DQN on Atari Games and An Initial Approach to Transfer Learning

Joseph Denby **Pranav Subramaniam** **Jing Yu**

Abstract

Deep Q-Networks (DQN) is a Q-learning method that learns a model which is a convolutional neural network (CNN). It is a popular method because it is model-free, and does not require context-specific information (in other words, it does not require the creation of hand-crafted features to represent the data). In addition, transferring neural network models between applications is also desirable for two reasons: evaluating the robustness of a model by evaluating its performance on other environments, and reducing time required to train a new model. In this paper, we implement DQN on Breakout. We also transfer the DQN model from Breakout to Pong and measure its performance. The results show that our implementation works well on Breakout, but does not transfer well to Pong.

1 Introduction

1.1 Motivation for DQN

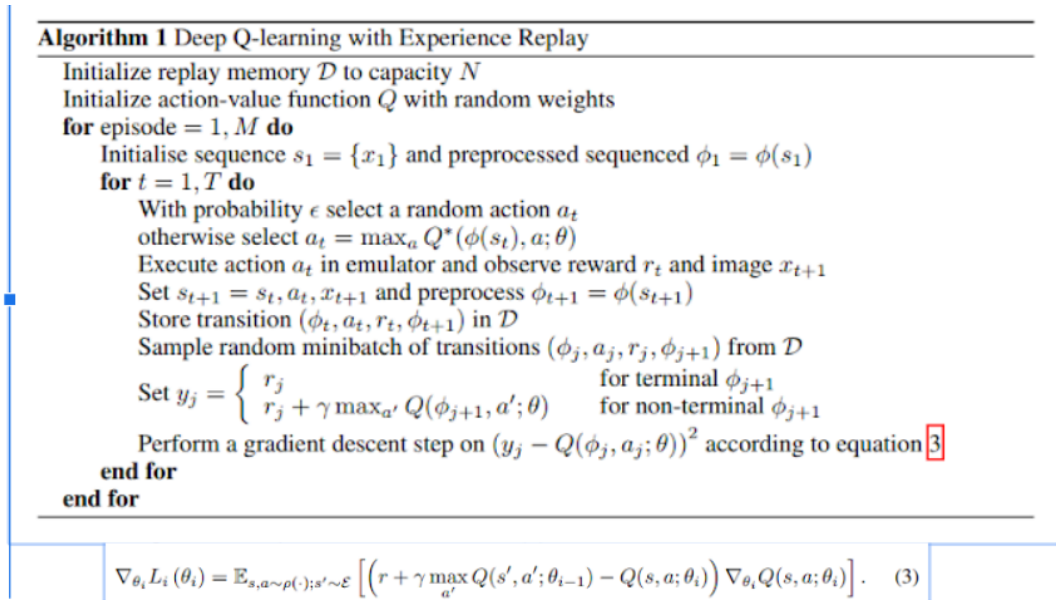
The basis for our implementation comes from [2]. We first explain the motivation behind DQN and why it was considered better than its alternatives according to this paper. The learning task of controlling agents directly from high-dimensional sensor data is a difficult problem with a long history. Before algorithms like DQN, most successful reinforcement learning (RL) applications that operated on such high-dimensional domains relied on hand-crafted features combined with linear value functions or policy representations, as mentioned by [2]. Clearly, such methods depend heavily on the quality of feature representation, making it difficult to compare such models in a quantitative way.

Advances in deep learning to bypass the need for hand-crafted features for representing high-dimensional sensor data by learning high-level features from raw sensor data had been developed. DQN is an algorithm that leverages these deep learning advances for RL.

1.2 Motivation for Transfer Learning

Transfer learning refers to a general technique in which one trains an ML model on one dataset and then uses the model for prediction on a new dataset, optionally reusing the trained model as a starting point for training a new model on the new dataset (this is called fine-tuning). There are two main reasons why transfer learning is useful:

1. Transfer learning is important for testing the robustness of ML models, especially deep neural networks.
2. If a ML model is found to be robust, it may be possible to use the model with minimal to no retraining on another dataset. That is, one may not need to completely train a new model. Since training time for deep learning algorithms (and RL algorithms) are notoriously high, this reason is of practical use.



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ 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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right]. \quad (3)$$

Figure 1: DQN Algorithm with Experience Replay, as described by Mnih et al.

1.3 Our Contributions

In our final project, we implement the DQN algorithm and transfer learning using the Atari game environments from OpenAI Gym, as this is the data used in [2]. Specifically, we implement DQN on the Breakout environment, and then observe the results of transfer learning on the Pong environment. We discuss the DQN algorithm, and details related to our implementation of it, in 2 and 3 respectively. In 3, we also discuss the details of our implementation for transfer learning. Then, in 4 and 5, we show and discuss the results of our DQN implementation on Breakout, and the results of transfer learning to Pong.

2 The DQN Algorithm with Experience Replay

2.1 High-level Ideas of DQN

Essentially, the DQN algorithm trains a neural network using a variant of the Q-learning algorithm discussed in class which uses stochastic gradient descent to update the weights. However, in addition to implementing this variant of Q-learning to update the weights of the neural network, most implementations of DQN (including ours) use experience replay. Experience replay randomly samples previous transitions, smoothing the training distribution over many past behaviors [2]. This helps alleviate the problems of correlated data and non-stationary distributions. Intuitively, for the Q-learning method there is the risk that successive states will be highly similar, so the neural network will completely forget about what it is like to be in a state that it has not seen in a while. Experience replay prevents this by showing old frames to the network.

2.2 The Algorithm

The DQN algorithm with experience replay, as described by [2], is shown in Figure 1. Firstly, note that the DQN algorithm above shares the properties of regular Q-learning: it is model-free, meaning it does not explicitly learn a model for its environment, and it is off-policy, meaning it greedily finds the action that maximizes the Q value while taking a sufficient number of exploratory actions. Note that this off-policy property is generally implemented using ϵ-greedy approaches. We explain our ϵ-greedy approach in section 3, but note that Figure 1 also accounts for the use of an ϵ-greedy approach.

2.3 Our Implementation

As mentioned in the introduction, we implemented DQN to learn to play (and win) Atari games, whose environments are provided from OpenAI's `gym` package. In particular, we used the `BreakoutDeterministic-v4` environment to train a neural net using DQN, following some of the details of the implementation specified by [2]. For Atari game environments, the input and output to the DQN algorithm is specified as follows:

1. State: Raw pixels (with some preprocessing)
2. Input: Stacks of 4 frame-action pairs
3. Output: Q function estimating future rewards for state-action pairs (equivalent to a policy from value iteration)

Note that both the 4th line of the algorithm described in Figure 1 and the state space we just described mention preprocessing. We preprocess the given Atari frames, which are 210x160 pixel images with a 128 color palette, by downsampling it to 105x80 pixels, converting the image from RGB to grayscale. The reason we do this is primarily because using the entire 210x160 pixels is computationally demanding.

Also, the reason why we describe the input as stacks of 4 frame-action pairs is because we are using the `BreakoutDeterministic-v4` environment, which skips every 4 frames, as opposed to using the `Breakout-v0` environment, which skips 2, 3, or 4 frames at random. The reason why these environments skip frames at all is because one frame does not always correspond to one timestep: an action may take more than a single frame to execute. The reason we use the `-v4` version of Breakout over the `-v0` version of Breakout is not only to be consistent with [2], but also because updating Q every 4 frames is more efficient than updating every frame.

Note that both the preprocessing step mentioned above and the frame skipping detail can be risks to DQN. Downsampling from RGB to grayscale can change the color of important components of the game in ways that hinder the agent's learning. For example, downsampling may turn the color of the ball in Breakout into the color of the background. Also, [2] note that skipping every four frames causes the laser to disappear in the Atari game, Space Invaders. However, no such problems with skipping four frames have been documented for Breakout, and no problems have been noted with downsampling from RGB to grayscale. Therefore, we assume these risks are not substantial.

Next, we explain how we implement experience replay. Note that in the algorithm in Figure 1, on the 11th line, we sample a minibatch of transitions from the memory buffer and perform gradient descent to fit the batch (that is, to update the weights of the neural net according to the new Q values). In our case, we sample 32 stacks of 4 frames from a memory buffer consisting of all previous elements (up to 600000 in our case) where each element is a tuple as described in line 6 of the algorithm in Figure 1, with an extra element indicating whether the current state is terminal or not. We also fill our memory buffer with 50000 random actions before beginning to update and use our estimate for Q.

3 Hyperparameters

Some parameters, such as experience replay sample size (also called minibatch size) were discussed above. However, there is a substantial number of parameters to tune in order to run DQN effectively. We go over them here, and present our choices for them.

3.1 γ : the Discount Factor

γ determines how much to value future rewards over immediate rewards. In our case, $\gamma = 0.99$. We choose this γ for two reasons:

1. Experience Replay is ineffective with a low γ, because in general, a low γ does not put enough weight on the frame we are trying to predict. Specifically, in a stack of four frames, a low γ would cause the agent to use only the immediate frame, and not the three other frames.
2. We most likely need a high γ to learn complex strategies in a game.

3.2 ϵ : the Exploration Likelihood

ϵ is the probability that, in a given iteration of the DQN algorithm, we will execute a randomly sampled action instead of executing an action based on the current estimate for Q (that is, ϵ is the ϵ from the ϵ-greedy method discussed in class). In general, we want to use ϵ-greedy methods to control how much the agent explores the state space. This is because an agent that does not have a good idea about the state space may not be able to exploit the state space to maximize its expected rewards. Therefore, we initially want the agent to explore much more than it exploits, and as iterations pass and the policy improves, we want to exploit. That is, we want ϵ to be large at first, but as iterations pass, we want ϵ to be small. To achieve this, we initialize ϵ to 1 and anneal ϵ linearly over 600,000 iterations (excluding the 50,000 iterations required to fill the memory buffer) until it is 0.1, at which point we keep it constant. Besides the reason that it is consistent with [2], the other reason we anneal to 0.1 and not lower, is because we do not perform as many iterations as might be required for the agent to learn a clear strategy. Many of the DQN implementations we found required 100M iterations, or 10 days of training [1]. After that many iterations, if an agent learns a sophisticated strategy, it may no longer need to explore, and so ϵ can be lower than 0.1.

3.3 Transfer Learning Details

In our case, transfer learning is simple: for each stack of 4 Pong frames, we choose the action that maximizes the Q-value, as predicted by our model from Breakout. However, the action spaces of Pong and Breakout differ: Breakout has 4 actions, whereas Pong has 6. Since Breakout's action space is a subset of Pong's action space, we give the Breakout model a one-hot encoded vector of actions for Pong that does not include entries for the two actions Breakout does not use.

4 Results

The implementation in [2] includes roughly five million training updates; due to time limitations, we were only able to train our model for 600000 iterations. As such, our model did not train enough to come close to the performance demonstrated in [2]. Nevertheless, our comparably small training time captures an initial slice of the results presented from elsewhere in the literature.

Since, at first, we are interested in replicating (a portion of) the results presented in [2], we evaluated our models by probing total reward per episode (i.e., game) and average Q-value per iteration. The former metric serves to directly track the model's ability on the game in question, while the latter provides a more direct picture of the model learning the game.

The total reward per game presented in Figure 2 clearly increases over the course of training. The model is only able to break roughly one brick for the first 1500 games, after which a dramatic boost in performance produces an average of 3 points per game. While the gains are not as dramatic as those presented in [2] (where the model was eventually able to score hundreds of points per game), they are substantial enough to clearly indicate successful learning.

The average Q value presented in Figure 3 corroborates this finding. Here, Q values tend dramatically upward between 300000 and 400000 iterations, indicating significant learning within that span of training. The initial drop within the first 110000 iterations is likely due to random model initialization - at first, the model's assessment of the value of state-action pairs is completely random and inaccurate, so these result indicate backing away from initial 'overconfidence'.

To visually assess final performance, we recorded the model playing a game of Breakout at the end of its training - the video is hosted here. The model's actions are not entirely comprehensible and certainly do not indicate mastery of the game. Due to its still nascent understanding of the game, it has likely only learned somewhat reliable actions for a small window of the game.

As mentioned above, transfer learning, which in our case involves training a DQN model on one Atari game and playing another, often includes some fine-tuning, whereby a trained model has some portion of its architecture frozen, but continues to learn on the new game for a small period of time. Due to time constraints, we were unable to implement any fine-tuning time; instead, we applied our model directly to Pong, another Atari game. This involved simply feeding frames offered by the Pong environment to the DQN trained on Breakout, predicting Q values for the action space, and picking actions based on those predictions. A visualization of this transfer model's performance on

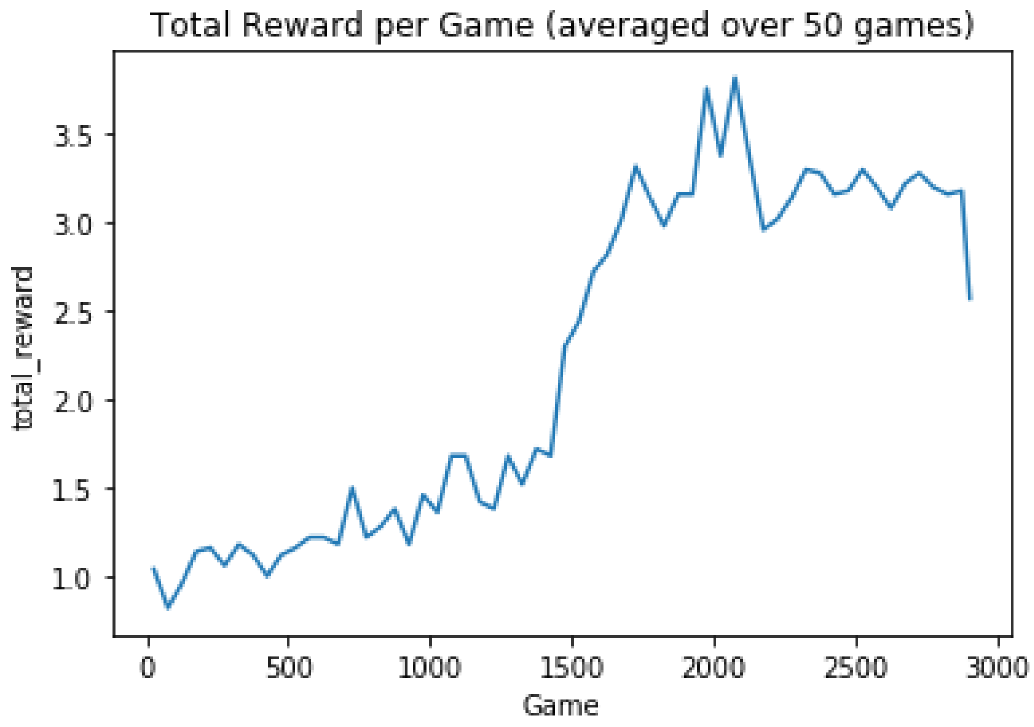


Figure 2: Total Reward per Game

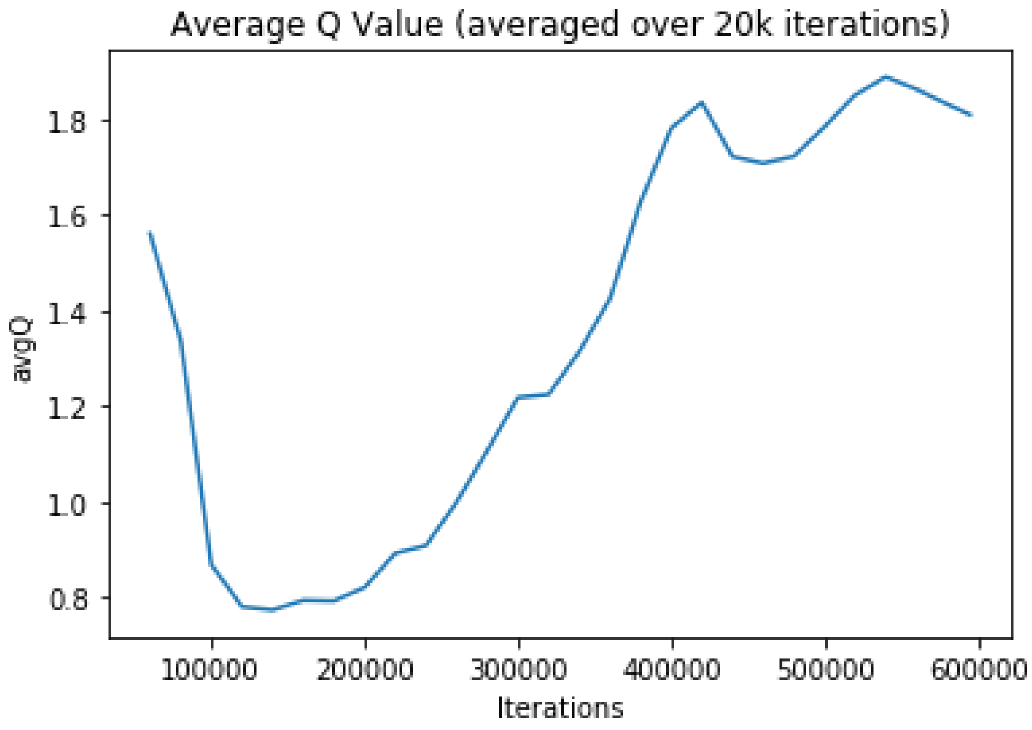


Figure 3: Average Q Value per Iteration

Pong is hosted here. Clearly, transfer learning is unsuccessful between these two games, as the agent repeatedly picks the same suboptimal action, yielding performance that is worse than chance. This is likely due to the clear differences between the observation spaces for each game - the model is trained on specifically Breakout's environment, so its procedure does not immediately generalize to Pong's environment. Some fine-tuning would surely ameliorate this issue by allowing the model to adjust to the novel observation space.

5 Discussion

While, at face value, our results fail to directly replicate those of [2] and do not demonstrate successful transfer learning, we nevertheless present some useful takeaways revelant to future efforts. First, implementing DQN successfully is extremely difficult, as the algorithm requires extensive training time to achieve even mediocre results on complex control tasks. Moreover, successful (and expedient) training involves careful consideration of both input data and hyperparameters.

Next, achieving decent performance on isolated Atari games with DQN is hard enough, but transferring learning between Atari games is even harder. DQN attempts to 'solve' games with complex observation spaces through function approximation - instead of trying to learn strict state-action evaluations (through, e.g., Vanilla Q Learning) which, DQN learns an abstract set of features that can map collections of similar observations to similar actions. Successful transfer learning requires learning features that are abstract enough to apply across games, which may have vastly different visual representations, objectives, and action spaces. Our results reinforce the idea that this is no trivial task, even across games with relatively sparse observation spaces. Sophisticated artificial agents trained through reinforcement learning should demonstrate performance that is robust across environments - transfer learning remains a key feature of intelligence that demands further study.

References

[1] [n. d.]. Beat Atari with Deep Reinforcement Learning! `https://becominghuman.ai/beat-atari-with-deep-reinforcement-learning-part-2-dqn-improvements-d3563f665a2c`. ([n. d.]). Accessed: 2019-06-06.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.