Deep Q-Networks (for Pong)

William Hill 2115261 | Ireton Liu 2089889 | Andrew Boyley 2090244

October 7, 2022

1 Introduction

A Deep Q-Network is a non-linear function approximation for the Q-function (i.e. state-action value function), which takes in the raw pixel values (that the agent would see on the screen) and computes the value associated with each possible action the agent could take, the maximum of which is the action chosen (under a greedy policy).

Our implementation is largely based off the RAIL Lab's implementation [1], although we have added comments explaining the logic of the various components of the solution.

2 Training Process

The goal of Deep Q-Learning (DQN) is to learn a set of weights w for a neural network, which takes normalised pixel values as input, and produces q-values for a number of output nodes, the largest of which corresponds to the action to take. The training process follows the Temporal Difference (TD) learning structure (specifically, Q-learning, which is off-policy TD control) and is executed as follows:

1. Initially 2 networks, a Policy Network and a Target Network, are initialised and made to be identical (i.e. set the weights of the Target Network to be equal to those of the Policy Network). The purpose of a separate Target Network is to fix the problem of highly-nonstationary targets (and so allow the network to converge instead of oscillating), where the weights of the Target Network are only periodically set to those of the Policy Network. In practice, the estimated q-value is taken from the Policy Network, and the target q-value is taken from the Target Network.

2. At state s, an action a is sampled using a policy. In this case, the ϵ-greedy policy is used, where a random action is taken with ϵ probability, otherwise the action is taken from the Policy Network. This action is then executed in the environment and the new state s' along with the reward r is observed.

3. This information (i.e. the transition tuple (s, a, r, s')) is stored in the agent's memory/Replay Buffer that is used for Experience Replay (with the oldest items being overwritten when the buffer becomes full). The purpose of Experience Replay is to break the correlation/dependence between successive states. This is required because the states do not constitute $i.i.d$ data: the next states are strongly correlated with prior states, and are, in fact, "generated" from those prior states (since the state space is so large). Consequently, the states we see are just because that is the trajectory we so happened to take, thus the agent is only exposed to a small subset of potential states. When this subset changes, in later episodes for example, the agent is at high risk of Catastrophic Forgetting, where the model forgets all the knowledge it has learnt previously when learning new information. Experience Replay mitigates these issues.

4. Step 2 and 3 are repeated for a number of iterations (this number is a hyper-parameter) before the training of the Q-Network begins. This ensures that the Replay Buffer is well populated with past experiences, thus ensuring that minimal correlation will occur between items in a randomly-sampled subset of the Replay Buffer.

5. For the training of the Policy Network, a subset of transitions (this is the minibatch) from the Replay Buffer is randomly selected to perform the parameter-update. This update is done using some optimiser (in this case, ADAM), where the model minimises the Mean Squared Error (between the Policy and Target Networks' q-values for items in the minibatch) given by the loss function:

$$L(w) = \mathbf{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w) \right)^2 \right]$$

where w^- are the weights of the Target Network, w are the weights of the Policy Network, r is the reward, and $\mathcal{D}$ is the Replay Buffer.

It is important that the Target Network does not always update with the Policy Network. That is, the weights of the target networks are "frozen" and only updated to the Policy Network's weights periodically.

3 Q-Network Architecture

The Neural Network contains 3 convolution layers and 2 fully connected layers, where the number of nodes in the output layer is equal to the number of actions that can be taken. The purpose of each layer is described as follows:

- The first hidden layer is a convolution layer that uses 32 filters, of size $8 \times 8 \times Number\_of\_Channels$, with a stride of 4. The $Number\_of\_Channels$ is the depth of the input image, which is 4. This means that 4 consecutive frames are concatenated and passed to the network. The input image has a 84×84 arrangement of pixels, which is required to ensure the correct number of features is available for the nodes of the fully-connected layers (which require a fixed input dimension).
- The second hidden layer is a convolution layer that has 64 filters, of size $4 \times 4 \times 32$, with a stride of 2.
- The third hidden layer is the last convolution layer that has 64 filters, of size $3 \times 3 \times 64$, with a stride of 1.
- The last hidden layer is a fully-connected layer, which maps from $64 * 7 * 7$ features (from the feature map following the convolutions) to 512 nodes (in the next layer).
- The output layer is a fully-connected layer, which maps from 512 nodes to $Number\_of\_Actions$ nodes.
- The Rectified Linear Unit(ReLU) is used as the activation function between the layers, with no activation being applied to the output layer.
- The chosen action is simply the action corresponding to the output node with the highest value.

This particular architecture allows us to compute the Q-value for every action given the input state (i.e. the input image) in a single pass.

4 Hyper-parameters

The following Hyper-parameters are used:

- **seed**

  Value: 42
  Role: The seed for the random number generator (of numpy, and the environment), which is used to initialise the Policy and Target Networks' weights, governs the random sampling from the Replay Buffer, and is used by the environment.

- **env**

Value: PongNoFrameskip-v4
Role: The environment isn't a hyper-parameter, but it is important that no frame skipping is used for the environment. Frame skipping is when an action is taken by the agent periodically (e.g. every fourth frame) and is simply repeated for the intermediate frames.

- **replay-buffer-size**

  Value: 5e3
  Role: This governs the maximum number of transition tuples, (s, a, r, s'), that can be stored in the Replay Buffer. This number should be large enough that sufficient transitions are stored such that a random subset has states with minimal dependence between them, but not so large that it uses too much memory.

- **learning-rate**

  Value: 1e-4
  Role: This is the size of the step that the Adam optimiser takes in the direction of the negative gradient when updating the parameters for the Policy Network.

- **discount-factor**

  Value: 0.99
  Role: The discount factor governs the (cumulative) relative weighting of future rewards to the return. This ensures that the agent balances short-term gains with the long-term goal of winning the game. Additionally, the discount factor allows us to reduce the impact of the high variance associated with long time-horizons. That is, we can take many possible trajectories from the current state (e.g. over many episodes), and the source of the variance is in the final reward of each trajectory, which may be quite different - the discount factor prevents this variance from wildly influencing our expected return.

- **num-steps**

  Value: 1e6
  Role: This is the total number of iterations (or moves) over which to train the agent. There may be many iterations (or moves) taken in an episode (a game played to completion/termination).

- **batch-size**

  Value: 256
  Role: This governs the number of transitions to randomly sample from the Replay Buffer (which form the minibatch) when performing a weight update for the Policy Network.

- **learning-starts**

  Value: 10000
  Role: This governs the number of iterations (or moves) of the game that need to be played (and, by extension, the number of transitions stored in the Replay Buffer) before the weights of the Policy Network can start being updated. This number needs to be large enough to ensure that sufficient transitions are stored in the Replay Buffer such that a random subset thereof has minimal dependence between its items.

- **learning-freq**

  Value: 5
  Role: This governs the number of iterations between each weight update for the Policy Network. A value greater than 1 serves to better inform a later weight update: more meaningful gradients will be computed, which will better guide the Policy Network to convergence.

- **use-double-dqn**

  Value: True
  Role: This governs whether the principle of Double Q-Learning is used. There is inherent error associated with Neural Networks, especially when they are used for function approximation. Double Q-Learning is a technique which introduces tighter bounds on this error, meaning the

final Q-values are more accurate overall. We use this principle when determining the target Q-values during the weight update of the Policy Network. Traditionally, the Target Network would be evaluated on the "next state" and the maximum value from the output nodes taken as the Q-value for the "next state". With Double Q-Learning, the Policy Network is used to determine the action to take given the "next state", and then the Q-value of the associated action from the Target Network is used as the Q-value for the "next state".

- **target-update-freq**

  Value: 1000
  Role: This governs the number of iterations between successive updates of the Target Network's parameters to the Policy Network's parameters. This value needs to be sufficiently high so as to prevent the problem of highly-non-stationary targets causing the Policy Network to fail to converge. This value should not be so large so as to cause overly-slow convergence of the Policy Network.

- **eps-start**

  Value: 1.0
  Role: This is the initial (and largest) value of ϵ in an ϵ-greedy policy. Initially, the portion of time spent exploring is larger than during later episodes.

- **eps-end**

  Value: 0.01
  Role: This is the smallest the value of ϵ in an ϵ-greedy policy can decay to, after which ϵ becomes fixed. This value ensures that the agent will always explore during training, but will be able to gradually exploit more of its knowledge as it learns more.

- **eps-fraction**

  Value: 0.1
  Role: This governs the fraction of the total number of steps (for which we are training) over which we linearly decay ϵ, before fixing it. This decay happens at the start of training and is fixed during the latter stages of training.

- **print-freq**

  Value: 1
  Role: This is not a hyper-parameter, but merely governs the interval between the printing of episodes' performance statistics.

References

[1] R. Lab, *DQN*, https://github.com/raillab/dqn, 2018.