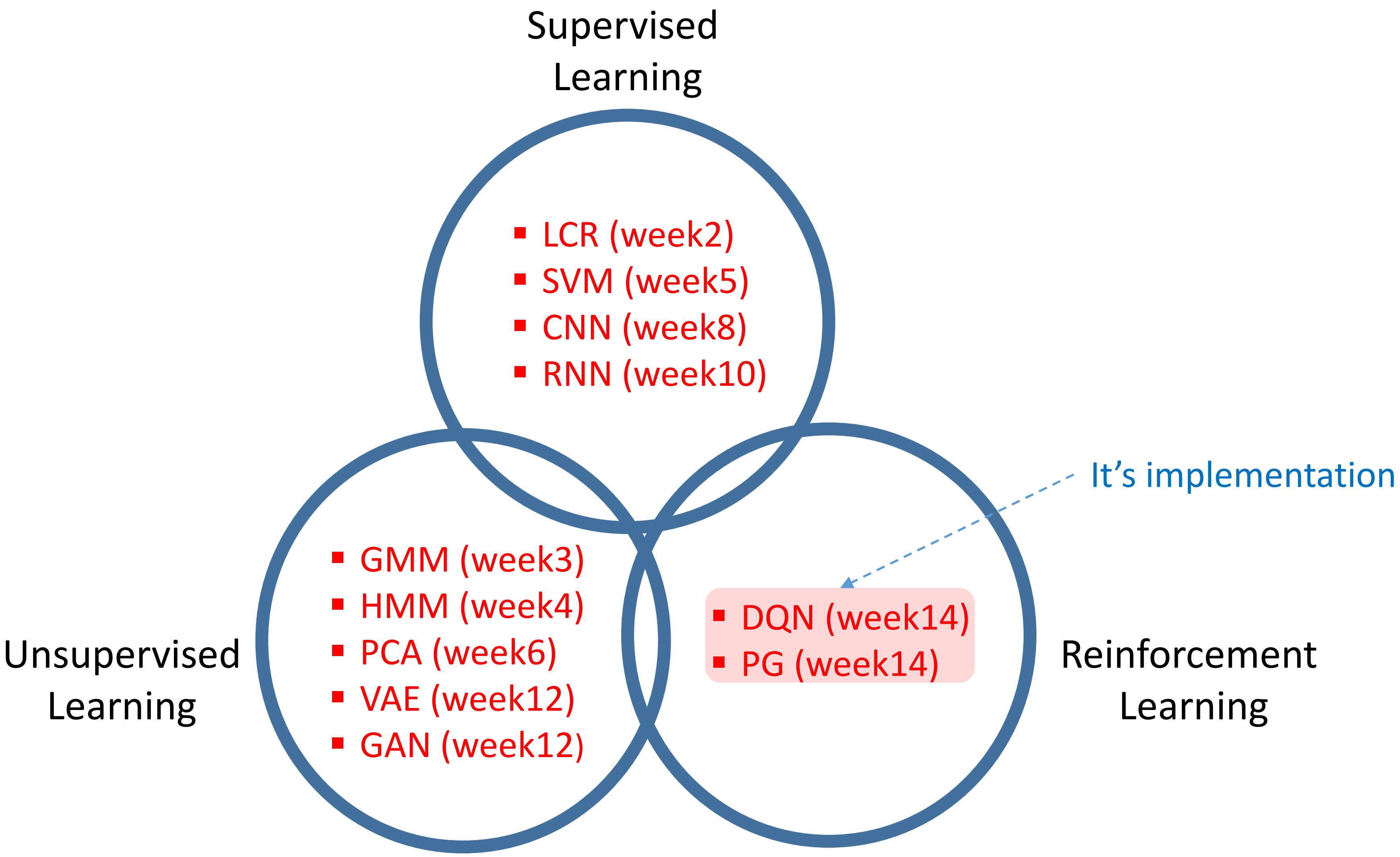
Practical Machine Learning

Lecture 15
Tensorflow – DQN/PG/AC implementation

Dr. Suyong Eum

OSAKA UNIVERSITY

Where we are



Supervised Learning
LCR (week2)
SVM (week5)
CNN (week8)
RNN (week10)
Unsupervised Learning
GMM (week3)
HMM (week4)
PCA (week6)
VAE (week12)
GAN (week12)
Reinforcement Learning
DQN (week14)
PG (week14)
It's implementation

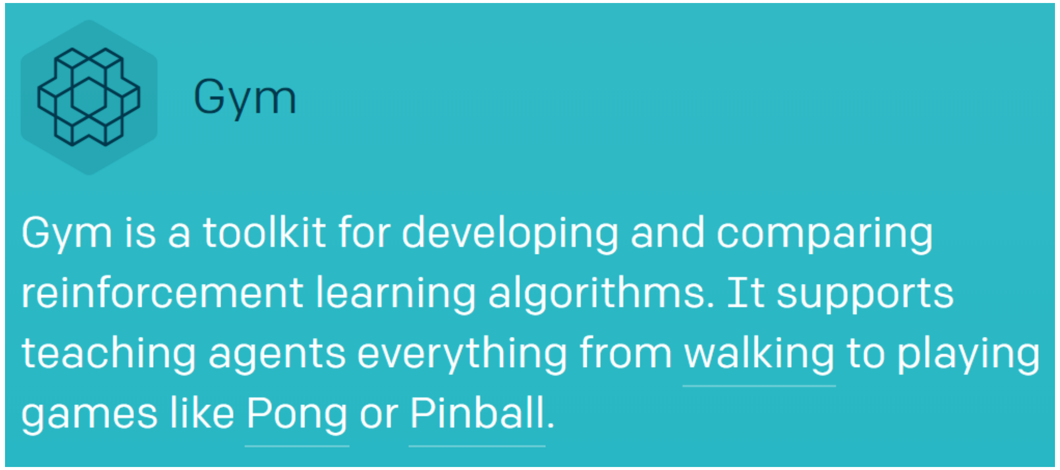
You are going to learn

- ❑ What OpenAI and Gym are,
- ❑ Implementations of
  - Deep Q-Network (DQN) – 2015
  - Policy Gradient (PG)
  - Actor Critic (AC)

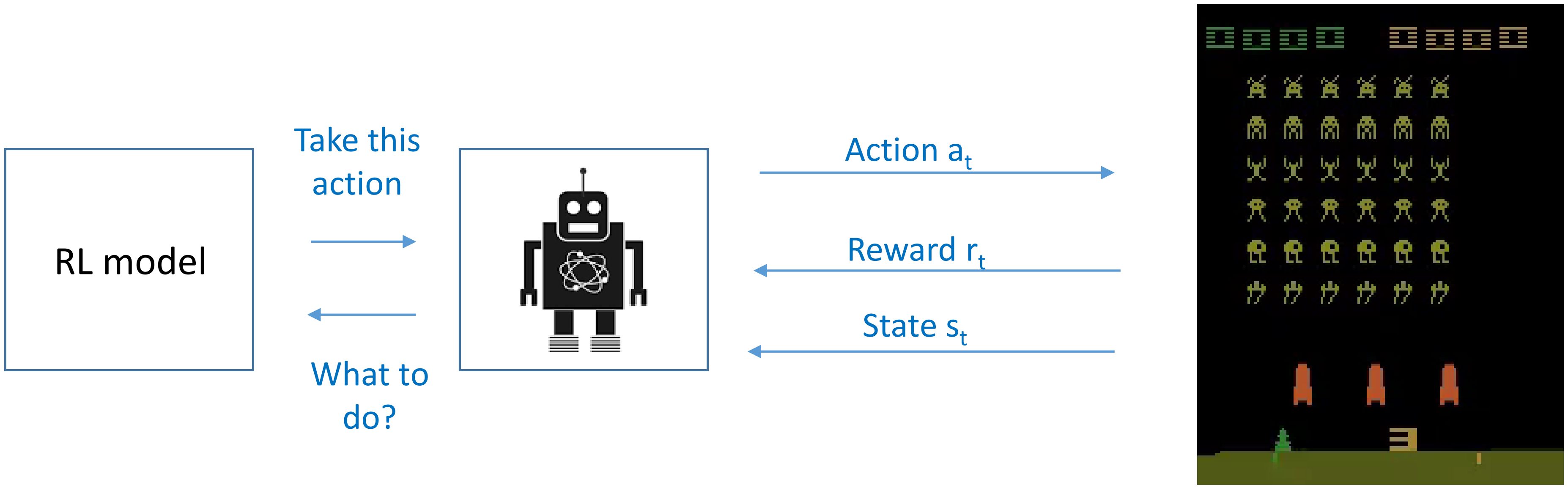
OpenAI Gym

- OpenAI :
  - A non-profit artificial intelligence (AI) research company that aims to promote and develop friendly AI in such a way to benefit humanity as a whole.
  - In October 2015, Elon Musk et al founded the organization.
  - On April 2016, OpenAI released a public beta of "OpenAI Gym", its platform for reinforcement learning research.

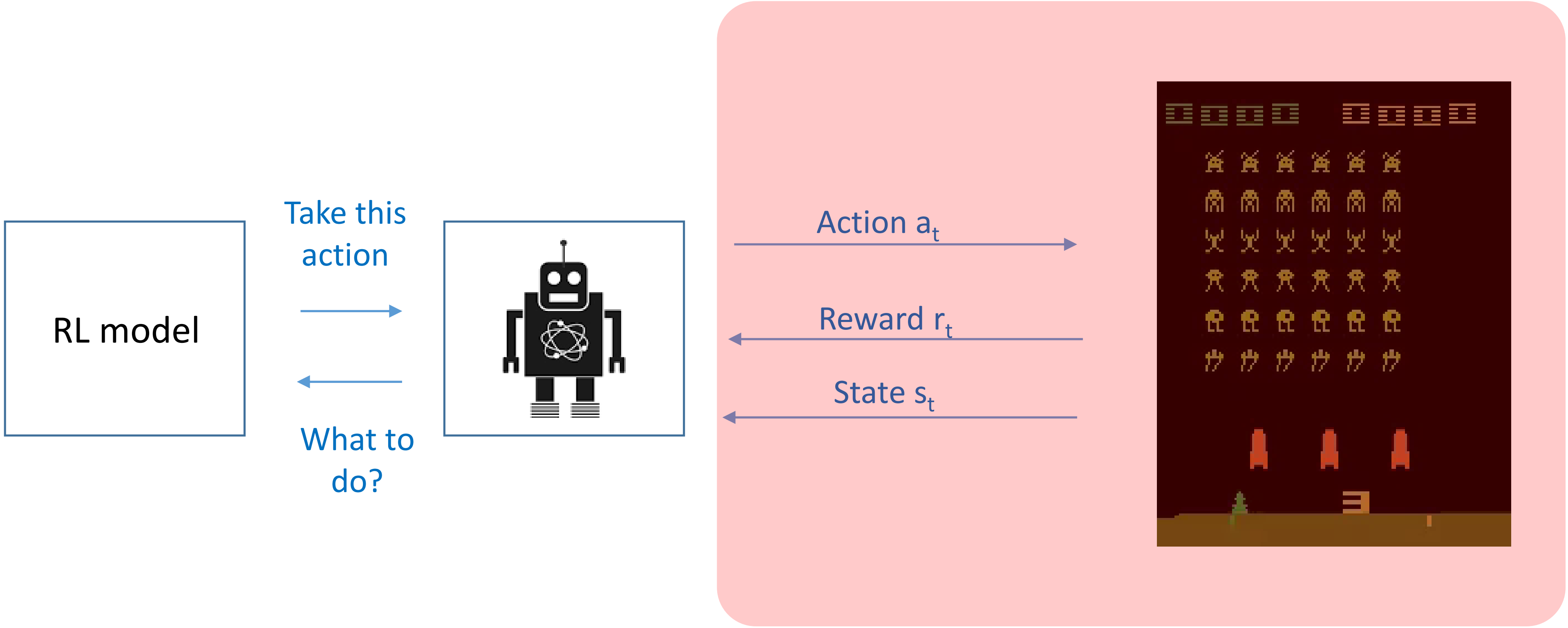
- OpenAI Gym
  - A toolkit for developing and comparing reinforcement learning algorithms
  - https://github.com/openai/gym
  - https://gym.openai.com/



OpenAI Gym framework

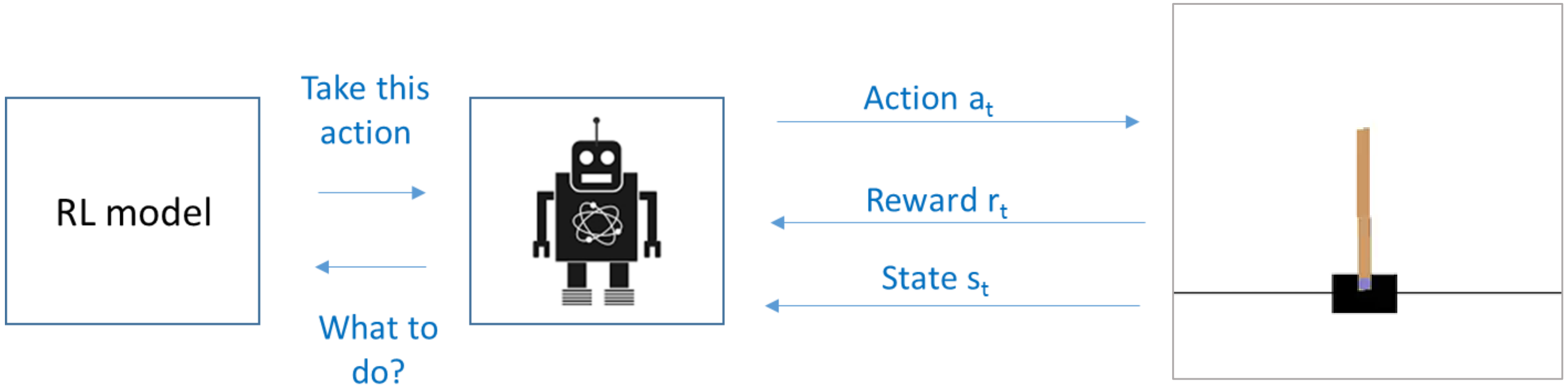


OpenAI Gym framework



Gym framework

OpenAI Gym framework: CartPole game



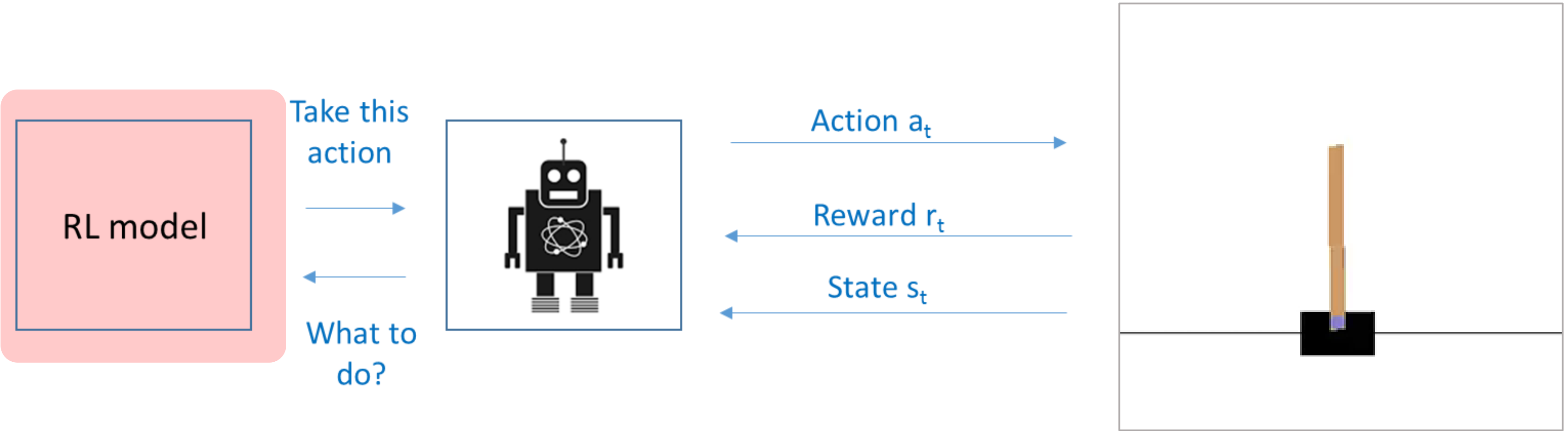

```python
import gym
env=gym.make('CartPole-v0')

# Initialize the current state
current_state = env.reset()

# Action as an input to gym
action = 1 # right

# gym returns values
next_state, reward, done, info = env.step(action)

# see the values
print("current state: ", current_state)
print("next state    : ", next_state)
print("reward        : ", reward)
print("finished?     : ", done)
```

- ❑ Which game you want to play?
- ❑ Initialization
- ❑ Two actions: right(1) or left (0)
- ❑ Asking to Gym box

```
current state:  [ 0.02754178 -0.01649114 -0.01287597 -0.01059267]
next state    :  [ 0.02721195  0.17881308 -0.01308783 -0.30731015]
reward        :  1.0
finished?     :  False
```

- ❑ State: 4 dimension vector
  - [0]: cart position [-2.4 to 2.4]
  - [1]: cart velocity [-Inf to Inf]
  - [2]: pole angle [-42.8˚ to 41.8˚]
  - [3]: pole velocity at tip [-Inf to Inf]

Reinforcement Learning (RL) algorithm



1) Deep Q-Networks (DQN)
2) Policy Gradient (PG)
3) Actor Critic (AC)

DQN implementation

https://github.com/hunkim/DeepLearningZeroToAll

- 07_3_dqn_2015_cartpole.py

DQN architecture (2015)

$\phi(s_t)$

Preprocessing:
RGB to grey

84x84x4

Convolutional Neural Network (CNN) → ReLU → Convolutional Neural Network (CNN) → ReLU → Convolutional Neural Network (CNN) → ReLU → Fully-connected layer → ReLU → Fully-connected layer

Convolutional Neural Network (CNN)	Convolutional Neural Network (CNN)	Convolutional Neural Network (CNN)	Fully-connected layer	Fully-connected layer
8x8x32 filters	4x4x64 filter	3x3x64 filter	512 units	# outputs units

DQN

$Q(s, a = E) = 0.7$

$Q(s, a = W) = 0.1$

$Q(s, a = S) = 0.4$

$Q(s, a = N) = 0.1$

- All possible actions
  - e.g, 2-18 actions

Algorithm (DQN 2015)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N → Data pool size and initialization

Initialize action-value function Q with random weights θ → Weight of 1st NN initialization

Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$ → Weight of 2nd NN initialization

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$ → Preprocessing, e.g., RGB to gray

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

    With probability ε select a random action a_t
    otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ → Action selection using E-greedy: off-policy

    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 → Experience replay

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

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ → Update NN (θ) without changing NN (θ^-)

    Every C steps reset $\hat{Q} = Q$ → Replace NN (θ^-) with NN (θ) every C steps

  End For

End For

- Human-level control through deep reinforcement learning, Feb. 26, 2015, Nature

DQN Code

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 acti

For episode $= 1, M$

Initialize sequenc

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

With probabi

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

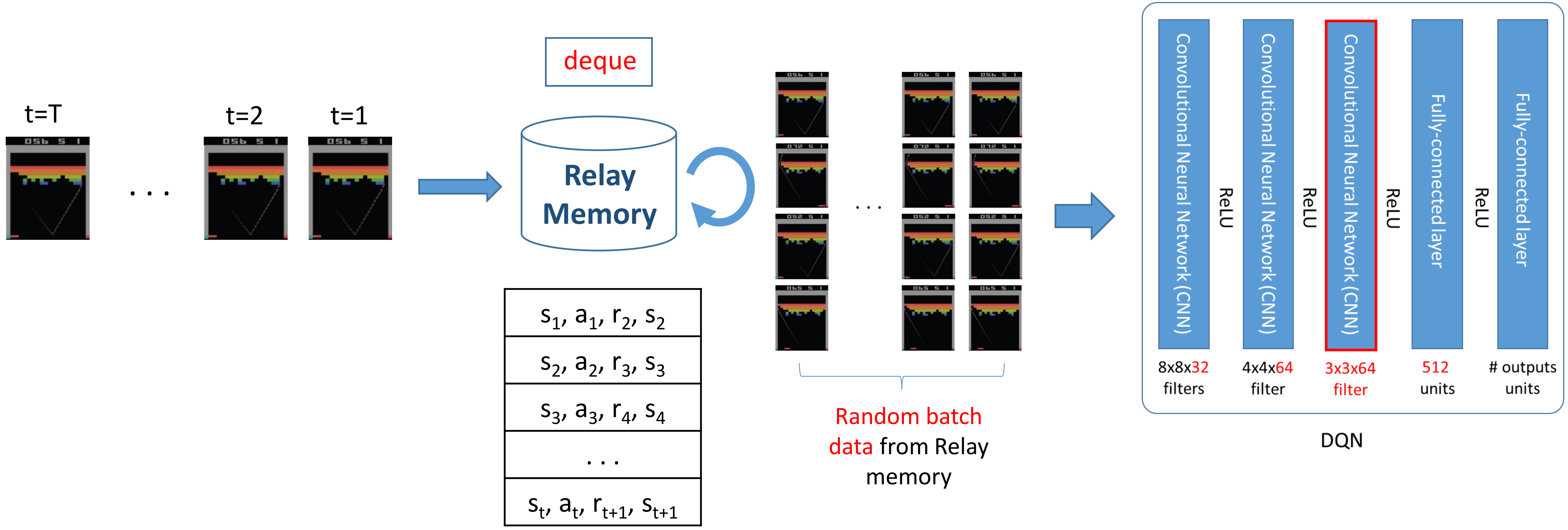
End For

- Data pool size and initialization
- deque(): list-like container with fast operation
- https://docs.python.org/2/library/collections.html

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code: experience replay

- ❑ Consecutive data frames are highly correlated
- ❑ Experience replay aims to remove the correlation between data samples



DQN Code

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

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

With probability ε select a ra

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

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

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

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

- Main Q-Net and target $\hat{Q}$-Net creation
- Copy from main Q-Net to target $\hat{Q}$-Net

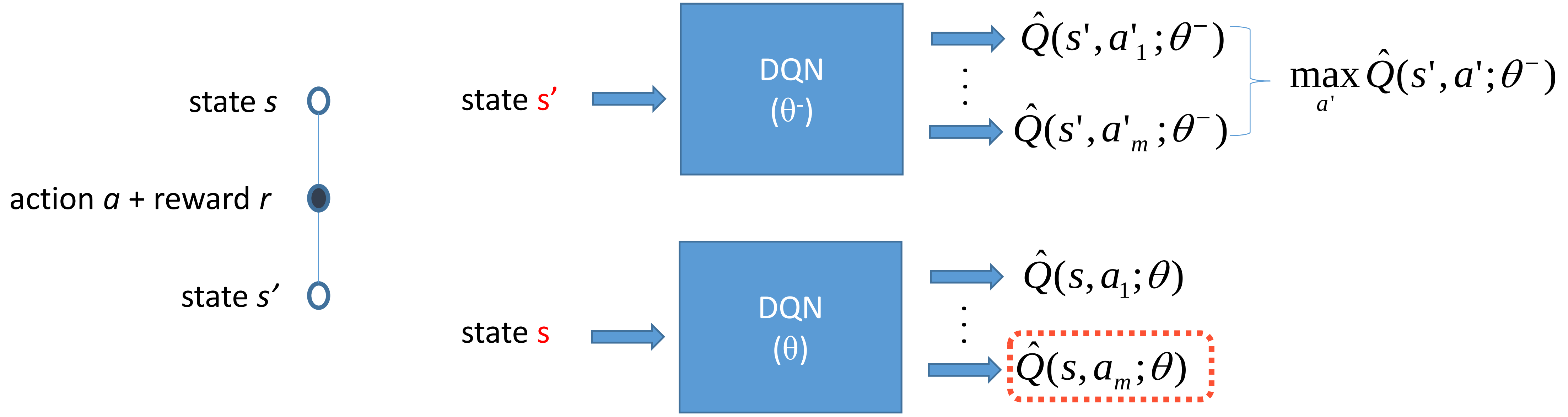
```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code: fixed Q-target

$$L_i(\theta_i) = \mathrm{E}_{(s,a,r,s')\sim U(D)}\left[\left(\overbrace{r + \gamma \max_{a'} Q(s',a'|\theta_i^-)}^{\text{Target Q network}} - \overbrace{Q(s,a,|\theta_i)}^{\text{Main Q network}}\right)^2\right]$$

- Sample data (s,a,r,s') randomly drawn from data pool U(D)
- Experience Replay

- Two different neural networks
- Fixed Q-target



DQN Code

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 prepr
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

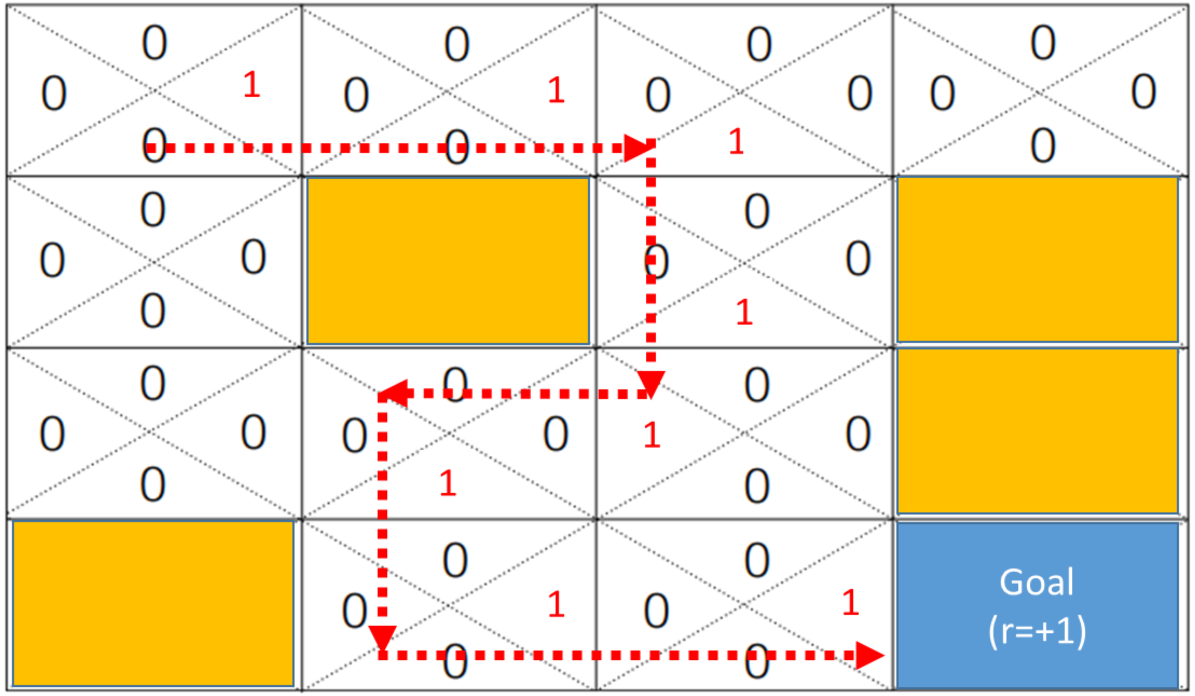
$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

    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

- Decaying E-greedy implementation

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code: Exploit vs Exploration



Decaying E-greedy policy

```
for i in range (1000)

e = 0.1 / (i+1)

if rand < e:
        action = random
else:
        action = argmax(Q(s,a))
```

→ Random to deterministic decision as iteration goes on

DQN Code

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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

- OpenAI GYM emulator

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

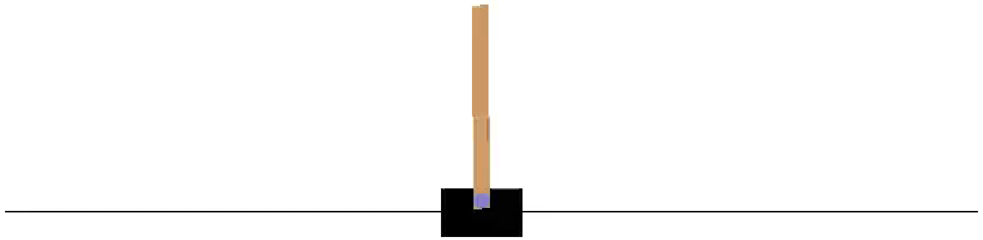
DQN Code: basic information

https://github.com/openai/gym/wiki/CartPole-v0

- Action:
  - 0: left
  - 1: right

```
# Get new state and reward from environment
next_state, reward, done, _ = env.step(action)
```

- State: 4 dimension vector
  - [0]: cart position [-2.4 to 2.4]
  - [1]: cart velocity [-Inf to Inf]
  - [2]: pole angle [-42.8° to 41.8°]
  - [3]: pole velocity at tip [-Inf to Inf]



- Initial state
  - Random values between ±0.05

10 = {ndarray} [-0.1062376 -0.37924474 0.19689548 0.9208114]

- Reward:
  - +1 each unit time if it is not fallen
- Episode Termination
  - Pole Angle is more than ±12°
  - Cart Position is more than ±2.4
  - Episode length is greater than 200 (unit time)
- Training Termination
  - Average reward is greater than or equal to 195 over 100 consecutive trials

DQN Code

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

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

E

- Experience replay implementation
- Wait until collecting enough number of input data (batch_size)
- Then, randomly sample "batch_size" number of inputs from the buffer

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code

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

- Training the main Q-Net given minibatch input

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code

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

    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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

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

  End For

End For

- Training the main Q-Net given minibatch input

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
```

```python
def replay_train(mainDQN: dqn.DQN, targetDQN: dqn.DQN, train_batch: list) -> float:
    """Trains `mainDQN` with target Q values given by `targetDQN`

    Args:
        mainDQN (dqn.DQN): Main DQN that will be trained
        targetDQN (dqn.DQN): Target DQN that will predict Q_target
        train_batch (list): Minibatch of replay memory
            Each element is (s, a, r, s', done)
            [(state, action, reward, next_state, done), ...]

    Returns:
        float: After updating `mainDQN`, it returns a `loss`
    """
    states = np.vstack([x[0] for x in train_batch])
    actions = np.array([x[1] for x in train_batch])
    rewards = np.array([x[2] for x in train_batch])
    next_states = np.vstack([x[3] for x in train_batch])
    done = np.array([x[4] for x in train_batch])

    X = states

    Q_target = rewards + DISCOUNT_RATE * np.max(targetDQN.predict(next_states), axis=1) * ~done

    y = mainDQN.predict(states)
    y[np.arange(len(X)), actions] = Q_target

    # Train our network using target and predicted Q values on each episode
    return mainDQN.update(X, y)
```

```python
            reward = -1

        # Save the experience to our buffer
        replay_buffer.append((state, action, reward, next_state, done))

        if len(replay_buffer) > BATCH_SIZE:
            minibatch = random.sample(replay_buffer, BATCH_SIZE)
            loss, _ = replay_train(mainDQN, targetDQN, minibatch)

        if step_count % TARGET_UPDATE_FREQUENCY == 0:
            sess.run(copy_ops)

        state = next_state
        step_count += 1

    print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code

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

    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 $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

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

- Training the main Q-Net given minibatch input

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
```

```python
def replay_train(mainDQN: dqn.DQN, targetDQN: dqn.DQN, train_batch: list) -> float:
    """Trains `mainDQN` with target Q values given by `targetDQN`

    Args:
        mainDQN (dqn.DQN): Main DQN that will be trained
        targetDQN (dqn.DQN): Target DQN that will predict Q_target
        train_batch (list): Minibatch of replay memory
            Each element is (s, a, r, s', done)
            [(state, action, reward, next_state, done), ...]

    Returns:
        float: After updating `mainDQN`, it returns a `loss`
    """
    states = np.vstack([x[0] for x in train_batch])
    actions = np.array([x[1] for x in train_batch])
    rewards = np.array([x[2] for x in train_batch])
    next_states = np.vstack([x[3] for x in train_batch])
    done = np.array([x[4] for x in train_batch])

    X = states

    Q_target = rewards + DISCOUNT_RATE * np.max(targetDQN.predict(next_states), axis=1) * ~done

    y = mainDQN.predict(states)
    y[np.arange(len(X)), actions] = Q_target

    # Train our network using target and predicted Q values on each episode
    return mainDQN.update(X, y)
```

```python
            reward = -1

        # Save the experience to our buffer
        replay_buffer.append((state, action, reward, next_state, done))

        if len(replay_buffer) > BATCH_SIZE:
            minibatch = random.sample(replay_buffer, BATCH_SIZE)
            loss, _ = replay_train(mainDQN, targetDQN, minibatch)

        if step_count % TARGET_UPDATE_FREQUENCY == 0:
            sess.run(copy_ops)

        state = next_state
        step_count += 1

    print("Episode: {}  steps: {}".format(episode, step_count))
```

DQN Code

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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

- Training the main Q-Net given minibatch input

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
```

```python
def replay_train(mainDQN: dqn.DQN, targetDQN: dqn.DQN, train_batch: list) -> float:
    """Trains `mainDQN` with target Q values given by `targetDQN`

    Args:
        mainDQN (dqn.DQN): Main DQN that will be trained
        targetDQN (dqn.DQN): Target DQN that will predict Q_target
        train_batch (list): Minibatch of replay memory
            Each element is (s, a, r, s', done)
            [(state, action, reward, next_state, done), ...]

    Returns:
        float: After updating `mainDQN`, it returns a `loss`
    """
    states = np.vstack([x[0] for x in train_batch])
    actions = np.array([x[1] for x in train_batch])
    rewards = np.array([x[2] for x in train_batch])
    next_states = np.vstack([x[3] for x in train_batch])
    done = np.array([x[4] for x in train_batch])

    X = states

    Q_target = rewards + DISCOUNT_RATE * np.max(targetDQN.predict(next_states), axis=1) * ~done

    y = mainDQN.predict(states)
    y[np.arange(len(X)), actions] = Q_target

    # Train our network using target and predicted Q values on each episode
    return mainDQN.update(X, y)
```

```python
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

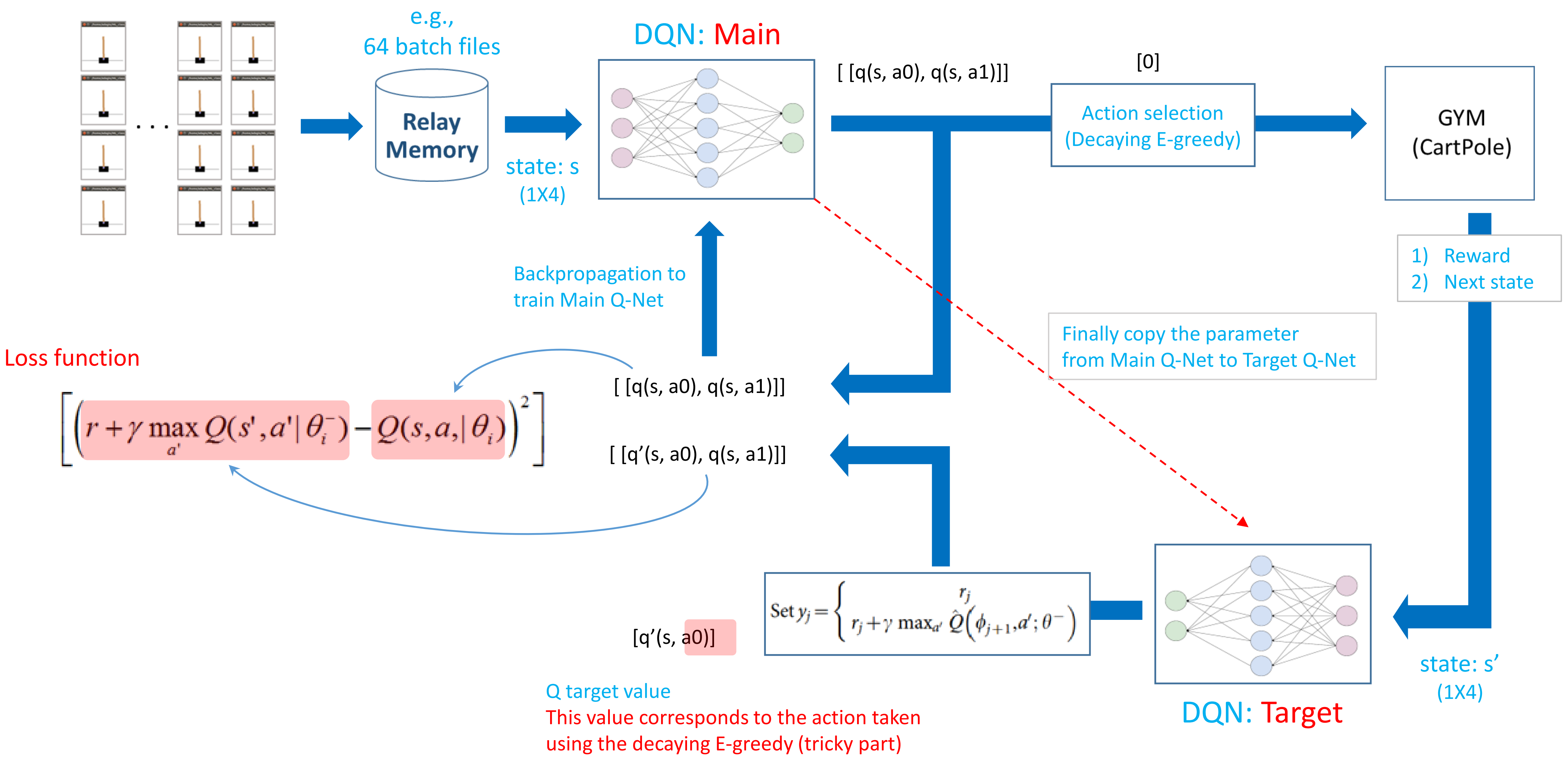
DQN Code

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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

- Copy NN (θ^-) with NN (θ) every C steps

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="main")
    targetDQN = dqn.DQN(sess, INPUT_SIZE, OUTPUT_SIZE, name="target")
    sess.run(tf.global_variables_initializer())

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    for episode in range(MAX_EPISODES):
        e = 1. / ((episode / 10) + 1)
        done = False
        step_count = 0
        state = env.reset()

        while not done:
            if np.random.rand() < e:
                action = env.action_space.sample()
            else:
                # Choose an action by greedily from the Q-network
                action = np.argmax(mainDQN.predict(state))

            # Get new state and reward from environment
            next_state, reward, done, _ = env.step(action)

            if done:  # Penalty
                reward = -1

            # Save the experience to our buffer
            replay_buffer.append((state, action, reward, next_state, done))

            if len(replay_buffer) > BATCH_SIZE:
                minibatch = random.sample(replay_buffer, BATCH_SIZE)
                loss, _ = replay_train(mainDQN, targetDQN, minibatch)

            if step_count % TARGET_UPDATE_FREQUENCY == 0:
                sess.run(copy_ops)

            state = next_state
            step_count += 1

        print("Episode: {}  steps: {}".format(episode, step_count))
```

Big picture for the implementation of DQN



PG implementation

http://karpathy.github.io/2016/05/31/rl/
https://github.com/hunkim/DeepLearningZeroToAll
- 08_1_pg_cartpole.py

PG Code

Algorithm 1 "Vanilla" policy gradient algorithm

Initialize policy parameter θ, baseline b
for iteration$=1, 2, \ldots$ **do**
  Collect a set of trajectories by executing the current policy
  At each timestep in each trajectory, compute
    the *return* $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$, and
    the *advantage estimate* $\hat{A}_t = R_t - b(s_t)$.
  Re-fit the baseline, by minimizing $\|b(s_t) - R_t\|^2$,
    summed over all trajectories and timesteps.
  Update the policy, using a policy gradient estimate $\hat{g}$,
    which is a sum of terms $\nabla_\theta \log \pi(a_t \mid s_t, \theta)\hat{A}_t$
end for

```python
for step in range(max_num_episodes):
    # Initialize x stack, y stack, and rewards
    xs = np.empty(shape=[0, input_size])
    ys = np.empty(shape=[0, 1])
    rewards = np.empty(shape=[0, 1])

    reward_sum = 0
    observation = env.reset()

    while True:
        x = np.reshape(observation, [1, input_size])

        # Run the neural net to determine output
        action_prob = sess.run(action_pred, feed_dict={X: x})

        # Determine the output based on our net, allowing for some randomness
        action = 0 if action_prob < np.random.uniform() else 1

        # Append the observations and outputs for learning
        xs = np.vstack([xs, x])
        ys = np.vstack([ys, action])  # Fake action

        # Determine the outcome of our action
        observation, reward, done, _ = env.step(action)
        rewards = np.vstack([rewards, reward])
        reward_sum += reward

        if done:
            # Determine standardized rewards
            discounted_rewards = discount_rewards(rewards)
            # Normalization
            discounted_rewards = (discounted_rewards - discounted_rewards.mean())
                                 /(discounted_rewards.std() + 1e-7)
            l, _ = sess.run([loss, train],
                            feed_dict={X: xs, Y: ys, advantages: discounted_rewards})
```

PG Code

Algorithm 1 "Vanilla" policy gradient algorithm

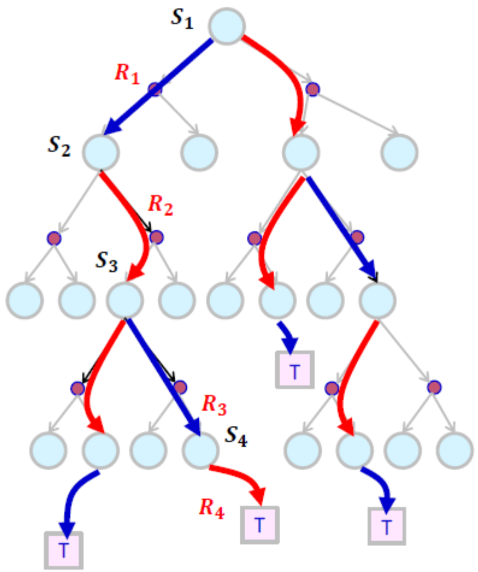
Initialize policy parameter θ, baseline b
for iteration$=1, 2, \ldots$ **do**
  Collect a set of trajectories by executing the current policy
  At each timestep in each trajectory, compute
   the *return* $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$, and
   the *advantage estimate* $\hat{A}_t = R_t - b(s_t)$.
  Re-fit the baseline, by minimizing $\|b(s_t) - R_t\|^2$,
   summed over all tra
  Update the policy, us
   which is a sum of te
end for

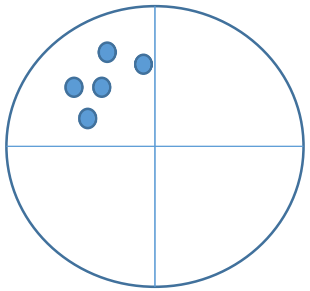
"np.vsack" stacks each event on a trajectory until an episode finish

```
for step in range(max_num_episodes):
    # Initialize x stack, y stack, and rewards
    xs = np.empty(shape=[0, input_size])
    ys = np.empty(shape=[0, 1])
    rewards = np.empty(shape=[0, 1])

    reward_sum = 0
    observation = env.reset()

    while True:
        x = np.reshape(observation, [1, input_size])

        # Run the neural net to determine output
        action_prob = sess.run(action_pred, feed_dict={X: x})

        # Determine the output based on our net, allowing for some randomness
        action = 0 if action_prob < np.random.uniform() else 1

        # Append the observations and outputs for learning
        xs = np.vstack([xs, x])
        ys = np.vstack([ys, action])  # Fake action

        # Determine the outcome of our action
        observation, reward, done, _ = env.step(action)
        rewards = np.vstack([rewards, reward])
        reward_sum += reward

        if done:
            # Determine standardized rewards
            discounted_rewards = discount_rewards(rewards)
            # Normalization
            discounted_rewards = (discounted_rewards - discounted_rewards.mean())
                                /(discounted_rewards.std() + 1e-7)
            l, _ = sess.run([loss, train],
                            feed_dict={X: xs, Y: ys, advantages: discounted_rewards})
```

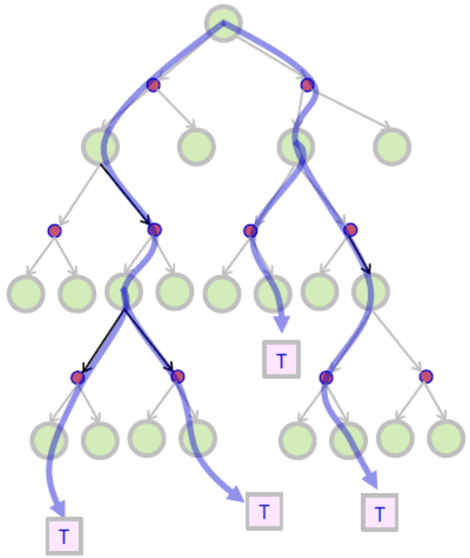
PG Code: Q learning vs Policy gradient

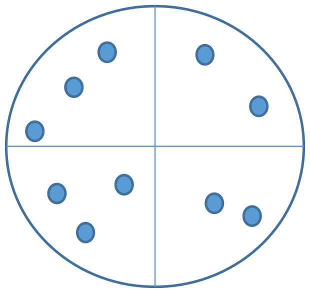
Q learning	Policy gradient
▪ Learning Q(s,a): modeling (Reward) values of actions - Value based approach: learning Q values	▪ Learning π(a): modeling probability of actions - Policy based approach: learning policy directly
▪ Deterministic policies: - e.g., cannot model rock-paper-scissors game	▪ Stochastic policies - e.g., can model rock-paper-scissors game
▪ Off-policy: an action is taken greedily - Greed search to calculate Q(s,a) and then determine a policy	▪ On-policy: an action is taken with a policy - Following a trajectory created by a policy and update it with given reward at the end.
▪ Learning update occurred step-by-step (bootstrapping) - Low variance but high bias	▪ Learning update occurred episode-by-episode - High variance but low bias





High bias





High variance

PG Code

Algorithm 1 "Vanilla" policy gradient algorithm

Initialize policy parameter θ, baseline b
for iteration$=1, 2, \ldots$ **do**
  Collect a set of trajectories by executing the current policy
  At each timestep in each trajectory, compute
   the *return* $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$, and
   the *advantage estimate* $\hat{A}_t = R_t - b(s_t)$.
  Re-fit the baseline, by minimizing $\|b(s_t) - R_t\|^2$,
   summed over all trajectories and timesteps.
  Update the policy, using a policy gradient estimate $\hat{g}$,
   which is a sum of terms $\nabla_\theta \log \pi(a_t \mid s_t, \theta)\hat{A}_t$
end for

```python
for step in range(max_num_episodes):
    # Initialize x stack, y stack, and rewards
    xs = np.empty(shape=[0, input_size])
    ys = np.empty(shape=[0, 1])
    rewards = np.empty(shape=[0, 1])

    reward_sum = 0
    observation = env.reset()

    while True:
        x = np.reshape(observation, [1, input_size])

        # Run the neural net to determine output
        action_prob = sess.run(action_pred, feed_dict={X: x})

        # Determine the output based on our net, allowing for some randomness
        action = 0 if action_prob < np.random.uniform() else 1

        # Append the observations and outputs for learning
        xs = np.vstack([xs, x])
        ys = np.vstack([ys, action])  # Fake action

        # Determine the outcome of our action
        observation, reward, done, _ = env.step(action)
        rewards = np.vstack([rewards, reward])
        reward_sum += reward

        if done:
            # Determine standardized rewards
            discounted_rewards = discount_rewards(rewards)
            # Normalization
            discounted_rewards = (discounted_rewards - discounted_rewards.mean())
                                 /(discounted_rewards.std() + 1e-7)
            l, _ = sess.run([loss, train],
                            feed_dict={X: xs, Y: ys, advantages: discounted_rewards})
```

If an episode finishes,

1) Discounted reward is calculated
2) Then, it is normalized
   - http://karpathy.github.io/2016/05/31/rl/

PG Code

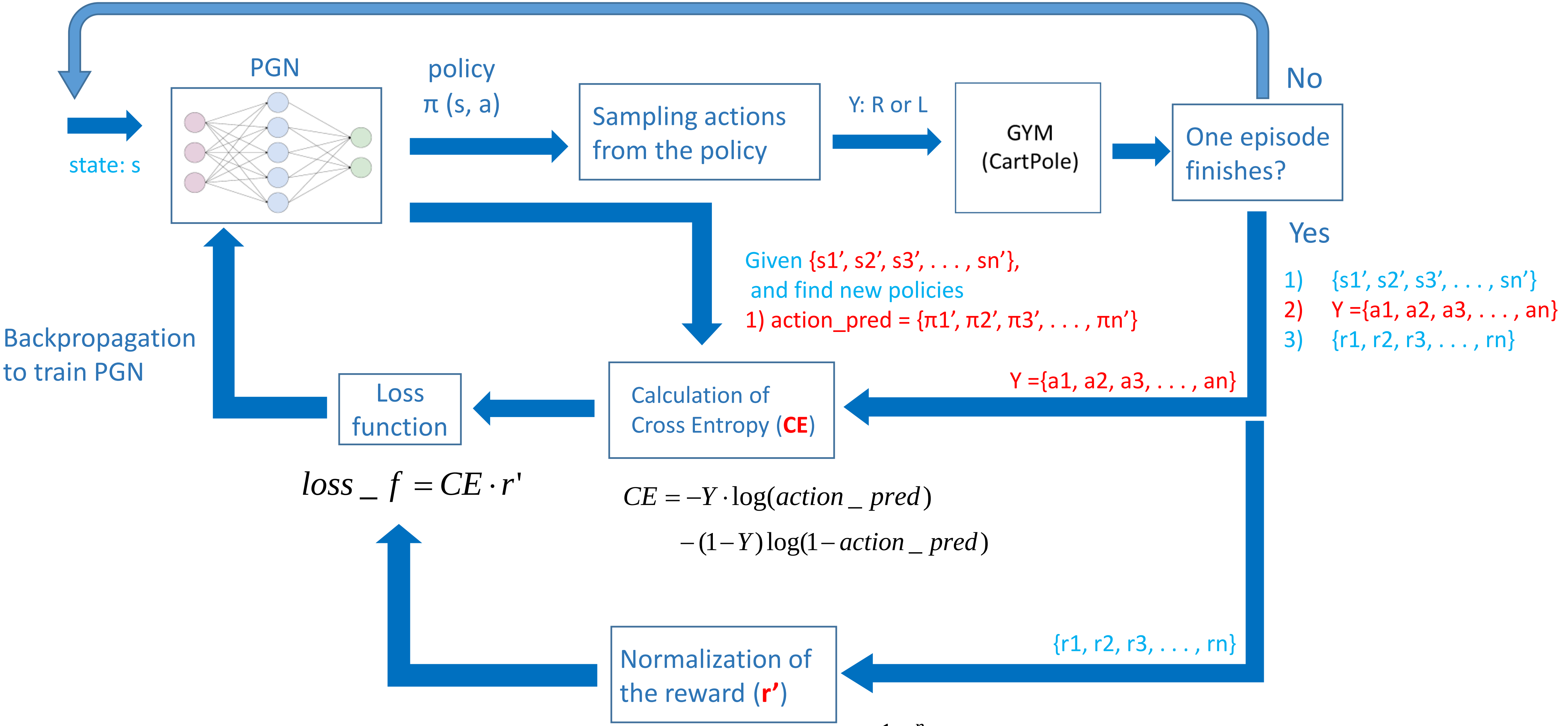
Algorithm 1 "Vanilla" policy gradient algorithm

Initialize policy parameter θ, baseline b
for iteration$=1, 2, \ldots$ **do**
  Collect a set of trajectories by executing the current policy
  At each timestep in each trajectory, compute
   the *return* $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$, and
   the *advantage estimate* $\hat{A}_t = R_t - b(s_t)$.
  Re-fit the baseline, by minimizing $\|b(s_t) - R_t\|^2$,
   summed over all trajectories and timesteps.
  Update the policy, using a policy gradient estimate $\hat{g}$,
   which is a sum of terms $\nabla_\theta \log \pi(a_t \mid s_t, \theta)\hat{A}_t$
end for

```
# Loss function: log likelihood * advantages
log_lik = -Y*tf.log(action_pred) - (1 - Y)*tf.log(1 - action_pred)
loss = tf.reduce_sum(log_lik * advantages)

# Learning
train = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)
```

```
for step in range(max_num_episodes):
    # Initialize x stack, y stack, and rewards
    xs = np.empty(shape=[0, input_size])
    ys = np.empty(shape=[0, 1])
    rewards = np.empty(shape=[0, 1])

    reward_sum = 0
    observation = env.reset()

    while True:
        x = np.reshape(observation, [1, input_size])

        # Run the neural net to determine output
        action_prob = sess.run(action_pred, feed_dict={X: x})

        # Determine the output based on our net, allowing for some randomness
        action = 0 if action_prob < np.random.uniform() else 1

        # Append the observations and outputs for learning
        xs = np.vstack([xs, x])
        ys = np.vstack([ys, action])  # Fake action

        # Determine the outcome of our action
        observation, reward, done, _ = env.step(action)
        rewards = np.vstack([rewards, reward])
        reward_sum += reward

        if done:
            # Determine standardized rewards
            discounted_rewards = discount_rewards(rewards)
            # Normalization
            discounted_rewards = (discounted_rewards - discounted_rewards.mean())
                                 /(discounted_rewards.std() + 1e-7)
            l, _ = sess.run([loss, train],
                            feed_dict={X: xs, Y: ys, advantages: discounted_rewards})
```

Big picture for the implementation of PG



$$loss\_f = CE \cdot r'$$

$$CE = -Y \cdot \log(action\_pred) - (1-Y)\log(1-action\_pred)$$

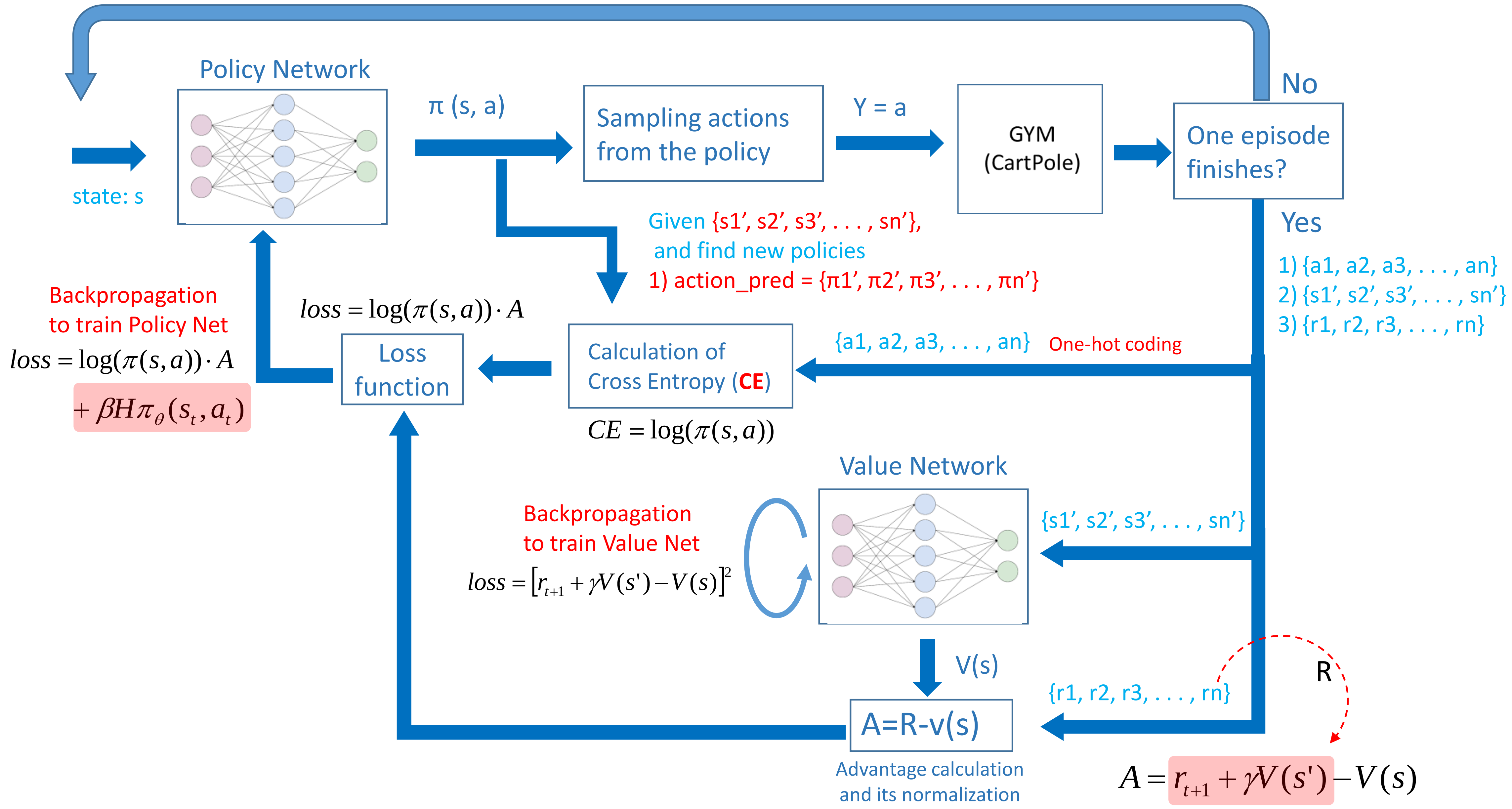
Using its mean $r_k^{'} = r_k - \frac{1}{n}\sum_{i=1}^{n} r_i$

Actor Advantage Critic (A2C)

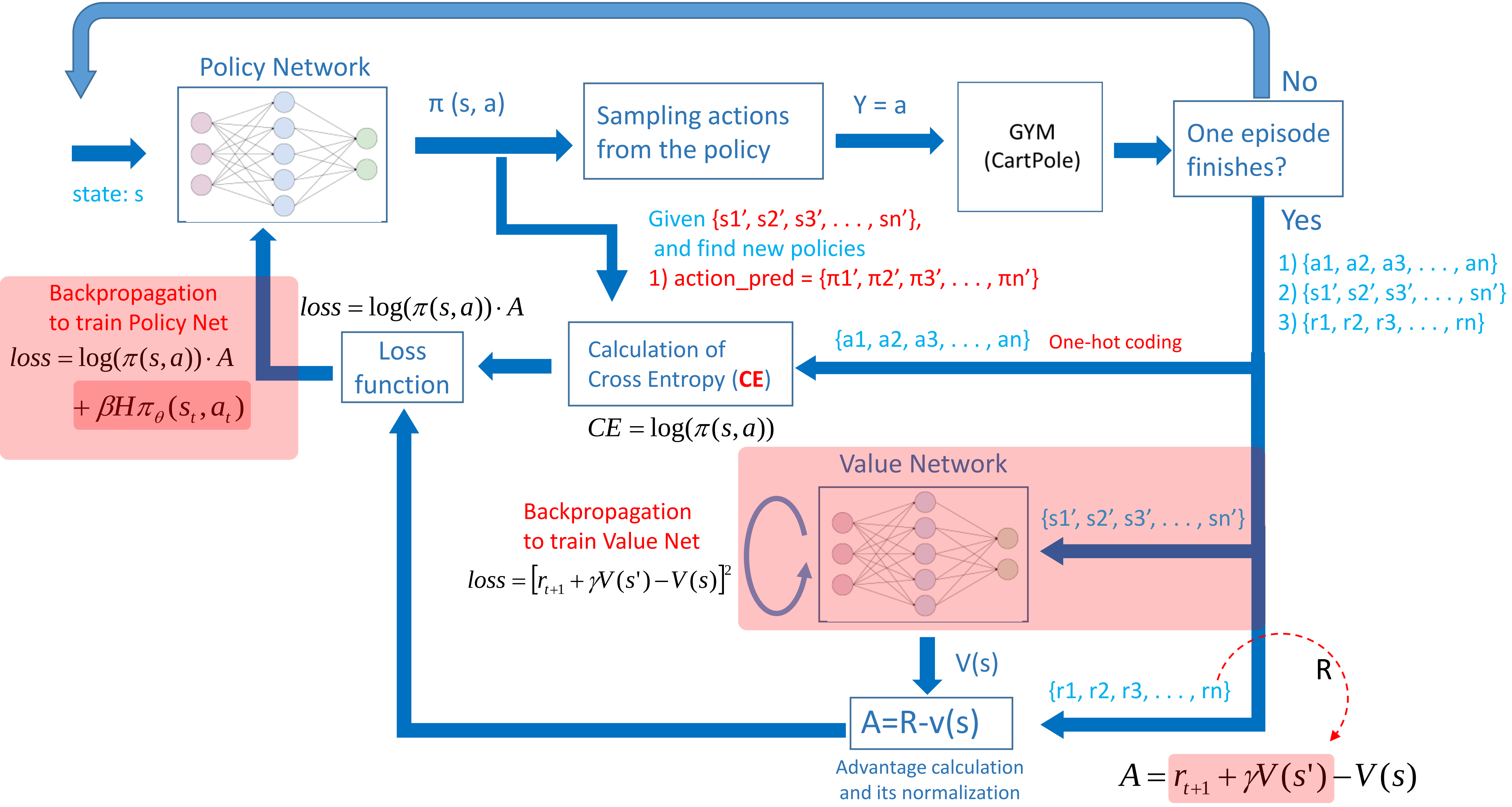
https://github.com/hunkim/DeepLearningZeroToAll

- 10_1_Actor_Critic.ipynb

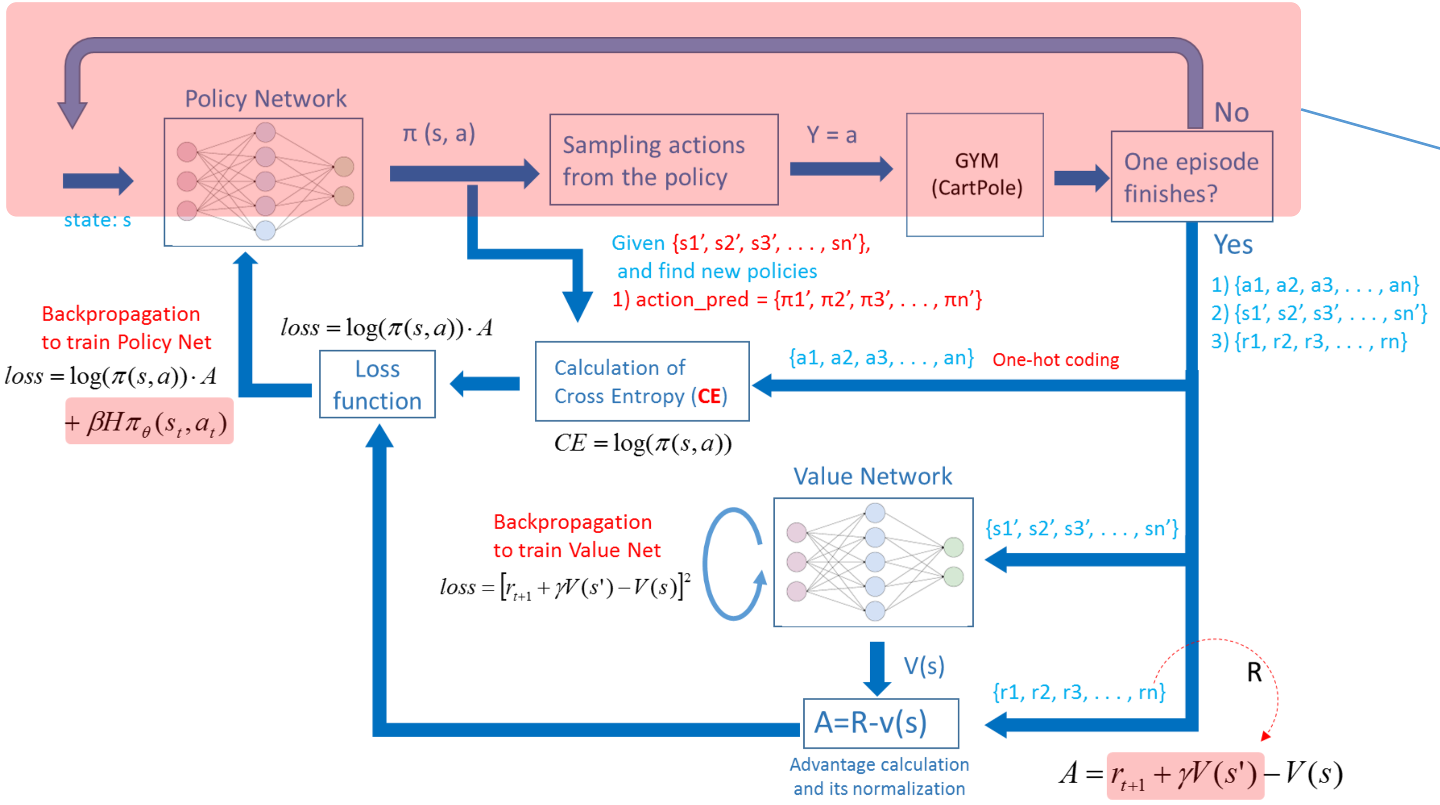
Big picture for the implementation of AC



Big picture for the implementation of AC



AC Code

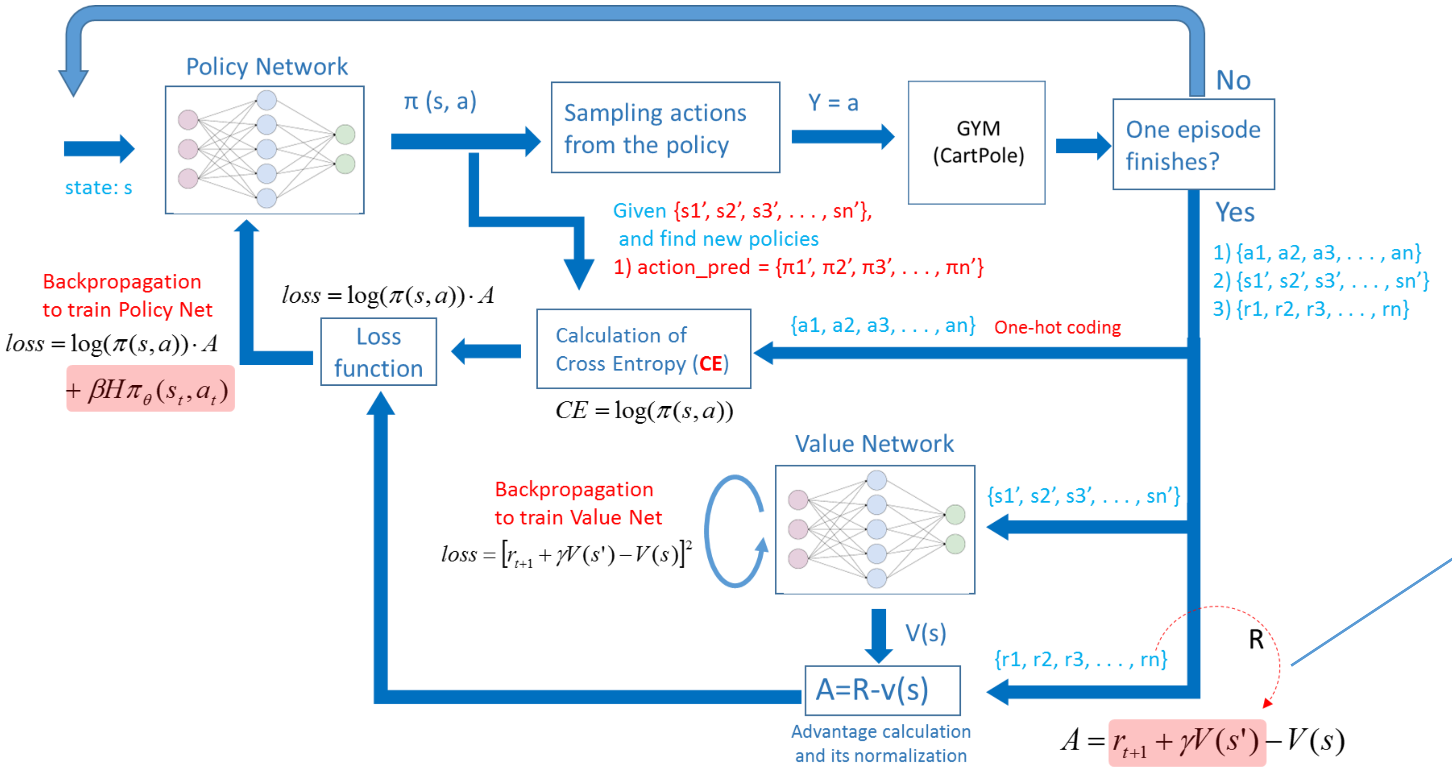



```python
for episode in range(MAX_EPISODES):
    s = env.reset()
    done = False

    s_list = []
    a_list = []
    r_list = []

    episode_r = 0

    while not done:

        s = preprocess_state(s)
        a = agent.choose_an_action(s)

        s2, r, done, info = env.step(a)

        s_list.append(s)
        a_list.append(a)
        r_list.append(r)

        s = s2

        episode_r += r

    a_list = preprocess_action(a_list, action_n)

    agent.train(np.vstack(s_list), a_list, r_list)
```

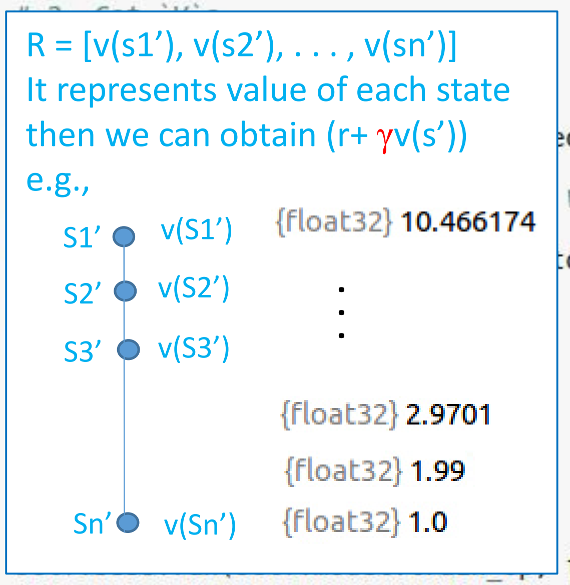
Until one episode finishes

One-hot encoding for cross entropy calculation

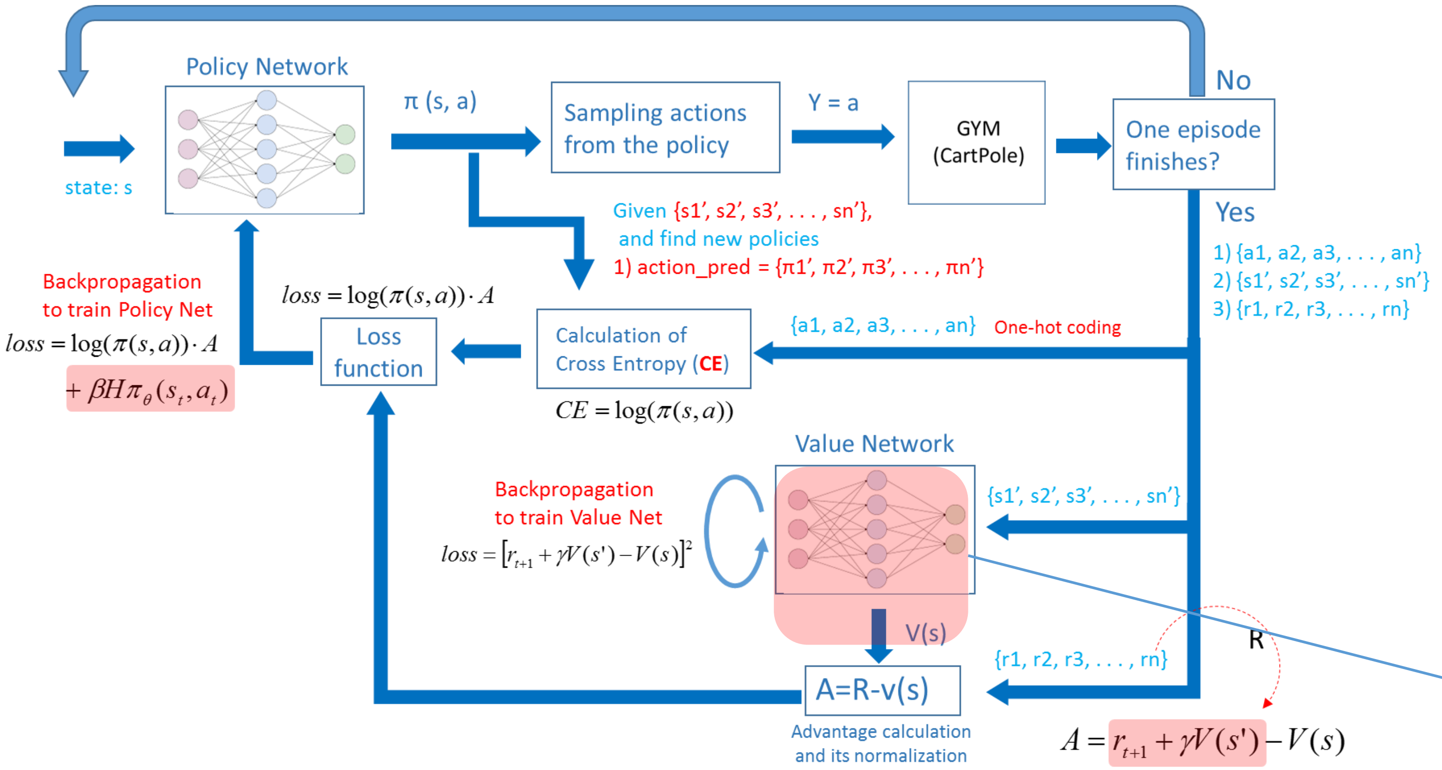
AC Code




```python
def train(self, S, A, R):
    """ Train the actor critic networks

    (1) Compute discounted rewards R
    (2) Compute advantage values A = R - V
    (3) Perform gradients updates

    """

    def discount_rewards(r, gamma=0.99):
        """ take 1D float array of rewards and compute dis
        discounted_r = np.zeros_like(r, dtype=np.float32)
        running_add = 0

        for t in reversed(range(len(r))):
            running_add = running_add * gamma + r[t]
            discounted_r[t] = running_add

        return discounted_r

    # 1. Get discounted `R`s
    R = discount_rewards(R)

                                                   ed_dict=feed)
                                                   V
                                                   td(ADV) + 1e-8)

                                                   feed_dict=feed)
```



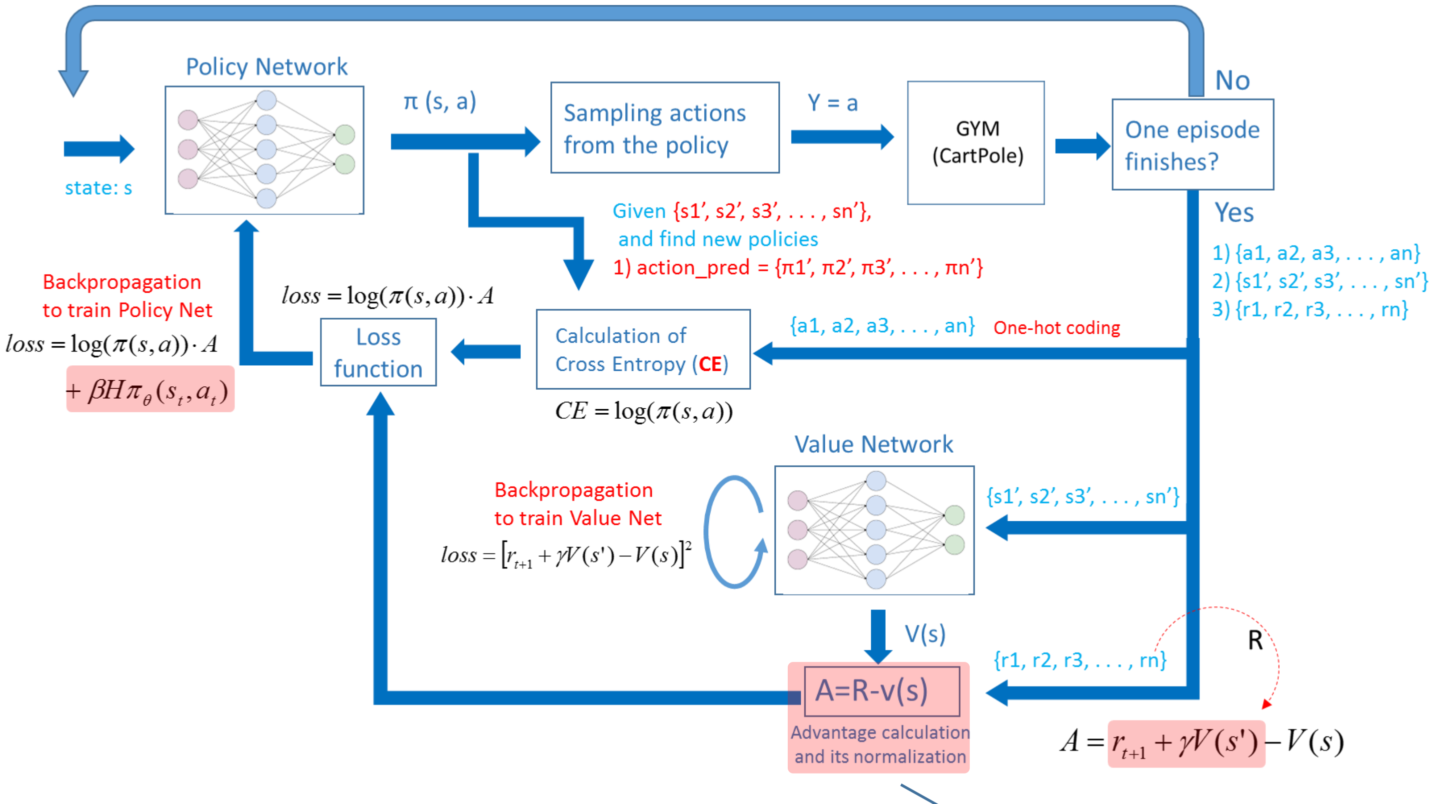
AC Code




```python
def train(self, S, A, R):
    """ Train the actor critic networks

    (1) Compute discounted rewards R
    (2) Compute advantage values A = R - V
    (3) Perform gradients updates

    """

    def discount_rewards(r, gamma=0.99):
        """ take 1D float array of rewards and compute dis
        discounted_r = np.zeros_like(r, dtype=np.float32)
        running_add = 0

        for t in reversed(range(len(r))):
            running_add = running_add * gamma + r[t]
            discounted_r[t] = running_add

        return discounted_r

    # 1. Get discounted `R`s
    R = discount_rewards(R)

    # 2. Get `V`s
    feed = {
        self.model.S: S
    }
    V = self.sess.run(self.model.V, feed_dict=feed)

    # 3. Get Advantage values, A = R - V
    ADV = R - V
    ADV = (ADV - np.mean(ADV)) / (np.std(ADV) + 1e-8)

    # 4. Perform gradient descents
    feed = {
        self.model.S: S,
        self.model.A: A,
        self.model.ADV: ADV,
        self.model.R: R
    }

    self.sess.run(self.model.train_op, feed_dict=feed)
```

Value Network produces value of each state: V(s)

AC Code



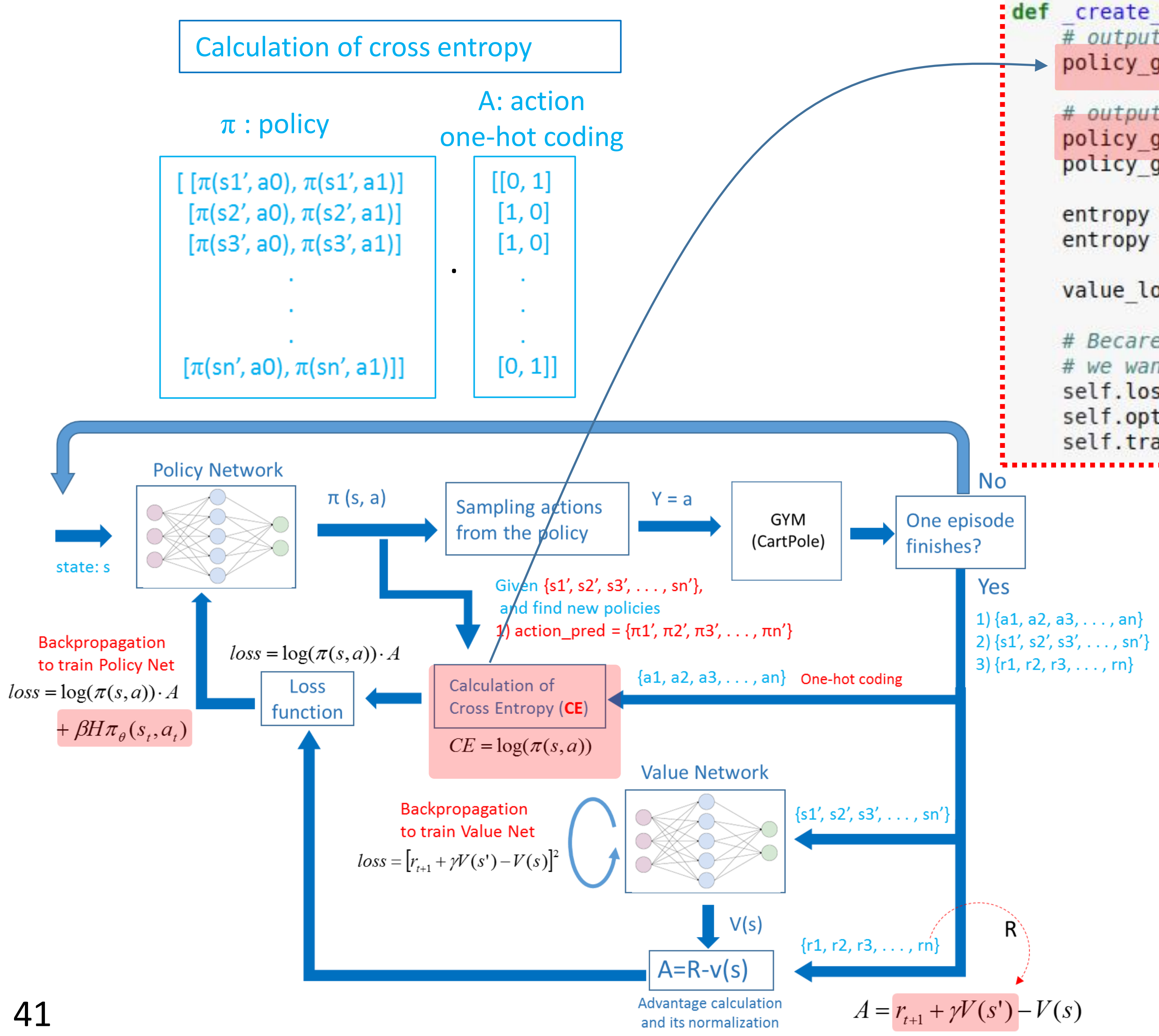
Advantage calculation

```python
def train(self, S, A, R):
    """ Train the actor critic networks

    (1) Compute discounted rewards R
    (2) Compute advantage values A = R - V
    (3) Perform gradients updates

    """

    def discount_rewards(r, gamma=0.99):
        """ take 1D float array of rewards and compute dis
        discounted_r = np.zeros_like(r, dtype=np.float32)
        running_add = 0

        for t in reversed(range(len(r))):
            running_add = running_add * gamma + r[t]
            discounted_r[t] = running_add

        return discounted_r

    # 1. Get discounted `R`s
    R = discount_rewards(R)

    # 2. Get `V`s
    feed = {
        self.model.S: S
    }
    V = self.sess.run(self.model.V, feed_dict=feed)

    # 3. Get Advantage values, A = R - V
    ADV = R - V
    ADV = (ADV - np.mean(ADV)) / (np.std(ADV) + 1e-8)

    # 4. Perform gradient descents
    feed = {
        self.model.S: S,
        self.model.A: A,
        self.model.ADV: ADV,
        self.model.R: R
    }

    self.sess.run(self.model.train_op, feed_dict=feed)
```

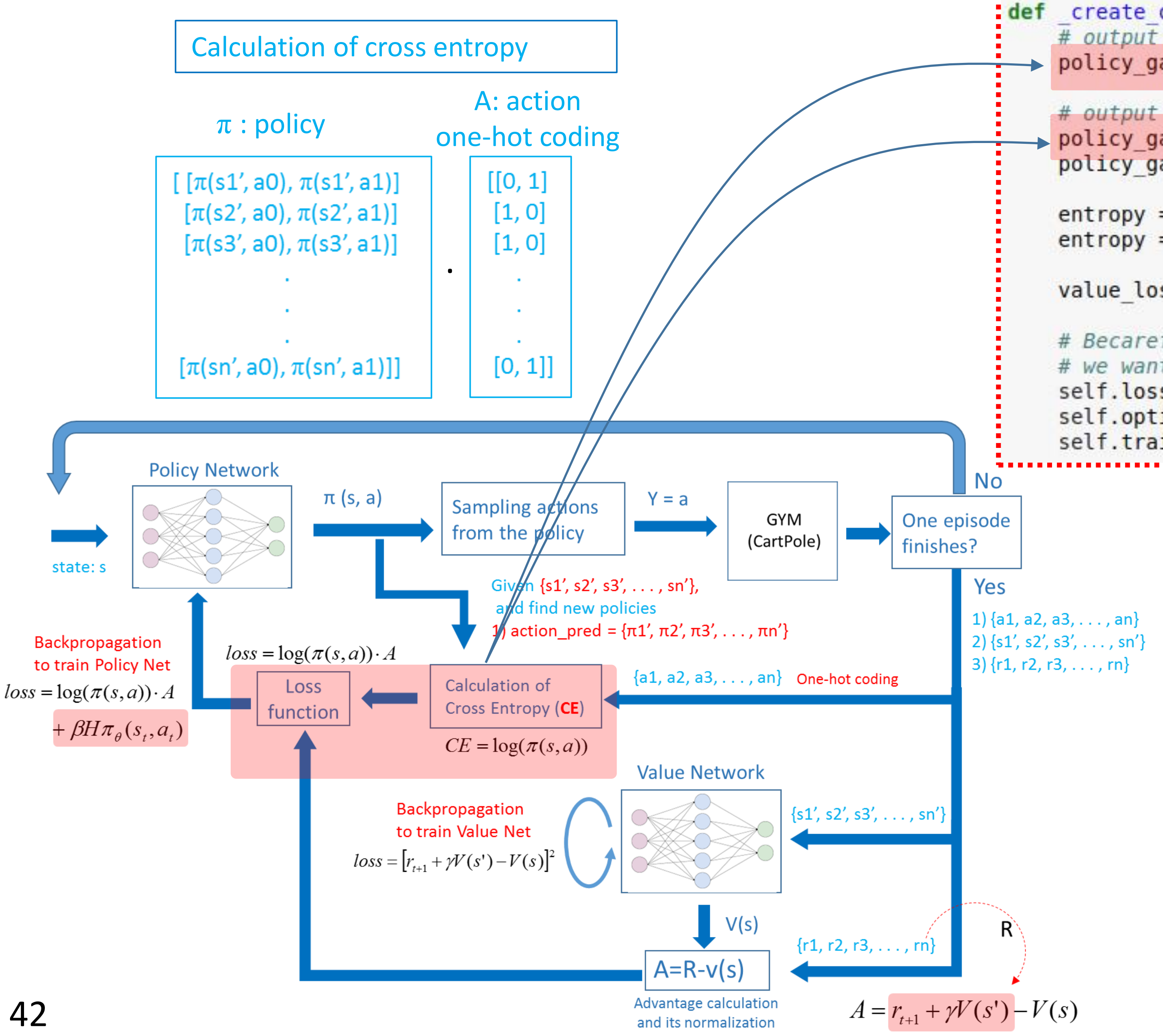
AC Code

Policy (probability) and Action(one-hot encoding)

```python
def _create_op(self):
    # output shape: [None]
    policy_gain = tf.reduce_sum(self.P * self.A, 1)

    # output shape: [None]
    policy_gain = tf.log(policy_gain) * self.ADV
    policy_gain = tf.reduce_sum(policy_gain, name="policy_gain")

    entropy = - tf.reduce_sum(self.P * tf.log(self.P), 1)
    entropy = tf.reduce_mean(entropy)

    value_loss = tf.losses.mean_squared_error(self.V, self.R, scope="value_loss")

    # Becareful negative sign because we only can minimize
    # we want to maximize policy gain and entropy (for exploration)
    self.loss = - policy_gain + value_loss - entropy * 0.01
    self.optimizer = tf.train.AdamOptimizer()
    self.train_op = self.optimizer.minimize(self.loss)
```



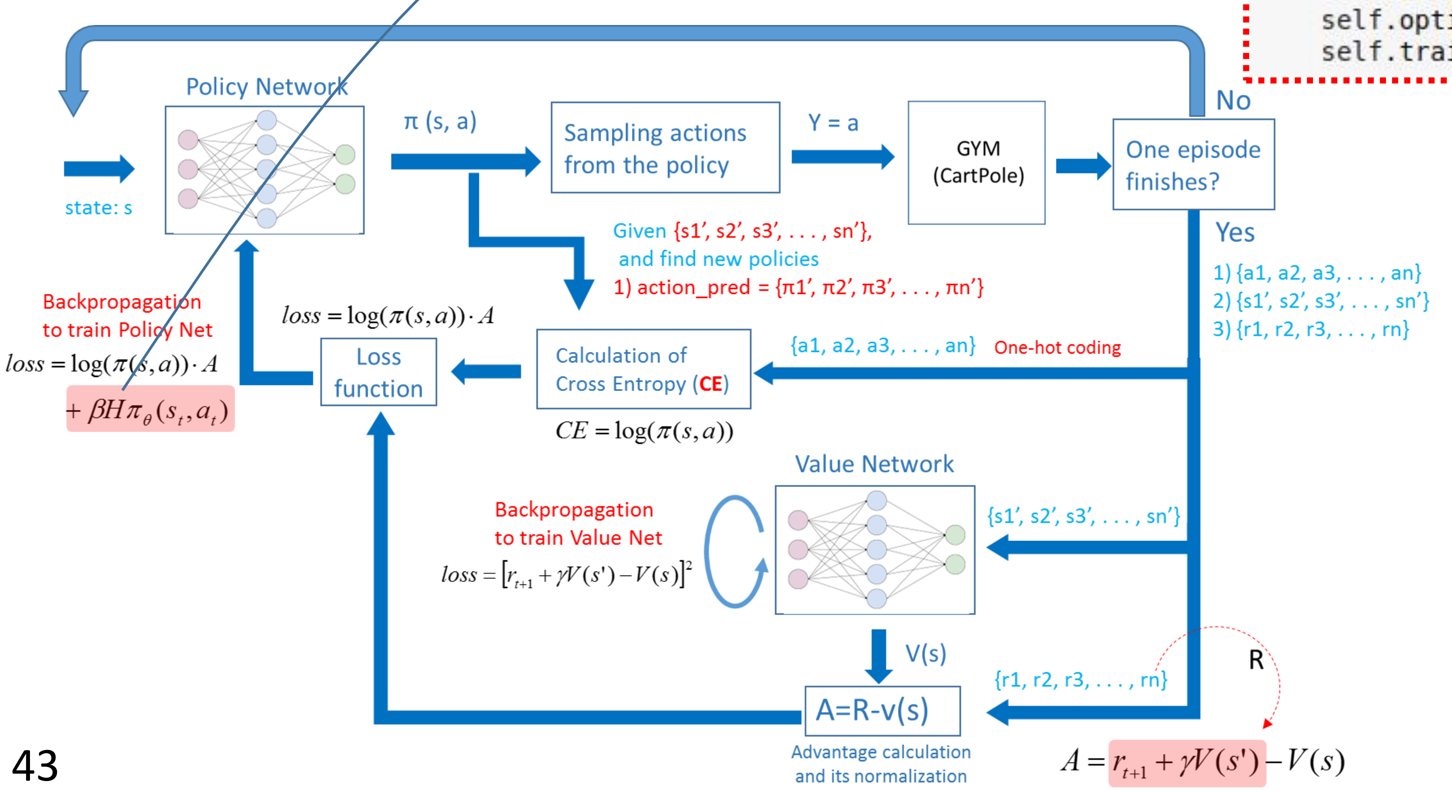
AC Code




```python
def _create_op(self):
    # output shape: [None]
    policy_gain = tf.reduce_sum(self.P * self.A, 1)

    # output shape: [None]
    policy_gain = tf.log(policy_gain) * self.ADV
    policy_gain = tf.reduce_sum(policy_gain, name="policy_gain")

    entropy = - tf.reduce_sum(self.P * tf.log(self.P), 1)
    entropy = tf.reduce_mean(entropy)

    value_loss = tf.losses.mean_squared_error(self.V, self.R, scope="value_loss")

    # Becareful negative sign because we only can minimize
    # we want to maximize policy gain and entropy (for exploration)
    self.loss = - policy_gain  + value_loss - entropy * 0.01
    self.optimizer = tf.train.AdamOptimizer()
    self.train_op = self.optimizer.minimize(self.loss)
```

AC Code

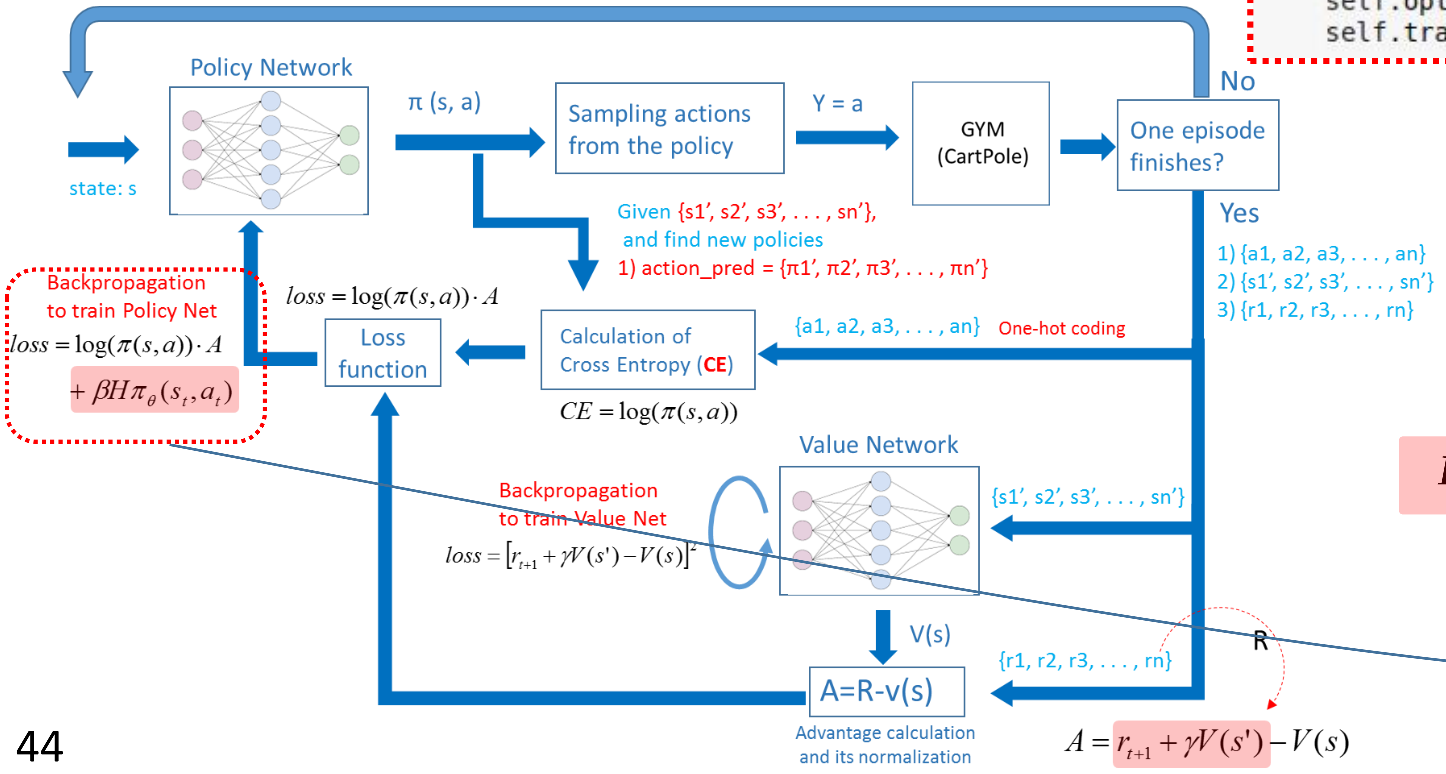
```python
def _create_op(self):
    # output shape: [None]
    policy_gain = tf.reduce_sum(self.P * self.A, 1)

    # output shape: [None]
    policy_gain = tf.log(policy_gain) * self.ADV
    policy_gain = tf.reduce_sum(policy_gain, name="policy_gain")

    entropy = - tf.reduce_sum(self.P * tf.log(self.P), 1)
    entropy = tf.reduce_mean(entropy)

    value_loss = tf.losses.mean_squared_error(self.V, self.R, scope="value_loss")

    # Becareful negative sign because we only can minimize
    # we want to maximize policy gain and entropy (for exploration)
    self.loss = - policy_gain  + value_loss - entropy * 0.01
    self.optimizer = tf.train.AdamOptimizer()
    self.train_op = self.optimizer.minimize(self.loss)
```



- Entropy regularization term
- This term tries to uniformize the probability distribution of actions defined in the first term.
  - Entropy is maximized when all actions from the policy π are same.
  - It aims to occur all action with equal probability (exploration)

AC Code: Loss function

```python
def _create_op(self):
    # output shape: [None]
    policy_gain = tf.reduce_sum(self.P * self.A, 1)

    # output shape: [None]
    policy_gain = tf.log(policy_gain) * self.ADV
    policy_gain = tf.reduce_sum(policy_gain, name="policy_gain")

    entropy = - tf.reduce_sum(self.P * tf.log(self.P), 1)
    entropy = tf.reduce_mean(entropy)

    value_loss = tf.losses.mean_squared_error(self.V, self.R, scope="value_loss")

    # Becareful negative sign because we only can minimize
    # we want to maximize policy gain and entropy (for exploration)
    self.loss = - policy_gain  + value_loss - entropy * 0.01
    self.optimizer = tf.train.AdamOptimizer()
    self.train_op = self.optimizer.minimize(self.loss)
```



Loss function to train policy Network

$$L(\theta) = \log(\pi_\theta(s_t, a_t)) \cdot (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) + \beta H\pi_\theta(s_t, a_t)$$

Backup Slides

DQN Code

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

- Training the main Q-Net and target $\hat{Q}$-Net given minibatch input

```python
# store the previous observations in replay memory
replay_buffer = deque(maxlen=REPLAY_MEMORY)

last_100_game_reward = deque(maxlen=100)

with tf.Session() as sess:
```

```python
def replay_train(mainDQN: dqn.DQN, targetDQN: d
    """Trains `mainDQN` with target Q values gi

    Args:
        mainDQN (dqn.DQN): Main DQN that will b
        targetDQN (dqn.DQN): Target DQN that wi
        train_batch (list): Minibatch of replay
            Each element is (s, a, r, s', done)
            [(state, action, reward, next_state

    Returns:
        float: After updating `mainDQN`, it ret
    """
    states = np.vstack([x[0] for x in train_bat
    actions = np.array([x[1] for x in train_bat
    rewards = np.array([x[2] for x in train_bat
    next_states = np.vstack([x[3] for x in trai
    done = np.array([x[4] for x in train_batch]

    X = states

    Q_target = rewards + DISCOUNT_RATE * np.max                                        done

    y = mainDQN.predict(states)
    y[np.arange(len(X)), actions] = Q_target

    # Train our network using target and predi                                      h episode
    return mainDQN.update(X, y)
```

```python
Q = [10, 20, 30]
y = np.array([[1,2], [4,7], [5,3]])

print ("Before: " , "\n" , y)

action = [1,0,1]
y[np.arange(3), action] = Q

print ("After: ", "\n",  y)
```

```
Before:
 [[1 2]
 [4 7]
 [5 3]]
After:
 [[ 1 10]
 [20  7]
 [ 5 30]]
```

- 3 batch samples
- Each sample: Q(R), Q(L)

- Insert target Q value into Y at location; a=[1,0,1]
- Q network needs to be trained to produce this value

Tricky part!

```python
            reward = -1

        # Save the experience to our buffer
        replay_buffer.append((state, action, reward, next_state, done))

        if len(replay_buffer) > BATCH_SIZE:
            minibatch = random.sample(replay_buffer, BATCH_SIZE)
            loss, _ = replay_train(mainDQN, targetDQN, minibatch)

        if step_count % TARGET_UPDATE_FREQUENCY == 0:
            sess.run(copy_ops)

        state = next_state
        step_count += 1

    print("Episode: {}  steps: {}".format(episode, step_count))
```