DQN(Deep Q-Network)

Paper Link: Human-level control through deep reinforcement learning

Key Features

- Q-learning with deep neural network for approximaing Q-value.
- DQN is an off-policy algorithm.
- DQN uses experience replay and target network for stable training.
- Target network is updated periodically (Every C steps reset $\hat{Q} = Q$).

Background

The basic idea behind Q-learning algorithms is to estimate the action-value function by using the Bellman equation as an iterative update,

$$Q_{i+1}(s,a) = E\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

Such value iteration algorithms converge to the optimal action-value function, $Q_i \rightarrow Q*$ as $i \rightarrow \infty$. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s,a;\theta) \approx Q*(s,a)$. In the past, it was generally a linear function approximator, but nowadays most non-linear function approximators such as neural networks are used. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = E_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right]$$

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator is used to represent the action-value(Q) function.

This instability has several causes:

- Deep learning algorithms assume that data samples are independent, while reinforcement learning usually results in highly correlated sequences of states.
- small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the Q and the target values.

Methods

Experience replay

To perform experience replay store the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t in a data set $D_t = \{e_1, \ldots, e_t\}$. During learning, we apply Q-learning updates, on samples (or mini -batches) of experience $(s, a, r, s') \sim U(D)$, drawn uniformly at random from the pool of stored samples.

This approach has several advantages over standard online Q-learning:

- each step of experience is potentially used in many weight updates, which allows for greater data efficiency.
- randomizing the samples breaks correlations between samples and therefore reduces the variance of the updates.

Target network

The second method is to use a separate network for generating the targets y_i in the Q-learning update. More precisely, every C updates we clone the network Q to obtain a target network $\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning targets y for the following C updates to Q.

- This modification makes the algorithm more stable where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ for all a and hence also increases the target y_j , possibly leading to oscillations or divergence of the policy.

- Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_j, making divergence or oscillations much more unlikely.

Algorithm

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

Implementation on JORLDY

- DQN JORLDY Implementation

```
# initialize action-value function Q
# initialize target action-value function Q'
# initialize replay memory D

def __init__(self, **kwargs,):
```

```
  ...

  # action-value function Q
  self.network = Network(
      network, state_size, action_size, D_hidden=hidden_size, head=head
  ).to(self.device)

  # target action-value function Q'
  self.target_network = Network(
      network, state_size, action_size, D_hidden=hidden_size, head=head
  ).to(self.device)

  ...

  # replay memory D
  self.buffer_size = buffer_size
  self.memory = ReplayBuffer(buffer_size)

  ...
```

```
# With probability ε select a random action a
# otherwise select a = argmaxQ

def act(self, state, training=True):
  self.network.train(training)
  epsilon = self.epsilon if training else self.epsilon_eval

  if np.random.random() < epsilon:
      batch_size = (
          state[0].shape[0] if isinstance(state, list) else state.shape[0]
      )
      action = np.random.randint(0, self.action_size, size=(batch_size, 1))
  else:
      action = (
          torch.argmax(self.network(self.as_tensor(state)), -1, keepdim=True)
          .cpu()
          .numpy()
      )
  return {"action": action}
```

```
# Store transition in D

def process(self, transitions, step):
  result = {}

  # Process per step
  self.memory.store(transitions)

  ...
```

```
# Sample random minibatch of transitions from D
# Set y = r if episode terminates at step else r+γmaxQ
# Perform a gradient descent step on smooth_L1_loss(y, Q(s))

def learn(self):
  transitions = self.memory.sample(self.batch_size)
  for key in transitions.keys():
      transitions[key] = self.as_tensor(transitions[key])

  state = transitions["state"]
  action = transitions["action"]
  reward = transitions["reward"]
  next_state = transitions["next_state"]
  done = transitions["done"]

  eye = torch.eye(self.action_size, device=self.device)
  one_hot_action = eye[action.view(-1).long()]
  q = (self.network(state) * one_hot_action).sum(1, keepdims=True)
  with torch.no_grad():
      max_Q = torch.max(q).item()
      next_q = self.target_network(next_state)
      target_q = (
          reward + (1 - done) * self.gamma * next_q.max(1, keepdims=True).values
      )

  loss = F.smooth_l1_loss(q, target_q)
  self.optimizer.zero_grad(set_to_none=True)
  loss.backward()
  self.optimizer.step()

  self.num_learn += 1

  result = {
      "loss": loss.item(),
      "epsilon": self.epsilon,
      "max_Q": max_Q,
  }

  return result
```

```
# Every C steps Q' = Q

def update_target(self):
  self.target_network.load_state_dict(self.network.state_dict())
```