C8: Atari Games with Deep Q Network (DQN)

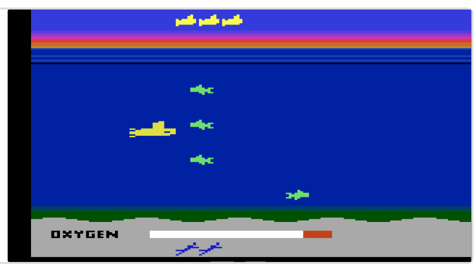
▼ Table of Content

https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Playing Atari with Deep Reinforcement Learning

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural

https://www.arxiv-vanity.com/papers/1312.5602/



Deep Q-Learning

- In Q-Learning, we estimate our Q-function through the construction of Q-table by finding the averaged return based on multiple iterations.
- Such exhaustive search across all possible state-action pairs for finding the optimal state soon faced a bottleneck when there are a large number of states and, in each state, there are a lot of actions to try.

- As such, Deep Q-Learning is introduced which uses NN with parameter/weights θ to approximate the Q-function with the goal that the approximated Q-function, $Q(s, a; \theta)$ is equiv/close to the optimal Q function, $Q^*(s, a)$.

$$Q(s, a\ ; \theta) \approx Q^*(s, a)$$

Loss Function & Target Network

- Before we go into details of the actual network architecture, let us properly define the end-goal that we wish to achieve or in more specific terms, the loss function that we wish to minimize.
- Recall that our Q-network is nothing more but a neural network that can approximate the Q-function.
  - Thus, the number of output neuron for Q-network is just number of action available given the current state.

$$f : S \to A$$

  - Recall that in Q-learning update rule, $r + \gamma\ max\ Q(s'a)$ is the target value and $Q(s, a)$ is the predicted value and we tried to minimize the difference by learning a right policy.

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma\ maxQ(s', a') - Q(s, a)]$$

- Hence, in DQN, our loss function is defined as the squared difference between the target and predicted value.

$$Loss = (y_i - Q(s, a; \theta))^2$$

  - Where our target value, y_i is defined as follows

$$y_i = r + \gamma\ argmax(Q(s', a'; \theta))$$

- Finally, the weights of Q-network is updated by minimizing the loss similar to how Q-learning works.

DQN Architecture

- Since the goal of a DQN is to approximate a Q-network, which learns from the rewards given by the environment after playing countless episodes of Atari games, a DQN is similar to a pure CNN with the first few layers being the convolutional network to learn from the pixel while the last few layers being the dense layer to approximate the Q-function
- The following are few characteristics of the CNN proposed in the paper and their reasoning behind the action:
    - Downsample 210x160 to 84x84 from RGB to Greyscaled input frame
        - To reduce the computational requirement for DQN
    - No pooling layer is used in the CNN
        - Retain positional information of pixels and features
        - Imagine a playing a Pong Game without the positional information of the ball, it is nearly impossible to do so.
    - Past four consecutive screen is passed in Forward Pass
        - Retain temporal information of the velocity of moving objects
    - Number of Output Neuron = Number of Action Space Possible
        - Get the Q-value for all actions available regardless of whether does a state allows

Tricks to Training a DQN

Experience Replay

- One more details when it comes to training a DQN is experience replay which is used to solve a problem of training NN for games.
    - In a single game, each consecutive pixels in a frame has high correlation from one state to another as the game environment simply does not change that rapidly in a typical Atari game.

- Such high correlation of input can cause a neural network to be highly biased and overfit with correlated experience.
- As such, we need someway to store and shuffle the input frames before feeding it forward for training of a Neural Network

- This is where Experience Replay comes in handy.
  - Experience Replay is simply a temporally Stack of replay buffer that contains of the transition information of a single action a, from one state s, to another s'. Which is denoted as follows:

$$\text{Exp Buffer} = < s, a, r, s' >$$

  - Before Starting Training Epoch
    - We store all the replay buffer to a fixed number before starting any training process.
  - During Every Training Epoch
    - Random batch of experience from the replay buffer is sampled for the training process.
  - After End of a Training Epoch
    - New batch of replay buffer is stored in the Experience Replay memory while the old ones are deleted

Target Network

- In the Q-learning update rule, there are two instance where a Q-function (which will be replaced by a NN) is called to calculate the target value $r + \gamma \ max_a(Q(s'a';\theta))$, and the predicted value $Q(s, a; \theta)$.
  - The problem rises when we tries to use the same Q-function for both the calculation of Q-value using the same weights from a neural network as there could be a lot of divergence between these two.
  - As small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values Q and the target values $r + \gamma \ maxQ(s', a')$.

> An analogy will be a dog chasing a meat hang at its own tail, it will not progress forward to the desired destination but just spinning around some random spot

- To solve this problem, we make used of Target Network $Q(s', a'; \theta')$ and Predicted Network $Q(s, a; \theta)$ which is two identical neural network with different weights.

$$Loss = (r + \gamma\ max_{a'}\ Q(s', a'; \theta') - Q(s, a; \theta))^2$$

  - The target network copies the weight from the actual Q network, and is frozen from time steps to time steps.
  - Freezing the target network for a while and then updating its weights with the actual Q network weights is what stabilizes the training.

> A follow up analogy will be fixing a extendible pole with meat at the back of a dog and changing its length from time-to-time so that the dog will chase the meat and walk in a more stable path

Rewards Clipping

- The final tricks for training a DQN for Atari games is rewards clipping whereby the reward value is clipped to +1 to -1 across all Atari games.

Epsilon-Greedy Strategy

- Like in Q-Learning, DQN make used of Epsilon-Greedy strategy to solve the Multi-Armed Bandit problem (aka Exploration and Exploitation trade-offs) in RL. Refer to the chapter if you are unfamiliar with the concept.

The Final Algorithm

- The steps involved in a DQN are as follows:

  1. The Game Screen (state s) are first preproced and feed to our DQN which returns the Q values of all possible actions in the state.

2. An action is selected using the epsilon-greedy policy:
    a. With probability epsilon, we select random action a
    b. With probability 1-epsilon, we select action that has maximum Q value, $a = argmax(Q(s,a;\theta))$
3. After selecting action a, we perform this action in state s and get the next state s' along with the reward r. Offcause s' refers to the preprocessed image of the next game screen
4. The transition is stored in our replay buffer as $< s,a,r,s' >$
5. Next, we sample some random batches of transitions from the replay buffer and calculate the loss which is the squared difference between the target Q and predicted Q

$$Loss = (r + \gamma \; max_{a'} Q(s',a';\theta') - Q(s,a;\theta))^2$$

6. Gradient descent is then performed wrt our actual network parameters θ to minimize the loss
7. After every k steps, we copy the actual network weights θ to our target network weights, θ'
8. We repeat these steps for M number of episodes.

Extra Tips and Tricks

Double DQN

Prioritized Experience Replay

Dueling Network Architechture