Sticking the Landing with Deep Q-Learning

Nathan Foote
Ann and H.J. Smead Aerospace Engineering Sciences
University of Colorado at Boulder
Boulder, Colorado
nathan.foote@colorado.edu

***Abstract*—Reinforcement learning environments require different techniques according to the specific state and action space they operate within. One such environment is the LunarLander-v2 environment in which a lunar lander agent is trained to land within a goal zone on a randomly generated lunar surface. The lander has a continuous state space and a discrete action space making it a candidate for DQN. This paper successfully implemented DQN on the LunarLander-v2 environment by training an agent to safely land in the goal zone.**

***Index Terms*—OpenAI, reinforcement learning, Q-learning, neural networks, DQN**

I. Introduction

Gymnasium, developed by the OpenAI company, provides environments to test and refine reinforcement learning algorithms and techniques [1]. These environments are Markov Decision Processes (MDPs) defined by states, actions, and rewards. One subset of environments uses the box2D physic engine created by Oleg Klimov [2]. In this paper, Deep Q-Learning (DQN) will be used to create a neural network agent to act on the LunarLander-v2 environment with the goal of landing in the designated landing zone.

II. Background and Related Work

A. Q-Learning

Q-Learning is a value-based, model-free reinforcement learning method meaning a Q-Learning agent learns the value of taking a particular action in a particular state by directly interacting with the environment. The ”Q” in Q-Learning is the action-value function Q(s,a) which represents the value of taking the action (a) in state (s) and can be represented by Bellman's expectation equation:

$$Q(s,a) = E_{r,s'}[r + \gamma max_{a'} Q(s',a')] \quad (1)$$

which states that the value of a given state action pair is equal to the expected rewards (r) for taking action (a) in state (s) plus the maximum value of all possible expected future rewards. The ”learning” in Q-Learning comes from the incremental estimation of the action value function by applying an incremental update rule with a constant learning rate.

$$\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x}) \quad (2)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a)) \quad (3)$$

Where α is the learning rate. The convergence of the action-value function is guaranteed by adopting an exploration policy such as ε-Greedy [3].

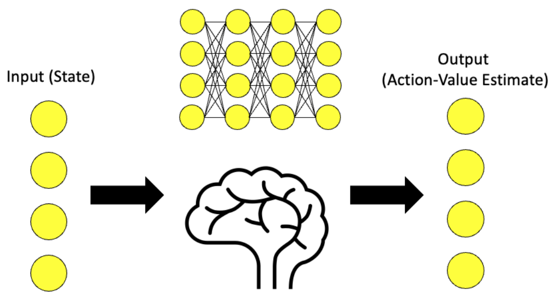


Fig. 1. Crude Neural Network Example

B. Neural Networks

Neural networks are machine learning algorithms that are designed to mimic the structure and function of the human brain. They consist of layers of interconnected nodes, or ”neurons,” that process and transmit information. The process of training a neural network involves adjusting the weights and biases of the connections between neurons, so that the network can learn to recognize patterns in data and make accurate predictions. An example of the structure of a neural network is shown in Fig. 1. A neural network is a differential function that maps inputs x to produce outputs y and parameterized by ϑ.

$$y = f_\theta(x) \quad (4)$$

The parameters of the network ϑ are tuned to minimize a loss function $l(f_\theta(x), y)$. The network and loss function are differentiable allowing for the gradient of the loss function with respect to the parameterization $\nabla_\theta l$ to incrementally improve the parameterization.

C. Deep Q-Learning

Deep-Q Learning is the combination of the Q-Learning reinforcement algorithm and neural networks made famous from Google's ”Human Level Control Through Deep Reinforcement Learning” [4]. In this paper, a deep convolutional neural network was used to approximate the action-value function to achieve human-level performance at playing classic Atari games. The neural network would take in the image of each frame and output a value estimate corresponding to each action possible using an Atari controller. The algorithm for DQN is shown below. The prepossessing steps required for DQN to interpret images are removed as they were not needed for the lander environment.

Algorithm 1 Deep Q Learning Algorithm with experience replay and target network

```
Initialize Experience replay memory $D$ to capacity $N$
Initialize action-value function $Q(s, a)$
with random weights $\theta$
Initialize target action-value function $\hat{Q}(s, a)$
with weights $\theta^- = \theta$
for $Episode = 1, M$ do
    Initialize sequence $s_1 = \{x_1\}$
    for t $= 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = argmax_a(Q_\theta(s, a))$
        Take action $a_t$,observe reward $r_t$ and next state $s_{t+1}$
        Set $s_{t+1} = s_t$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
        Sample random mini-batch of transitions from $D$
        Set target for each transition
$y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
with respect to $\theta$
        Every C steps reset $\hat{Q} = Q$
    end for
end for
```

Notably, there are a few tricks that need to be implemented in order for DQN to work. First, a memory of the agents experience is captured in an experience replay buffer. This buffer collects the experience of the agent up to a fixed size after which a mini-batch of data is randomly sampled for training the neural network. As experience is collected, the oldest experience is thrown away. This solves the issue of sampling from highly correlated experiences. The second trick is to have a fixed target that is a copy of the trained neural network that is only updated after a set amount of steps in the environment. This prevents the trained network from chasing a moving target as it is being updated.

III. Problem Formulation

The LunarLander-v2 environment is a classic reinforcement learning environment consisting of a lander navigating in 2D space in order to land in a designated area of the lunar surface show in Fig. 2.

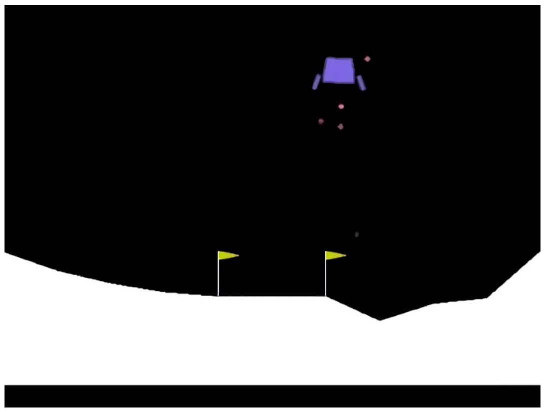


Fig. 2. LunarLander-v2 Environment

A. Action Space

There are two possible action spaces for the LunarLander-v2 environment: discrete and continuous. DQN is only applicable to discrete action spaces so that is what was used. There are four discrete actions the lander can take:

- 0: do nothing
- 1: fire left orientation engine
- 2: fire main engine
- 3: fire right orientation engine

where the engine is either fully firing or completely off for each action.

B. State (Observation Space)

The state of the lander is represented by an 8-dimensional vector:

1) x-coordinate
2) y-coordinate
3) x-linear velocity
4) y-linear velocity
5) Angle
6) Angular velocity
7) Left leg contact
8) Right leg contact

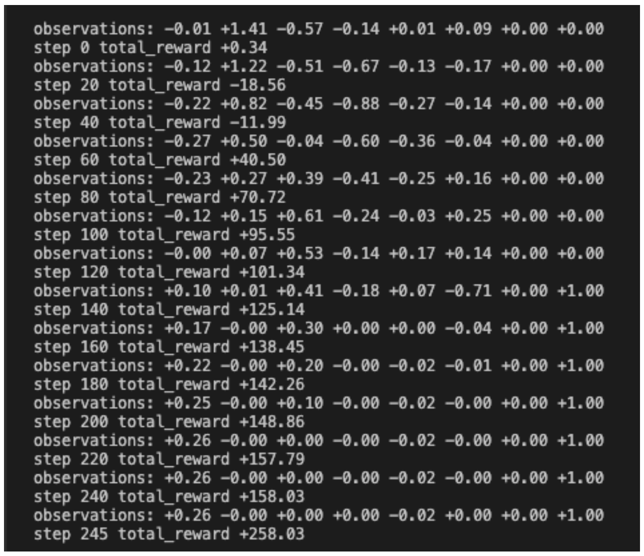
C. Rewards

After every step, a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode. For each step, the reward:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.
- is decreased the more the lander is tilted (angle not horizontal).
- is increased by 10 points for each leg that is in contact with the ground.
- is decreased by 0.03 points each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.
- if the lander crashes, collect a reward of -100 points.
- if the lander comes to rest, collect a reward of +100 points.

D. Transitions

The transition function in the LunarLander-v2 environment is a complex and nonlinear function that depends on the state of the agent, the action taken, and the dynamics of the



```
observations: -0.01 +1.41 -0.57 -0.14 +0.01 +0.09 +0.00 +0.00
step 0 total_reward +0.34
observations: -0.12 +1.22 -0.51 -0.67 -0.13 -0.17 +0.00 +0.00
step 20 total_reward -18.56
observations: -0.22 +0.82 -0.45 -0.88 -0.27 -0.14 +0.00 +0.00
step 40 total_reward -11.99
observations: -0.27 +0.50 -0.04 -0.60 -0.36 -0.04 +0.00 +0.00
step 60 total_reward +40.50
observations: -0.23 +0.27 +0.39 -0.41 -0.25 +0.16 +0.00 +0.00
step 80 total_reward +70.72
observations: -0.12 +0.15 +0.61 -0.24 -0.03 +0.25 +0.00 +0.00
step 100 total_reward +95.55
observations: -0.00 +0.07 +0.53 -0.14 +0.17 +0.14 +0.00 +0.00
step 120 total_reward +101.34
observations: +0.10 +0.01 +0.41 -0.18 +0.07 -0.71 +0.00 +1.00
step 140 total_reward +125.14
observations: +0.17 -0.00 +0.30 +0.00 +0.00 -0.04 +0.00 +1.00
step 160 total_reward +138.45
observations: +0.22 -0.00 +0.20 -0.00 -0.02 -0.01 +0.00 +1.00
step 180 total_reward +142.26
observations: +0.25 -0.00 +0.10 -0.00 -0.02 -0.00 +0.00 +1.00
step 200 total_reward +148.86
observations: +0.26 -0.00 +0.00 -0.00 -0.02 -0.00 +0.00 +1.00
step 220 total_reward +157.79
observations: +0.26 -0.00 +0.00 -0.00 -0.02 -0.00 +0.00 +1.00
step 240 total_reward +158.03
observations: +0.26 -0.00 +0.00 +0.00 -0.02 +0.00 +0.00 +1.00
step 245 total_reward +258.03
```

Fig. 3. LunarLander-v2 acting with heuristic policy

environment. When the agent takes an action, the environment updates the state of the agent based on the laws of physics. The new state of the agent is determined by:

- The previous state of the agent
- The action taken by the agent
- The gravitational forces acting on the agent
- The thrust forces generated by the agent's engines
- The contact forces between the agent and the landing pad (if any)

The environment is terminated when the lander crashes (the lander body gets in contact with the moon), the lander gets outside of the viewport (x coordinate is greater than 1) or the lander is not awake. From the Box2D docs, a body that is not awake is a body that doe not move and does not collide with any other body.

E. *Success and Heuristic Policy*

The environment is considered solved upon receiving a reward of 200 points. A heuristic policy is provided with the LunarLander-v2 environment that regularly receives an award above 200 as shown in Fig. 3 and demonstrates the behavior the agent should exhibit after training. The heuristic policy follows a few principles: keep the angle of the lander pointing toward the center, move the center of the screen, move to a point above the goal, and then slowly descend to the surface.

IV. Solution Approach

Implementing DQN came in two parts: the hyper-parameters used for the algorithm and how the algorithm was implemented in Julia.

A. *Hyper-parameters*

The hyper-parameters for DQN are shown below:

- Episodes = 5000
- Steps = 5000
- Experience Replay Buffer Size = 10,000
- Batch size = 64
- Learning Rate = 0.001
- Discount = 0.99
- Epsilon = $max(0.001, 1.0 - (0.9 * (step/500, 000)$

1) Episode: An episode is a complete sequence of interactions between the agent and the environment, starting from an initial state and ending in a terminal state. In the LunarLander-v2 environment, an episode is completed when the agent lands the Lunar Lander safely on the landing pad or crashes it, or when a maximum number of steps have been taken. A maximum number of episodes of 5000 was chosen.

2) Steps: A step is a single interaction between the agent and the environment, where the agent takes an action based on the current state and receives a reward and a new state from the environment. A maximum of 5000 steps in the environment was chosen. Normally, the LunarLander-v2 environment will return a "truncated" Boolean value after 1000 steps to terminate the environment. Testing with the implementation in Julia found it to be easier and faster to not check for this value. Investigating how to incorporate the "truncated" value into the training may be done in the future.

3) Experience Replay Buffer Size: The experience replay is limited to a set size to reduce the amount of storage required to store the agent's experience. If the replay has reached its size limit, the oldest experience is thrown away as new experience is gained. Google's implementation of DQN used a buffer size of 1,000,000 but 10,000 was chosen to accommodate the limitations of the computer that was used.

4) Batch size: The batch size is a the size of the data that is randomly sampled from the experience replay buffer. The random sampling from the buffer solves the problems of highly correlated samples and size-1 batches that are inherent in basic DQN. The batch size is typically small when compared to the size of the experience replay buffer. When Schaul et al. proposed the new technique of prioritized experience replay, they used a minibatch size of 60 for a buffer size of 1,000,000 [5]. A minibatch size of 64 was chosen to for this implementation.

5) Learning Rate: The learning rate dictates how quickly the agent updates the weights of the neural network that is approximating the action-value function. The training preformed on the neural network utilized the ADAM optimizer which stands for Adaptive Moment Estimation [6]. ADAM adaptively adjusts the learning rate to converge faster and more reliably than stochastic optimization algorithms. A learning rate of 0.0001 was chosen for the ADAM optimizer.

6) Discount: The discount, γ, discounts the value of expected future rewards and is set to 0.99 per tradition.

7) Epsilon: It is important for the neural network to start with an ε of 1.0 to explore randomly according to the ε-Greedy policy. The ε is gradually decayed as the network is trained and has learned more about the environment.

8) Network Structure: Previous work on the LunarLander-v2 problem has been conducted with DQN [7]. Gadgil et al. used a 3 layer network with 128 neurons in the hidden layers,

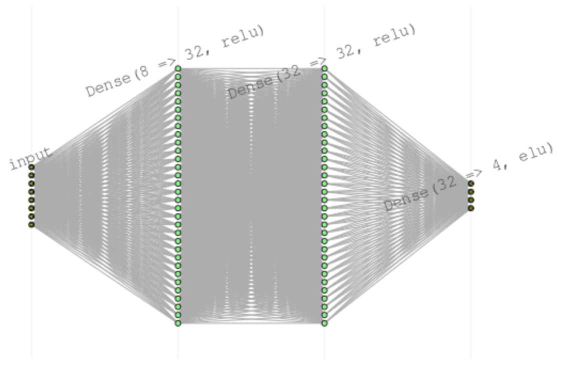


Fig. 4. LunarLander Neural Network

ReLU activation for the hidden layers, and LINEAR activation for the output layer. However, their network was trained to handle more uncertainty in the environmental conditions than the normal LunarLander-v2 problem. Therefore, the same structure was used for this paper but with 32 neurons instead. A picture of the structure that was used can be found in Fig. 4.

B. *Implementation in Julia*

The implementation in Julia involved three main steps. First the buffer was populated up to capacity by taking steps in the environment by executing random actions and collecting the transition information.

Next, an experience replay function is used in the main training loop of the algorithm to execute actions according to an ε-greedy policy. The function first checks if the buffer is at capacity. If the buffer is at capacity, the first element in the buffer is removed. Then the buffer simulates a step in the environment and then adds it to the buffer.

Lastly, a random minibatch of data is taken and sent to a training function that utilizes the Flux.jl machine learning library to perform the training on the network.

A few additional tricks were required to achieve the goal of the agent landing on the landing pad. There is the concept of global steps and episode steps. The global steps are the total number of steps that the agent has taken in the environment and episode steps are the number of steps taken per episode. This is important because the target network must be updated according to the number of global steps taken during training which is not clear in how the algorithm is written.

Also, Google's DQN agent was not trained every step to further reduce the correlation of data the network was training with. This update frequency was implemented in Julia by training the network every four steps in the environment.

The last additional trick that was needed was to keep a record of the average rewards and save the network that was either had enough score to pass or that had the best reward. DQN showed great instability while training and would cycle between converging on negative and positive rewards. So it was important to determine if a "good enough" network was trained and stop the program when that threshold was reached. Because the goal for this project was to land in the goal zone with the agent, a threshold score of 100 was chosen which corresponds to the reward collected upon coming to rest on the surface.

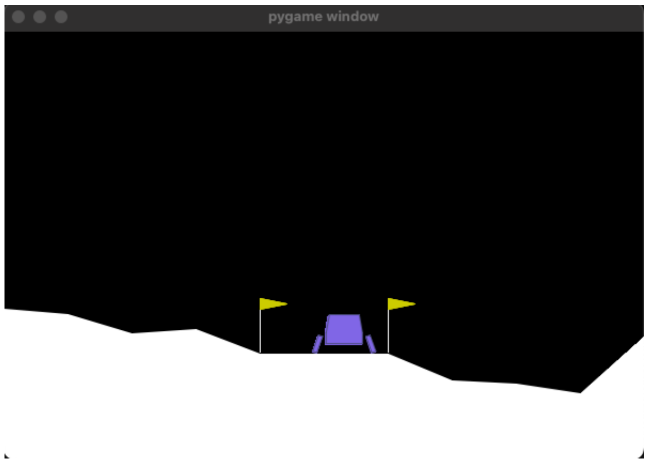


Fig. 5. Successful Agent Landing

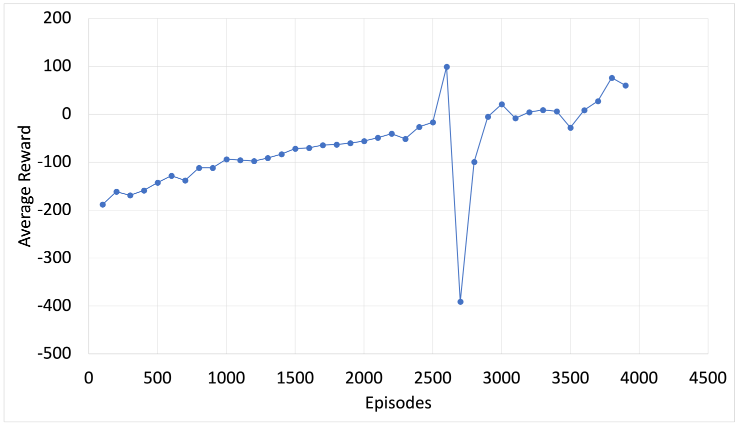


Fig. 6. Reward vs Episodes showing instability during training

V. Results

DQN was successfully implemented on the LunarLander-v2 environment by training the agent to land in the goal zone. A picture of the agent's successful landing can be shown in Fig. 5.

After much troubleshooting, the solution approach outlined in section IV was conducted. The first set of promising results occurred when the code didn't save the best preforming network, but it gave a helpful insight to the nuances of DQN. A plot of a running 100 episode average reward is shown in Fig. 6.

The reward has an overall upward trend showing the network learning until around 3000 episodes. The agent began to deviated from the high reward behavior, even collecting negative rewards even after 4000 episodes of training.

With this information a new network was trained and its results can be shown in Fig. 7. This network implemented the threshold reward and terminated after the agent collected an average reward above 100 points over 100 episodes. The training terminated after 3900 episodes. The agent can reliably land in the center of the goal, but misses out on the full 200 points that the environment would normally consider a success. It is believed that this is because the agent takes too much time during the slow landing phase. If the agent were to land quicker, than it would collect less negative reward for firing it's engines.

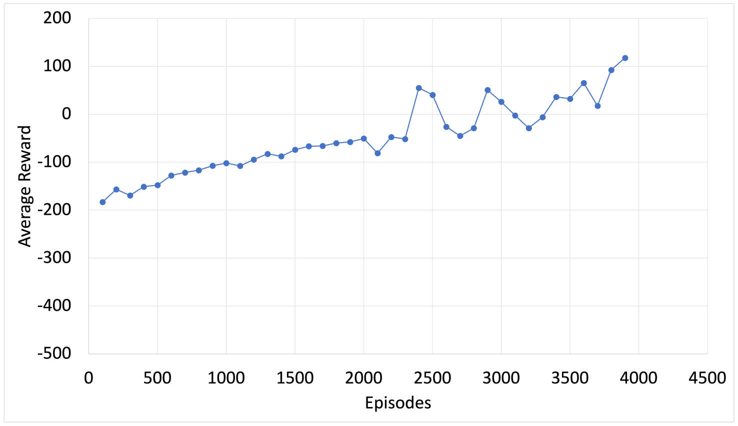


Fig. 7. Reward vs Episode of final network

Next steps for this project would be to train a network that can reliably collect the 200 point reward. Also, DQN could be further improved using Double Q-Learning like what as implemented by Hasselt et al. at Google Deep Mind [8].

VI. Contributions and Release

Nathan was responsible for all aspects of the project including the code and the paper that was written. The author grants permission for the report to be posted publicly.

References

[1] https://gymnasium.farama.org

[2] https://box2d.org

[3] Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, 2nd Ed. MIT Press, 2018. Available online: http://incompleteideas.net/book/the-book-2nd.html

[4] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," arXiv preprint arXiv:1511.05952, 2015. [Online]. Available: https://arxiv.org/abs/1511.05952. [Accessed: May 7, 2023].

[6] D. P. Kingma and J. Ba, "ADAM: A Method for Stochastic Optimization," in Proceedings of the 3rd International Conference on Learning Representations (ICLR), San Diego, CA, USA, May 7-9, 2015, pp. 1-13. [Online]. Available: https://arxiv.org/abs/1412.6980. [Accessed: May 7, 2023].

[7] S. Gadgil, Y. Xin, and C. Xu, "Solving the lunar lander problem under uncertainty using reinforcement learning," 2020 SoutheastCon, 2020.

[8] H. V. Hasselt et al., "Deep Reinforcement Learning with Double Q-Learning," AAAI Conference on Artificial Intelligence, Austin, TX, USA, 2015

VII. Appendix

The code that was implemented is shown on the following pages.

In [1]:
```julia
using PyCall

# Import openai gymnasim
gym = pyimport("gym")

# Create the LunarLander environment
env = gym.make(
    "LunarLander-v2",
    continuous = false,
    gravity = -10.0,
    enable_wind = false,
    wind_power = 15.0,
    turbulence_power = 0.0,
)

# Reset the environment
env.reset(seed=42)
```

Out[1]:
```
(Float32[0.00229702, 1.4181306, 0.23264714, 0.32046658, -0.0026548817, -0.052698076, 0.0, 0.0], Dict{Any, An
y}())
```

In [2]:
```julia
using Plots: scatter, scatter!, plot, plot!
using Flux
using BSON: @save
using Printf: @printf
using CommonRLInterface
using BenchmarkTools
using ProgressMeter
```

In [6]:
```
# ϵ-greedy policy
function ϵ_greedy(q, ϵ, s::Vector{Float32})

    # If ϵ is greater than a random number, chose a random action
    if rand() < ϵ
        return rand([0,1,2,3])

    # Otherwise return the greedy action
    else
        argmax(q(s))-1
    end
end

# ϵ-decay function
function ϵ_decay(min_ϵ::Float64, n_anneal_step::Int64, stp::Int64)
    ep_cal = 1.0 - (0.9 * (stp/n_anneal_step))
    updated_ϵ = max(min_ϵ, ep_cal)
    return updated_ϵ
end
```

Out[6]:
```
ϵ_decay (generic function with 1 method)
```

In [8]:

```julia
# Experience Replay Buffer Function
function experience_replay(D::Vector{Tuple{Vector{Float32}, Int64, Float64, Vector{Float32}, Bool}},
        state::Vector{Float32},
        N::Int64,
        ϵ::Float64)

    L::Int64 = length(D)

    # If buffer is at capacity, remove the oldest experience
    if L >= N
        deleteat!(D,1)
    end

    # Take an action according to ϵ_greedy policy with ϵ equal to epoch/epochs
    action::Int64 = ϵ_greedy(q, ϵ, state)

    # Take the greedy action, observe the new state, and collect the reward
    state_prime::Vector{Float32}, reward::Float64, done::Bool, _, _ = env.step(action::Int64)

    # Push the experience into the buffer
    push!(D, (state::Vector{Float32}, action::Int64, reward::Float64, state_prime::Vector{Float32}, done::I

    return D, state_prime
end
```

Out[8]:

```
experience_replay (generic function with 1 method)
```

In [9]:

```julia
# Function to train the neural network
function train!(q, q̂, minibatch)
    # Stocastic Gradient Descent
            Flux.Optimise.train!(Flux.params(q), minibatch, ADAM(0.0001)) do state, action, reward, state_p

            # Loss function
            if done
                return (q(state)[action+1] - reward)^2
            else
                return (q(state)[action+1] - (reward + 0.99*maximum(q̂(state_prime))))^2
            end #end loss function
        end #end do
end
```

Out[9]:

```
train! (generic function with 1 method)
```

In [10]:
```
# Evaluate the current network
# MODIFIED FUNCTION FROM HW5 SOLUTION
function evaluate(q, env::PyObject;
        episodes=100,
        discount=0.99
        )

    # Set the cumlative reward to 0.0
    rsum::Float32 = 0.0

    # For each episode
    for _ in 1:episodes

        terminated = false

        # Reset the environment and observe the current state
        s, _ = env.reset()

        # Set the discount to 1
        disc = 1.0

        # While the environment is not terminated/truncated and the discount is above 0.005
        while !terminated && disc > 0.01

            # Detemine the greedy action
            a = argmax(q(s)) - 1

            # Take the greedy action, observe the new state, and collect the reward
            sp, r, terminated, _, _ = env.step(a)

            # Add the reward to the cumlative reward
            rsum += disc*r

            # Mulitply dict by discount to farther discount future rewards
            disc *= discount

            # Set s' to s
            s = sp
        end

    end

    # Return the cumlative reward devided by the number of episodes
    return rsum/episodes
```

```
end
```

Out[10]:

```
evaluate (generic function with 1 method)
```

In [169]:
```julia
# Train the neural network
function DQN_train!(q, env,
        episodes, # Episodes
        max_steps, # Max steps per episodes
        N, # Experience replay buffer capacity
        batch_size, # Minibatch size
        ϵ # Epsilon
        )

    # Initialize the best network
    q_best = deepcopy(q)

    #### Logging ####
    all_rewards = Vector{Float64}()
    globalsteps_taken::Int64 = 0
    start = time()
    best_reward = 100.0

    #### Initalize the target network ####
    q̂ = deepcopy(q)

    #### Initialize buffer ####
    println("Initializing Experience Replay Buffer")

    # Reset the environment and observe the current state
    state::Vector{Float32}, _ = env.reset()

    # Determine the greedy action
    action::Int64 = ϵ_greedy(q, ϵ, state)

    # Take the greedy action, observe the new state, and collect the reward
    state_prime::Vector{Float32}, reward::Float64, done::Bool, _, _ = env.step(action)

    # Create the buffer from the experience tuple
    D = [(state, action, reward, state_prime, done)]

    # Initalize experience replay to capacity N
    while length(D) < N
        D, state = experience_replay(D, state, N, ϵ)

        if D[end][5]
            state, _ = env.reset()
        end
    end

    #### Start DQN ####
    println("DQN Start!!!")
    for ep in 1:episodes

        # Reset environment
        state, _ = env.reset()

        # Reset episode reward and steps taken per episode
        episode_reward::Float64 = 0.0
        steps_taken::Int64 = 0

        #### Take steps for each episode ####
        for stp in 1:max_steps

            # Decay ϵ
            ϵ = ϵ_decay(0.001,500_000,globalsteps_taken)

            #### Sampling: Filling the experience replay
            # With probability ϵ select a random action aₜ
            # otherwise select aₜ=argmaxₐQ_θ(s,a)
            # Execute action aₜ in emulator and observe reward rₜ and sₜ₊₁
            # Store transition (sₜ, aₜ, rₜ, sₜ₊₁, done)
            D, state = experience_replay(D, state, N, ϵ)

            # Record reward collected
            episode_reward += D[end][3]
            globalsteps_taken += 1

            #### Training ####
            # Train every 4 steps
            if globalsteps_taken % 4 == 0
                # Randomly sample from the buffer
                minibatch = rand(D, batch_size)

                # Stocastic Gradient Descent
                train!(q,q̂,minibatch)
            end

            # Freeze target every 10_000 steps
            if globalsteps_taken % 10_000 == 0
                q̂ = deepcopy(q)
            end

            # If done, break
            if D[end][5]
```

```
                break
            end

        end #end stp

        # Record the reward collected
        push!(all_rewards,episode_reward)

        # Logging every 100 episodes
        if ep % 100 == 0
            ave_reward = mean(all_rewards[end-99:end])

            @printf("Episode: %8.0f | Step: %8.0f | ϵ: %8.5f | Avg. Reward: %8.3f\n",
                ep, globalsteps_taken,ϵ, ave_reward)
            flush(stdout)

            if ave_reward > best_reward
                @printf("Reward above 100 found\n")
                q_best = deepcopy(q)
                @printf("Total Time: %8.3f\n", time() - start)
                return q_best, all_rewards
            end
        end

    end #end episodes

    q_best = deepcopy(q)
    @printf("Total Time: %8.3f\n", time() - start)
    # Return the best network, and the collected rewards
    return q_best, all_rewards

end #end DQN_train!
```

Out[169]:
```
DQN_train! (generic function with 1 method)
```

In [170]:
```
# Hyperparameters
episodes = 5000; # Episodes
max_steps = 5000; # Max_steps per episodes
N = 10_000; # Maximum buffer size
batch_size = 64; # Minibatch size
ϵ = 1.0; # Epsilon
```

In [171]:
```
# Create the neural network
q = Chain(Dense(8, 32, relu), Dense(32, 32, relu), Dense(32, 4,elu))
```

Out[171]:
```
Chain(
  Dense(8 => 32, relu),                 # 288 parameters
  Dense(32 => 32, relu),                # 1_056 parameters
  Dense(32 => 4, elu),                  # 132 parameters
)                   # Total: 6 arrays, 1_476 parameters, 6.141 KiB.
```

In [172]:

```
# Train the network
q_trained, rewards = DQN_train!(q, env, episodes, max_steps, N, batch_size, ϵ)
```

```
Initializing Experience Replay Buffer
DQN Start!!!
Episode:      100 | Step:     9169 | ϵ:  0.98350 | Avg. Reward: -183.112
Episode:      200 | Step:    18633 | ϵ:  0.96646 | Avg. Reward: -156.712
Episode:      300 | Step:    28193 | ϵ:  0.94925 | Avg. Reward: -169.200
Episode:      400 | Step:    38029 | ϵ:  0.93155 | Avg. Reward: -150.984
Episode:      500 | Step:    47709 | ϵ:  0.91413 | Avg. Reward: -147.861
Episode:      600 | Step:    57241 | ϵ:  0.89697 | Avg. Reward: -127.561
Episode:      700 | Step:    71962 | ϵ:  0.87047 | Avg. Reward: -121.511
Episode:      800 | Step:    81838 | ϵ:  0.85269 | Avg. Reward: -116.696
Episode:      900 | Step:    92165 | ϵ:  0.83410 | Avg. Reward: -107.383
Episode:     1000 | Step:   102716 | ϵ:  0.81511 | Avg. Reward: -101.824
Episode:     1100 | Step:   117922 | ϵ:  0.78774 | Avg. Reward: -107.731
Episode:     1200 | Step:   128557 | ϵ:  0.76860 | Avg. Reward:  -94.439
Episode:     1300 | Step:   139739 | ϵ:  0.74847 | Avg. Reward:  -82.687
Episode:     1400 | Step:   155421 | ϵ:  0.72024 | Avg. Reward:  -87.597
Episode:     1500 | Step:   167116 | ϵ:  0.69919 | Avg. Reward:  -73.955
Episode:     1600 | Step:   178657 | ϵ:  0.67842 | Avg. Reward:  -66.611
Episode:     1700 | Step:   190491 | ϵ:  0.65712 | Avg. Reward:  -66.102
Episode:     1800 | Step:   202283 | ϵ:  0.63589 | Avg. Reward:  -60.178
Episode:     1900 | Step:   219541 | ϵ:  0.60483 | Avg. Reward:  -57.601
Episode:     2000 | Step:   242115 | ϵ:  0.56419 | Avg. Reward:  -50.480
Episode:     2100 | Step:   280926 | ϵ:  0.49433 | Avg. Reward:  -81.259
Episode:     2200 | Step:   309695 | ϵ:  0.44255 | Avg. Reward:  -47.370
Episode:     2300 | Step:   341567 | ϵ:  0.38518 | Avg. Reward:  -51.635
Episode:     2400 | Step:   446841 | ϵ:  0.19569 | Avg. Reward:   55.050
Episode:     2500 | Step:   525802 | ϵ:  0.05356 | Avg. Reward:   40.498
Episode:     2600 | Step:   560277 | ϵ:  0.00100 | Avg. Reward:  -26.117
Episode:     2700 | Step:   617227 | ϵ:  0.00100 | Avg. Reward:  -45.071
Episode:     2800 | Step:   702067 | ϵ:  0.00100 | Avg. Reward:  -28.990
Episode:     2900 | Step:   767462 | ϵ:  0.00100 | Avg. Reward:   50.760
Episode:     3000 | Step:   861152 | ϵ:  0.00100 | Avg. Reward:   26.069
Episode:     3100 | Step:   886049 | ϵ:  0.00100 | Avg. Reward:   -2.357
Episode:     3200 | Step:   901650 | ϵ:  0.00100 | Avg. Reward:  -28.834
Episode:     3300 | Step:   920362 | ϵ:  0.00100 | Avg. Reward:   -6.057
Episode:     3400 | Step:   942574 | ϵ:  0.00100 | Avg. Reward:   36.286
Episode:     3500 | Step:   970224 | ϵ:  0.00100 | Avg. Reward:   32.334
Episode:     3600 | Step:  1002672 | ϵ:  0.00100 | Avg. Reward:   65.408
Episode:     3700 | Step:  1083543 | ϵ:  0.00100 | Avg. Reward:   17.559
Episode:     3800 | Step:  1167579 | ϵ:  0.00100 | Avg. Reward:   92.560
Episode:     3900 | Step:  1224601 | ϵ:  0.00100 | Avg. Reward:  117.708
Reward above 100 foundTotal Time: 1383.001
```

```
Out[172]: (Chain(Dense(8 => 32, relu), Dense(32 => 32, relu), Dense(32 => 4, elu)), [-170.03313259125207, -82.667126611
          3522, -192.6009913649434, -346.1995599340764, -156.16567810860056, -89.50224875667666, -210.04557208854726, -
          246.47301698357293, -158.8668399374058, -124.74871972597623  …  160.67354349202387, 264.47610383172355, 243.4
          8080085505663, 236.8883042066439, 204.52091761884796, 262.3658304063878, 150.3648537526253, 179.4089187821668
          4, 194.27490695693604, 158.16524939165706])
```

```
In [183]: evaluate(q_trained, env)
```

```
Out[183]: 28.342928f0
```

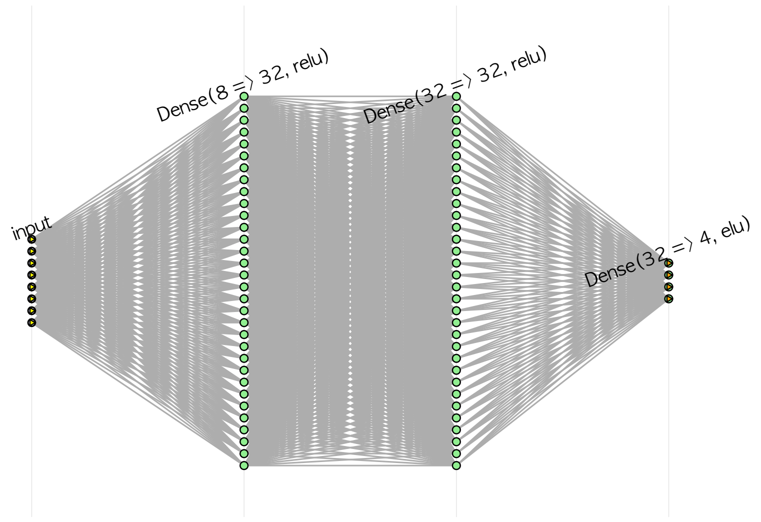
```
In [175]: model = q_trained
```

```
Out[175]: Chain(
            Dense(8 => 32, relu),                 # 288 parameters
            Dense(32 => 32, relu),                # 1_056 parameters
            Dense(32 => 4, elu),                  # 132 parameters
          )                   # Total: 6 arrays, 1_476 parameters, 6.141 KiB.
```

```
In [176]: @save "LunarLanderNew.bson" model
```

```
In [178]: using ChainPlots
```

```
In [181]: using Plots
In [179]:  1 plot(q)
Out[179]:
```



```
GKS: could not find font Sans.ttf
```