Reinforcement Learning with Deep Q-Networks: Attempting the Cartpole Problem

Sai Madhavan G - IMT2021101

March 27, 2024

Abstract

This report presents a Deep Q-Network (DQN) approach to solving the Cartpole problem in the OpenAI Gym environment as a part of the assignment for AI 832. I discuss the design and implementation of the DQN algorithm, experimentation with hyperparameters, and adaptation to learn from input images. Results and observations from the training process are analyzed and discussed. The code and accompanying observations can be accessed on GitHub[1].

1 Introduction

Reinforcement Learning (RL) has gained significant attention in recent years for its ability to solve complex sequential decision-making problems. The Cartpole problem, a classic RL benchmark, involves balancing a pole on a cart by applying appropriate actions. In this report, I employ a Deep Q-Network (DQN) to tackle this problem, aiming to achieve stable and effective control policies.

2 Background

2.1 Deep Q-Networks (DQN)

DQN is a deep learning-based approach to RL that combines Q-learning with neural networks. It uses a neural network to approximate the Q-function, which represents the expected cumulative reward for taking a particular action in a given state.

2.2 Cartpole Problem

The Cartpole problem is a classic benchmark problem in the field of reinforcement learning (RL). It involves balancing a pole (attached by a joint to a cart) in an upright position by moving the cart left or right along a frictionless track. The goal is to keep the pole balanced for as long as possible, which requires making continuous adjustments to the cart's position based on the pole's angle.

In the Cartpole environment, the state is represented by a four-dimensional vector, consisting of the cart's position, the cart's velocity, the pole's angle, and the pole's angular velocity. The action space consists of two discrete actions: pushing the cart left or right.

The task is considered successful if the pole remains upright within a predefined threshold (-12° to 12°) for a certain duration of time steps (1000 steps). The environment provides a reward of +1 for each time step the pole remains upright. The episode terminates if the pole falls beyond a certain angle threshold or if the cart moves outside the frame.

3 Methodology and Implementation

The code for the below described implementation can be found at **dqn.py** in the repo [1].

3.1 Environment

I utilized the "CartPole-v1" environment [2] of the OpenAI gym package on python. I limited the total number of steps per episode to be 1000, thereby making the task episodic.

3.2 Algorithm

The DQN training loop was adapted from DeepMind's seminal work [3] on training an RL agent to play the Atari game using deep Q-learning. Below is the pseudocode of my implementation:

Algorithm 1 DQN Training Loop

```
0: procedure TrainingLoop
0:   Initialize DQN parameters
0:   Initialize replay buffer
0:   Initialize neural network
0:   for each episode do
0:     Reset environment and get initial state
0:     Set terminal state to False
0:     while not terminal state do
0:       Select action using epsilon-greedy policy
0:       Take action and observe next state and reward
0:       Store experience in replay buffer
0:       Train the network
0:     end while
0:     Store episode reward and average Q value
0:     Update tensorboard (if enabled)
0:     Perform inference if it's time
0:     Check for termination condition
0:   end for
0: end procedure=0
```

3.3 Neural Network

Given the relatively small dimensions of input, which is the size of state space (4) and output, which is the size of the action space (2), I used a small ANN consisting of 2 hiddnen layers consisting of roughly 6000 parameters. The ReLU activation functions were used at each neuron. Dropout was not used as it significantly impaired the training process, even for small values. I speculate this was due to the sparse nature of data sampled from the replay buffer when compared to datasets of other tasks. The loss function used was the mean squared error function, which was applied between the prediction and estimated Q value. The Adam optimizer was used to make a step after every step of an episode. The implementation was carried out in PyTorch.

3.4 Termination Condition

As mentioned earlier, the total number of steps per episode was capped at 1000. So when an agent achieves a score of 1000 on two consecutive inference runs, I halted training and considered the model to be "converged". If an agent doesn't converge even after 300 episodes, I halt the training as training for more than 300 episodes was not feasible given the time and hardware constraints.

3.5 Metrics Tracked

- **Rewards/episode:** The sum of all rewards accumulated in each episode

- **Inference rewards:** The rewards accumulated when the agent was run in inference mode (without epsilon) every 5 episodes.

- **Inference score:** Average reward on running 10 episodes of inference mode after model reaches termination condition.

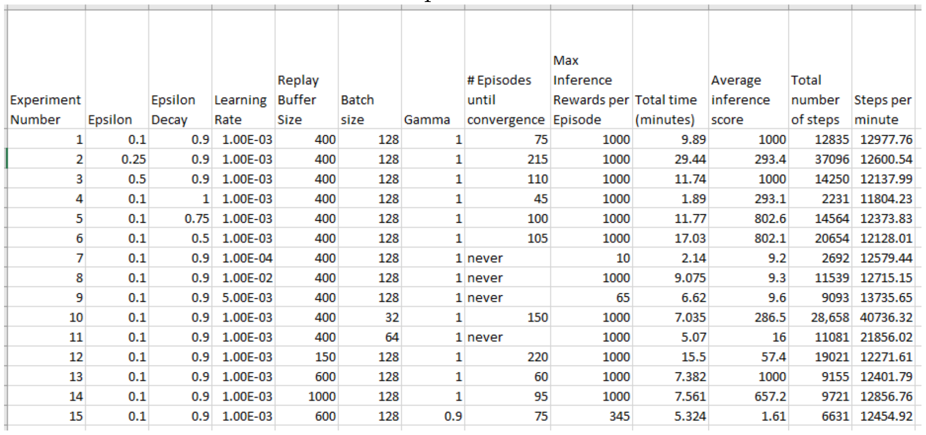
- **Average Q value:** Average of sum of Q-values predicted in each batch in each episode.

  The above, and a few more metrics were tracked and logged real time, using tensorboard. You can view the data across runs by running *tensorboard –logdir=runs* after cloning the repo.

4 Experiments with Hyper-parameters

Since every run with a specific set of parameters took several minutes on average to converge, the comprehensive experimentation spanning across the domains of all the hyper parameters was infeasible. Therefore, the discussion in this section will be an analysis of a few values of the hyper-parameters that will hopefully give us an understanding on how the hyper-paramters affect both the training as well as the results.

Table 1: Experiments' Results



Experiment Number	Epsilon	Epsilon Decay	Learning Rate	Replay Buffer Size	Batch size	Gamma	# Episodes until convergence	Max Inference Rewards per Episode	Total time (minutes)	Average inference score	Total number of steps	Steps per minute
1	0.1	0.9	1.00E-03	400	128	1	75	1000	9.89	1000	12835	12977.76
2	0.25	0.9	1.00E-03	400	128	1	215	1000	29.44	293.4	37096	12600.54
3	0.5	0.9	1.00E-03	400	128	1	110	1000	11.74	1000	14250	12137.99
4	0.1	1	1.00E-03	400	128	1	45	1000	1.89	293.1	2231	11804.23
5	0.1	0.75	1.00E-03	400	128	1	100	1000	11.77	802.6	14564	12373.83
6	0.1	0.5	1.00E-03	400	128	1	105	1000	17.03	802.1	20654	12128.01
7	0.1	0.9	1.00E-04	400	128	1	never	10	2.14	9.2	2692	12579.44
8	0.1	0.9	1.00E-02	400	128	1	never	1000	9.075	9.3	11539	12715.15
9	0.1	0.9	5.00E-03	400	128	1	never	65	6.62	9.6	9093	13735.65
10	0.1	0.9	1.00E-03	400	32	1	150	1000	7.035	286.5	28,658	40736.32
11	0.1	0.9	1.00E-03	400	64	1	never	1000	5.07	16	11081	21856.02
12	0.1	0.9	1.00E-03	150	128	1	220	1000	15.5	57.4	19021	12271.61
13	0.1	0.9	1.00E-03	600	128	1	60	1000	7.382	1000	9155	12401.79
14	0.1	0.9	1.00E-03	1000	128	1	95	1000	7.561	657.2	9721	12856.76
15	0.1	0.9	1.00E-03	600	128	0.9	75	345	5.324	1.61	6631	12454.92

4.1 Experiments with ϵ

Experiments 1-3 were experiments with different initial values of ϵ, namely, 0.1, 0.25, 0.5. ϵ denotes the chance that a random action is chosen in each step while training. From Table 1, it is clear that a value of 0.1 seems to be the best in terms of both time taken for convergence as well as inference time performance.

However, there doesn't seem to be a linear correlation as experiment 2 performed worse than experiment 3 in spite of having a lesser ϵ value.

4.2 Experiments with ϵ-decay

Experiments 1 and 4-6 were experiments with different initial values of ϵ-decay, namely, 0.9, 1, 0.75, 0.5. ϵ-decay is the factor that is periodically (every 10 episodes) multiplied with ϵ. The best performing experiment was experiment 1, indicating that a slow decay of 0.9 works well with a small epsilon value of 0.1. Time-wise, although experiment 4 was the quickest among all the experiments, it is clear from the low inference score that the training terminated earlier than when it was supposed to. This is probably due to the un-restrained stochasticity that persisted throughout the experiment. The inference scores of lesser values are also poor, implying 0.9 is around the "Goldilocks" zone for training.

4.3 Experiments with learning rate

The earlier experiments were all conducted with a constant LR of 0.001 in the Adam optimizer. So, experiments 7, 8 and 9 focused in trying different learning rates. Experiment 7 used a very small LR of 10^{-4}. This value turned out to be very low since not only the experiment did not converge in 300 episodes, but the max reward on inference never exceeded 10, indicating not enough was learnt from each batch. Experiment 8 used a large LR value of 0.01. This experiment too, never reached convergence. The maximum reward of 1000 was reached in just 35 episodes, however, the optimizer quickly "over-shot" and never reached convergence again as apparent from the plot (Image 1). Experiment 9 used a LR of $5{\cdot}10^{-3}$, which too, yielded underwhelming results. This indicates that 10^{-3} seems to be an ideal value for learning rate as it balances the trade off between

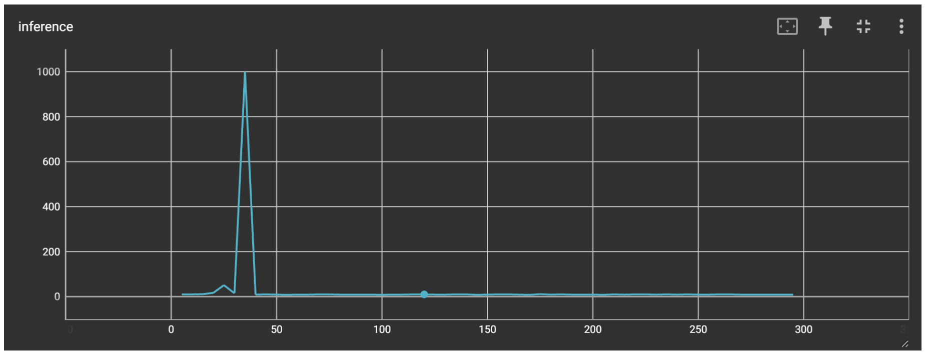


Figure 1: Inference rewards for experiment 8 (lr = 0.01)

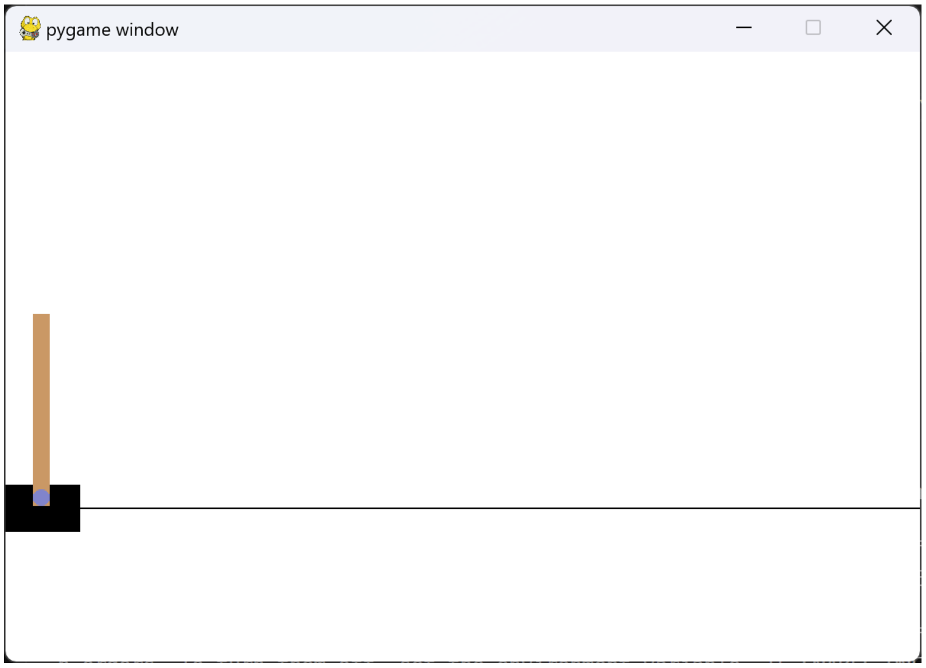


Figure 2: The cart after stabilizing in experiment 10

4.4 Experiments with batch size

In each update step, a random subset of the replay buffer was taken for applying the algorithm. I experimented with 3 values of batch size: 32, 64 and 128. I could not exceed 128 due to hardware constraints. From experiments 10 and 11, it is also clear that decreasing the batch size is adversely affecting the performance. However, a batch size of 32 seems to perform better than 64. Interestingly, in experiment 10, the cart seemed to almost always drift off to the left before stabilizing as shown in Figure 2 or fall instantly. This seems to suggest that the small sample size seemed to bias the agent heavily towards episodes where the cart drifted to the left. The training episodes seemed to be significantly high in both experiments 10 and 11, since relatively little information was learnt in each backprop step. However, the training time in term of number of steps taken per minute is significantly faster. The ideal batch size seems to be as high as possible, which in my experiments turned out to be 128.

4.5 Experiments with buffer size

The replay buffer is a queue-like data structure that stores tuples containing current state, action, reward and next state information. In each step, a random

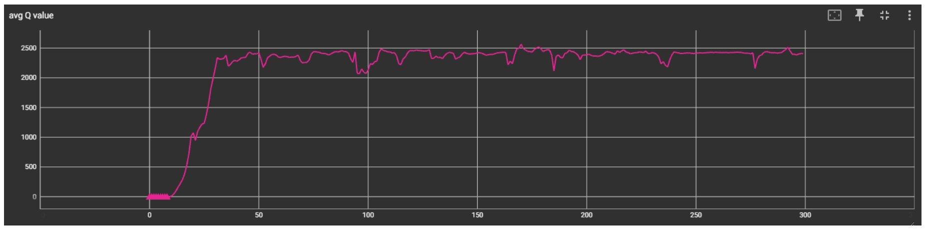


Figure 3: Average Q value over episodes while running experiment 15

batch is sampled from here. Keeping the batch size constant (128), I tried runs with four different sizes of replay buffer: 150, 400, 600 and 1000. From experiment 12, it is clear that a low buffer size value is very bad. This is possibly due to the fact that the agent is only learning from the most recent steps and a lot of the randomness or exploration is the lost. Even high values of buffer size like 1000 (experiment 14) seem to suffer from poor inference time performance, possibly because a large "context-length" can inhibit the agent improving if old mistakes are sampled repeatedly. The ideal values seem midway, as values of 400 and 600 performed very well, with 600 outperforming 400 slightly.

4.6 Experiments with γ

Since I have made sure that the episodes cannot exceed 1000 steps, a γ value less than one is not required for preventing the infinte return problem. Therefore, a value of $\gamma = 1$ seems to work the best. Experiment 15, which used a $\gamma = 0.9$ never reached convergence. The plot visualizing it's average Q values over episodes (Figure 3) platued quite quickly due to the γ factor and the agent was never able to converge.

4.7 Conclusion

From the above experiments, we can make broad conclusions on the following:

- ϵ **value:** Should be less (0.1 in my experiments)
- ϵ**-decay value:** Should be quite high, but not 1 (0.9 in my experiments)
- **Learning rate value:** Can be around 0.001, although this might vary with other optimizer configurations
- **Batch size:** Should be as high as possible, given the hardware constraints (128 in my experiments)
- **Replay buffer size:** Should be around 4-5 times the size of the batch (600 in my experiments)
- γ **value:** Can be 1 for episodic tasks

5 DQN with Image Inputs

In this section, I detail my attempts at recreating the above DQN training loop and inference, but when the image of the frame is the state instead of the 4-value vector.

5.1 Approach

I replicated the approach uded in the Atari paper [3]. I decided to use a CNN as a feature extractor, after which, I passed a sequence of these features to a neural network identical the earlier set up.

First I created a dataset, with the cart and pole in different postions and the label being a tuple of the normalized value of the position of the cart and the angle of the pole. This dataset had 20,000 train examples and 5,000 test examples.

For the CNN, I finetuned a ResNet18 model pretrained on the ImageNet dataset, on this dataset that I created. I used MSELoss between groundtruth and predicted value so that the model could perform regression. I stopped training once the validation loss reached order of 10^{-4}.

I then froze the weights of this CNN, took the values of the last layer of this model and passed their sequence to a neural network identical to the mode in the earlier section. Then, the training loop was similar. The weights of the CNN were not updated. A sequence length of 4 gave me the best results. The other hyperparameters of the experiment were identical to the best hyperparameters from Section 4.7.

5.1.1 Results

Although the agent never reached a score as high as 1000, I had to halt it at around 250 episodes due to time constraints.

- **Max Inference Rewards per Episode:** 330
- **Episodes taken:** 230
- **Time taken (minutes):** 35
- **Average inference score:** 135.40
- **Total steps:** 13953
- **Steps per minute:** 400 (approx.)

5.2 Other Experiments:

Although I didn't have the time to experiment with the hyper-parameters when training, I did perform around 20 experiments with different design decisions and architectures.

Most notably, I experimented with different values of sequence lengths (2, 4 and 8) and found 4 worked the best. I also experimented with different ways of initializing a sequence, like initializing with all zeros, using a special value etc. I found replicating the first frame four times worked the best.

I also tried to train the network without freezing the feature extractor, i.e., training the feature extractor as well. However, it was both time and computationally expensive and didn't give impressive results in the first 100 episodes.

I also replaced the neural network with a CNN that took a sequence of images concatenated along the x-axis as the input and directly predicted the action. This too, turned out to be very expensive computationally and didn't yield promising results.

6 Demo

To run inference of the best performing models, run the following commands from in the root directory of the repo [1].

```
python3 dqn.py best-params.json inference 10
```

```
python3 image_dqn.py best-image-params.json inference 10
```

References

[1] https://github.com/SaiMadhavanG/rl-assignment-1

[2] https://www.gymlibrary.dev/environments/classic_control/cart_pole/

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).