DQN with experience replay

DEEP REINFORCEMENT LEARNING IN PYTHON

Timothée Carayol

Principal Machine Learning Engineer, Komment

datacamp

Introduction to experience replay

- Barebone DQN agent learns only from latest experience
  - Consecutive updates are highly correlated
  - Agent is forgetful
- Solution: Experience Replay
  - Store experiences in a buffer
  - At each step, learn from a random batch of past experiences

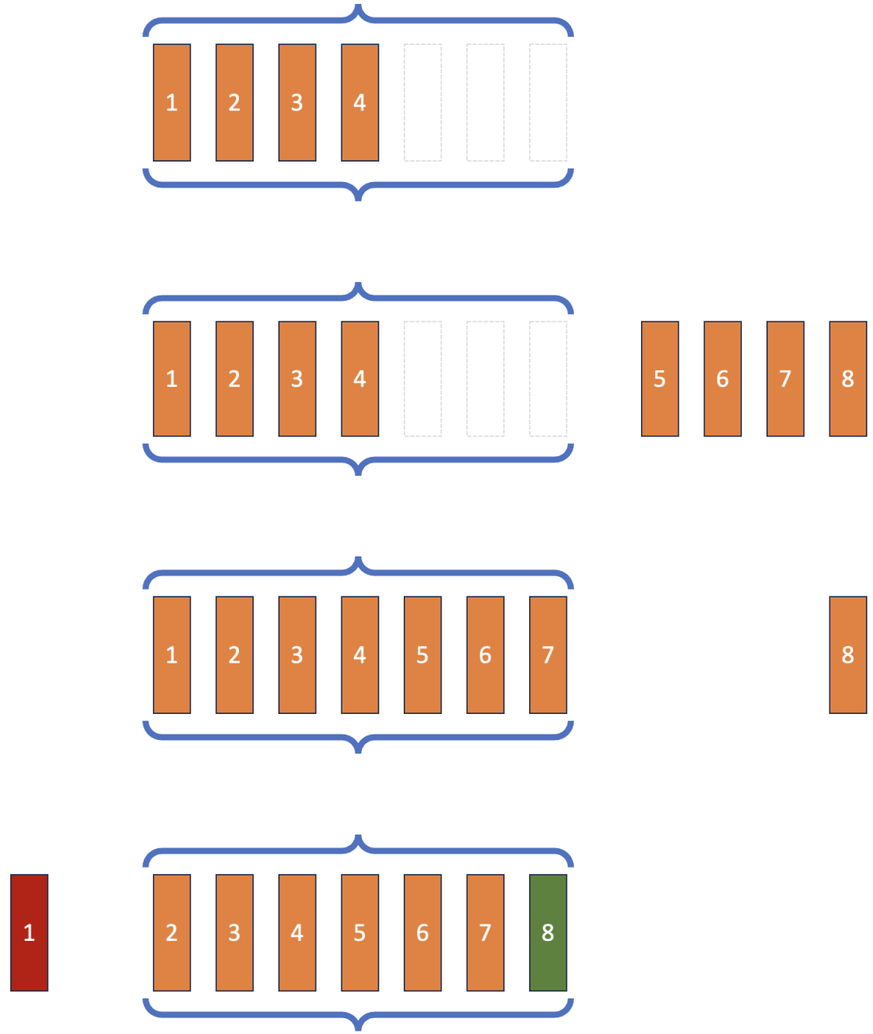
The Double-Ended Queue

```
from collections import deque

# Instantiate with limited capacity
buffer = deque([1,2,3,4], maxlen=7)

# Extend to the right side
buffer.extend([5,6,7,8])
```

- Beyond capacity, oldest items get dropped



Implementing Replay Buffer

```
import random

class ReplayBuffer:
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, state, action,
             reward, next_state, done):
        experience_tuple = (state, action,
                            reward, next_state, done)
        self.memory.append(experience_tuple)

    def __len__(self):
        return len(self.memory)

...
```

- Replay memory: `deque` with limited capacity
- `.push()`:
  - Experience as transition tuple
  - Append experience to buffer
  - At capacity: drops oldest experience

Implementing Replay Buffer

```
...
def sample(self, batch_size):
    batch = random.sample(self.memory, batch_size)
    states, actions, rewards, next_states, dones = (
      zip(*batch))

    states_tensor = torch.tensor(
      states, dtype=torch.float32)
    ... # repeat identically for
        # rewards, next_states, dones
    actions_tensor = torch.tensor(
      actions, dtype=torch.long).unsqueeze(1)

    return states_tensor, actions_tensor,
rewards_tensor, next_states_tensor, dones_tensor
```

- Randomly draw from past experiences
- `batch` : from list of transition tuples...
- ...to tuple of lists...
- ...to tuple of PyTorch tensors

Integrating Experience Replay in DQN

1. Before training loop:

   `replay_buffer = ReplayBuffer(10000)`

2. In training loop, after action selection:

```
replay_buffer.push((state, action,
                    reward, next_state, done))
if len(replay_buffer) >= batch_size:
    states, actions, rewards, next_states, dones = (
           replay_buffer.sample(batch_size))
    q_values = (
        q_network(states).gather(1, actions).squeeze(1))
    next_states_q_values = q_network(next_states).amax(1)
    target_q_values = (
                rewards +
                gamma * next_states_q_values * (1-dones))
    loss = nn.MSELoss()(target_q_values, q_values)
```

- Initialize replay buffer
- Push latest transition to buffer

If buffer length $\geq$ `batch_size` :

- Draw a random batch from the buffer and proceed with loss calculation
- Loss calculation conceptually unchanged
- Mean Squared Bellman Error on a replay memory batch
  - Learning is more stable and more efficient

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

The complete DQN algorithm

DEEP REINFORCEMENT LEARNING IN PYTHON

Timothée Carayol

Principal Machine Learning Engineer, Komment

datacamp

The DQN algorithm

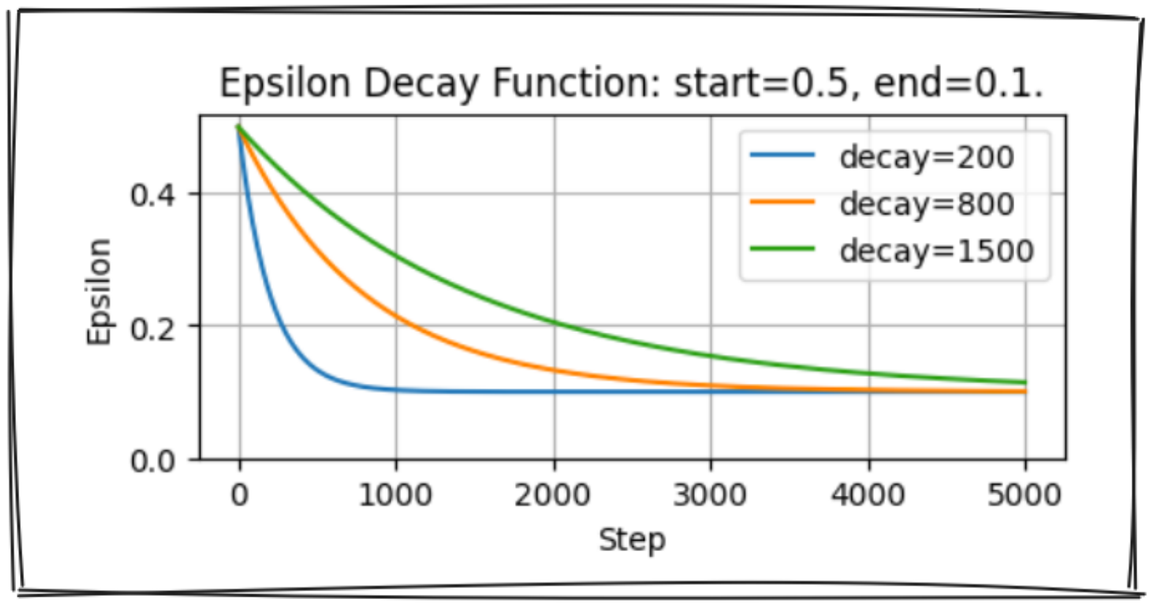
- We studied DQN with Experience Replay
- Close to DQN as first published (2015)
- We still miss two components:
  - Epsilon-greediness -> more exploration
  - Fixed Q-targets -> more stable learning

Epsilon-greediness in the DQN algorithm

- Implement Decayed Epsilon-greediness in `select_action()`

```
def select_action(q_values, step, start, end, decay):
    # Calculate the threshold value for this step
    epsilon = (
        end + (start-end) * math.exp(-step / decay))
    # Draw a random number between 0 and 1
    sample = random.random()
    if sample < epsilon:
        # Return a random action index
        return random.choice(range(len(q_values)))
    # Return the action index with highest Q-value
    return torch.argmax(q_values).item()
```

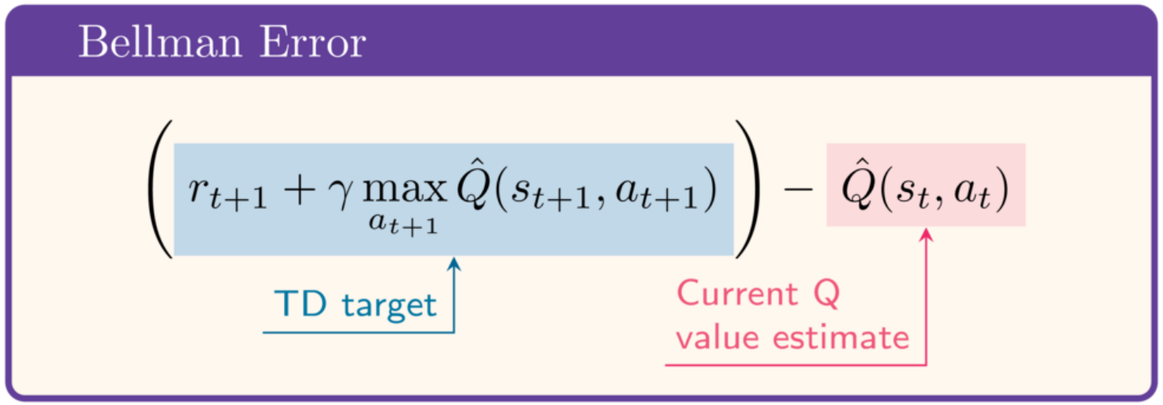
- $\varepsilon = end + (start - end) \cdot e^{-\frac{step}{decay}}$
- Take random action with probability ε
- Take highest value action with probability $1 - \varepsilon$

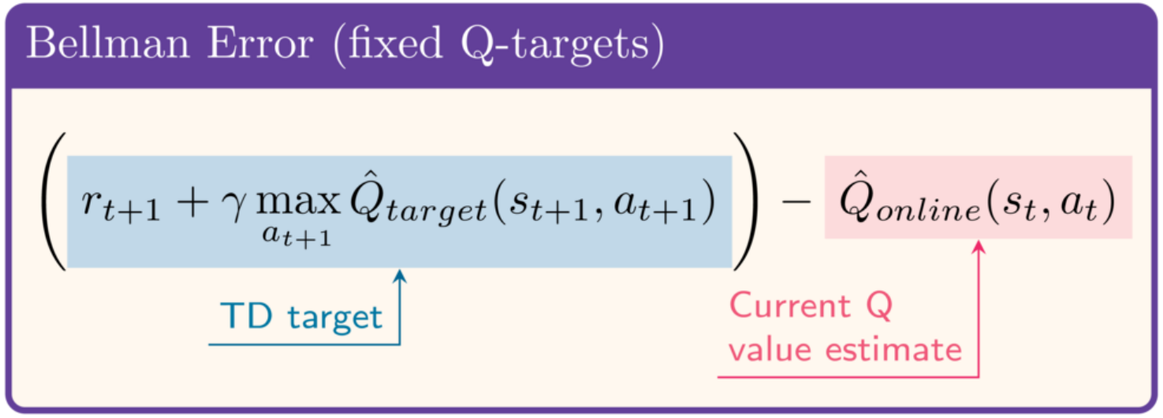


Fixed Q-targets

- In Bellman Error:
  - Q-Network in both Q-Value and TD-Target calculation
  - Instability from shifting target

- Introduce target network to stabilize target





Implementing fixed Q-targets

```
online_network = QNetwork(state_size, action_size)
target_network = QNetwork(state_size, action_size)
target_network.load_state_dict(
                        online_network.state_dict())

def update_target_network(
                    target_network, online_network, tau):
  target_net_state_dict = target_network.state_dict()
  online_net_state_dict = online_network.state_dict()
  for key in online_net_state_dict:
    target_net_state_dict[key] = (
         online_net_state_dict[key] * tau +
         target_net_state_dict[key] * (1 - tau))
    target_network.load_state_dict(
                          target_net_state_dict)
  return None
```

- Initially Online Network = Target Network
- A network's state dict contains all weights:

```
fc1.weight :
       tensor(
              [[-0.5059, -0.6368],
              [-0.4193,  0.6154]]
              )
fc1.bias :
       tensor([ 0.2465, -0.5916])
fc2.weight :
       tensor(
              [[-0.1208, -0.2308],
              [-0.2805,  0.5415]]
              )
```

- Each step, every weight of Target Network gets a bit closer to Online Network

Loss calculation with fixed Q-targets

```
# In the inner loop, after action selection
if len(replay_buffer) >= batch_size:
  states, actions, rewards, next_states, dones =
      replay_buffer.sample(64)
  q_values = (online_network(states)
              .gather(1, actions).squeeze(1))
  with torch.no_grad():
    next_q_values = (
        target_network(next_states).amax(1))
    target_q_values = (
        rewards + gamma * next_q_values * (1 - dones))
  loss = torch.nn.MSELoss()(target_q_values, q_values)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
  update_target_network(
                target_network, online_network, tau)
```

- Q-values use `online_network`
- Target Q-values use `target_network`
- Use `torch.no_grad()` to disable gradient tracking for target Q-values
- Still use Mean Squared Bellman Error for loss calculation
- Use `update_target_network()` to slowly update `target_network`

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

Double DQN

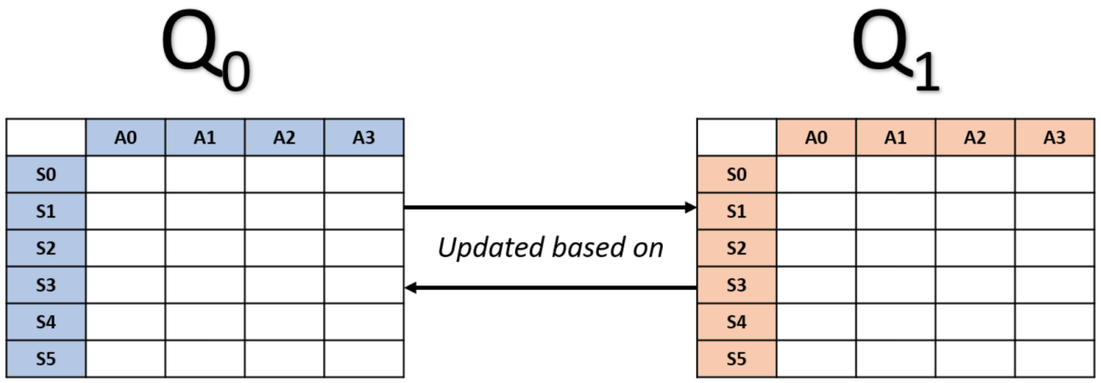
DEEP REINFORCEMENT LEARNING IN PYTHON

Timothée Carayol

Principal Machine Learning Engineer, Komment

Double Q-learning

- Q-learning overestimates Q-values, compromising learning efficiency
- This is due to maximization bias
- Double Q-Learning eliminates bias by decoupling action selection and value estimation



$$max_a = argmax(Q_1(s'))$$

$$Q_1(s,a) = (1-\alpha)Q_1(s,a) + \alpha[r + \gamma Q_0(s', max_a)]$$

New Q value

Old Q value

Maximum Q value given the next state

The idea behind DDQN

- Start from complete DQN (with fixed Q-targets)
- In DQN TD target:
  - Action selection: target network
  - Value estimation: target network
- In DDQN TD target:
  - Action selection: *online* network
  - Value estimation: target network
- Not exactly double Q-learning (no alternating Q-networks)
- Most of the benefit, with minimal change

TD Error (DQN with fixed Q-targets)

$$\left(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}_{target}(s_{t+1}, a_{t+1}) \right) - \hat{Q}_{online}(s_t, a_t)$$

TD target

Current Q value estimate

TD Error (DDQN)

$$\left(r_{t+1} + \gamma \max_{\tilde{a}} \hat{Q}_{target}(s_{t+1}, \tilde{a}) \right) - \hat{Q}_{online}(s_t, a_t)$$

TD target

Current Q value estimate

with $\tilde{a} = \arg\max_{a_{t+1}} \hat{Q}_{online}(s_{t+1}, a_{t+1})$

Double DQN implementation

DQN:

```
... # instantiate online and target networks
q_values = (online_network(states)
            .gather(1, actions).squeeze(1))
with torch.no_grad():
    #
    #
    next_q_values = (target_network(next_states)
        .amax(1))
    target_q_values = (rewards +
        gamma * next_q_values * (1 - dones))
loss = torch.nn.MSELoss()(q_values, target_q_values)
... # gradient descent
... # target network update
```

DDQN:

```
... # instantiate online and target networks
q_values = (online_network(states)
            .gather(1, actions).squeeze(1))
with torch.no_grad():




    target_q_values = (rewards +
        gamma * next_q_values * (1 - dones))
loss = torch.nn.MSELoss()(q_values, target_q_values)
... # gradient descent
... # target network update
```

Double DQN implementation

DQN:

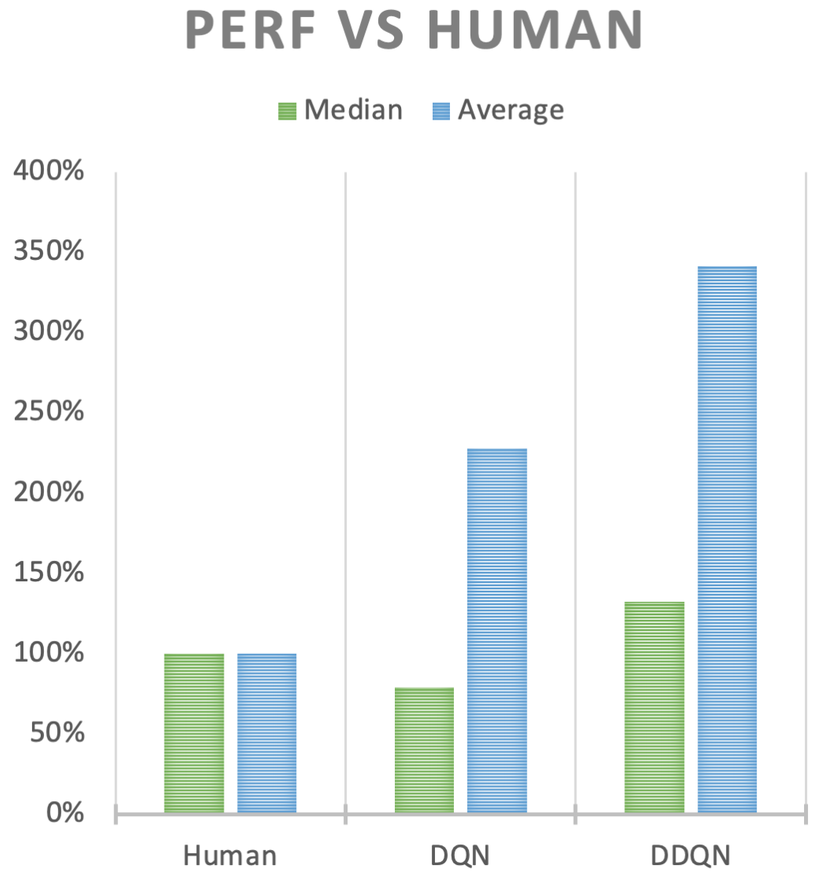
```
... # instantiate online and target networks
q_values = (online_network(states)
            .gather(1, actions).squeeze(1))
with torch.no_grad():
    next_actions = (target_network(next_states)
       .argmax(1).unsqueeze(1))
    next_q_values = (target_network(next_states)
       .gather(1, next_actions).squeeze(1))
    target_q_values = (rewards +
        gamma * next_q_values * (1 - dones))
loss = torch.nn.MSELoss()(q_values, target_q_values)
... # gradient descent
... # target network update
```

DDQN:

```
... # instantiate online and target networks
q_values = (online_network(states)
            .gather(1, actions).squeeze(1))
with torch.no_grad():
    next_actions = (online_network(next_states)
       .argmax(1).unsqueeze(1))
    next_q_values = (target_network(next_states)
       .gather(1, next_actions).squeeze(1))
    target_q_values = (rewards +
        gamma * next_q_values * (1 - dones))
loss = torch.nn.MSELoss()(q_values, target_q_values)
... # gradient descent
... # target network update
```

DDQN performance

- Compare performance of DDQN, DQN and human players on Atari games
- DDQN: higher scores than original DQN
- May not always be true -> try both



[1] https://arxiv.org/abs/2303.11634

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

Prioritized experience replay

DEEP REINFORCEMENT LEARNING IN PYTHON

Timothée Carayol

Principal Machine Learning Engineer, Komment

datacamp

Not all experiences are created equal

- Experience Replay:
  - Uniform sampling of experiences may overlook important memories
- Prioritized Experience Replay:
  - Assign priority to each experience, based on TD errors
  - Focus on experiences with high learning potential

Prioritized Experience Replay (PER)

```
for step = 1 to T do:
    # Take optimal action according to value function
    # Observe next state and reward
    # Append transition to replay buffer
      # Give it highest priority (1)
    # Sample a batch of past transitions
      # Based on priority (2)
    # Calculate TD errors for the batch
    # Calculate the loss and update the Q Network
      # Use importance sampling weights (4)
    # Update priority of sampled transitions (3)
# Increase importance sampling over time. (5)
```

(1) New transitions are appended with highest priority $p_i = \max_k(p_k)$

(2) Sample transition i with probability

$$P(i) = p_i^\alpha / \sum_k p_k^\alpha \qquad (0 < \alpha < 1)$$

(3) Sampled transitions have their priority updated to their TD error: $p_i = |\delta_i| + \varepsilon$

(4) Use importance sampling weights

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \qquad (0 < \beta < 1)$$

(5) Progressively increase β towards 1

Implementing PER

```
def __init__(self, capacity, alpha=0.6, beta=0.4, beta_increment=0.001, epsilon=0.00
    # Initialize memory buffer
    self.memory = deque(maxlen=capacity)
    # Store parameters and initialize priorities
    self.alpha, self.beta, self.beta_increment, self.epsilon = (alpha, beta, beta_in
    self.priorities = deque(maxlen=capacity)


...
```

Implementing PER

```
...

def push(self, state, action, reward, next_state, done):
    # Append experience to memory buffer
    experience_tuple = (state, action, reward, next_state, done)
    self.memory.append(experience_tuple)
    # Set priority of new transition to maximum priority
    max_priority = max(self.priorities) if self.memory else 1.0
    self.priorities.append(max_priority)

...
```

Implementing PER

```
def sample(self, batch_size):
    priorities = np.array(self.priorities)
    # Calculate sampling probabilities
    probabilities = priorities**self.alpha / np.sum(priorities**self.alpha)
    # Randomly select sampled indices
    indices = np.random.choice(len(self.memory), batch_size, p=probabilities)
    # Calculate weights
    weights = (1 / (len(self.memory) * probabilities)) ** self.beta
    weights /= np.max(weights)
    states, actions, rewards, next_states, dones = zip(*[self.memory[idx] for idx in indices])
    weights = [weights[idx] for idx in indices]
    states, actions, rewards, next_states, dones = (zip(*[self.memory[idx] for idx in indices]))
    # Return tensors
    states = torch.tensor(states, dtype=torch.float32)
    ... # Repeat for rewards, next_states, dones, weights
    actions = torch.tensor(actions, dtype=torch.long).unsqueeze(1)
    return (states, actions, rewards, next_states, dones, indices, weights)
```

Implementing PER

```
...

def update_priorities(self, indices, td_errors: torch.Tensor):
    # Update priorities for sampled transitions
    for idx, td_error in zip(indices, td_errors.abs()):
        self.priorities[idx] = abs(td_error.item()) + self.epsilon

def increase_beta(self):
    # Increment beta towards 1
    self.beta = min(1.0, self.beta + self.beta_increment)
```

PER in the DQN training loop

1. In pre-loop code:

```
buffer = PrioritizedReplayBuffer(capacity)
```

2. At the start of each episode:

```
buffer.increase_beta()
```

3. At every step:

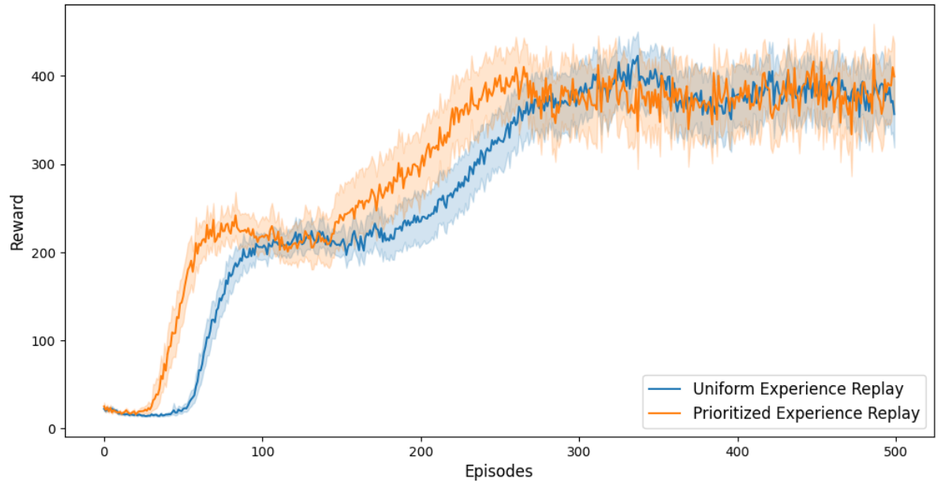
```
# After selecting an action
buffer.push(state, action, reward,
            next_state, done)
...
# Before calculating the TD errors:
replay_buffer.sample(batch_size)
...
# After calculating the TD errors
buffer.update_priorities(indices, td_errors)
loss = torch.sum(weights * (td_errors ** 2))
```

PER In Action: Cartpole

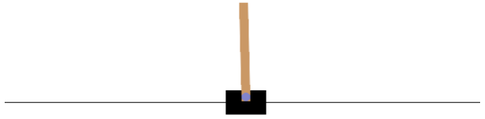
100 training runs in the Cartpole environment:

1. with Prioritized Experience Replay
2. with Uniform Experience Replay

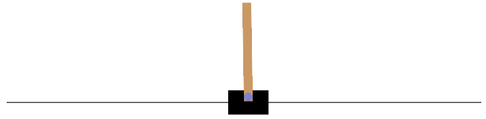
- **Faster** learning and **better performance** with PER than uniform experience replay



After 100 epochs:

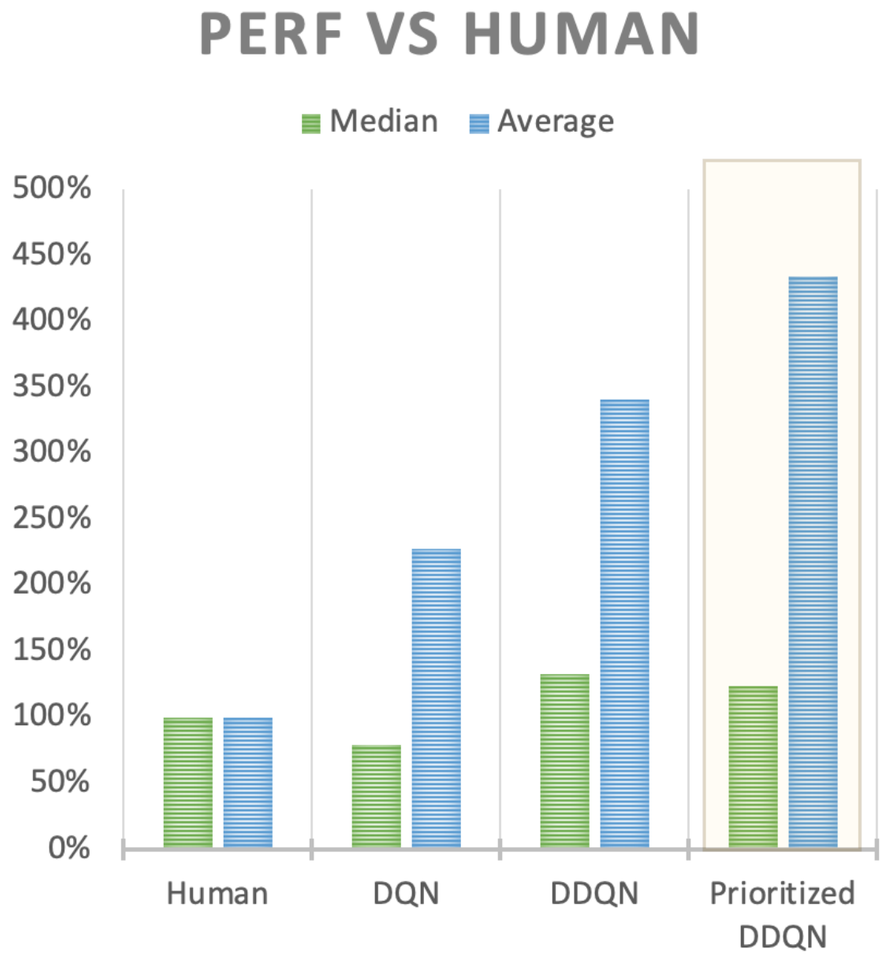


After 400 epochs:



PER In Action: Atari environments

- Substantial performance boost with PER in Atari environments



[1] https://arxiv.org/abs/2303.11634

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON