obtained by different models.

5(b) Comparing Sampling Strategies

We see from figure 6 that the cost increases almost exponentially while using Gibbs Sampling, whereas it oscillates between reasonable values in the case of uniform sampling. This is probably because Gibbs Sampling over biases certain states in an undesirable manner, leading to exploding costs. Exploding costs form a cascading effect where the same mini-batch gets chosen over and over again since we sample high cost states more.

From figure 7 that maximum Q-value explodes almost exponentially, further confirming that fact that in our experiment Gibbs Sampling performs worse than uniform sampling. However, we cannot come to an absolute conclusion that it is worse than uniform sampling. Better probability assignments might be required for this.

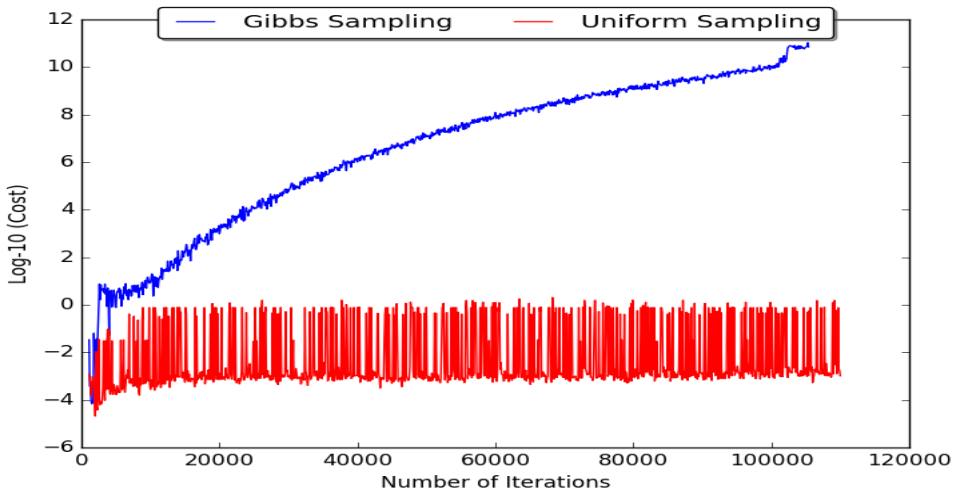

Figure 6: Plot of log(cost) versus log(Number of Iterations)

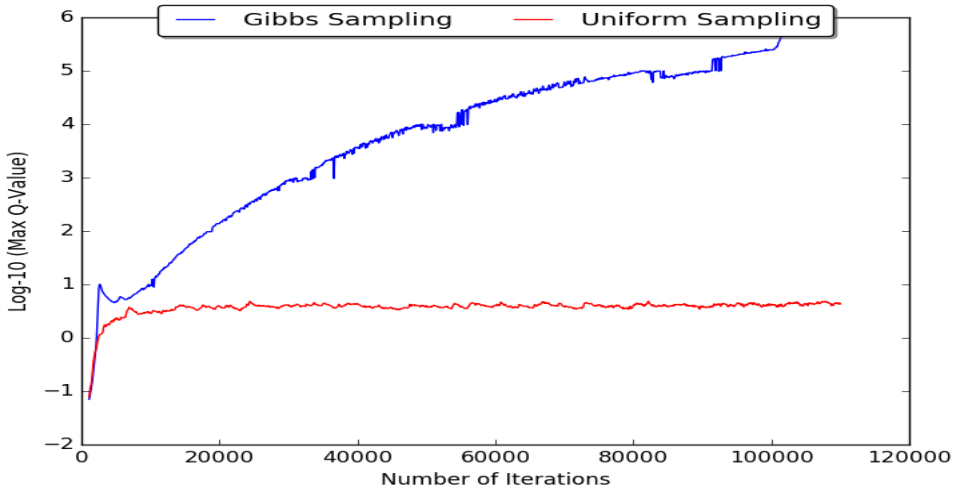

Figure 7: Plot of the average scores obtained by different models.

6 Future Work

We have several directions in mind to take this forward. One way to improve accuracy is to probably use a bigger network, a bigger experience replay buffer size and smaller learning rates. We would also like to implement Double-Q-Learning and check the effects of Gibbs Sampling on it. We came up with the concept of *Happy Experience Replay* where we only remember experiences leading to high rewards. This is one way to incorporate prior knowledge that we are actually playing Atari Breakout instead of any other Atari game. We believe that this will help speed up the initial learning process by not learning on useless observations.

References

[1] Playing Atari with Deep Reinforcement Learning, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller; DeepMind Technologies.

[2] OpenAI Gym by Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba (on archive).

[3] Tensorflow: https://www.tensorflow.org/versions/r0.11/tutorials/deep_cnn/index.html

[4] Deep Reinforcement Learning with Double Q-learning, Hado van Hasselt, Arthur Guez, David Silver, AAAI2016

[5] Asynchronous Methods for Deep Reinforcement Learning, Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu, https://arxiv.org/abs/1602.01783