ECE 6254 Spring 2021 Final Project

Deep Q-Learning applied to Atari Breakout

Team members: Pranjal Gupta, Sahana Joshi, Viren Sugandh

Introduction:

In this project, we explore different Reinforcement Learning (RL) and Deep Reinforcement Learning algorithms on a few environments. Specifically, we explore Q-Learning on a Taxi environment, Deep Q-Learning, Double DQN, Dueling architectures and A2C on CartPole and finally Deep Q-Learning on Atari Breakout. The Taxi environment has a finite state space and can be solved by using a basic Q-Learning algorithm. The CartPole has a continuous state space and is solved by using Deep Q Learning, but our agent only has to process a few sensory inputs to take actions. In Breakout, our agent has to learn directly from the pixel data, which is both extremely large state space and also a complex visual input that needs to be comprehended.

In RL, an agent interacts with the environment, moves to a new state and receives an immediate reward from the environment. The agent aims to learn a policy, or map from observations to actions in order to maximize its returns (expected sum of rewards). In model-free methods, no model of the environment is required, and the agent learns about the environment by taking actions and sampling from it. In many practical problems, the states of the MDP are high dimensional and cannot be solved by traditional RL algorithms. This is where Deep Reinforcement Learning (DRL) algorithms incorporate deep learning to solve such MDPs, often representing the policy or other learned functions as a neural network and train these from its experiences. These networks are efficient function approximators, even in environments with very large state or action spaces.

Deep reinforcement learning constitutes a set of architectures/ algorithms of RL where the state-space of the problem is too large to be captured as tabular data. A popular application of Deep RL is in creating agents which can play games, as the state space is of the order of the number of pixels. Using deep networks also substitutes the task of feature engineering, as convolutional neural networks can take raw pixels as input and construct features from them. The objective of our project is to create an agent using Deep RL which can learn to play Breakout from scratch.

The work was divided as follows:

- Pranjal implemented the Q-Learning on Taxi, Deep Q-Learning, double DQN and duelling architectures for the CartPole. He also implemented the DQN architecture for Atari Breakout and trained the model with lives lost added.
- Sahana implemented A2C methods on Cartpole and Breakout environments.
- Viren performed the training, evaluation and created the videos of our agent playing the game.

Detailed Description

The Atari Breakout environment is much more complex and requires understanding pixel data directly, which is an extremely hard task. This is why we first experimented with a few simpler environments. To speed up our experiments, we also tested various other algorithms on these environments. This helped us get results quickly and also debug faster. We used the Open AI gym environment for all our experiments.

Taxi-v3

In this environment (shown in Figure 1), we have an agent driving a taxi (represented by the yellow square) that needs to pick up the passenger at the blue spot and drop them off at the red spot. There are barriers that it cannot cross, represented by '|'. The actions the agent can take are move north, move south, move east, move west, pickup and drop-off. The agent gets a reward of -1 for every step that it takes. It gets a reward of -10 every time it tries to pickup or drop-off the passenger at the wrong spot (a penalty) and gets a reward of +20 if it drops the passenger off at the right spot. Using only these rewards, our agent must figure out the optimal path to take.

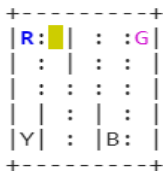

Figure 1: Taxi environment

If we use a random agent to solve this problem, the number of time steps taken averaged over 100 trials is 2674.98, and the Penalties incurred is 1432.78. We then use a RL algorithm called Q-learning, where we store a table that contains the value of being in a particular state, and taking a particular action. This value is an estimate of the future reward that we can obtain, starting from the current state, and taking a particular action. This is called the action value, or Q-value of the current state. After training our agent with Q-Learning over 100,000 episodes, the average time steps taken evaluated over 100 trials is 12.78 and our agent incurs no penalties. The reward plot obtained during evaluation is shown in Figure 2.

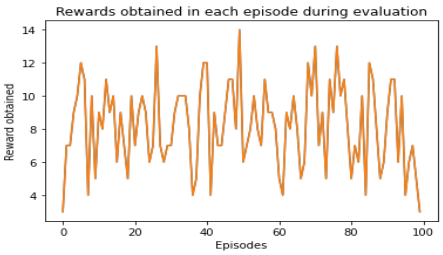

Figure 2: Evaluation results on taxi

CartPole-v1

In this environment (shown in Figure 3), the agent has to balance a pole by moving a cart. The actions it can take are move left or move right. It cannot move past a certain distance, and it also cannot tilt the pole beyond a certain angle. The agent observes only the cart position, cart velocity, pole angle and pole velocity. Every step it balances the pole, it gets a reward of +1. The goal is to balance the pole as long as possible.

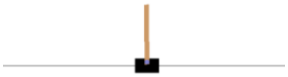

Figure 3: CartPole environment

If we try to use the Q-learning algorithm here, our Q-table would be extremely large and it would be computationally infeasible to implement it. This is because our state space is continuous. That's why the Deep Q-Learning algorithm is suitable here. We use Neural Networks as function approximators to estimate the Q-value of each state. The network is trained from the agents experiences (more detail on DQN later). The results obtained by training our agent on the DQN algorithm is shown in Figure 4. Initially, our agent is exploring the state space a lot, to learn more about its environment, which is why the rewards obtained are poor. After it learns enough about its environment, it starts to use this information to exploit and get higher rewards. In Reinforcement Learning, as in life, there must be a fine balance between exploration and exploitation.

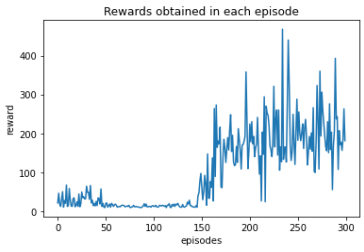

Figure 4: DQN training results

In DQN we have two networks, one called the main network and one called the target network. The target network is updated with the weights of the main network periodically. It is mainly present for stability. During training in DQN, our target action values (labels) are obtained by maximizing over the action values of the current state obtained from the target network. Since these values are noisy, we will probably obtain unfavourable actions. This method also almost always leads to overestimation of the true action value. Thus, the DQN leads to suboptimal policies. This is why we decided to try double DQN for the same task.

In double DQN, we maximize over the action values of the current state obtained from the main network, to take an action and evaluate the action taken by using the target network. In this way, we avoid over estimating our action values and this leads to a better policy. The results obtained by training our agent on the double DQN algorithm are shown in Figure 5. We can see a better growth of the rewards when the agent starts to learn.

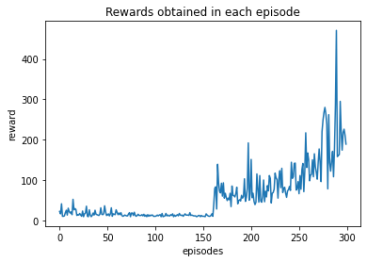

Figure 5: double DQN training results

Next we tried the Dueling architecture. Instead of directly predicting a single Q-value for each action, the Dueling architecture splits the final layer into two streams that represent the value and advantage function. The value function depends only on the state and the advantage function depends on the state and action. Intuitively, the Dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. In this way, the duelling architecture can more quickly identify during policy evaluation the actions that are redundant. It also quickly identifies the correct action to take under critical circumstances. The results obtained by using the Dueling architecture together with DQN are shown in Figure 6.

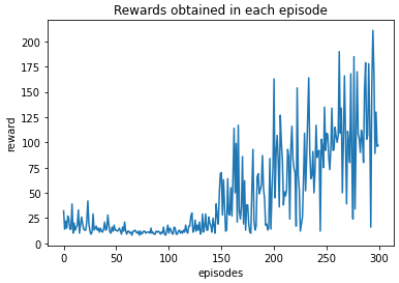

Figure 6: Dueling architecture training results

Actor Critic Methods (A2C)

Actor critic architectures fall into the category of model-free algorithms, where the state-space is too large to be expressed in a Q table, and hence use function approximators to estimate the policy or the state value. These algorithms use two function approximators: the actor and the critic. The task of the actor is to learn the policy directly: it does so using the policy gradient updates (described below) and the task of the critic is to estimate the state-value function. The general idea is that over iterations, the actor gets better at picking the best action and the critic network gets better at evaluating the true value of the states the environment passes through with

the actions chosen by the actor. This is an on-policy learning method, we are updating the same policy that we are following.

The critic network is similar to the Q network of the DQN: the updates happen based on the loss function between the observed rewards in an episode (or within a given number of steps) and the current estimate of the value of the state. The actor network updates are done using the policy gradients. The policy optimization function is the expected reward when following that policy, and we know a reward is generated for every state transition. So, the policy optimization function can be expressed as the expected value of reward-weighted probability density function of the transitions.

When the update is used in the exact form as shown above, it leads the REINFORCE algorithm. But the reward term associated with the transition adds a lot of variance to the policy updates, hence we introduce *baselines:* which are estimates of the reward or some modification of it. Depending on the baseline chosen, we have different policy gradient algorithms, in A2C, the reward is replaced by the advantage function which is the difference between the action-value function and the state-value function.

We tried A2C algorithm on two games: Cartpole and Breakout. In either case the environment was provided by OpenAI Gym. We used the same cartpole environment as in the previous parts. The results are shown in Figure 7. During evaluation, we observed that the agent can get an average reward of **199.**

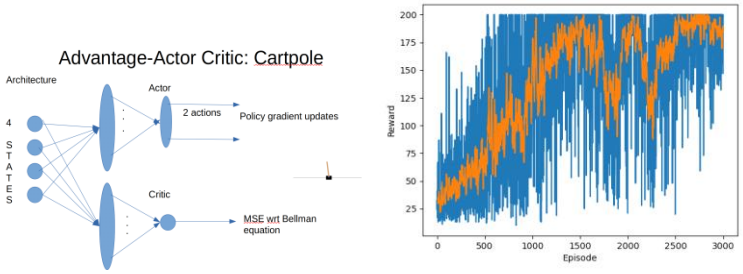

Figure 7: (left) A2C architecture, (right) Reward/episode plot during training

The breakout setup (environment explained later) is the same as explained before. The architecture (shown in Figure 8) is also similar to the one shown above, the main difference being that we use convolutional layers as feature extractors on the state space. We flatten the output of the convolutional layers and feed it into the actor and critic networks.

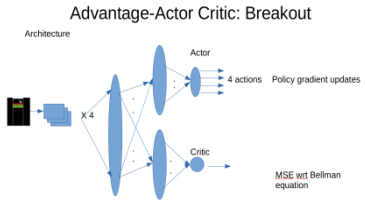

Figure 8: A2C architecture for Breakout

Atari Breakout

In this environment (shown in Figure 9), we have an agent controlling a slider. The agent hits a ball with the slider in order to break the tiles. The agent can either move left or right. In each step, the agent observes image frames from the previous 4 time steps and uses that information to take an action. The agent gets a reward of +1 for every tile it breaks. It has 5 lives in total and loses a life every time it misses the ball. The goal is to break as many tiles as possible before running out of lives.

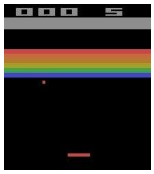

Figure 9: Atari Breakout environment

Since our state space is very large, we again use Deep Q-Learning. The overall architecture of our model is shown in Figure 10.

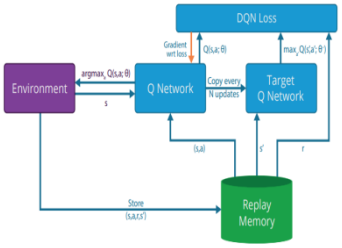

Figure 10 : Overview of the DQN architecture

Each individual block is explained below:

- The Environment: The agent interacts with the environment by taking actions. In response, the environment returns frames that represent the state of the game after taking that action. The frames from 4 previous time steps are stacked and returned as states. We convert each frame from 210x160x3 RGB images to 84x84 grayscale to speed up computation.
- The Q-network and the target Q-network: The Q-network and the target network have the same architecture. The input is the state obtained from the environment. Our network architecture consists of convolutional layers followed by dense layers. The outputs of the network are action values of the input state. The main network is used to select actions and is trained from the experiences sampled from the experience replay block. The target network is periodically updated with the weights of the main network. This helps keep the training labels stationary for a while and thus provides training stability.

- Replay Memory: It is a buffer that consists of past experiences in the form of (state, action, reward, next state, episode termination flag). It has a finite capacity so that the network is updated with only relatively recent experiences. Batches of these experiences are drawn from this block during training. We draw these samples randomly to break any correlations between samples. The target Q values used in training the main network are calculated from these experiences.
- DQN loss: The loss function is Huber loss, which is less sensitive to outliers. We also chose the Adam optimizer for gradient descent. The remaining hyperparameters such as filter sizes, replay memory lengths etc. were chosen to match the value in the Deepmind paper. The experiences obtained from replay memory contain information on the action taken and the reward obtained. The Huber loss is calculated between the Q-value for the action taken (obtained from the Q-network) and the calculated target Q-values (obtained by using Bellman Optimality on the next state Q-value obtained from target Q-network and the reward obtained from experience). The weights of the Q-network are updated by backpropogating this loss.

The evaluation was performed during training every fixed period of time. It lasted for a fixed number of frames. In evaluation the agent maximized over Q-values obtained from the main network for the current state (exploitation) to take actions. The frames were recorded and the total reward obtained in each evaluation episode was calculated. Using these frames, we make a video of the agent playing the game during each evaluation stage. In reinforcement learning, the training performance of the agent is a massive underestimation (unlike supervised learning) of its true capability, due to its constant exploration during training. This is why we don't report any training results.

Results:

We evaluated the agent over ten trials and found the average score our agent obtained. We also made a modification in our original implementation and considered a life lost during an episode as a terminal state in the replay memory. This incentivizes the agent to treat each life as valuable and avoid losing a life at all cost. The agent performed much better with this change. In both cases the agent performed better than the human benchmark which is the average score of 30. Our results are summarized in Table 1

Implementation	Average Score	Best Score
Without accounting lives lost within episode	144.6	182
Accounting lives lost within episode	320.3	367

Table 1: Results sumary

After enough training our agent was able to learn the optimal policy of "tunnelling" where the agent digs a hole in the blocks early on and aims the ball through the hole to score many easy points from the ball bouncing around the top for a long time. This shows that our agent is indeed very farsighted, as is expected from an agent trained with reinforcement learning techniques.

References

[1]. Main reference: https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756

[2]. A2C paper: https://arxiv.org/pdf/1602.01783v2.pdf

[3]. A2C blog: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

[4]. Q-Table for taxi environment: https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

[5]. minDQN: https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc#:~:text=Critically%2C%20Deep%20Q%2DLearning%20replaces,process%20uses%202%20neural%20networks

[6]. OpenAI: https://gym.openai.com/