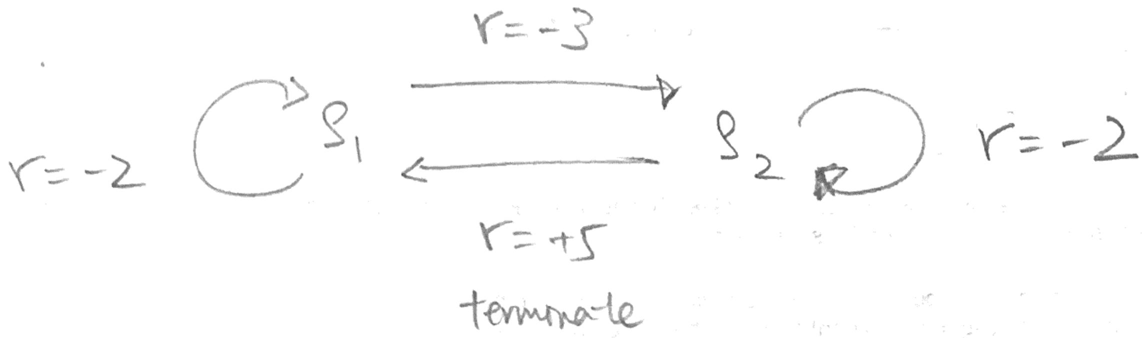
1] $S = \{s_1, s_2\}$ $A = \{\text{stay}, \text{go}\}$



a] $\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t) = \sum_{t=0}^{\infty} \gamma^t r_t(s_1, \text{stay})$

$= \sum_{t=0}^{\infty} \gamma^t (-2)$, since $0 < \gamma \leq 1$, this is a geometric series.

$= \boxed{\dfrac{-2}{1-\gamma}}$

b] Consider π_e that goes to terminate ASAP.

i.e. $\pi_e(s_1) = \text{go}$, $\pi_e(s_2) = \text{go}$.

Now consider its sum of discounted rewards.

$$\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t) = \gamma^0 r_0(s_1, go)$$

$$+ \gamma^1 r_1(s_2, go)$$

$$= \underline{-3 + \gamma(5)}$$

Let's compare this with result from 1a. (π_{1a})

We can use this π_e if the following holds.

$$-3 + \gamma(5) - \left(\frac{-2}{1-\gamma}\right) \geq 0$$

$$\frac{(-3+5\gamma)(1-\gamma) + 2}{1-\gamma} \geq 0$$

$1-\gamma \rightarrow$ denominator is always >0 since $\gamma < 1$.

So inequality depends on numerator.

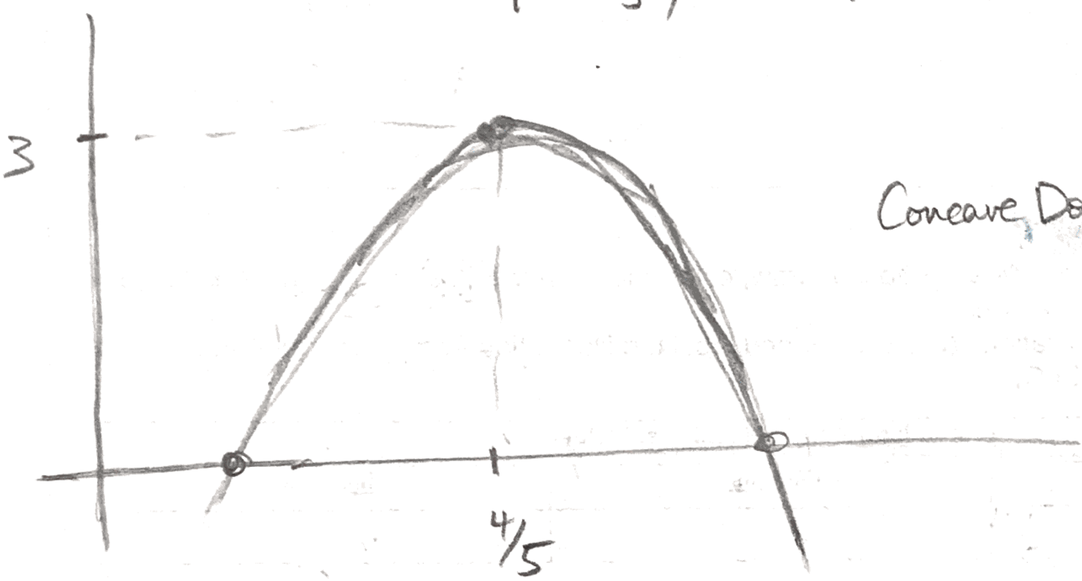
numerator $= -3+\cancel{5\gamma}+\cancel{3}\gamma-5\gamma^2+2$

$$f(\gamma) = -5\gamma^2 + 8\gamma - 1$$

$$f(\gamma) = -5\left(\gamma^2 - \frac{8}{5}\gamma + \frac{4}{5}\right) + 3$$

$$= -5\left(\gamma - \frac{4}{5}\right)^2 + 3.$$

$\therefore$ when $\gamma = \frac{4}{5}$, $f(\gamma) = 3$



Concave Down, since -5

need to find roots to determine

when $f(\gamma) \geq 0$; this is when we should

use π_e, o.w. when $f(\gamma) < 0$, use π_{1a}

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-8 \pm \sqrt{64 - 4(-5)(-1)}}{-10}$$

$$= \frac{-8 \pm 2\sqrt{11}}{-10} = \frac{4 \pm \sqrt{11}}{5} \approx (.137, 1.46)$$

$$\text{Optimal Policy} = \begin{cases} (go, go) & \text{if } .137 \leq \gamma \leq 1.46 \\ (stay, stay, \ldots, stay) & \text{o.w.} \end{cases}$$

1c) $V^0 = [0, 0]$

$$V^1(s_1) = \max \{ r(s_1, stay) + V^0(s_1), \; r(s_1, go) + V^0(s_2) \}$$

$$= \max \{ -2 + 0, -3 + 0 \} = -2$$

$$V^1(s_2) = \max \{ r(s_2, stay) + V^0(s_2), r(s_2, go) \}$$

$$= \max \{ -2 + 0, 5 \} = 5$$

$\therefore V^1 = [-2, 5]$

$$V^2(s_1) = \max \{ r(s_1, stay) + V^1(s_1), r(s_1, go) + V^1(s_2) \}$$

$$= \max \{ -2 - 2, -3 + 5 \} = 2$$

$$V^2(s_2) = \max \{ r(s_2, stay) + V^1(s_2); r(s_2, go) \}$$

$$= \max \{ -2 + 5, 5 \} = 5$$

$\therefore V^2 = [2, 5]$

1c)

$$V^3(s_1) = \max\{r(s_1, stay) + V^2(s_1),\ r(s_1, go) + V^2(s_2)\}$$

$$= \max\{-2+2,\ -3+5\} = 2$$

$$V^3(s_2) = \max\{r(s_2, stay) + V^2(s_2),\ r(s_2, go)\} = 5$$

$$\underbrace{V^3 = [2, 5]}$$

$$\boxed{\therefore V^* = [2, 5]}$$

2| a| $\|v^i - V^*\|_\infty = \max_{s \in S} |v^i(s) - V^*(s)|$

$i=1;\quad \max\{\underbrace{|v'(s_1) - V^*(s_1)|}_{4}, \underbrace{|V'(s_2) - V^*(s_2)|}_{0}\} = 4$

$i=2;\quad \max\{|v^2(s_1) - V^*(s_1)|, |v^2(s_2) - V^*(s_2)|\} = 0$

$i=3;\quad \max\{|v^3(s_1) - V^*(s_1)|, |v^3(s_2) - V^*(s_2)|\} = 0$

b| $T(v) = \max_a \sum_{s'} p(s'|s,a)[r(s,a) + rV(s')]$

$T(v') = \max_a \sum_{s'} p(s'|s,a)[r(s,a) + rv'(s')]$

$\|Tv - Tv'\|_\infty$

$= \|\max_a \sum_{s'} p(s'|s,a)[r(s,a) + rv(s')] - \max_a \sum_{s'} p(s'|s,a)$
$[r(s,a) + rv'(s')]\|_\infty$

$\leq \max_a \|\sum_{s'} p(s'|s,a)[r(s,a) + rv(s')] - \sum_{s'} p(s'|s,a)$
$[r(s,a) + rv'(s')]\|_\infty \quad \|\max_a f(a) - \max_a (g(a))\|_\infty$
$\leq \max_a \|f(a) - g(a)\|_\infty$

$$= \max_a \| \sum_{s'} p(s'|s\,a] [r v(s') - r v'(s')] \|_\infty$$

$$= \max_a \| \sum_{s'} p(s'|s,a)\, r(v(s') - v'(s')) \|_\infty$$

$$= r \max_a \| \sum_{s'} p(s'|s,a)(v(s') - v'(s')) \|_\infty$$

$$\leq \gamma \| v(s') - v'(s') \|_\infty$$

2c) $\| V^{n+1} - V^* \|_\infty = \| V^* - V^{n+1} \|_\infty$

$$= \| \sum_{t=1}^{\infty} V^{n+1+t} - V^{n+t} \|_\infty$$

$$\leq \sum_{t=1}^{\infty} \| V^{n+1+t} - V^{n+1} \|_\infty$$

$$\leq \sum_{t=1}^{\infty} \gamma^t \| V^{n+1} - V^n \|_\infty$$

$$= \| V^{n+1} - V^n \|_\infty \cdot \frac{r}{1-r}$$

$\therefore \| V^{n+1} - V^* \|_\infty \leq \frac{\gamma}{1-\gamma} \| V^{n+1} - V^n \|_\infty$

if $\| V^{n+1} - V^n \|_\infty < \epsilon$

then

$\| V^{n+1} - V^* \|_\infty \leq \frac{\gamma}{1-\gamma} \epsilon$

2d] Suppose $V^* = T^{(\infty)}(0)$

$$T(V^*) = T(T^{\infty}(0)) = T^{\infty}(0) = V^*$$

$$\therefore \; T(V^*) = V^*$$

Suppose $V^* = T^{\infty}(0)$

$$\| T^n(v) - T^{(n)}(v') \|_\infty \leq \gamma^n \| v - v' \|_\infty \quad \underline{\text{by } 2b}$$

$$\| T^{\infty}(v) - T^{\infty}(0) \|_\infty \leq \gamma \| v - 0 \|_\infty = 0$$

$$\| T^{\infty}(v) - V^* \|_\infty = 0$$

$$T^{\infty}(v) = V^*$$

4a]

$$\nabla_\theta J(\theta) = \nabla_\theta E_{\tau \sim \pi_\theta}[R(\tau)] \quad \text{--- (A)}$$

if $R(\tau)$ is changed to $R(\tau) - b$,

then $\nabla_\theta J(\theta) = \nabla_\theta E_{\tau \sim \pi_\theta}[R(\tau) - b]$

s.t. b is not a function of θ

$$= E_{\tau \sim \pi_\theta}[\nabla_\theta(R(\tau) - b)]$$

$$= E_{\tau \sim \pi_\theta}[\nabla_\theta R(\tau)]$$

$$= \nabla_\theta E_{\tau \sim \pi_\theta}[R(\tau)] \quad \text{--- (B)}$$

$\therefore$ (A) = (B)

Dynamic Programming (30 points)

In this assignment, we will implement a few dynamic programming algorithms, namely, policy iteration and value iteration and run them on a simple MDP - the Frozen Lake environment.

The sub-routines for these algorithms are present in `vi_and_pi.py` and must be filled out to test your implementation.

The deliverables are located at the end of this notebook and show the point distrbution for each part.

In [1]:
```
%load_ext autoreload
%autoreload 2

import numpy as np
import gym
import time

from IPython.display import clear_output

from lake_envs import *
from vi_and_pi import *

np.set_printoptions(precision=3)

env_d = gym.make("Deterministic-4x4-FrozenLake-v0")
env_s = gym.make("Stochastic-4x4-FrozenLake-v0")
```

Render Mode

The variable `RENDER_ENV` is set `True` by default to allow you to see a rendering of the state of the environment at every time step. However, when you complete this assignment, you must set this to `False` and re-run all blocks of code. This is to prevent excessive amounts of rendered environments from being included in the final PDF.

IMPORTANT: SET `RENDER_ENV` TO FALSE BEFORE SUBMISSION!

In [2]:
```
RENDER_ENV = False
```

Part 1: Value Iteration

For the first part, you will implement the familiar value iteration update from class.

In `vi_and_pi.pi` and complete the `value_iteration` function.

In [3]:
```
####################################################
# Use this space for debugging                     #
# Make sure to delete this code before submission #
####################################################
pass
####################################################
```

Run the cell below to train value iteration and render a single episode of following the policy obtained at the end of value iteration.

You should expect to get an Episode reward of `1.0`.

In [4]:
```
print("\n" + "-"*25 + "\nBeginning Value Iteration\n" + "-"*25)

V_vi, p_vi = value_iteration(env_d.P, env_d.nS, env_d.nA, gamma=0.9, tol=1e-3)
render_single(env_d, p_vi, 100, show_rendering=RENDER_ENV)
```

```
-------------------------
Beginning Value Iteration
-------------------------
Episode reward: 1.000000
```

Part 2: Policy Iteration

In this question, you will implement policy iteration.

In class, we studied the value iteration update:

$$V_{t+1}(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V_t(s')\right]$$

This is used to compute the value function V^* corresponding to the optimal policy π^*. We can alternatively compute the value function V^π corresponding to an arbitrary policy π, with a similar update loop:

$$V_{t+1}^{\pi}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma V_t^{\pi}(s')\right]$$

On convergence, this will give us V^π, which is the first step of a policy iteration update.

The second step involves policy refinement, which will update the policy to take actions greedily with respect to V^π:

$$\pi_{new} \leftarrow \arg\max_a \left[r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s')\right]$$

A single update of policy iteration involves the two above steps: (1) policy evaluation (which itself is an inner loop which will converge to V^π and (2) policy refinement. In the first part of assignment, you will implement the functions for policy evaluation, policy improvement (refinement) and policy iteration.

In `vi_and_pi.pi` and complete the `policy_evaluation`, `policy_improvement` and `policy_iteration` functions. Run the blocks below to test your algorithm.

In [5]:
```
####################################################
# Use this space for debugging                     #
# Make sure to delete this code before submission #
####################################################
pass
####################################################
```

In [6]:
```
print("\n" + "-"*25 + "\nBeginning Policy Iteration\n" + "-"*25)

V_pi, p_pi = policy_iteration(env_d.P, env_d.nS, env_d.nA, gamma=0.9, tol=1e-3)
render_single(env_d, p_pi, 100, show_rendering=RENDER_ENV)
```

```
-------------------------
Beginning Policy Iteration
-------------------------
Episode reward: 1.000000
```

Part 3: VI on Stochastic Frozen Lake

Now we will apply our implementation on an MDP where transitions to next states are stochastic. Modify your implementation of value iteration as needed so that policy iteration and value iteration work for stochastic transitions.

In [7]:
```
####################################################
# Use this space for debugging                     #
# Make sure to delete this code before submission #
####################################################
pass
####################################################
```

In [8]:
```
print("\n" + "-"*25 + "\nBeginning Value Iteration\n" + "-"*25)

V_vi, p_vi = value_iteration(env_s.P, env_s.nS, env_s.nA, gamma=0.9, tol=1e-3)
render_single(env_s, p_vi, 100, show_rendering=RENDER_ENV)
```

```
-------------------------
Beginning Value Iteration
-------------------------
Episode reward: 1.000000
```

Part 4: PI on Stochastic Frozen Lake

Now, we will run policy iteration on stochastic frozen lake.

In [9]:
```
####################################################
# Use this space for debugging                     #
# Make sure to delete this code before submission #
####################################################
pass
####################################################
```

In [10]:
```
print("\n" + "-"*25 + "\nBeginning Policy Iteration\n" + "-"*25)

V_pi, p_pi = policy_iteration(env_s.P, env_s.nS, env_s.nA, gamma=0.9, tol=1e-3)
render_single(env_s, p_pi, 100, show_rendering=RENDER_ENV)
```

```
-------------------------
Beginning Policy Iteration
-------------------------
Episode reward: 1.000000
```

Evaluate All Policies

Now, we will first test the value iteration implementation on two kinds of environments - the dererministic FrozenLake and the stochastic FrozenLake. We will also run the same for policy iteration

Deliverable 1 (10 points)

Run value iteration on deterministic FrozenLake. You should get a reward of 1.0 for full credit.

In [11]:
```
print("\nValue Iteration on Deterministic FrozenLake:")
V_vi, p_vi = value_iteration(env_d.P, env_d.nS, env_d.nA, gamma=0.9, tol=1e-3)
evaluate(env_d, p_vi, max_steps=100, max_episodes=2)
```

```
Value Iteration on Deterministic FrozenLake:
> Average reward over 2 episodes:                    1.0
> Percentage of episodes goal reached:               100%
```

Deliverable 2 (10 points)

Run value iteration on stochastic FrozenLake. Note that this time, running the same policy over multiple episodes will result in different outcomes (final reward) due to stochastic transitions in the environment, and even the optimal policy may not succeed in reaching the goal state 100% of the time.

You should get a reward of 0.7 or higher over 1000 episodes for full credit.

In [12]:
```python
print("\nValue Iteration on Stochastic FrozenLake:")
V_vi, p_vi = value_iteration(env_s.P, env_s.nS, env_s.nA, gamma=0.9, tol=1e-3)
evaluate(env_s, p_vi, max_steps=100, max_episodes=1000)
```

```
Value Iteration on Stochastic FrozenLake:
> Average reward over 1000 episodes:                        0.725
> Percentage of episodes goal reached:                      95%
```

Deliverable 3 (5 points)

Run policy iteration on deterministic FrozenLake. You should get a reward of 1.0 for full credit.

In [13]:
```python
print("Policy Iteration on Deterministic FrozenLake:")
V_pi, p_pi = policy_iteration(env_d.P, env_d.nS, env_d.nA, gamma=0.9, tol=1e-3)
evaluate(env_d, p_pi, max_steps=100, max_episodes=2)
```

```
Policy Iteration on Deterministic FrozenLake:
> Average reward over 2 episodes:                           1.0
> Percentage of episodes goal reached:                      100%
```

Deliverable 4 (5 points)

Run policy iteration on stochastic FrozenLake.

You should get a reward of 0.7 or higher over 1000 episodes for full credit.

In [14]:
```python
print("Policy Iteration on Stochastic FrozenLake:")
V_pi, p_pi = policy_iteration(env_s.P, env_s.nS, env_s.nA, gamma=0.9, tol=1e-3)
evaluate(env_s, p_pi, max_steps=100, max_episodes=1000)
```

```
Policy Iteration on Stochastic FrozenLake:
> Average reward over 1000 episodes:                        0.731
> Percentage of episodes goal reached:                      93%
```

Submission Reminder

PLEASE RE-RUN THE NOTEBOOK WITH `RENDER_ENV` SET TO FALSE BEFORE SUBMISSION!

Q-Learning & DQNs (30 points + 5 bonus points)

In this section, we will implement a few key parts of the Q-Learning algorithm for two cases - (1) A Q-network which is a single linear layer (referred to in RL literature as "Q-learning with linear function approximation") and (2) A deep (convolutional) Q-network, for some Atari game environments where the states are images.

Optional Readings:

- **Playing Atari with Deep Reinforcement Learning**, Mnih et. al., https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf)
- **The PyTorch DQN Tutorial** https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html (https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html)

Note: **The bonus credit for this question applies to both sections CS 7643 and CS 4803**

In [3]:
```
%load_ext autoreload
%autoreload 2

import numpy as np
import gym

import torch
import torch.nn as nn
import torch.optim as optim

from core.dqn_train import DQNTrain
from utils.test_env import EnvTest
from utils.schedule import LinearExploration, LinearSchedule
from utils.preprocess import greyscale
from utils.wrappers import PreproWrapper, MaxAndSkipEnv

from linear_qnet import LinearQNet
from cnn_qnet import ConvQNet

if torch.cuda.is_available():
    device = torch.device('cuda', 0)
else:
    device = torch.device('cpu')
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Part 1: Setup Q-Learning with Linear Function Approximation

Training Q-networks using (Deep) Q-learning involves a lot of moving parts. However, for this assignment, the scaffolding for the first 3 points listed below is provided in full and you must only complete point 4. You may skip to point 4 if you only care about the implementation required for this assignment.

1. **Environments**: We will use the standardized OpenAI Gym framework for environment API calls (read through http://gym.openai.com/docs/ (http://gym.openai.com/docs/) if you want to know more details about this interface). Specifically, we will use a custom Test environment defined in `utils/test_env.py` for initial sanity checks and then Gym-Atari environments later on.

1. **Exploration**: In order to train any RL model, we require experience or "data" gathered from interacting with the environment by taking actions. What policy should we use to collect this experience? Given a Q-network, one may be tempted to define a greedy policy which always picks the highest valued action at every state. However, this strategy will in most cases not work since we may get stuck in a local minima and never explore new states in the environment which may lead to a better reward. Hence, for the purpose of gathering experience (or "data") from the environment, it is useful to follow a policy that deviates from the greedy policy slightly in order to explore new states. A common strategy used in RL is to follow an ϵ-greedy policy which with probability $0 < \epsilon < 1$ picks a random action instead of the action provided by the greedy policy.

1. **Replay Buffers**: Data gathered from a single trajectory of states and actions in the environment provides us with a batch of highly correlated (non IID) data, which leads to high variance in gradient updates and convergence. In order to ameliorate this, replay buffers are used to gather a set of transitions i.e. (state, action, reward, next state) tuples, by executing multiple trajectories in the environment. Now, for updating the Q-Network, we will first wait to fill up our replay buffer with a sufficiently large number of transitions over multiple different trajectories, and then randomly sample a batch of transitions to compute loss and update the models.

1. **Q-Learning network, loss and update**: Finally, we come to the part of Q-learning that we will implement for this assignment -- the Q-network, loss function and update. In particular, we will implement a variant of Q-Learning called "Double Q-Learning", where we will maintain two Q networks -- the first Q network is used to pick actions and the second "target" Q network is used to compute Q-values for the picked actions. Here is some referance material on the same - Blog 1 (https://towardsdatascience.com/double-q-learning-the-easy-way-a924c4085ec3), Blog 2 (https://medium.com/@ameetsd97/deep-double-q-learning-why-you-should-use-it-bedf660d5295), but we will not need to get into the details of Double Q-learning for this assignment. Now, let's walk through the steps required to implement this below.

   - **Linear Q-Network**: In `linear_qnet.py` , define the initialization and forward pass of a Q-network with a single linear layer which takes the state as input and outputs the Q-values for all actions.
   - **Setting up Q-Learning**: In `core/dqn_train.py` , complete the functions `process_state` , `forward_loss` and `update_step` and `update_target_params` . The loss function for our Q-Networks is defined for a single transition tuple of (state, action, reward, next state) as follows. $Q(s_t, a_t)$ refers to the state-action values computed by our first Q-network at the current state and and for the current actions, $Q_{target}(s_{t+1}, a_{t+1})$ refers to the state-action values for the next state and all possible future actions computed by the target Q-Network

$$
\begin{aligned}
Q_{sample}(s_t) &= r_t \text{ if done} \\
&= r_t + \gamma \max_{a_{t+1}} Q_{target}\,(s_{t+1}, a_{t+1}) \text{ otherwise} \\
\text{Loss} &= \left(Q_{sample}(s_t) - Q(s_t, a_t)\right)^2
\end{aligned}
$$

Deliverable 1 (15 points)

Run the following block of code to train a Linear Q-Network. You should get an average reward of ~4.0, full credit will be given if average reward at the final evaluation is above 3.5

In [4]:
```python
from configs.p1_linear import config as config_lin

env = EnvTest((5, 5, 1))

# exploration strategy
exp_schedule = LinearExploration(env, config_lin.eps_begin,
        config_lin.eps_end, config_lin.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_lin.lr_begin, config_lin.lr_end,
        config_lin.lr_nsteps)

# train model
model = DQNTrain(LinearQNet, env, config_lin, device)
model.run(exp_schedule, lr_schedule)
```

```
Evaluating...
Average reward: 1.60 +/- 0.00

 1001/10000 [==>...........................] - ETA: 3s - Loss: 0.2968 - Avg_R: 0.5500 - Max_R: 3.1000 - eps: 0.8020 - Grads: 1.2544 - Max_Q: 0.6897
- lr: 0.0042

Evaluating...
Average reward: 3.80 +/- 0.00

 2001/10000 [=====>........................] - ETA: 3s - Loss: 0.3410 - Avg_R: 2.0550 - Max_R: 4.1000 - eps: 0.6040 - Grads: 1.1430 - Max_Q: 1.8823
- lr: 0.0034

Evaluating...
Average reward: 3.90 +/- 0.00

 3001/10000 [========>.....................] - ETA: 3s - Loss: 0.3964 - Avg_R: 2.0800 - Max_R: 4.0000 - eps: 0.4060 - Grads: 0.7165 - Max_Q: 2.5491
- lr: 0.0026

Evaluating...
Average reward: 3.80 +/- 0.00

 4001/10000 [===========>..................] - ETA: 2s - Loss: 0.3749 - Avg_R: 2.9400 - Max_R: 4.1000 - eps: 0.2080 - Grads: 0.6549 - Max_Q: 2.7646
- lr: 0.0018

Evaluating...
Average reward: 4.10 +/- 0.00

 5001/10000 [==============>...............] - ETA: 2s - Loss: 0.0746 - Avg_R: 3.9900 - Max_R: 4.1000 - eps: 0.0100 - Grads: 0.5588 - Max_Q: 2.7375
- lr: 0.0010

Evaluating...
Average reward: 4.10 +/- 0.00

 6001/10000 [=================>............] - ETA: 1s - Loss: 0.0002 - Avg_R: 3.9000 - Max_R: 4.1000 - eps: 0.0100 - Grads: 0.0812 - Max_Q: 2.9415
- lr: 0.0010

Evaluating...
Average reward: 4.10 +/- 0.00

 7001/10000 [====================>.........] - ETA: 1s - Loss: 0.0001 - Avg_R: 4.1000 - Max_R: 4.1000 - eps: 0.0100 - Grads: 0.0516 - Max_Q: 2.6616
- lr: 0.0010

Evaluating...
Average reward: 4.10 +/- 0.00

 8001/10000 [=======================>......] - ETA: 0s - Loss: 0.0055 - Avg_R: 4.1000 - Max_R: 4.1000 - eps: 0.0100 - Grads: 0.1861 - Max_Q: 2.8178
- lr: 0.0010

Evaluating...
Average reward: 4.10 +/- 0.00

 9001/10000 [==========================>...] - ETA: 0s - Loss: 0.0001 - Avg_R: 4.0000 - Max_R: 4.1000 - eps: 0.0100 - Grads: 0.0183 - Max_Q: 2.7897
- lr: 0.0010

Evaluating...
Average reward: 4.10 +/- 0.00

10001/10000 [==============================] - 4s - Loss: 0.0001 - Avg_R: 4.1000 - Max_R: 4.1000 - eps: 0.0100 - Grads: 0.0306 - Max_Q: 2.8013 - l
r: 0.0010

- Training done.
Evaluating...
Average reward: 4.10 +/- 0.00
```

You should get a final average reward of over 4.0 on the test environment.

Part 2: Q-Learning with Deep Q-Networks

In `cnn_qnet.py` , implement the initialization and forward pass of a convolutional Q-network with architecture as described in this DeepMind paper:

"Playing Atari with Deep Reinforcement Learning", Mnih et. al. (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf))

Deliverable 2 (10 points)

Run the following block of code to train our Deep Q-Network. You should get an average reward of ~4.0, full credit will be given if average reward at the final evaluation is above 3.5

In [20]:
```python
from configs.p2_cnn import config as config_cnn

env = EnvTest((80, 80, 1))

# exploration strategy
exp_schedule = LinearExploration(env, config_cnn.eps_begin,
        config_cnn.eps_end, config_cnn.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_cnn.lr_begin, config_cnn.lr_end,
        config_cnn.lr_nsteps)

# train model
model = DQNTrain(ConvQNet, env, config_cnn, device)
model.run(exp_schedule, lr_schedule)
```

```
Evaluating...
Average reward: 0.50 +/- 0.00

Populating the memory 150/200...

Evaluating...
Average reward: 0.50 +/- 0.00

 301/1000 [========>.....................] - ETA: 2s - Loss: 0.0899 - Avg_R: 0.3250 - Max_R: 3.1000 - eps: 0.4060 - Grads: 4.2915 - Max_Q: 0.1761 - 
lr: 0.0002

Evaluating...
Average reward: 0.50 +/- 0.00

 401/1000 [===========>..................] - ETA: 2s - Loss: 0.0347 - Avg_R: 0.0750 - Max_R: 2.3000 - eps: 0.2080 - Grads: 1.3289 - Max_Q: 0.2084 - 
lr: 0.0001

Evaluating...
Average reward: 0.50 +/- 0.00

 501/1000 [==============>...............] - ETA: 2s - Loss: 0.0188 - Avg_R: 0.4900 - Max_R: 1.9000 - eps: 0.0100 - Grads: 1.0697 - Max_Q: 0.2152 - 
lr: 0.0001

Evaluating...
Average reward: 0.50 +/- 0.00

 601/1000 [=================>............] - ETA: 2s - Loss: 0.2072 - Avg_R: 2.9450 - Max_R: 4.0000 - eps: 0.0100 - Grads: 3.6084 - Max_Q: 0.3536 - 
lr: 0.0001

Evaluating...
Average reward: 3.80 +/- 0.00

 701/1000 [====================>.........] - ETA: 1s - Loss: 0.0938 - Avg_R: 3.9250 - Max_R: 4.0000 - eps: 0.0100 - Grads: 3.8625 - Max_Q: 0.4915 - 
lr: 0.0001

Evaluating...
Average reward: 3.80 +/- 0.00

 801/1000 [=======================>......] - ETA: 1s - Loss: 0.0779 - Avg_R: 3.8500 - Max_R: 4.0000 - eps: 0.0100 - Grads: 1.8714 - Max_Q: 0.6081 - 
lr: 0.0001

Evaluating...
Average reward: 4.00 +/- 0.00

 901/1000 [==========================>...] - ETA: 0s - Loss: 0.0088 - Avg_R: 3.9750 - Max_R: 4.0000 - eps: 0.0100 - Grads: 2.9653 - Max_Q: 0.6976 - 
lr: 0.0001

Evaluating...
Average reward: 3.90 +/- 0.00

1001/1000 [==============================] - 7s - Loss: 0.0059 - Avg_R: 3.8250 - Max_R: 4.0000 - eps: 0.0100 - Grads: 1.9607 - Max_Q: 0.7689 - lr: 
0.0001

- Training done.
Evaluating...
Average reward: 4.00 +/- 0.00
```

You should get a final average reward of over 4.0 on the test environment, similar to the previous case.

Part 3: Playing Atari Games from Pixels - using Linear Function Approximation

Now that we have setup our Q-Learning algorithm and tested it on a simple test environment, we will shift to a harder environment - an Atari 2600 game from OpenAI Gym: Pong-v0 (https://gym.openai.com/envs/Pong-v0/ (https://gym.openai.com/envs/Pong-v0/)), where we will use RGB images of the game screen as our observations for state.

No additional implementation is required for this part, just run the block of code below (will take around 1 hour to train). We don't expect a simple linear Q-network to do well on such a hard environment - full credit will be given simply for running the training to completion irrespective of the final average reward obtained.

You may edit `configs/p3_train_atari_linear.py` if you wish to play around with hyperparamters for improving performance of the linear Q-network on Pong-v0, or try another Atari environment by changing the `env_name` hyperparameter. The list of all Gym Atari environments are available here: https://gym.openai.com/envs/#atari (https://gym.openai.com/envs/#atari)

Deliverable 3 (5 points)

Run the following block of code to train a linear Q-network on Atari Pong-v0. We don't expect the linear Q-Network to learn anything meaingful so full credit will be given for simply running this training to completion (without errors), irrespective of the final average reward.

In [ ]:
```
from configs.p3_train_atari_linear import config as config_lina

# make env
env = gym.make(config_lina.env_name)
env = MaxAndSkipEnv(env, skip=config_lina.skip_frame)
env = PreproWrapper(env, prepro=greyscale, shape=(80, 80, 1),
                    overwrite_render=config_lina.overwrite_render)

# exploration strategy
exp_schedule = LinearExploration(env, config_lina.eps_begin,
        config_lina.eps_end, config_lina.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_lina.lr_begin, config_lina.lr_end,
        config_lina.lr_nsteps)

# train model
model = DQNTrain(LinearQNet, env, config_lina, device)
print("Linear Q-Net Architecture:\n", model.q_net)
model.run(exp_schedule, lr_schedule)
```

```
Evaluating...

Linear Q-Net Architecture:
 LinearQNet(
  (layer): Linear(in_features=25600, out_features=6, bias=True)
)

Average reward: -20.92 +/- 0.04

130601/500000 [======>.......................] - ETA: 1647s - Loss: 0.1784 - Avg_R: -20.6000 - Max_R: -18.0000 - eps: 0.8825 - Grads: 17.9362 - Max
_Q: 6.9595 - lr: 0.0001
```

Part 4: [BONUS] Playing Atari Games from Pixels - using Deep Q-Networks

This part is extra credit and worth 5 bonus points. We will now train our deep Q-Network from Part 2 on Pong-v0.

Again, no additional implementation is required but you may wish to tweak your CNN architecture in `cnn_qnet.py` and hyperparameters in `configs/p4_train_atari_cnn.py` (however, evaluation will be considered at no farther than the default 5 million steps, so you are not allowed to train for longer). Please note that this training may take a very long time (we tested this on a single GPU and it took around 6 hours).

The bonus points for this question will be allotted based on the best evaluation average reward (EAR) before 5 million time stpes:

1. EAR >= 0.0 : 4/4 points
2. EAR >= -5.0 : 3/4 points
3. EAR >= -10.0 : 3/4 points
4. EAR >= -15.0 : 1/4 points

Deliverable 4: (5 bonus points)

Run the following block of code to train your DQN:

In [ ]:
```
from configs.p4_train_atari_cnn import config as config_cnna


# make env
env = gym.make(config_cnna.env_name)
env = MaxAndSkipEnv(env, skip=config_cnna.skip_frame)
env = PreproWrapper(env, prepro=greyscale, shape=(80, 80, 1),
                    overwrite_render=config_cnna.overwrite_render)

# exploration strategy
exp_schedule = LinearExploration(env, config_cnna.eps_begin,
        config_cnna.eps_end, config_cnna.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_cnna.lr_begin, config_cnna.lr_end,
        config_cnna.lr_nsteps)

# train model
model = DQNTrain(ConvQNet, env, config_cnna, device)
print("CNN Q-Net Architecture:\n", model.q_net)
model.run(exp_schedule, lr_schedule)
```

In [ ]: