In [1]:
```
import random
import itertools
from collections import namedtuple

import gym
import numpy as np
import torch
import torch.nn.functional as F
```

In [2]:
```
env = gym.make('CartPole-v0').unwrapped
```

```
WARN: gym.spaces.Box autodetected dtype as <class 'numpy.float32'>. Please pro
vide explicit dtype.
```

In [3]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [4]:
```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))

class ReplayBuffer:

    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.index = 0

    def add(self, state, action, next_state, reward):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.index] = Transition(state, action, next_state, reward)
        self.index = (self.index + 1) % self.capacity
        ## TODO: create a new Transition tuple and add it to the buffer in the cur
        # then increment the current index making sure to loop the index around wh

    def get_sample(self, batch_size):
        return random.sample(self.buffer, batch_size)
```

In [5]:
```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_DECAY = 200
EPS_END = 0.05
TARGET_UPDATE = 10

state_size = env.observation_space.shape[0]

class DQN(torch.nn.Module):

    def __init__(self):
        super(DQN, self).__init__()
        self.linear1 = torch.nn.Linear(state_size, 32)
        self.linear2 = torch.nn.Linear(32, 32)
        self.output = torch.nn.Linear(32, 2)
        self.steps = 0

    def forward(self, x):
        ## TODO: use linear1, linear2 and output to create a 3 layer net with relu
#          raise NotImplementedError('TODO')
        ### BEGIN Solution
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.relu(x)
        x = self.output(x)
        return x
        ### END Solution

    def get_action(self, state, _eval=False):
        eps_thresh = EPS_END + (EPS_START - EPS_END) * np.exp(-1.0 * self.steps /
        self.steps += 1
        if random.random() > eps_thresh or _eval:
            with torch.no_grad():
                # TODO: get the index of the max value of the output of the networ
                # HINT: look at pytorch's max function
                index = self.forward(state).max(1)[1]
                return index.view(1, 1)
        else:
            return torch.tensor([[random.randrange(2)]], device=device, dtype=torc
```

In [6]:
```
net = DQN().to(device)
target_net = DQN().to(device)
target_net.load_state_dict(net.state_dict())
target_net.eval()
```

Out[6]:
```
DQN(
  (linear1): Linear(in_features=4, out_features=32, bias=True)
  (linear2): Linear(in_features=32, out_features=32, bias=True)
  (output): Linear(in_features=32, out_features=2, bias=True)
)
```

In [7]:
```
optimizer = torch.optim.RMSprop(net.parameters())
replay_buffer = ReplayBuffer(10000)
```

In [8]:
```
def train_step():
    if len(replay_buffer.buffer) < BATCH_SIZE:
        return

    samples = replay_buffer.get_sample(BATCH_SIZE)
    batch = Transition(*zip(*samples))

    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype

    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # TODO: get the values of the output of our net
    state_batch_values = net.forward(state_batch)
    # TODO: from the above values choose only the ones that correspond to the acti
    # HINT: remember that for each sample in the batch their will be 2 output valu
    state_action_values = state_batch_values.gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0

    # TODO: compute the expected state_action values from the next state values us
    # belman equation V(s_t) = V(s_{t+1}) * GAMMA + current_reward
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # TODO: Compute L1 loss between `state_action_values` and `expected_state_acti
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsq

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in net.parameters():
        # clamp gradients between -1 and 1
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

In [9]:

```
from tqdm import tqdm
num_episodes = 1000
rewards = []
for i_episode in tqdm(range(num_episodes)):
    # Initialize the environment and state
    env.reset()
    state = torch.tensor(env.state, device=device, dtype=torch.float32).view(1, -1
    reward_sum = 0
    for t in itertools.count():
        # Select and perform an action
        action = net.get_action(state)
        next_state, reward, done, _ = env.step(action.item())
        reward_sum += reward
        reward = torch.tensor([reward], device=device)

        next_state = torch.tensor(next_state, device=device, dtype=torch.float32).
        # Store the transition in memory
        replay_buffer.add(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the target network)
        train_step()
        if done:
            rewards.append(reward_sum)
            break
    # Update the target network
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(net.state_dict())

print('Complete')
env.render()
env.close()
```

```
100%|████████████████████████████████████████████████████████████████████████
████████████████████████████████████████████████| 1000/1000 [00:19<00:00, 51.
30it/s]

Complete
```

In [10]:

```
print(np.mean(rewards))
print(np.max(rewards))
print(len(rewards))
```

```
10.015
90.0
1000
```

In [ ]: