BCS Project - OORL

GOAL- Reproducing the results of the paper '*Playing Atari with Deep Reinforcement Learning*'

Team Members :
- ➢ **Sajal Goyal** (180651)
- ➢ **Aditya Gupta** (190061)
- ➢ **Padmaja Chavan** (190580)
- ➢ **Arka Das** (190175)

1. **Description**

Atari 2600 is a challenging RL testbed that presents agents with a high dimensional visual input (210 x 160 RGB video) and a diverse and interesting set of tasks that were designed to be difficult for humans players.
The network is not provided with any game specific information or hand-designed visual features, it learns just from the video input, the reward, terminal signals, and the set of possible actions.

1.1 Problems faced while training RL model :

- ❏ Scalar reward signal that is frequently sparse, noisy, delayed and the model takes a long time for slight increase in prediction accuracy as compared to deep learning models.
- ❏ Data samples are not independent, sequences of highly correlated states.
- ❏ The data distribution changes as the model learns new behaviours, so there is no fixed underlying distribution.

1.2 — How to overcome these problems :

- ❏ From raw video data, the Deep Q-network with CNN is built, with stochastic gradient descent to update weights.
- ❏ For correlated data and non-stationary distribution, an experience replay mechanism is used which randomly samples from previous transitions.

2. **Background**

2.1 — What agent can observe:

- ❏ An image from the game emulator (no access to the internal state)
- ❏ Reward which depends on the whole prior sequence of actions and observations.

2.2 — It's impossible to understand the current situation fully from only the current screen. We overcome this by -

- ❏ Using sequences of actions and observations and learning strategies.
- ❏ This gives rise to a finite Markov decision process in which each sequence is a distinct state.

2.3 — The goal is to maximizes future rewards as

- ❏ Future rewards are discounted so that the value does not shoot up to infinity for infinite processes.
- ❏ Optimal action-value Q*(s; a) as the maximum expected return achievable.

2.4 — How the RL training job works ?

- ❏ The basic idea behind many RL is to estimate the action value function, by using the Bellman equation as an iterative update, such value iteration converges to optimal action value function. But in the real case scenario, it is impractical. Instead we use a nonlinear function approximator with weights as a Q-network.
- ❏ Q-network is trained by minimizing a sequence of loss function which is an error between a Q-network and target, the action value function by the Bellman equation.

2.5 — Characteristics of the work

- ❏ Model-free — using samples from the emulator without constructing an estimate of the emulator.
- ❏ Off-policy — greedy strategy with some adequate exploration of the state space.

3. Deep Reinforcement Learning

- ❏ The goal is to connect a RL algorithm to an deep neural network which operates directly on RGB images and by using stochastic gradient updates.
- ❏ Utilizing experience replay which is applying Q-learning updates to samples of experience, random from the pool of stored samples.

3.1 — Advantages over standard online Q-learning

- ❏ Each step of experience is potentially used in many weight updates.
- ❏ Due to the strong correlations between the samples; randomizing the samples breaks these correlations and reduces the variance of the updates
- ❏ Experience Replay**.

4. Preprocessing and Model Architecture

4.1 — Preprocessing

- ❏ The raw frames (210 × 160) are preprocessed by converting RGB to gray-scale
- ❏ Down-sampling (110×84)
- ❏ Final cropping to (84 x 84) region to roughly capture the playing area.

4.2 — Model architecture

- ❏ The input to the neural network : 84 × 84 × 4 image (using the last 4 frames of a history)
- ❏ The first hidden layer : 16 filters (each 8 × 8 dimension) with stride 4
- ❏ The second hidden layer : 32 filters(each 4 × 4 dimension) with stride 2
- ❏ The final hidden layer: fully-connected and consists of 256 rectifier units
- ❏ The output layer : fully connected linear layer with a single output for each valid action

5. Experiments

- ❏ RMSProp with minibaches of size 32.
- ❏ e-greedy policy → e decreases linearly from 1 to 0.1 over the first million frames, and fixed at 0.1 thereafter.
- ❏ The agent sees and selects actions on every k^{th} frame instead of every frame, and its last action is repeated on skipped frames.

6. Conclusion

- ❏ Uses only raw pixels as input.
- ❏ Q-learning that combines stochastic minibatch updates with experience replay memory.
- ❏ Although RL takes a lot of time to train, it is capable of doing tasks which are impossible by pure deep learning.
- ❏ This network can also be applied to a range of Atari 2600 games.

****Experience Replay** : It is the key method that prevents our network from diverging. In fact, I can resist showing you the amazing results that adding experience replay gets us according to “Human-level control through deep reinforcement learning”: The idea behind experience replay is quite simple: at each Q-learning iteration, you play one step in the game, but instead of updating the model based on that last step, you add all the relevant information from the step you just took (current state, next state, action taken, reward and whether the next state is terminal) to a finite-size memory (of 1,000,000 elements in this case), and then call fit_batch on a sample of that memory (of 32 elements in our case). Before doing any iterations on the neural network, we prefill the memory with a random policy up to a certain number of elements (50,000 in our case).The reason why experience replay is helpful has to do with the fact that in reinforcement learning, successive states are highly similar. This means that there is a significant risk that the network will completely forget about what it’s like to be in a state it hasn’t seen in a while. Replaying experience prevents this by still showing old frames to the network.

GITHUB - https://github.com/sajalgoyal113/tensorpack

******

The github link contains the colab notebook Atari.ipynb which we created to train an existing tensorpack implementation of the paper for getting average scores of 427 and maximum scores of 863 in Atari-breakout. This is highly training intensive and took a lot of time (around 10 hours for 80 epochs).

******