

Deep Q-Network (DQN) Implementation

Presented by Dongmin Lee

AI Frenz

August, 2019

Outline

1. Environment: CartPole
2. Deep Q-Network (DQN)
 - Learning process
 - Import & Hyperparameter
 - Main loop
 - Train model
 - Train & TensorBoard
 - Learning curve & Test

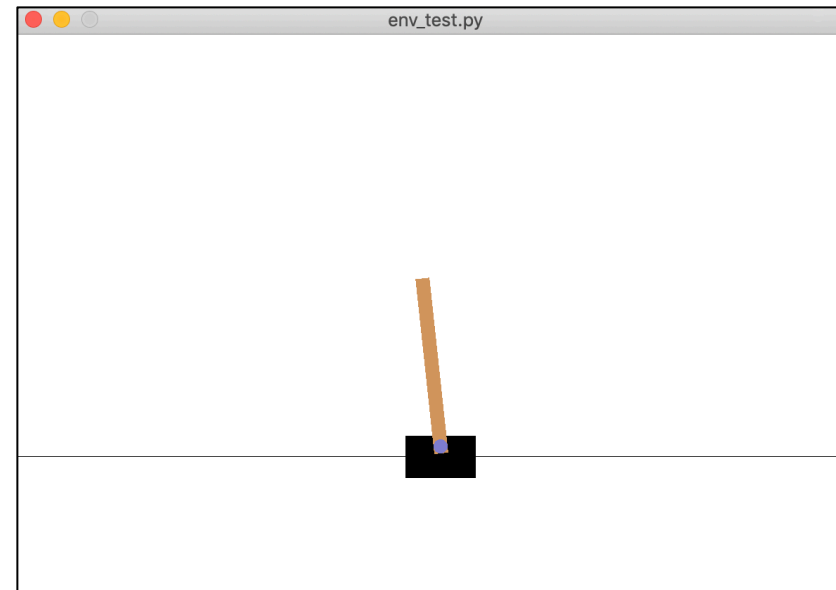
CartPole

- Environment name: CartPole-v1
- States: Continuous observation spaces

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	$\sim -41.8^\circ$	$\sim 41.8^\circ$
3	Pole Velocity At Tip	-Inf	Inf

- Actions: **Discrete** action spaces

Num	Action
0	Push cart to the left
1	Push cart to the right



CartPole

- Test code

```
17 import gym
18
19 env = gym.make('CartPole-v1')
20
21 for episode in range(10000):
22     done = False
23     state = env.reset()
24
25     while not done:
26         env.render()
27
28         action = env.action_space.sample()
29         next_state, reward, done, _ = env.step(action)
30
31         print('state: {} | action: {} | reward: {} | next_state: {} | done: {}'.format(
32             state, action, reward, next_state, done))
33
34         state = next_state
35
36     if done:
37         break
```

```
state: [-0.02315321 -0.04640906 0.01296667 -0.01212249] | action: 0 | reward: 1.0
| next_state: [-0.02408139 -0.24171455 0.01272422 0.28462321] | done: False
state: [-0.02408139 -0.24171455 0.01272422 0.28462321] | action: 1 | reward: 1.0
| next_state: [-0.02891568 -0.04677637 0.01841668 -0.00401958] | done: False
state: [-0.02891568 -0.04677637 0.01841668 -0.00401958] | action: 0 | reward: 1.0
| next_state: [-0.02985121 -0.24215753 0.01833629 0.29441667] | done: False
```

Deep Q-Network (DQN)

DQN (Final version)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$

For episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1, T$ **do**

 With probability ε select a random action a_t

 otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

 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 D

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

 Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

 Every C steps reset $\hat{Q} = Q$

End For

End For

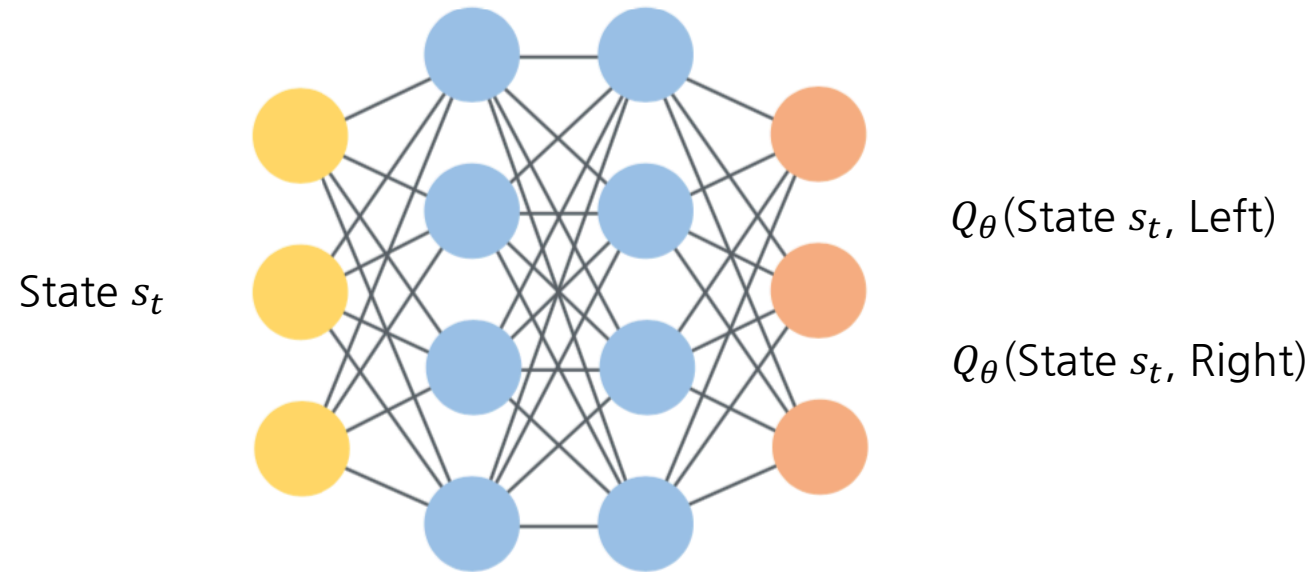
Deep Q-Network (DQN)

Learning process

1. Collect dataset $\{(s_i, a_i, r_i, s'_i)\}$ using some policy, add them to **Buffer**
2. **Sample a mini-batch** $\{(s_j, a_j, r_j, s'_j)\}$ from **Buffer**
3. Compute the **target** $y_j^- \triangleq r_j + \gamma \max_a Q_{\theta^-}(s'_j, a)$ for all j
4. Update $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta} (Q_\theta(s_j, a_j) - y_j^-)$
5. **Target parameters θ^- are synchronized as θ every N steps**

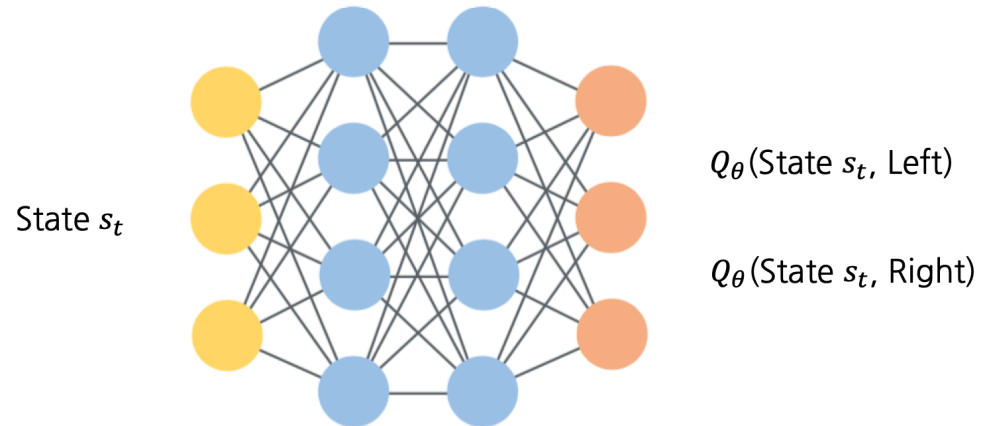
Deep Q-Network (DQN)

Q-Network of CartPole Environment



Deep Q-Network (DQN)

Q-Network of CartPole Environment



```
35 class QNet(nn.Module):
36     def __init__(self, state_size, action_size, args):
37         super(QNet, self).__init__()
38         self.fc1 = nn.Linear(state_size, args.hidden_size)
39         self.fc2 = nn.Linear(args.hidden_size, action_size)
40
41     def forward(self, x):
42         x = torch.tanh(self.fc1(x))
43         q_values = self.fc2(x)
44
45         return q_values
```


Learning process

1. Collect dataset $\{(s_i, a_i, r_i, s'_i)\}$ using some policy, add them to **Buffer**
2. **Sample a mini-batch** $\{(s_j, a_j, r_j, s'_j)\}$ from **Buffer**

```
123     q_values = q_net(torch.Tensor(state))
124     action = get_action(q_values, action_size, args.epsilon)
125
126     next_state, reward, done, _ = env.step(action)
127
128     next_state = np.reshape(next_state, [1, state_size])
129     reward = reward if not done or score == 499 else -1
130     mask = 0 if done else 1
131
132     replay_buffer.append((state, action, reward, next_state, mask))
133
134     state = next_state
135     score += reward
136
137     if steps > args.initial_exploration:
138         args.epsilon -= args.epsilon_decay
139         args.epsilon = max(args.epsilon, 0.1)
140
141     mini_batch = random.sample(replay_buffer, args.batch_size)
```

Learning process

1. Collect dataset $\{(s_i, a_i, r_i, s'_i)\}$ using **some policy**, add them to **Buffer**
2. **Sample a mini-batch** $\{(s_j, a_j, r_j, s'_j)\}$ from **Buffer**

```
123     q_values = q_net(torch.Tensor(state))
124     action = get_action(q_values, action_size, args.epsilon)
125
126     next_state, reward, done, _ = env.step(action)
127
128     next_state = np.reshape(next_state, [1, state_size])
129     reward = reward if not done or score == 499 else -1
130     mask = 0 if done else 1
131
132     replay_buffer.append((state, action, reward, next_state, mask))
133
134     state = next_state
135     score += reward
136
137     if steps > args.initial_exploration:
138         args.epsilon -= args.epsilon_decay
139         args.epsilon = max(args.epsilon, 0.1)
140
141     mini_batch = random.sample(replay_buffer, args.batch_size)
```

Learning process

1. Collect dataset $\{(s_i, a_i, r_i, s'_i)\}$ using **some policy**, add them to **Buffer**
2. **Sample a mini-batch** $\{(s_j, a_j, r_j, s'_j)\}$ from **Buffer**

ϵ -Greedy:

- Choose $a_t \in \arg \max_a Q_\theta(s_t, a)$ with probability $(1 - \epsilon)$
- Choose a random action with probability ϵ

```
47 def get_action(q_values, action_size, epsilon):
48     if np.random.rand() <= epsilon:
49         return random.randrange(action_size)
50     else:
51         _, action = torch.max(q_values, 1)
52         return action.numpy()[0]
```

- Start with high ϵ and gradually reduce

```
137         if steps > args.initial_exploration:
138             args.epsilon -= args.epsilon_decay
139             args.epsilon = max(args.epsilon, 0.1)
```

Learning process

3. Compute the **target** $y_j^- \triangleq r_j + \gamma \max_a Q_{\theta}(s'_j, a)$ for all j
4. Update $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_{\theta}}{d\theta} (Q_{\theta}(s_j, a_j) - y_j^-)$

```
69     criterion = torch.nn.MSELoss()
70
71     # get Q-value
72     q_values = q_net(torch.Tensor(states))
73     q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)
74
75     # get target
76     target_next_q_values = target_q_net(torch.Tensor(next_states))
77     target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]
78
79     loss = criterion(q_value, target.detach())
80     optimizer.zero_grad()
81     loss.backward()
82     optimizer.step()
```

Learning process

3. Compute the **target** $y_j^- \triangleq r_j + \gamma \max_a Q_{\theta}(s'_j, a)$ for all j
4. Update $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_{\theta}}{d\theta} (Q_{\theta}(s_j, a_j) - y_j^-)$

```
69     criterion = torch.nn.MSELoss()
70
71     # get Q-value
72     q_values = q_net(torch.Tensor(states))
73     q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)
74
75     # get target
76     target_next_q_values = target_q_net(torch.Tensor(next_states))
77     target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]
78
79     loss = criterion(q_value, target.detach())
80     optimizer.zero_grad()
81     loss.backward()
82     optimizer.step()
```

Learning process

3. Compute the **target** $y_j^- \triangleq r_j + \gamma \max_a Q_\theta(s'_j, a)$ for all j
4. Update $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta} (Q_\theta(s_j, a_j) - y_j^-)$

Gather: `gather(dim, index)`

```
tensor = torch.Tensor([[1,3], [3,2], [5,4], [1,4], [4,2]])
index = torch.LongTensor([[0], [1], [0], [0], [1]])

tensor = tensor.gather(1, index)
print(tensor)
...
tensor([[1.],
        [2.],
        [5.],
        [1.],
        [2.]])
...
```

Learning process

5. Target parameters θ^- are synchronized as θ every N steps

```
146         if steps % args.update_target == 0:  
147             update_target_model(q_net, target_q_net)
```

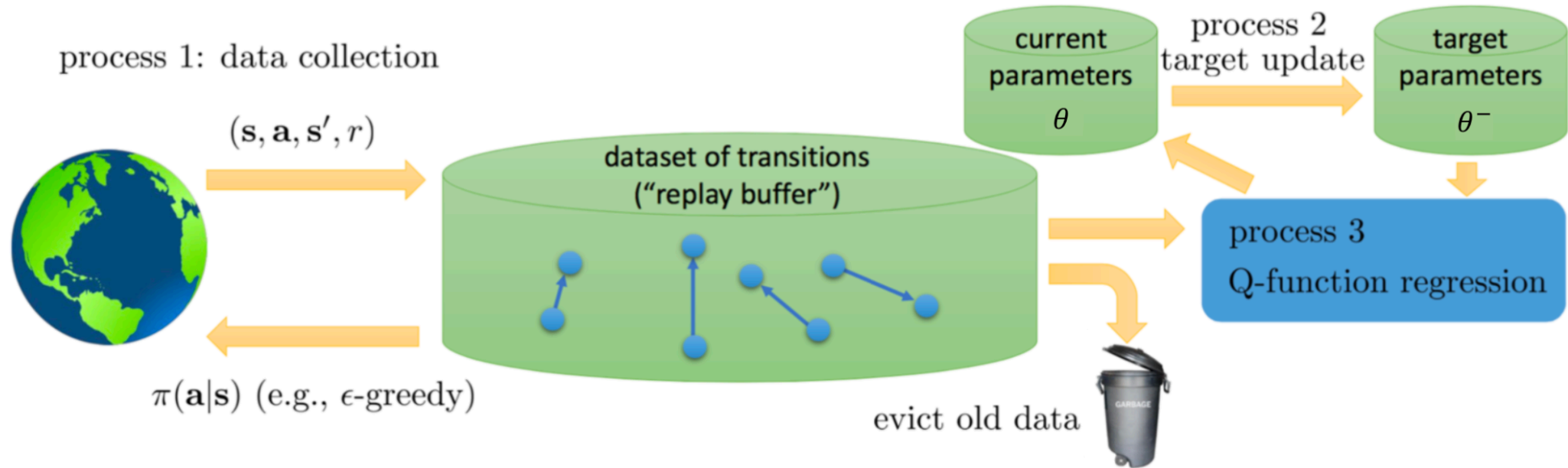
```
54     def update_target_model(q_net, target_q_net):  
55         target_q_net.load_state_dict(q_net.state_dict())
```

- First, Save target network parameters: $\theta^- \leftarrow \theta$

```
101         update_target_model(q_net, target_q_net)
```

Learning process

DQN Algorithm



Import & Hyperparameter

```
1  import os
2  import gym
3  import random
4  import argparse
5  import numpy as np
6  from collections import deque
7
8  import torch
9  import torch.nn as nn
10 import torch.optim as optim
11 from torch.utils.tensorboard import SummaryWriter
12
13 parser = argparse.ArgumentParser()
14 parser.add_argument('--env_name', type=str, default="CartPole-v1")
15 parser.add_argument('--save_path', default='./save_model/', help='')
16 parser.add_argument('--render', action="store_true", default=False)
17 parser.add_argument('--gamma', type=float, default=0.99)
18 parser.add_argument('--buffer_size', type=int, default=10000)
19 parser.add_argument('--batch_size', type=int, default=32)
20 parser.add_argument('--hidden_size', type=int, default=64)
21 parser.add_argument('--learning_rate', type=float, default=1e-3)
22 parser.add_argument('--initial_exploration', type=int, default=1000)
23 parser.add_argument('--epsilon', type=float, default=1.0)
24 parser.add_argument('--epsilon_decay', type=float, default=5e-5)
25 parser.add_argument('--update_target', type=int, default=100)
26 parser.add_argument('--max_iter_num', type=int, default=1000)
27 parser.add_argument('--log_interval', type=int, default=10)
28 parser.add_argument('--goal_score', type=int, default=400)
29 parser.add_argument('--tensorboard_flag', type=str, default=True)
30 parser.add_argument('--logdir', type=str, default='./logs',
31                    help='tensorboard logs directory')
32 args = parser.parse_args()
```

Main loop

Initialize:

- Seed - random number
- Q-Net & Target Q-Net
- Optimizer
- Target parameter θ^-
- TensorBoard
- Replay buffer

```
85  def main():
86      env = gym.make(args.env_name)
87      env.seed(500)
88      torch.manual_seed(500)
89
90      state_size = env.observation_space.shape[0]
91      action_size = env.action_space.n
92      print('state size:', state_size)
93      print('action size:', action_size)
94
95      print('args', args)
96
97      q_net = QNet(state_size, action_size, args)
98      target_q_net = QNet(state_size, action_size, args)
99      optimizer = optim.Adam(q_net.parameters(), lr=args.learning_rate)
100
101      update_target_model(q_net, target_q_net)
102
103      if args.tensorboard_flag:
104          writer = SummaryWriter()
105
106      replay_buffer = deque(maxlen=args.buffer_size)
107      running_score = 0
108      steps = 0
```

Main loop

Repeat steps

1. Collect dataset $\{(s_i, a_i, r_i, s'_i)\}$ using some policy, add them to **Buffer**

```
110     for episode in range(args.max_iter_num):
111         done = False
112         score = 0
113
114         state = env.reset()
115         state = np.reshape(state, [1, state_size])
116
117         while not done:
118             if args.render:
119                 env.render()
120
121             steps += 1
122
123             q_values = q_net(torch.Tensor(state))
124             action = get_action(q_values, action_size, args.epsilon)
125
126             next_state, reward, done, _ = env.step(action)
127
128             next_state = np.reshape(next_state, [1, state_size])
129             reward = reward if not done or score == 499 else -1
130             mask = 0 if done else 1
131
132             replay_buffer.append((state, action, reward, next_state, mask))
133
134             state = next_state
135             score += reward
```

Main loop

Repeat steps

1. Collect dataset $\{(s_i, a_i, r_i, s'_i)\}$ using some policy, add them to **Buffer**
2. **Sample a mini-batch** $\{(s_j, a_j, r_j, s'_j)\}$ from **Buffer**
3. Update Q-Network parameter θ
4. Target parameters θ^- are synchronized as θ every N steps

```
137         if steps > args.initial_exploration:
138             args.epsilon -= args.epsilon_decay
139             args.epsilon = max(args.epsilon, 0.1)
140
141             mini_batch = random.sample(replay_buffer, args.batch_size)
142
143             q_net.train(), target_q_net.train()
144             train_model(q_net, target_q_net, optimizer, mini_batch)
145
146             if steps % args.update_target == 0:
147                 update_target_model(q_net, target_q_net)
```

Main loop

Repeat steps

- Compute running score
- Print logs & Visualize with TensorBoard
- Running score > 400
 - Save Q-Network parameter
 - Exit

```
149     score = score if score == 500.0 else score + 1
150     running_score = 0.99 * running_score + 0.01 * score
151
152     if episode % args.log_interval == 0:
153         print('{ } episode | running_score: {:.2f} | epsilon: {:.2f}'.format(
154             episode, running_score, args.epsilon))
155         if args.tensorboard_flag:
156             writer.add_scalar('log/score', float(score), episode)
157
158     if running_score > args.goal_score:
159         if not os.path.isdir(args.save_path):
160             os.makedirs(args.save_path)
161
162         ckpt_path = args.save_path + 'model.pth.tar'
163         torch.save(q_net.state_dict(), ckpt_path)
164         print('Running score exceeds 400. So end')
165         break
```

Train model

- Mini-batch → Numpy array
 - mini_batch - (32, 5)
- Divide Mini-batch
 - states - (32, 4)
 - next_states - (32, 4)
 - actions - (32)
 - rewards - (32)
 - masks - (32)
- List → Torch tensor
 - actions - (32)
 - rewards - (32)
 - masks - (32)

```
57 def train_model(q_net, target_q_net, optimizer, mini_batch):
58     mini_batch = np.array(mini_batch)
59     states = np.vstack(mini_batch[:, 0])
60     actions = list(mini_batch[:, 1])
61     rewards = list(mini_batch[:, 2])
62     next_states = np.vstack(mini_batch[:, 3])
63     masks = list(mini_batch[:, 4])
64
65     actions = torch.LongTensor(actions)
66     rewards = torch.Tensor(rewards)
67     masks = torch.Tensor(masks)
```

Train model

3. Compute the **target** $y_j^- \triangleq r_j + \gamma \max_a Q_{\theta}(s'_j, a)$ for all j
4. Update $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_{\theta}}{d\theta} (Q_{\theta}(s_j, a_j) - y_j^-)$

```
69     criterion = torch.nn.MSELoss()
70
71     # get Q-value
72     q_values = q_net(torch.Tensor(states))
73     q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)
74
75     # get target
76     target_next_q_values = target_q_net(torch.Tensor(next_states))
77     target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]
78
79     loss = criterion(q_value, target.detach())
80     optimizer.zero_grad()
81     loss.backward()
82     optimizer.step()
```

Train model

- $Q_{\theta}(s_j, a_j)$
 - q_values - (32, 2)
 - action.unsqueeze(1) - (32, 1)
 - q_value - (32)
- Target $y_j^- \triangleq r_j + \gamma \max_a Q_{\theta}(s'_j, a)$
 - target_next_q_values - (32, 2)
 - target_next_q_values.max(1)[0] - (32)
 - target - (32)

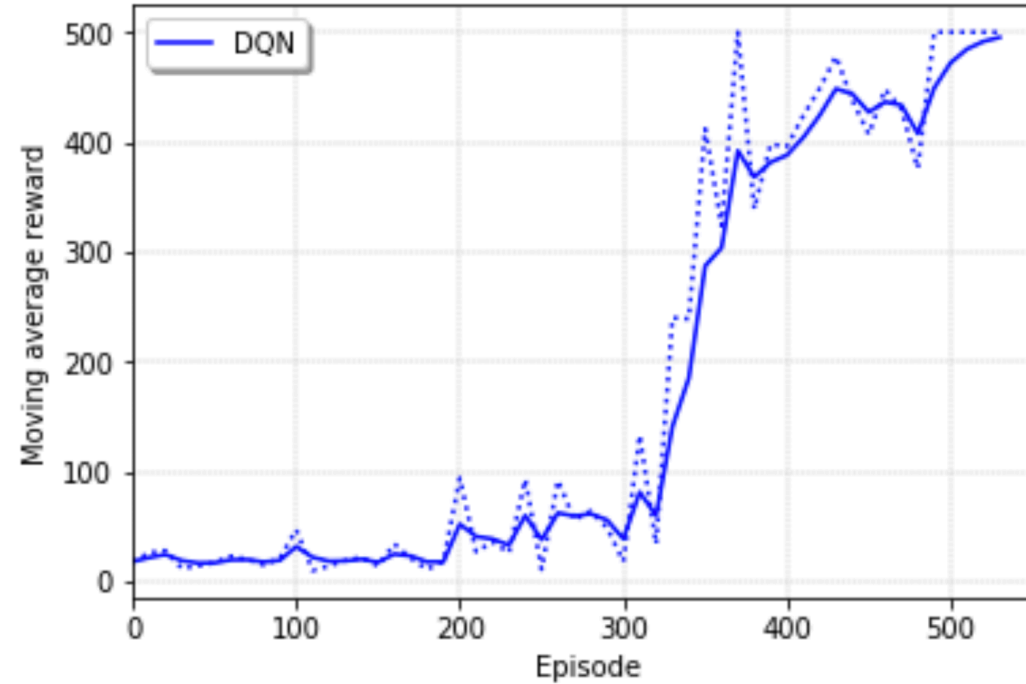
```
69 criterion = torch.nn.MSELoss()
70
71 # get Q-value
72 q_values = q_net(torch.Tensor(states))
73 q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)
74
75 # get target
76 target_next_q_values = target_q_net(torch.Tensor(next_states))
77 target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]
78
79 loss = criterion(q_value, target.detach())
80 optimizer.zero_grad()
81 loss.backward()
82 optimizer.step()
```


Train & TensorBoard

- Terminal A - Train
 - `conda activate env_name`
 - `python train.py`
- Terminal B - TensorBoard
 - `conda activate env_name`
 - `tensorboard --logdir=runs`
 - (In browser) `localhost:6006`

Learning curve & Test

- Learning curve



- Test
 - python test.py

Any Questions?

Thank You