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

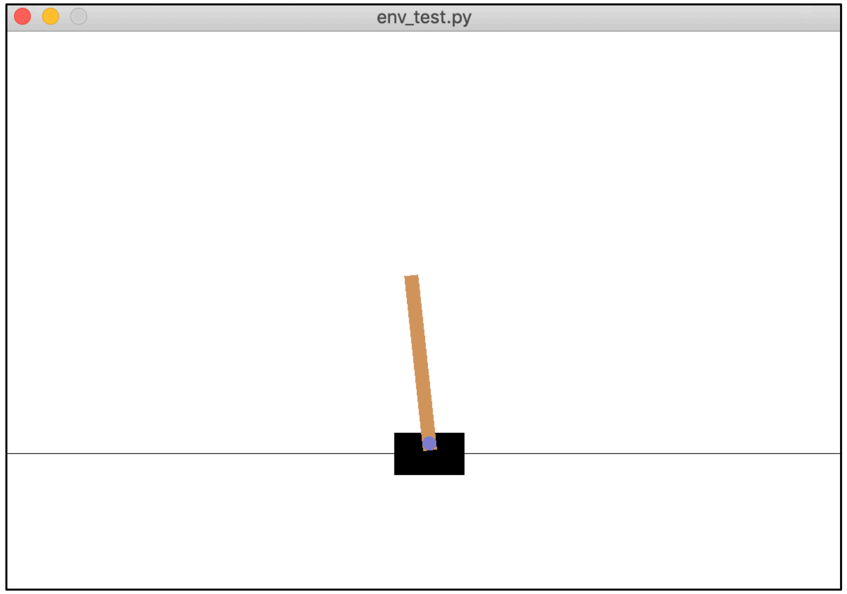
CartPole

- 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	~ -41.8°	~ 41.8°
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

```python
import gym

env = gym.make('CartPole-v1')

for episode in range(10000):
    done = False
    state = env.reset()

    while not done:
        env.render()

        action = env.action_space.sample()
        next_state, reward, done, _ = env.step(action)

        print('state: {} | action: {} | reward: {} | next_state: {} | done: {}'.format(
                state, action, reward, next_state, done))

        state = next_state

        if done:
            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 $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        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 $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
        Every C steps reset $\hat{Q} = Q$
    End For
End For

source : https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Deep Q-Network (DQN)

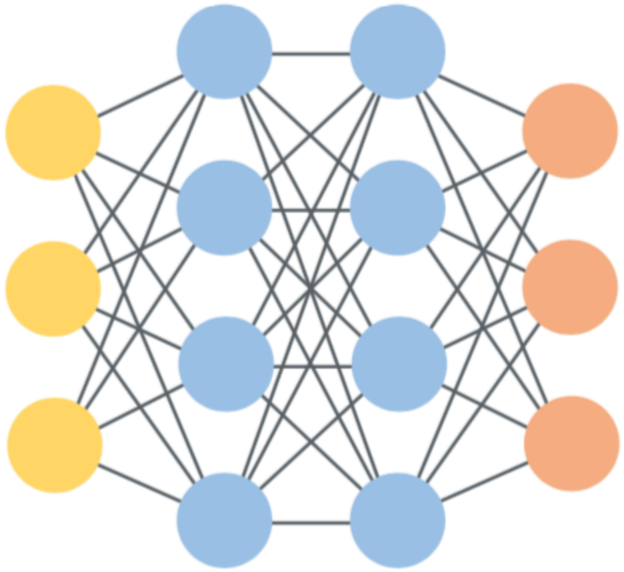
Learning process

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** $y_j^- \triangleq r_j + \gamma \max_a Q_{\theta^-}\left(s_j', a\right)$ for all j
4. Update $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta}\left(Q_\theta\left(s_j, a_j\right) - y_j^-\right)$
5. **Target parameters θ^- are synchronized as θ every N steps**

Deep Q-Network (DQN)

Q-Network of CartPole Environment

State s_t

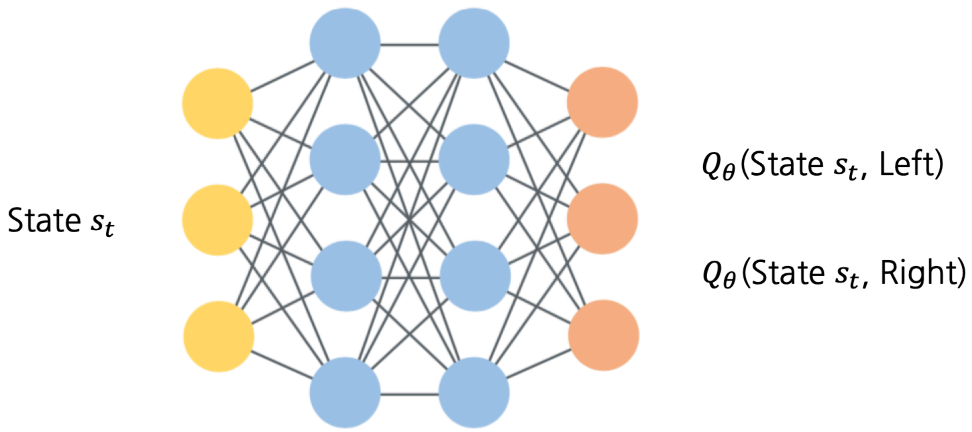


Q_θ(State s_t, Left)

Q_θ(State s_t, Right)

Deep Q-Network (DQN)

Q-Network of CartPole Environment




```python
class QNet(nn.Module):
    def __init__(self, state_size, action_size, args):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(state_size, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, action_size)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        q_values = self.fc2(x)

        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**

```python
q_values = q_net(torch.Tensor(state))
action = get_action(q_values, action_size, args.epsilon)

next_state, reward, done, _ = env.step(action)

next_state = np.reshape(next_state, [1, state_size])
reward = reward if not done or score == 499 else -1
mask = 0 if done else 1

replay_buffer.append((state, action, reward, next_state, mask))

state = next_state
score += reward

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

    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**

```
            q_values = q_net(torch.Tensor(state))
            action = get_action(q_values, action_size, args.epsilon)

            next_state, reward, done, _ = env.step(action)

            next_state = np.reshape(next_state, [1, state_size])
            reward = reward if not done or score == 499 else -1
            mask = 0 if done else 1

            replay_buffer.append((state, action, reward, next_state, mask))

            state = next_state
            score += reward

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

                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

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

- Start with high ϵ and gradually reduce

```
            if steps > args.initial_exploration:
                args.epsilon -= args.epsilon_decay
                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 $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta}\left(Q_\theta(s_j, a_j) - y_j^-\right)$

```
    criterion = torch.nn.MSELoss()

    # get Q-value
    q_values = q_net(torch.Tensor(states))
    q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)

    # get target
    target_next_q_values = target_q_net(torch.Tensor(next_states))
    target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]

    loss = criterion(q_value, target.detach())
    optimizer.zero_grad()
    loss.backward()
    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 $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta}\left(Q_\theta(s_j, a_j) - y_j^-\right)$

```
    criterion = torch.nn.MSELoss()

    # get Q-value
    q_values = q_net(torch.Tensor(states))
    q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)

    # get target
    target_next_q_values = target_q_net(torch.Tensor(next_states))
    target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]

    loss = criterion(q_value, target.detach())
    optimizer.zero_grad()
    loss.backward()
    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 $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta}\left(Q_\theta(s_j, a_j) - y_j^-\right)$

- Gather: gather(dim, index)

```python
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**

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

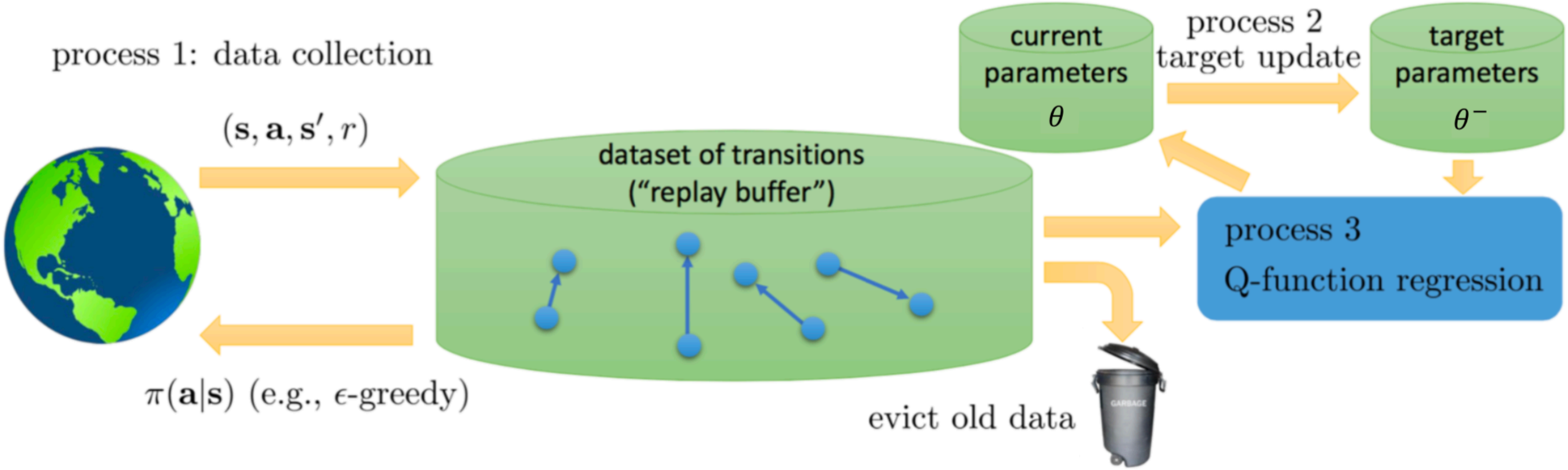
```
def update_target_model(q_net, target_q_net):
    target_q_net.load_state_dict(q_net.state_dict())
```

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

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

Learning process

DQN Algorithm



source : http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_7.pdf

Import & Hyperparameter

```
import os
import gym
import random
import argparse
import numpy as np
from collections import deque

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter

parser = argparse.ArgumentParser()
parser.add_argument('--env_name', type=str, default="CartPole-v1")
parser.add_argument('--save_path', default='./save_model/', help='')
parser.add_argument('--render', action="store_true", default=False)
parser.add_argument('--gamma', type=float, default=0.99)
parser.add_argument('--buffer_size', type=int, default=10000)
parser.add_argument('--batch_size', type=int, default=32)
parser.add_argument('--hidden_size', type=int, default=64)
parser.add_argument('--learning_rate', type=float, default=1e-3)
parser.add_argument('--initial_exploration', type=int, default=1000)
parser.add_argument('--epsilon', type=float, default=1.0)
parser.add_argument('--epsilon_decay', type=float, default=5e-5)
parser.add_argument('--update_target', type=int, default=100)
parser.add_argument('--max_iter_num', type=int, default=1000)
parser.add_argument('--log_interval', type=int, default=10)
parser.add_argument('--goal_score', type=int, default=400)
parser.add_argument('--tensorboard_flag', type=str, default=True)
parser.add_argument('--logdir', type=str, default='./logs',
                    help='tensorboard logs directory')
args = parser.parse_args()
```

Main loop

Initialize:

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

```
def main():
    env = gym.make(args.env_name)
    env.seed(500)
    torch.manual_seed(500)

    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    print('state size:', state_size)
    print('action size:', action_size)

    print('args', args)

    q_net = QNet(state_size, action_size, args)
    target_q_net = QNet(state_size, action_size, args)
    optimizer = optim.Adam(q_net.parameters(), lr=args.learning_rate)

    update_target_model(q_net, target_q_net)

    if args.tensorboard_flag:
        writer = SummaryWriter()

    replay_buffer = deque(maxlen=args.buffer_size)
    running_score = 0
    steps = 0
```

Main loop

Repeat steps

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

```python
    for episode in range(args.max_iter_num):
        done = False
        score = 0

        state = env.reset()
        state = np.reshape(state, [1, state_size])

        while not done:
            if args.render:
                env.render()

            steps += 1

            q_values = q_net(torch.Tensor(state))
            action = get_action(q_values, action_size, args.epsilon)

            next_state, reward, done, _ = env.step(action)

            next_state = np.reshape(next_state, [1, state_size])
            reward = reward if not done or score == 499 else -1
            mask = 0 if done else 1

            replay_buffer.append((state, action, reward, next_state, mask))

            state = next_state
            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**

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

                mini_batch = random.sample(replay_buffer, args.batch_size)

                q_net.train(), target_q_net.train()
                train_model(q_net, target_q_net, optimizer, mini_batch)

                if steps % args.update_target == 0:
                    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

```python
            score = score if score == 500.0 else score + 1
            running_score = 0.99 * running_score + 0.01 * score

            if episode % args.log_interval == 0:
                print('{} episode | running_score: {:.2f} | epsilon: {:.2f}'.format(
                    episode, running_score, args.epsilon))
                if args.tensorboard_flag:
                    writer.add_scalar('log/score', float(score), episode)

            if running_score > args.goal_score:
                if not os.path.isdir(args.save_path):
                    os.makedirs(args.save_path)

                ckpt_path = args.save_path + 'model.pth.tar'
                torch.save(q_net.state_dict(), ckpt_path)
                print('Running score exceeds 400. So end')
                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)

```python
def train_model(q_net, target_q_net, optimizer, mini_batch):
    mini_batch = np.array(mini_batch)
    states = np.vstack(mini_batch[:, 0])
    actions = list(mini_batch[:, 1])
    rewards = list(mini_batch[:, 2])
    next_states = np.vstack(mini_batch[:, 3])
    masks = list(mini_batch[:, 4])

    actions = torch.LongTensor(actions)
    rewards = torch.Tensor(rewards)
    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 $\theta \leftarrow \theta - \alpha \sum_j \frac{dQ_\theta}{d\theta}\left(Q_\theta(s_j, a_j) - y_j^-\right)$

```
    criterion = torch.nn.MSELoss()

    # get Q-value
    q_values = q_net(torch.Tensor(states))
    q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)

    # get target
    target_next_q_values = target_q_net(torch.Tensor(next_states))
    target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]

    loss = criterion(q_value, target.detach())
    optimizer.zero_grad()
    loss.backward()
    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)

```
criterion = torch.nn.MSELoss()

# get Q-value
q_values = q_net(torch.Tensor(states))
q_value = q_values.gather(1, actions.unsqueeze(1)).view(-1)

# get target
target_next_q_values = target_q_net(torch.Tensor(next_states))
target = rewards + masks * args.gamma * target_next_q_values.max(1)[0]

loss = criterion(q_value, target.detach())
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

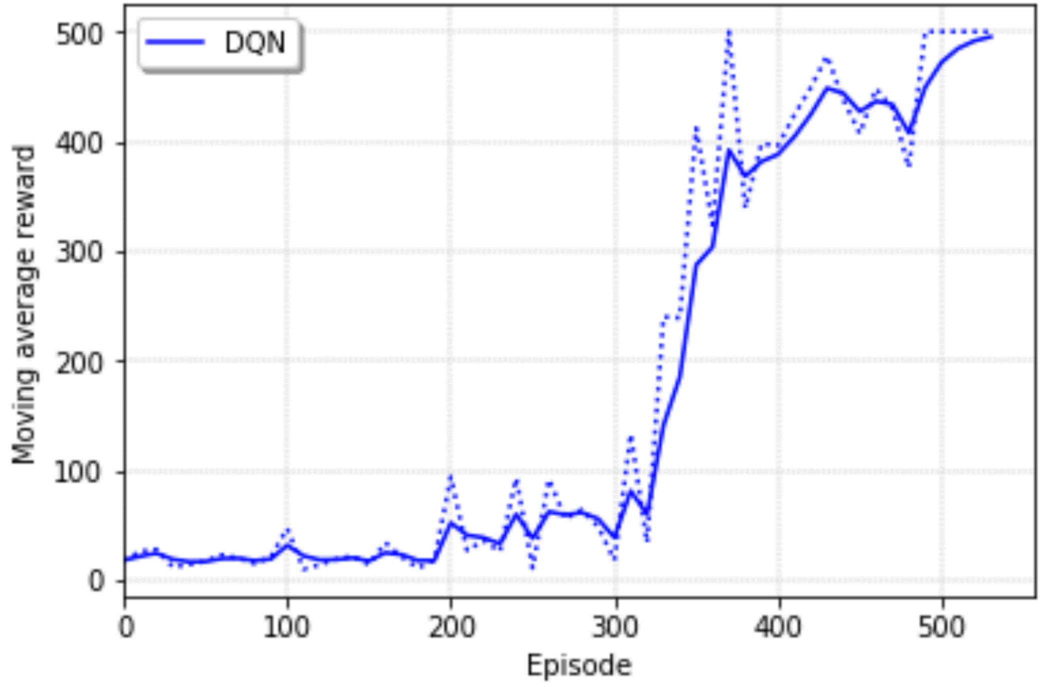
Train & TensorBoard

- 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