Multi Agent Systems

- Lab 8 -

Deep Q-Network

Q-Learning with Linear Value Function Approximation – General Formulation

- Use *features* to represent state and action $x(s,a)=\begin{pmatrix} x_1(s,a) \\ x_2(s,a) \\ \dots \\ x_n(s,a) \end{pmatrix}$

- Q-function represented as ***weighted linear combination of features***

$$\hat{Q}(s,a,\boldsymbol{w})=\boldsymbol{x}(s,a)^T\boldsymbol{w}=\sum_{j=1}^{n} x_j(s,a)w_j$$

- ***Learn*** weights **w** through stochastic gradient descent updates $\nabla_w J(w)=\nabla_w E_\pi[(Q^\pi(s,a)-\hat{Q}^\pi(s,a,\boldsymbol{w}))^2]$

Q-Learning with Linear Value Function Approximation –

- VFA relies on **features** that have to convey information for learning.
  - Features are **handcrafted** based on human knowledge and intuition about a specific problem

- **Function approximation** + **off-policy control** + **bootstrapping** (the “deadly triad”) can fail to converge

- **=>** attempt to improve on one aspect, function approximation, by leveraging Deep Neural Networks as universal function approximators

DQN – TD Target as Loss objective

- For Q-Function, instead of the actual gain per episode under current policy $Q^{\pi}(s,a)$ use **TD-target** $r+\gamma max_{a'}\hat{Q}(s',a',\mathbf{w})$

- ***Learn NN*** weights **w** through stochastic gradient descent updates

$$\nabla_w J(w)=\nabla_w(r+\gamma max_{a'}\hat{Q}(s',a';\mathbf{w})-\hat{Q}(s,a;\mathbf{w}))^2$$

In practice – Huber Loss

$$\Delta w=\alpha(r+\gamma max_{a'}\hat{Q}(s',a';\mathbf{w})-\hat{Q}(s,a;\mathbf{w}))\nabla_w\hat{Q}(s,a;\mathbf{w})$$

Experience Replay

- TD Learning Target requires behavioral samples *(s, a, r, s')*
- Problems in learning Q values through NN trained with SGD arise when inputs (s) are highly correlated (e.g. consecutive frames in a game)
- **=>** alleviate issue through **experience replay:** store a series of behavioral samples (in a R**eplay Buffer)**, **shuffle them** and **pick a sample** at each training step
- **Replay Buffer** management subject of ongoing research

DQN Procedure

procedure DQN (<S, A, γ>, ε, nn_*model, mem_size, batch_size***)**
Init **replay_buffer,** Init ***nn_model,*** Init ***optimizer***

for all episodes **do**
s ← initial state
while s not final state **do**
pick action a using ε-Greedy (s, nn_model, ε)
execute a → get reward r and next state *s'*
replay_buffer.push((s,a,r,s'))
learn(nn_model, replay_buffer, batch_size, optimizer)
s ← s'
end while
end for

procedure learn *(nn, memory, batch_size, optimizer)*

batch_s, batch_a, batch_r, batch_s' = memory.sample(batch_size)

current_q = nn(batch_s).gather(batch_a)
next_state_q = $\mathit{argmax}_{a'}$ *nn(batch_s')*
td_target = r + γ next_state_q
loss = huber_loss (current_q, td_target)

optimizer.zero_grad()
loss.backward()
optimizer.step()

procedure ε-Greedy (*s, NN_model,* ε)
[$\hat{q}(s,a_1)$, …, $\hat{q}(s,a_n)$] = *NN_model(s)*
with prob ε: return *random(A)*
with prob 1-ε: return argmax_a [$\hat{q}(s,a_1)$, …, $\hat{q}(s,a_n)$]
end

DQN – Fixed targets

- Standard DQN can still suffer from the “deadly triad” leading to convergence issues

- => improvement: maintain **2 Q-Learning models $\mathbf{Q_{model}}$ and $\mathbf{Q_{target}}$ –** whereby Q_{target} is “delayed” in updating its parameters => ***fixed target*** *(for a while – e.g. 50, 100 steps)*
  - Weights of $\mathbf{Q_{target}}$ are updated periodically to those of $\mathbf{Q_{model}}$

DQN – Fixed Target Procedure

procedure DQN (<S, A, γ>, ε, *nn_model, nn_target, mem_size, batch_size, delay_steps***)**

Init **replay_buffer,** Init ***nn_model,*** Init ***optimizer***
steps = 0

for all episodes **do**
s ← initial state
while s not final state **do**
pick action a using ε-Greedy (s, **nn_model**, ε)
execute a → get reward r and next state *s'*
replay_buffer.push((s,a,r,s'))
learn(nn_model, nn_target, replay_buffer,
batch_size, optimizer)
s ← s'
steps += 1

if steps % delay_steps == 0
then nn_target ← nn_model
end while
end for

procedure learn *(model, target, memory, batch_size, optimizer)*

batch_s, batch_a, batch_r, batch_s' =

memory.sample(batch_size)

current_q = ***model****(batch_s).gather(batch_a)*
next_state_q = $argmax_{a'}$ ***target****(batch_s')*
td_target = r + γ next_state_q
loss = huber_loss (current_q, td_target)

optimizer.zero_grad()
loss.backward()
optimizer.step()

procedure ε-Greedy (*s, NN_model*, ε)
[$\hat{q}(s,a_1)$, …, $\hat{q}(s,a_n)$] = *NN_model(s)*
with prob ε: return *random(A)*
with prob 1-ε: return $argmax_a$ [$\hat{q}(s,a_1)$, …, $\hat{q}(s,a_n)$]
end

Double-DQN

- Standard DQN can still suffer from the “deadly triad” leading to convergence issues

- => improvement: interplay between two modes in **Q** and **$\mathbf{Q}_{\mathbf{target}}$** in setting the TD target
  - reduce **overestimations** by **decomposing** the max operation in the target into ***action selection*** and ***action evaluation***
  - ***evaluate*** *the greedy policy according to the* ***online network****, but using the* ***target network to estimate its value.***

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

DDQN Procedure

```
procedure DQN (<S, A, γ>, ε, nn_model, nn_target,
mem_size, batch_size, delay_steps)
  Init replay_buffer, Init nn_model, Init optimizer
  steps = 0

  for all episodes do
     s ← initial state
    while s not final state do
       pick action a using ε-Greedy (s, nn_model, ε)
      execute a → get reward r and next state s'
      replay_buffer.push( (s,a,r,s') )
      learn(nn_model, nn_target, replay_buffer,
            batch_size, optimizer)
      s ← s'
      steps += 1

      if steps % delay_steps == 0
       then nn_target ← nn_model
    end while
  end for
```

```
procedure learn (model, target, memory,
batch_size, optimizer)

  batch_s, batch_a, batch_r, batch_s' =

     memory.sample(batch_size)

   current_q = model(batch_s).gather(batch_a)
   next_state_q =
target(batch_s').gather(argmax_a'
                                   model(batch_s'))
   td_target =  r + γ next_state_q
   loss = huber_loss (current_q, td_target)

   optimizer.zero_grad()
   loss.backward()
   optimizer.step()
```

procedure ε-Greedy (*s, NN_model,* ε)

$[\hat{q}(s,a_1), \ldots, \hat{q}(s,a_n)]$ = *NN_model(s)*

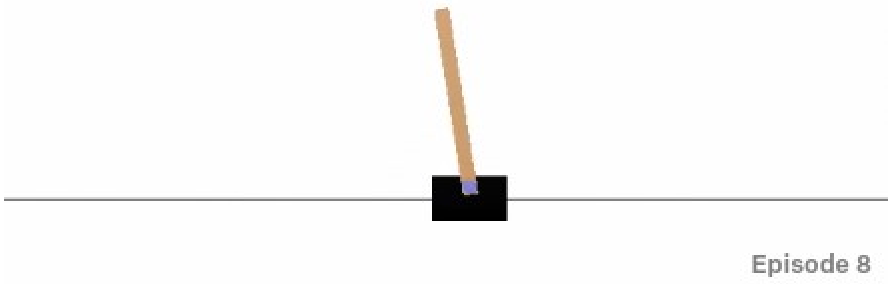
with prob ε: return *random(A)*

with prob 1-ε: return $argmax_a\,[\hat{q}(s,a_1), \ldots, \hat{q}(s,a_n)]$

end

OpenAI Gym CartPole Environment

- Cartpole-v1 environment in OpenAI Gymnasium:
  - Objective: keep a pendulum upright for as long as possible
  - 2 actions: left (force = -1), right (force = +1)
  - Reward: +1 for every timestep that the pole remains upright
  - Game ends when pole more the 15° from vertical OR cart moves > 2.4 units from center



OpenAI Gym CartPole Environment

- Cartpole-v1 environment in OpenAI Gym:
  - **Explore with:**
    - The SGD **learning rate**
    - The **target_update_freq** parameter (10, 100, 200)
    - The **learning_freq** parameter (1, 4, 8)
  - **Use** an ε=decay(init=0.9, end=0.05, nr_iterations) – see `eps_generator` in code – **explore different decay rates**

- **Plot** agent learning curves for each case

- Compare agent learning curves for **DQN** against **Double DQN**