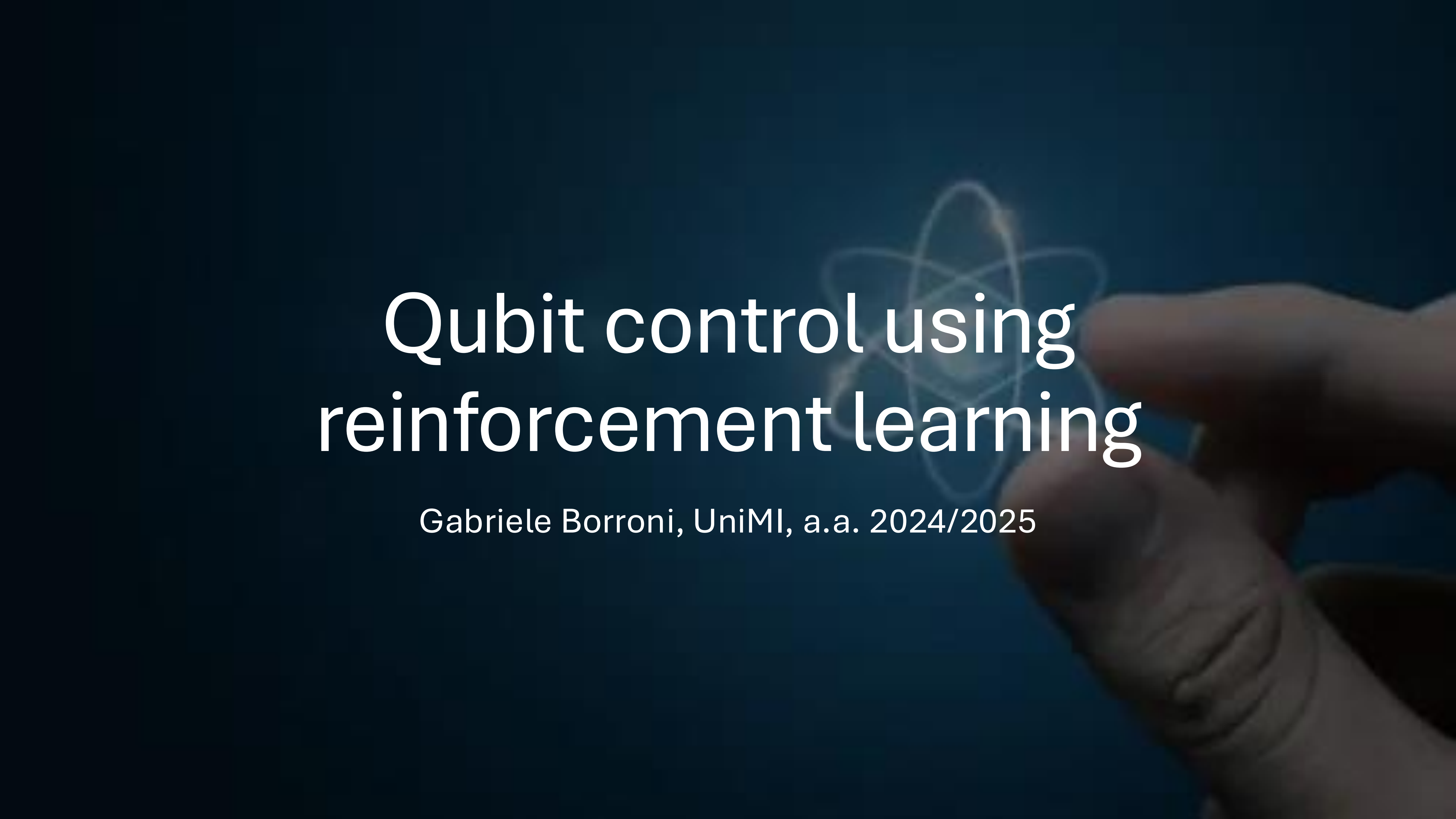


Qubit control using reinforcement learning

Gabriele Borroni, UniMI, a.a. 2024/2025

Qubits

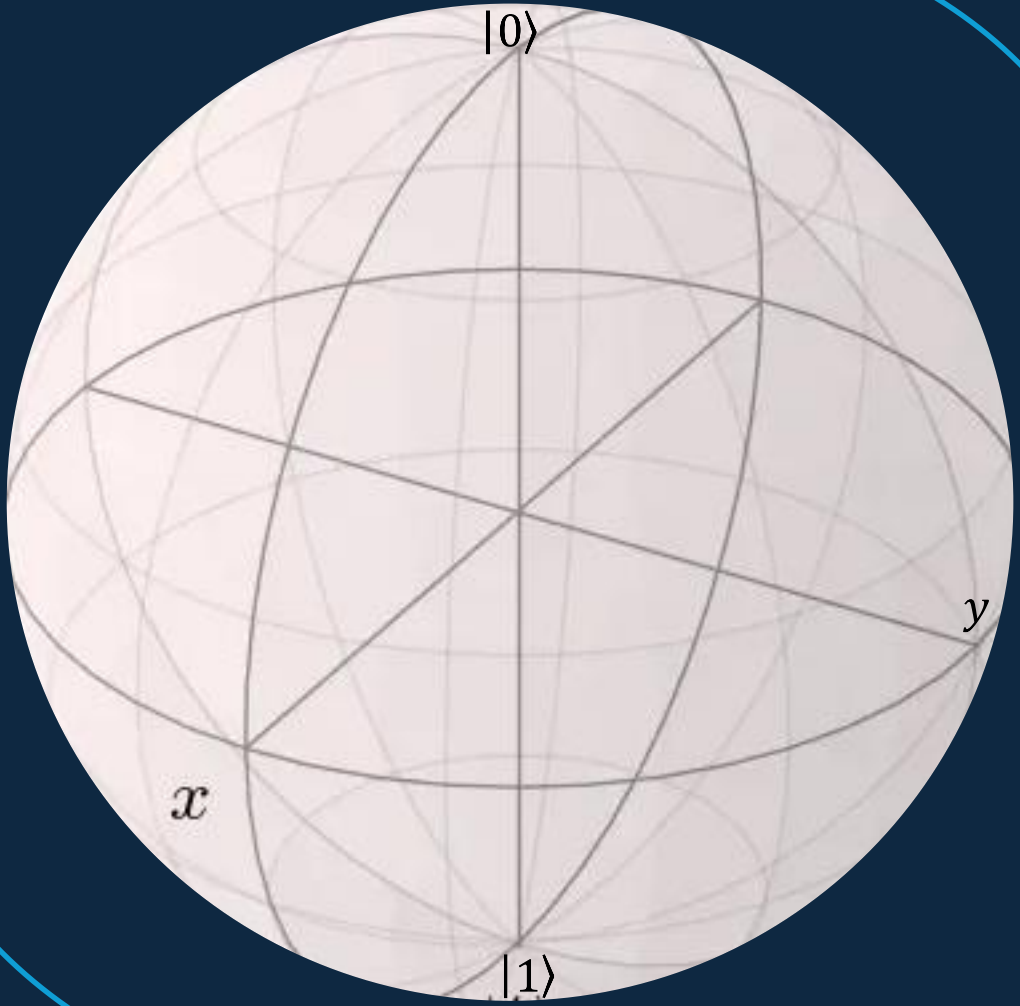
- 2-levels quantum systems:

$$\psi = \alpha|0\rangle + \beta|1\rangle$$

- Most effective representation is the Bloch-sphere:

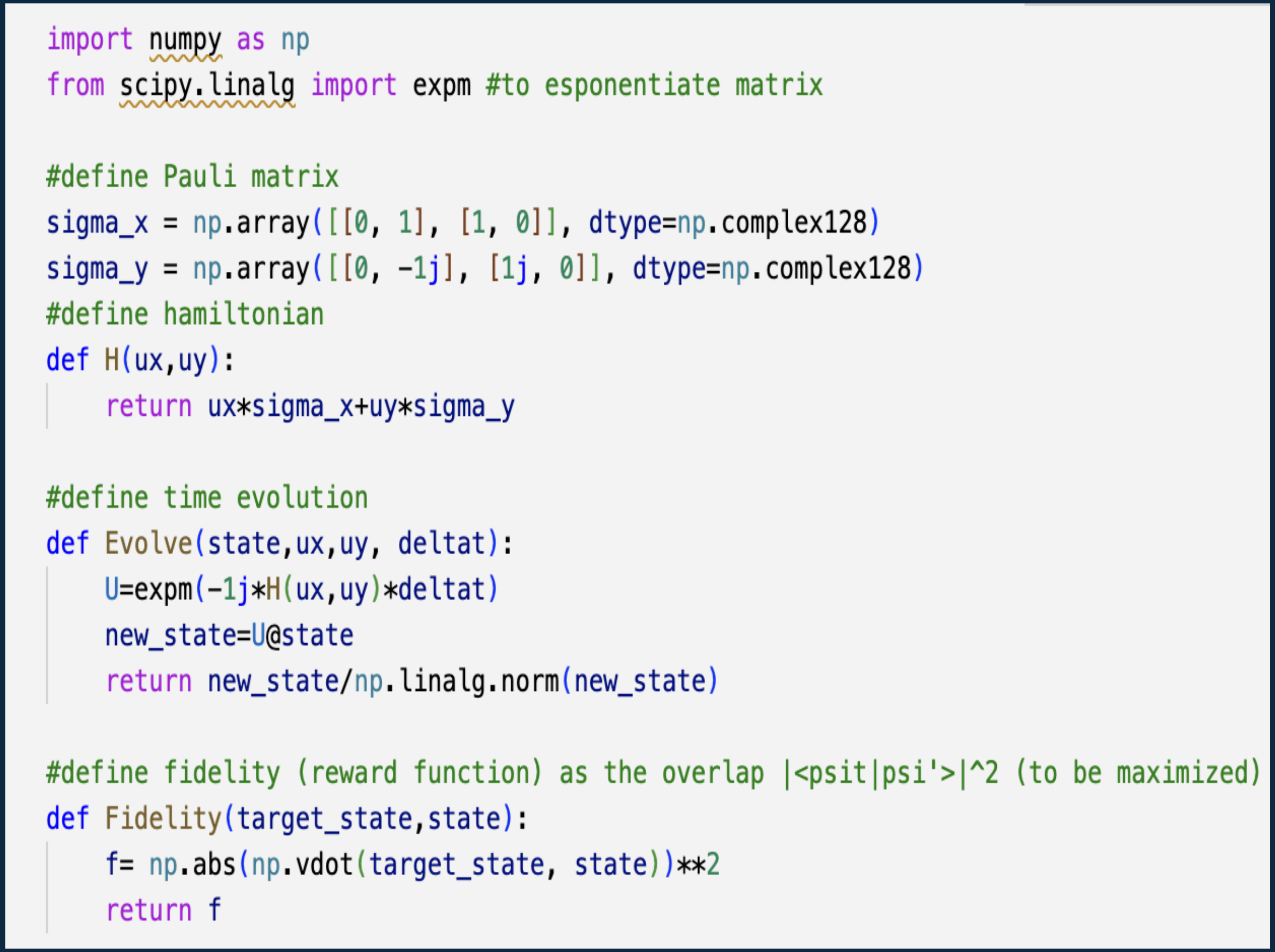
$$\psi = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\ |1\rangle$$

$$(\sin\theta\cos\phi\,, \sin\theta\sin\phi, \cos\theta)$$



Spin Qubits

- Physical realizations of Qubit using an electron spin confined in a magnetic field: $\psi = \alpha|\uparrow\rangle + \beta|\downarrow\rangle$
- The Hamiltonian can be modeled as $H(\mathrm{t}) = u_x(\mathrm{t})\sigma_x + u_y(\mathrm{t})\sigma_y$
- Where $U = \left(u_x, u_y\right)$ parametrizes the action of an extern E.M. field
- Time evolution:
  $$\psi(t + \Delta t) = e^{-iH(t)\Delta t}|\psi(t)\rangle$$



```python
import numpy as np
from scipy.linalg import expm #to esponentiate matrix

#define Pauli matrix
sigma_x = np.array([[0, 1], [1, 0]], dtype=np.complex128)
sigma_y = np.array([[0, -1j], [1j, 0]], dtype=np.complex128)
#define hamiltonian
def H(ux,uy):
    return ux*sigma_x+uy*sigma_y

#define time evolution
def Evolve(state,ux,uy, deltat):
    U=expm(-1j*H(ux,uy)*deltat)
    new_state=U@state
    return new_state/np.linalg.norm(new_state)

#define fidelity (reward function) as the overlap |<psit|psi'>|^2 (to be maximized)
def Fidelity(target_state,state):
    f= np.abs(np.vdot(target_state, state))**2
    return f
```

Reinforcement learning

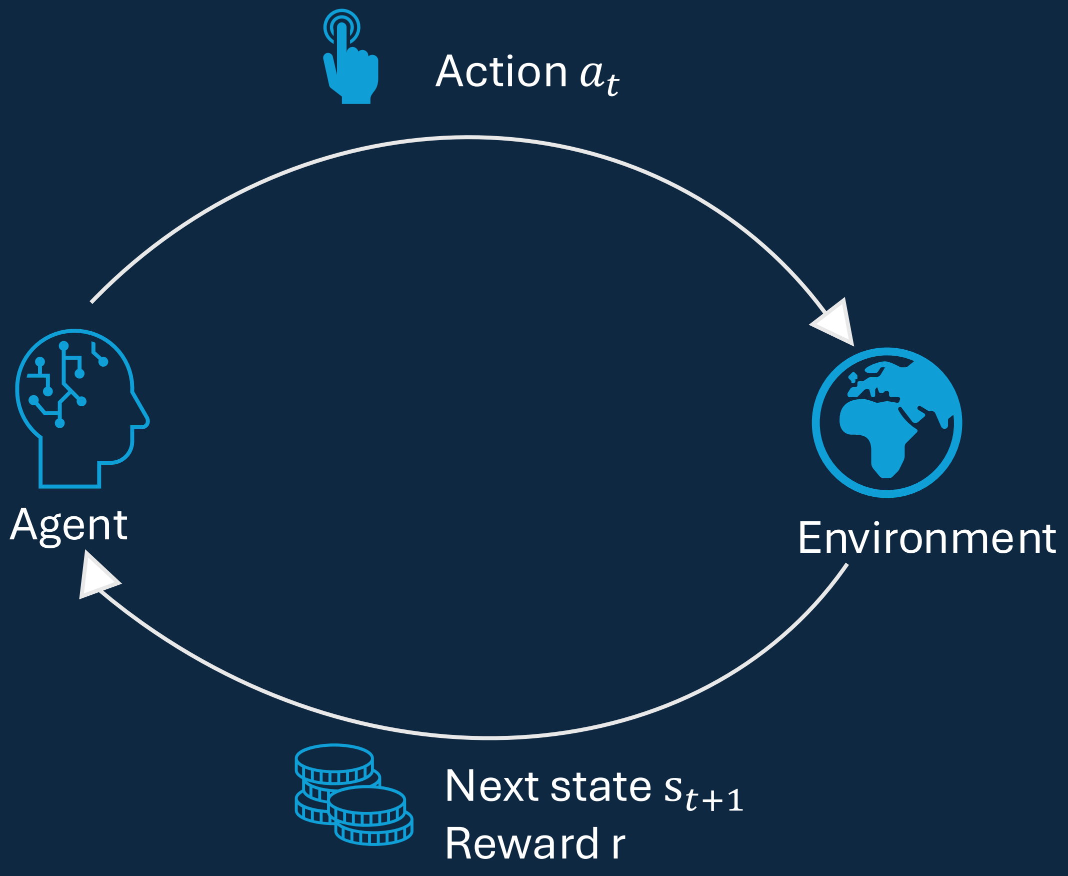
- Agent interacts with the environment and receives a reward based on its action.

- The better the action, the greater the reward

- The agent follows a policy, wich tells it the action to take in each state:
$$\pi(s_t) = a_t$$

- The agent aims to find a policy that maximizes the expected cumulative (discounted) reward:

$$\pi^* = \arg\max_{\pi} E\left[\sum_{t\geq 0} \gamma^t r_t \mid \pi\right]$$



Deep Q-Learning (DQN)

- The core is the action-value function:

$$Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^{t} r_{t} \mid s_{0} = s,\ a_{0} = a,\ \pi]$$

- The best policy is the one that follows the optimal Q-function:

$$Q^{*} = \max_{\pi} Q^{\pi}(s,a) \ \rightarrow \ \pi^{*} = arg \max_{a} Q^{*}(s,a)$$

- It can be recursively computed using the bellman equation:

$$Q^{*}(s,a) = \mathbb{E}\left[r + \gamma max_{a'}\left(Q(s', a')\right)\middle| s,a\right]$$

- At each step, the update follows:

$$Q^{i+1}(s,a) = \mathbb{E}\left[r + \gamma max_{a'}\left(Q^{i}(s', a')\right)\middle| s,a\right] \rightarrow Q^{*}$$

Training loop

- $Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s,\, a_0 = a,\, \pi]$

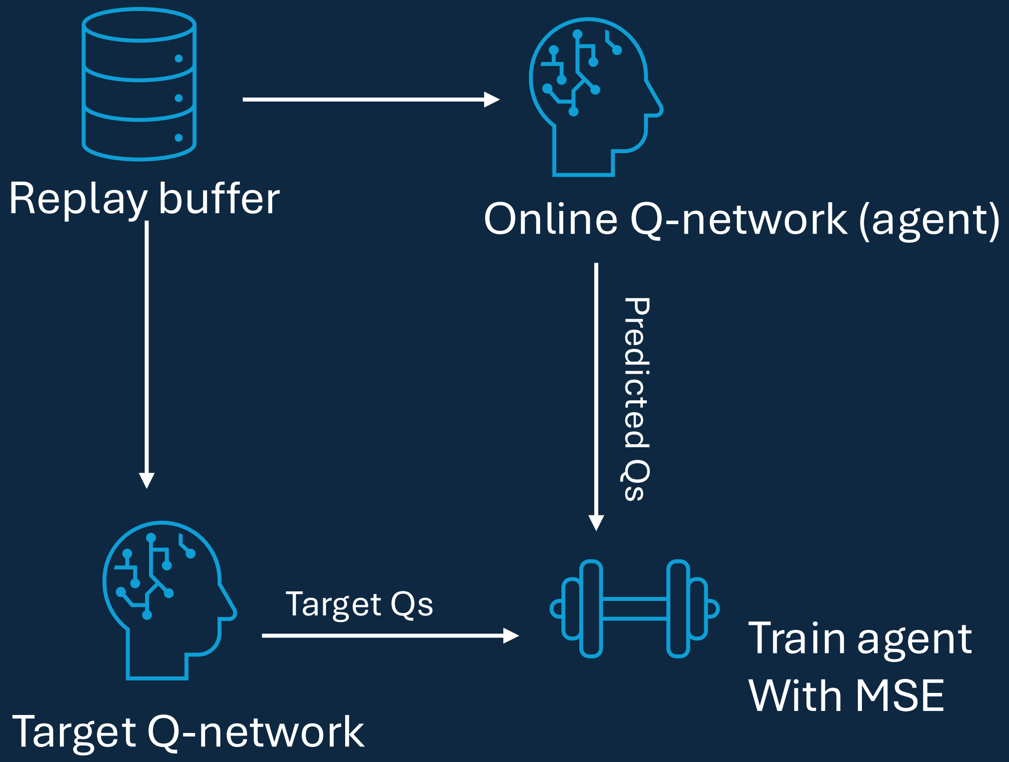
  evaluates expected reward of a state-action pair, following the current policy. Here it is parametrized by a DNN.

- the agent stores experiences in a replay buffer, then samples minibatches to compute target values y using a target DNN, (a copy of the agent updated periodically)

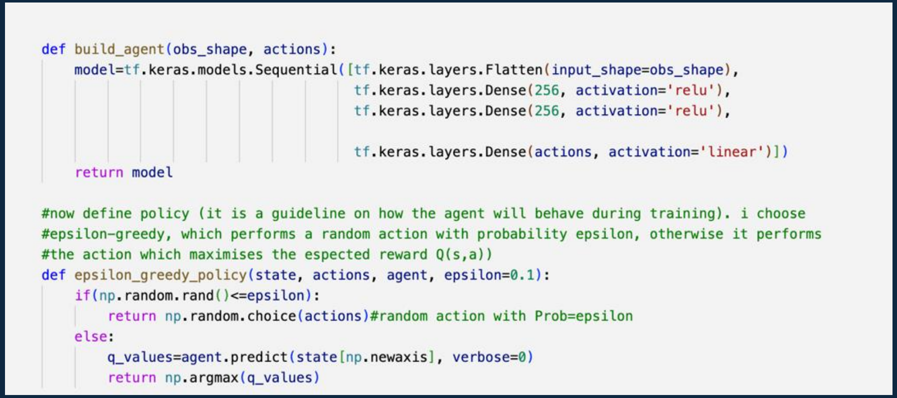
  $$y = r + \gamma max_{a'} Q(s', a'; \theta^{-})$$

- The online network is trained by minimizing the MSE loss between target and predicted Q-values:

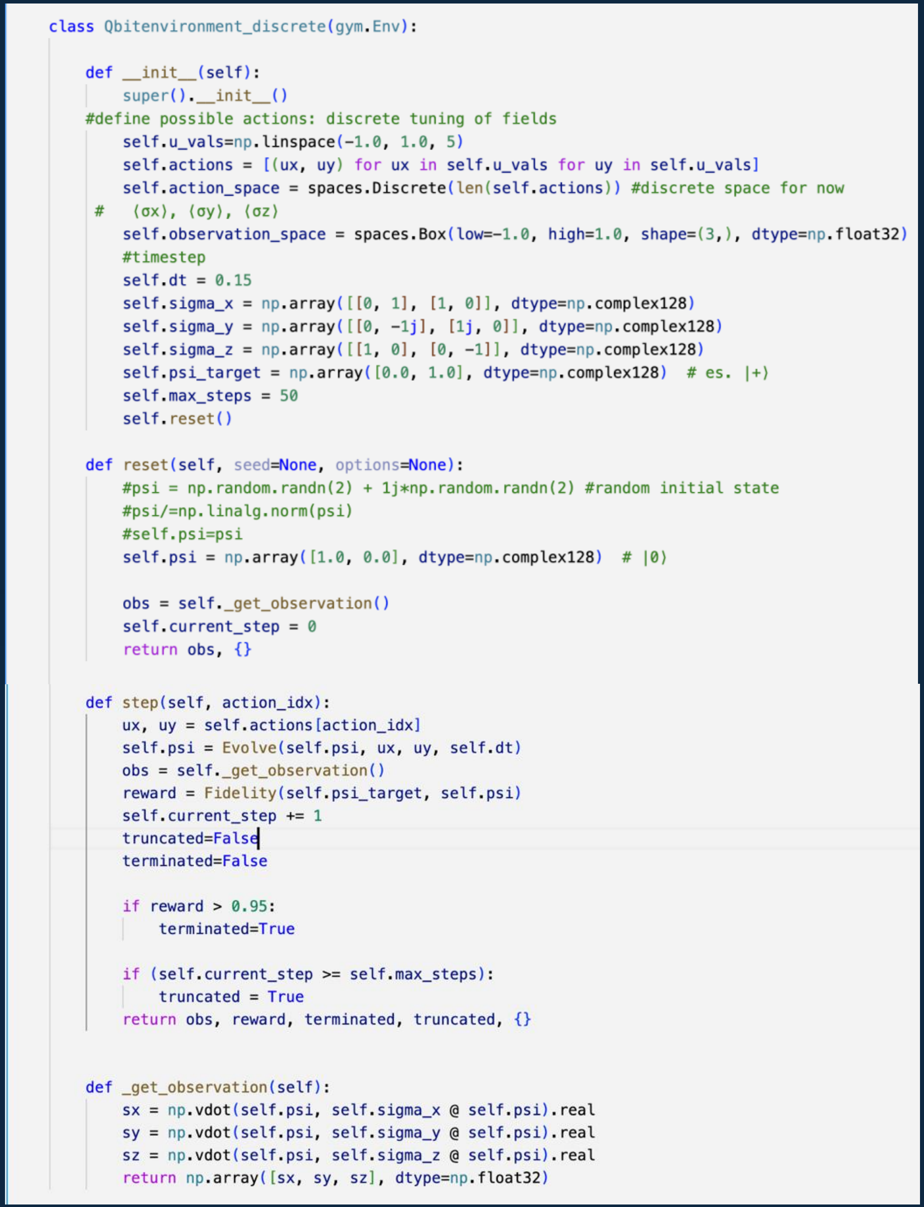
$$L(\theta) = MSE(Q(s_i, a_i, \theta), y_i)$$



First approach: discretized action space

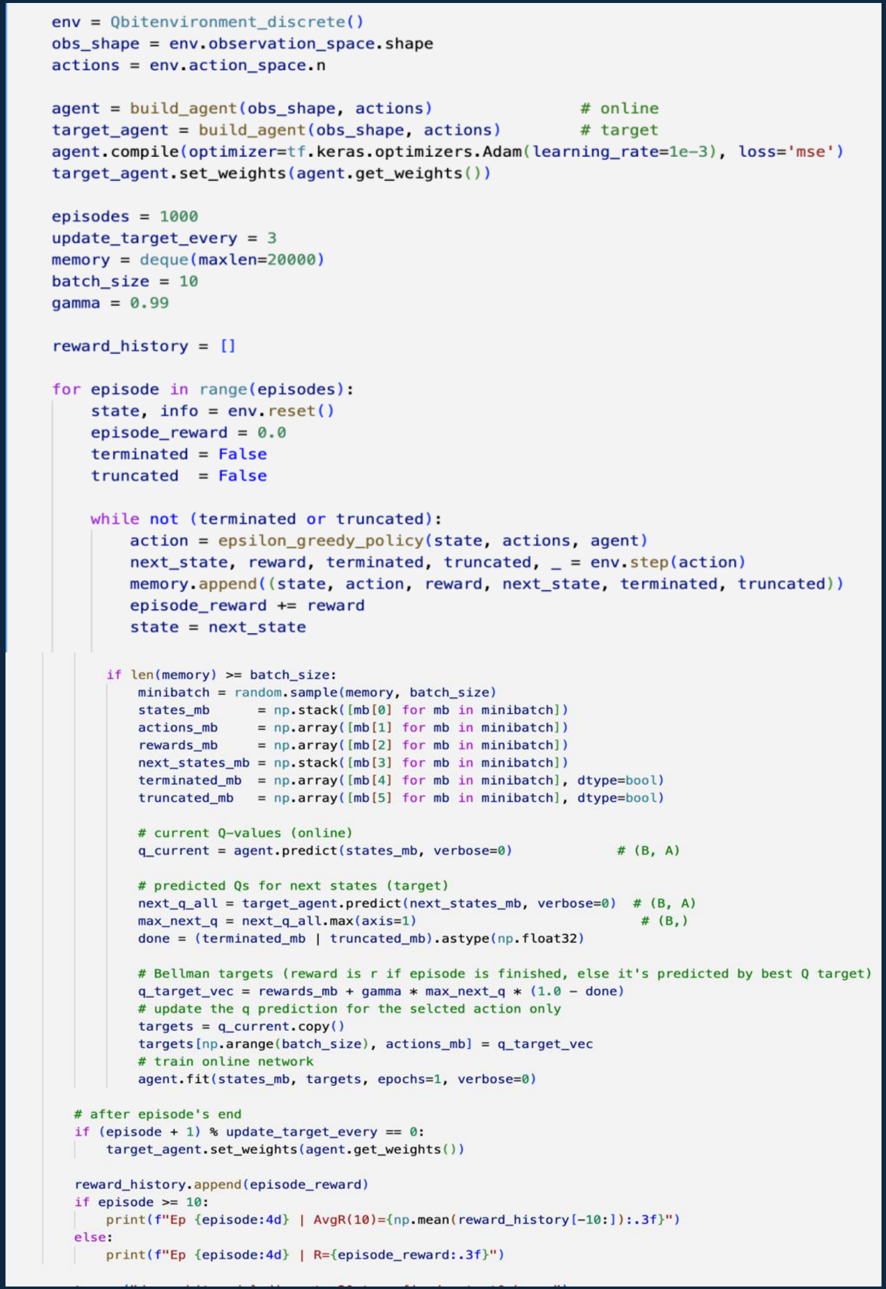


```python
def build_agent(obs_shape, actions):
    model=tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=obs_shape),
                                      tf.keras.layers.Dense(256, activation='relu'),
                                      tf.keras.layers.Dense(256, activation='relu'),

                                      tf.keras.layers.Dense(actions, activation='linear')])
    return model

#now define policy (it is a guideline on how the agent will behave during training). i choose
#epsilon-greedy, which performs a random action with probability epsilon, otherwise it performs
#the action which maximises the espected reward Q(s,a))
def epsilon_greedy_policy(state, actions, agent, epsilon=0.1):
    if(np.random.rand()<=epsilon):
        return np.random.choice(actions)#random action with Prob=epsilon
    else:
        q_values=agent.predict(state[np.newaxis], verbose=0)
        return np.argmax(q_values)
```



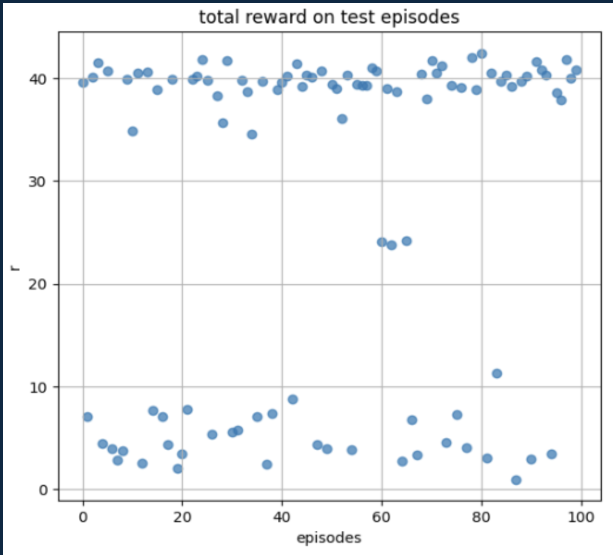
```python
class Qbitenvironment_discrete(gym.Env):

    def __init__(self):
        super().__init__()
    #define possible actions: discrete tuning of fields
        self.u_vals=np.linspace(-1.0, 1.0, 5)
        self.actions = [(ux, uy) for ux in self.u_vals for uy in self.u_vals]
        self.action_space = spaces.Discrete(len(self.actions)) #discrete space for now
     #   ⟨σx⟩, ⟨σy⟩, ⟨σz⟩
        self.observation_space = spaces.Box(low=-1.0, high=1.0, shape=(3,), dtype=np.float32)
        #timestep
        self.dt = 0.15
        self.sigma_x = np.array([[0, 1], [1, 0]], dtype=np.complex128)
        self.sigma_y = np.array([[0, -1j], [1j, 0]], dtype=np.complex128)
        self.sigma_z = np.array([[1, 0], [0, -1]], dtype=np.complex128)
        self.psi_target = np.array([0.0, 1.0], dtype=np.complex128)  # es. |+⟩
        self.max_steps = 50
        self.reset()

    def reset(self, seed=None, options=None):
        #psi = np.random.randn(2) + 1j*np.random.randn(2) #random initial state
        #psi/=np.linalg.norm(psi)
        #self.psi=psi
        self.psi = np.array([1.0, 0.0], dtype=np.complex128)  # |0⟩

        obs = self._get_observation()
        self.current_step = 0
        return obs, {}

    def step(self, action_idx):
        ux, uy = self.actions[action_idx]
        self.psi = Evolve(self.psi, ux, uy, self.dt)
        obs = self._get_observation()
        reward = Fidelity(self.psi_target, self.psi)
        self.current_step += 1
        truncated=False
        terminated=False

        if reward > 0.95:
            terminated=True

        if (self.current_step >= self.max_steps):
            truncated = True
        return obs, reward, terminated, truncated, {}


    def _get_observation(self):
        sx = np.vdot(self.psi, self.sigma_x @ self.psi).real
        sy = np.vdot(self.psi, self.sigma_y @ self.psi).real
        sz = np.vdot(self.psi, self.sigma_z @ self.psi).real
        return np.array([sx, sy, sz], dtype=np.float32)
```

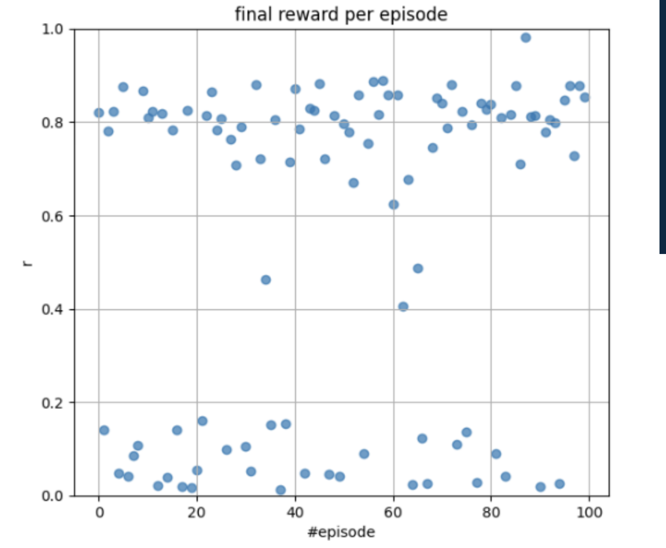
Training loop

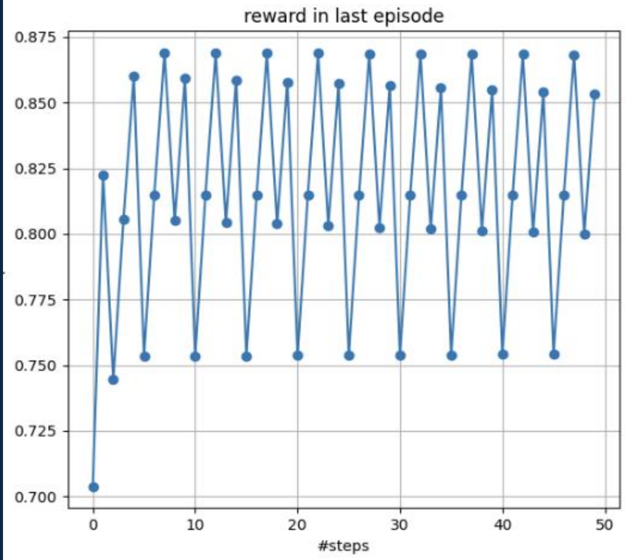


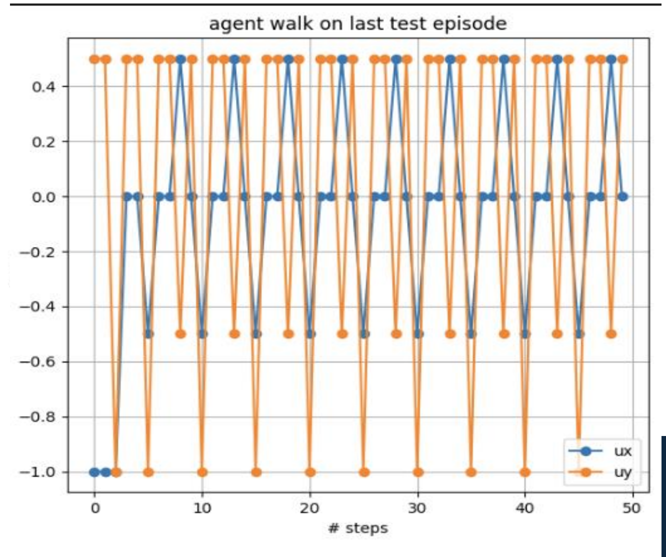
```python
env = Qbitenvironment_discrete()
obs_shape = env.observation_space.shape
actions = env.action_space.n

agent = build_agent(obs_shape, actions)                # online
target_agent = build_agent(obs_shape, actions)         # target
agent.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3), loss='mse')
target_agent.set_weights(agent.get_weights())

episodes = 1000
update_target_every = 3
memory = deque(maxlen=20000)
batch_size = 10
gamma = 0.99

reward_history = []

for episode in range(episodes):
    state, info = env.reset()
    episode_reward = 0.0
    terminated = False
    truncated  = False

    while not (terminated or truncated):
        action = epsilon_greedy_policy(state, actions, agent)
        next_state, reward, terminated, truncated, _ = env.step(action)
        memory.append((state, action, reward, next_state, terminated, truncated))
        episode_reward += reward
        state = next_state

        if len(memory) >= batch_size:
            minibatch = random.sample(memory, batch_size)
            states_mb      = np.stack([mb[0] for mb in minibatch])
            actions_mb     = np.array([mb[1] for mb in minibatch])
            rewards_mb     = np.array([mb[2] for mb in minibatch])
            next_states_mb = np.stack([mb[3] for mb in minibatch])
            terminated_mb  = np.array([mb[4] for mb in minibatch], dtype=bool)
            truncated_mb   = np.array([mb[5] for mb in minibatch], dtype=bool)

            # current Q-values (online)
            q_current = agent.predict(states_mb, verbose=0)             # (B, A)

            # predicted Qs for next states (target)
            next_q_all = target_agent.predict(next_states_mb, verbose=0)  # (B, A)
            max_next_q = next_q_all.max(axis=1)                            # (B,)
            done = (terminated_mb | truncated_mb).astype(np.float32)

            # Bellman targets (reward is r if episode is finished, else it's predicted by best Q target)
            q_target_vec = rewards_mb + gamma * max_next_q * (1.0 - done)
            # update the q prediction for the selcted action only
            targets = q_current.copy()
            targets[np.arange(batch_size), actions_mb] = q_target_vec
            # train online network
            agent.fit(states_mb, targets, epochs=1, verbose=0)

    # after episode's end
    if (episode + 1) % update_target_every == 0:
        target_agent.set_weights(agent.get_weights())

    reward_history.append(episode_reward)
    if episode >= 10:
        print(f"Ep {episode:4d} | AvgR(10)={np.mean(reward_history[-10:]):.3f}")
    else:
        print(f"Ep {episode:4d} | R={episode_reward:.3f}")
```

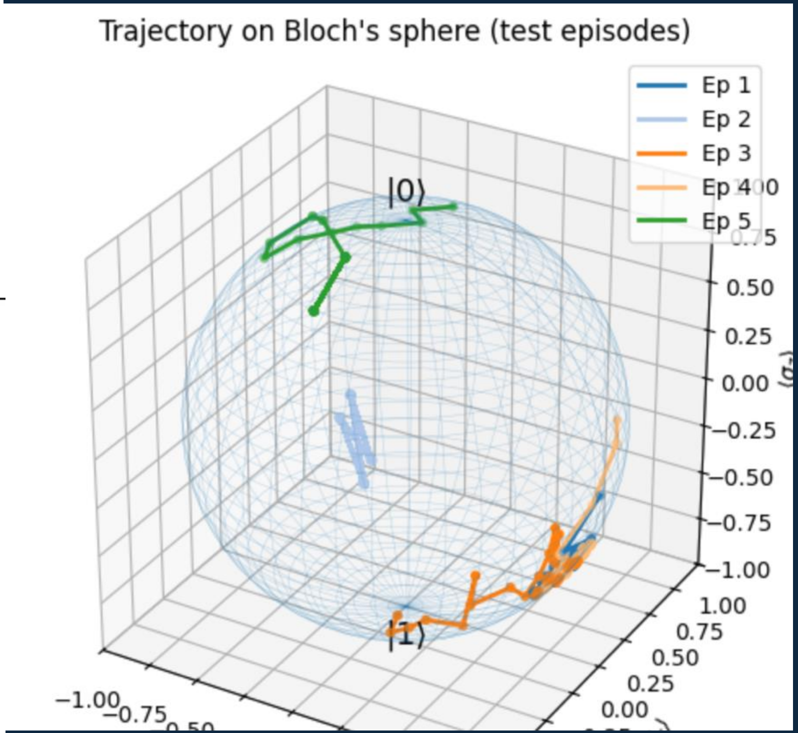
Results


total reward on test episodes
r
episodes
0
10
20
30
40
0
20
40
60
80
100


final reward per episode
r
#episode
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100


reward in last episode
#steps
0.700
0.725
0.750
0.775
0.800
0.825
0.850
0.875
0
10
20
30
40
50


agent walk on last test episode
# steps
ux
uy
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0
10
20
30
40
50


Trajectory on Bloch's sphere (test episodes)
Ep 1
Ep 2
Ep 3
Ep 400
Ep 5
|0⟩
|1⟩
⟨σz⟩
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
1.00
0.75
0.50
0.25
0.00
−1.00
−0.75

Continuous action space- Soft Actor Critic (SAC)

Ingredients:

- Actor learns to maximize:

$$J(\pi) = \mathbb{E}\left[\sum_t \gamma^t r(s,t) - \alpha log\big(\pi(a|s)\big)\right]$$

- Stochastic policy (actor): π_θ.
- Actor returns the parameters of a PDF (Gaussian) from which the action is sampled
- 2-Q-values networks (critic)
- Temperature for entropy regularization α
- Replay buffer

Advantages:

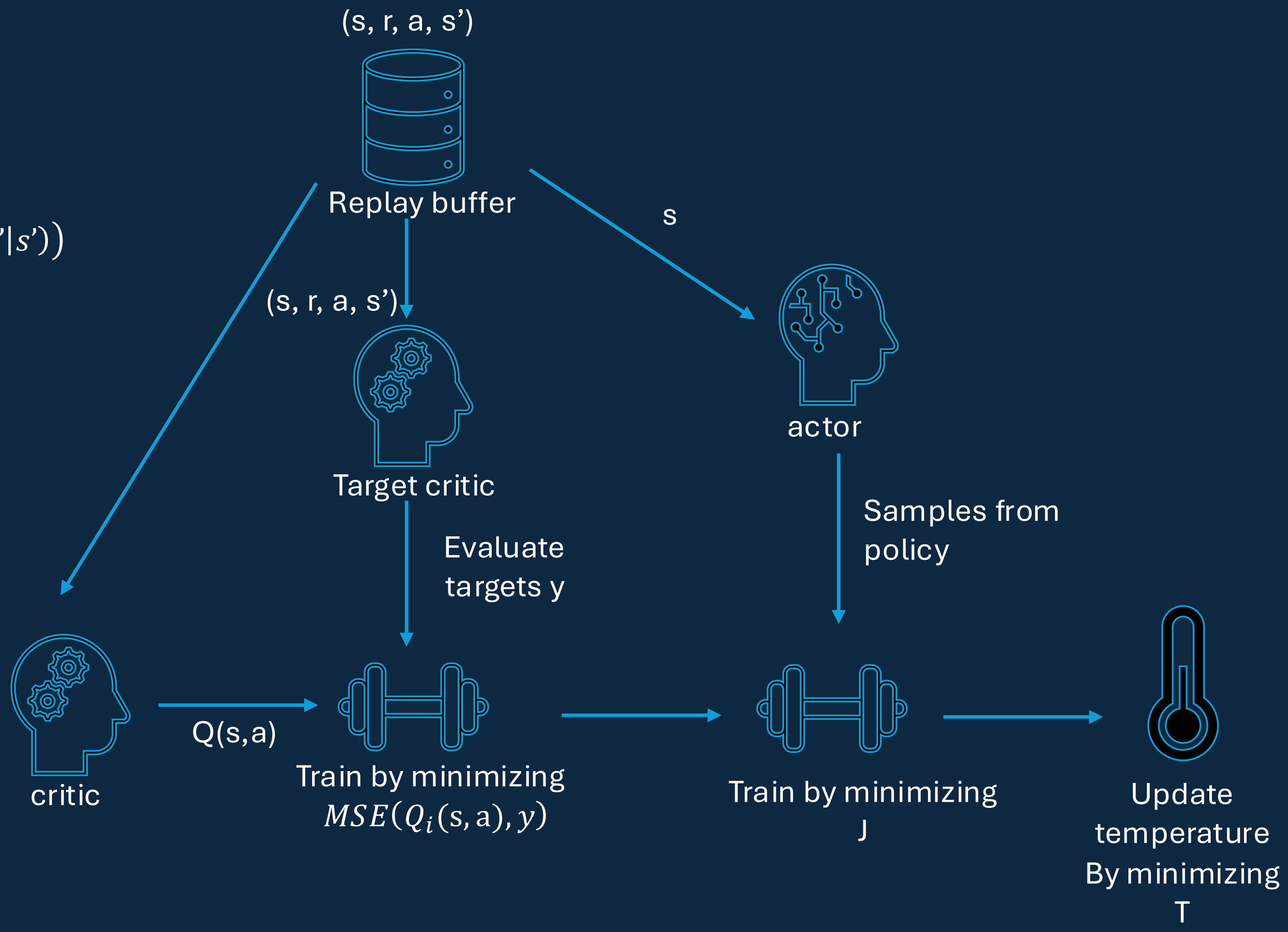
- Handles continuous action space efficiently thanks to the stochastic policy
- Stable exploration-exploitation trade off throug entropy regularization
- Reduced overestimation bias by using two Q-networks

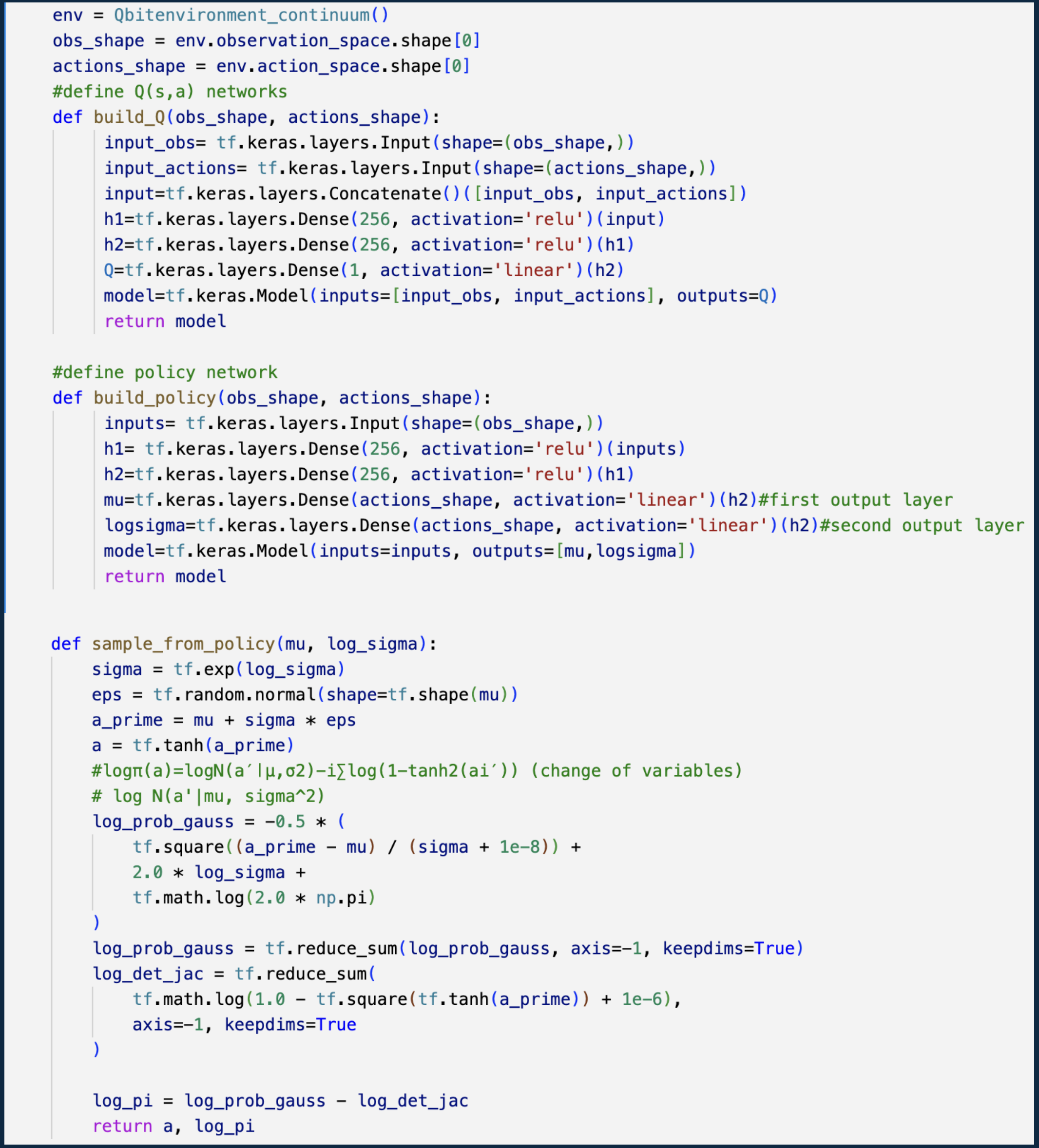
Training loop

- Target evaluation:
  $y_i = r + \gamma\big(\min\big(Q_1(s'.a'), Q_2(s',a')\big) - \alpha \log \pi_\theta(a'|s')\big)$

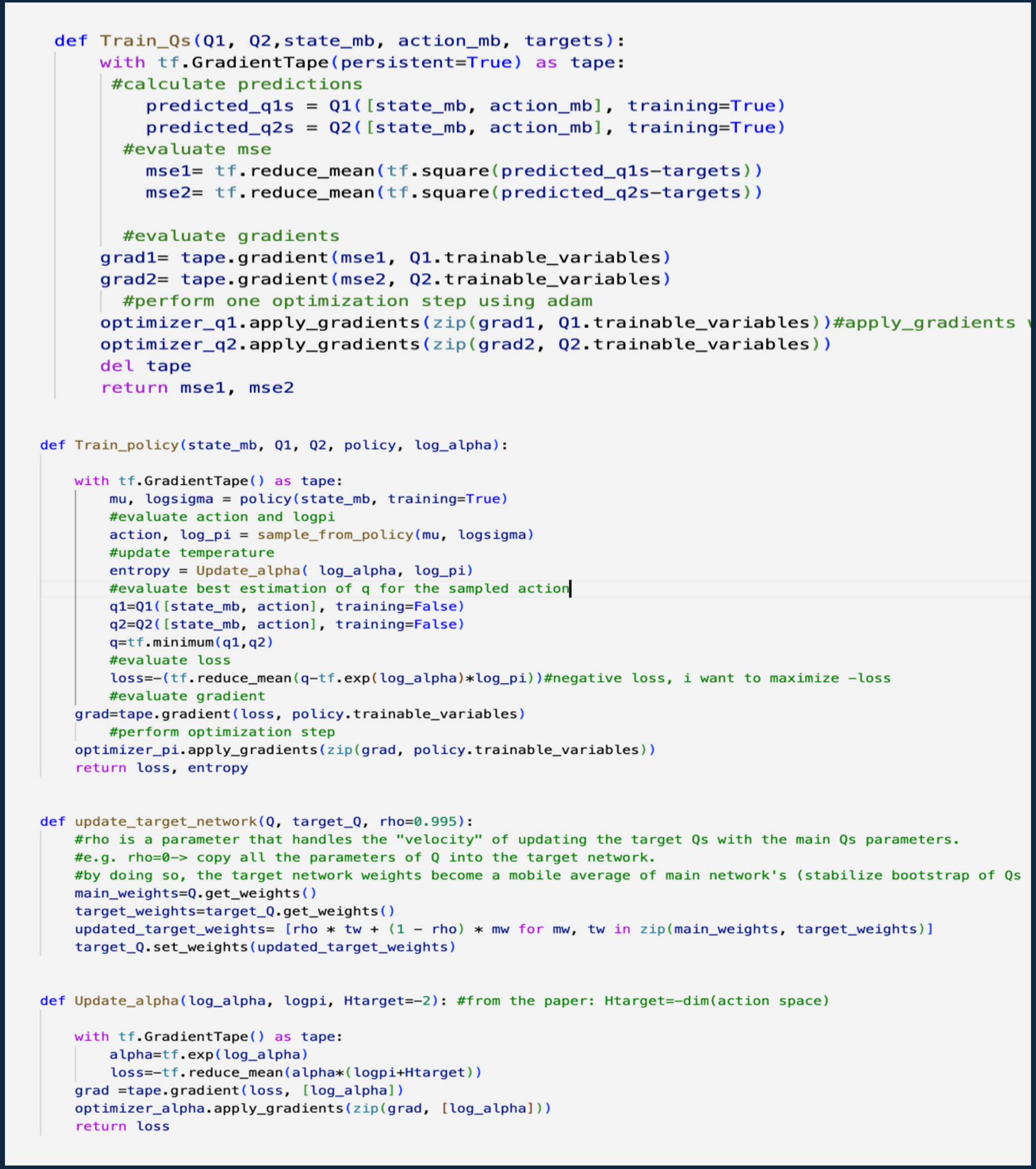
- Actor loss:
  $J(\pi) = \mathbb{E}[\alpha \log \pi - \min(Q_1(s,a), Q_2(s,a))]$

- Temperature update loss:
  $T(\alpha) = \ \mathbb{E}[-\alpha(\log\big(\pi(s|a) + H_{target}\big]]$



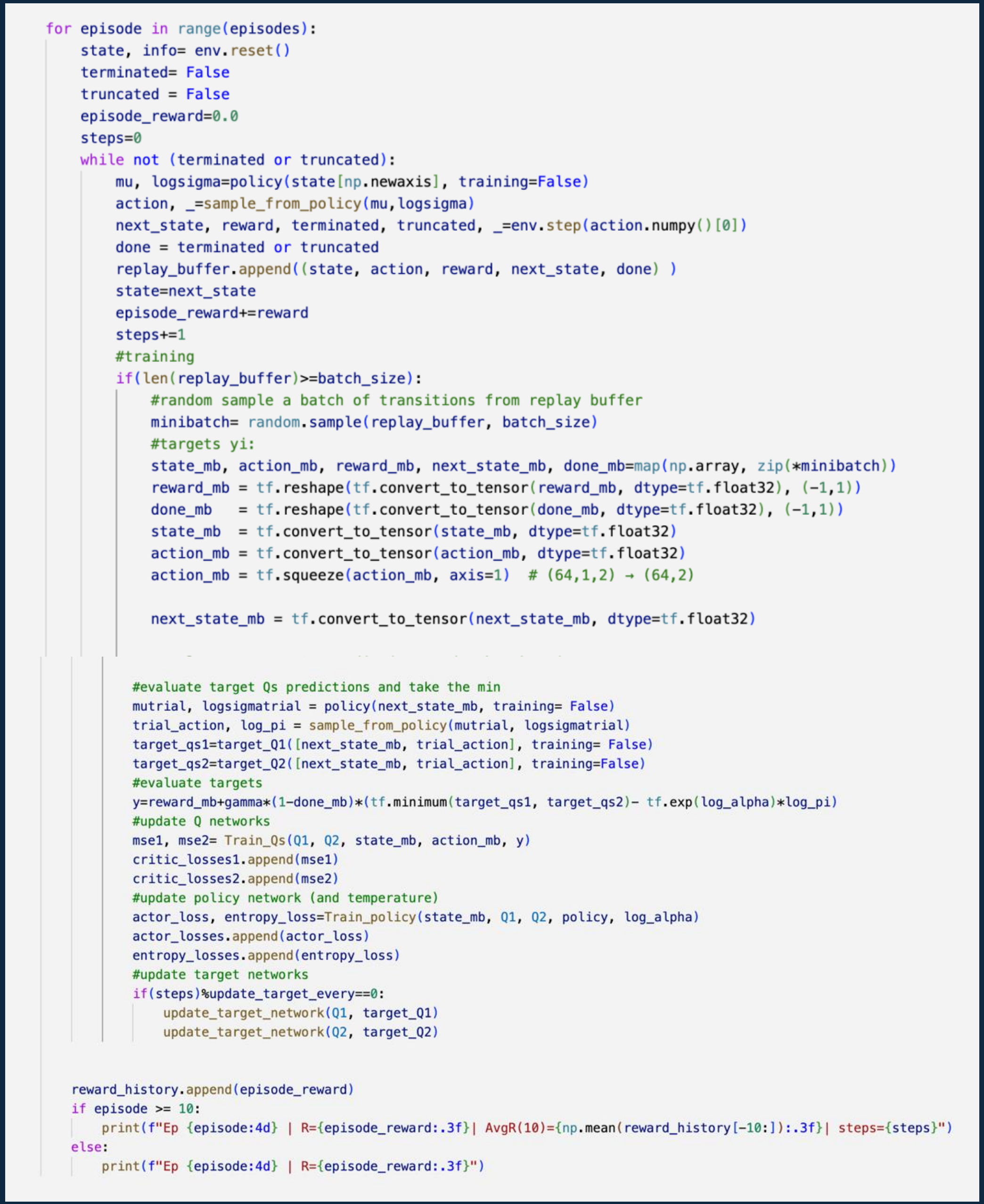


```python
env = Qbitenvironment_continuum()
obs_shape = env.observation_space.shape[0]
actions_shape = env.action_space.shape[0]
#define Q(s,a) networks
def build_Q(obs_shape, actions_shape):
    input_obs= tf.keras.layers.Input(shape=(obs_shape,))
    input_actions= tf.keras.layers.Input(shape=(actions_shape,))
    input=tf.keras.layers.Concatenate()([input_obs, input_actions])
    h1=tf.keras.layers.Dense(256, activation='relu')(input)
    h2=tf.keras.layers.Dense(256, activation='relu')(h1)
    Q=tf.keras.layers.Dense(1, activation='linear')(h2)
    model=tf.keras.Model(inputs=[input_obs, input_actions], outputs=Q)
    return model

#define policy network
def build_policy(obs_shape, actions_shape):
    inputs= tf.keras.layers.Input(shape=(obs_shape,))
    h1= tf.keras.layers.Dense(256, activation='relu')(inputs)
    h2=tf.keras.layers.Dense(256, activation='relu')(h1)
    mu=tf.keras.layers.Dense(actions_shape, activation='linear')(h2)#first output layer
    logsigma=tf.keras.layers.Dense(actions_shape, activation='linear')(h2)#second output layer
    model=tf.keras.Model(inputs=inputs, outputs=[mu,logsigma])
    return model


def sample_from_policy(mu, log_sigma):
    sigma = tf.exp(log_sigma)
    eps = tf.random.normal(shape=tf.shape(mu))
    a_prime = mu + sigma * eps
    a = tf.tanh(a_prime)
    #logπ(a)=logN(a′|μ,σ2)−i∑log(1−tanh2(ai′)) (change of variables)
    # log N(a'|mu, sigma^2)
    log_prob_gauss = -0.5 * (
        tf.square((a_prime - mu) / (sigma + 1e-8)) +
        2.0 * log_sigma +
        tf.math.log(2.0 * np.pi)
    )
    log_prob_gauss = tf.reduce_sum(log_prob_gauss, axis=-1, keepdims=True)
    log_det_jac = tf.reduce_sum(
        tf.math.log(1.0 - tf.square(tf.tanh(a_prime)) + 1e-6),
        axis=-1, keepdims=True
    )

    log_pi = log_prob_gauss - log_det_jac
    return a, log_pi
```



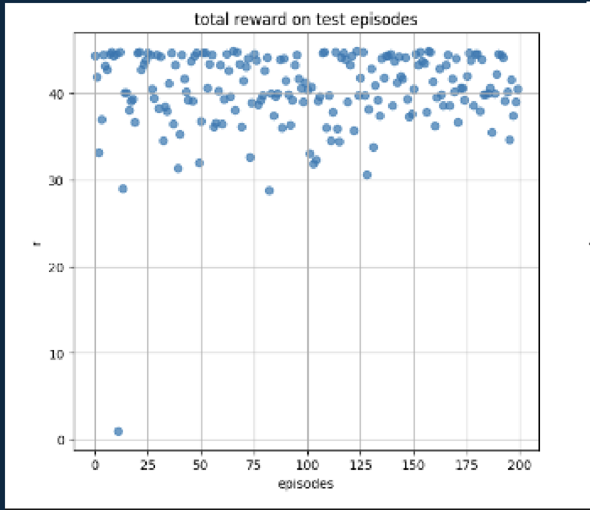
```python
def Train_Qs(Q1, Q2,state_mb, action_mb, targets):
    with tf.GradientTape(persistent=True) as tape:
     #calculate predictions
        predicted_q1s = Q1([state_mb, action_mb], training=True)
        predicted_q2s = Q2([state_mb, action_mb], training=True)
      #evaluate mse
        mse1= tf.reduce_mean(tf.square(predicted_q1s-targets))
        mse2= tf.reduce_mean(tf.square(predicted_q2s-targets))

      #evaluate gradients
    grad1= tape.gradient(mse1, Q1.trainable_variables)
    grad2= tape.gradient(mse2, Q2.trainable_variables)
      #perform one optimization step using adam
    optimizer_q1.apply_gradients(zip(grad1, Q1.trainable_variables))#apply_gradients
    optimizer_q2.apply_gradients(zip(grad2, Q2.trainable_variables))
    del tape
    return mse1, mse2


def Train_policy(state_mb, Q1, Q2, policy, log_alpha):

    with tf.GradientTape() as tape:
        mu, logsigma = policy(state_mb, training=True)
        #evaluate action and logpi
        action, log_pi = sample_from_policy(mu, logsigma)
        #update temperature
        entropy = Update_alpha( log_alpha, log_pi)
        #evaluate best estimation of q for the sampled action
        q1=Q1([state_mb, action], training=False)
        q2=Q2([state_mb, action], training=False)
        q=tf.minimum(q1,q2)
        #evaluate loss
        loss=-(tf.reduce_mean(q-tf.exp(log_alpha)*log_pi))#negative loss, i want to maximize -loss
        #evaluate gradient
    grad=tape.gradient(loss, policy.trainable_variables)
        #perform optimization step
    optimizer_pi.apply_gradients(zip(grad, policy.trainable_variables))
    return loss, entropy


def update_target_network(Q, target_Q, rho=0.995):
    #rho is a parameter that handles the "velocity" of updating the target Qs with the main Qs parameters.
    #e.g. rho=0-> copy all the parameters of Q into the target network.
    #by doing so, the target network weights become a mobile average of main network's (stabilize bootstrap of Qs
    main_weights=Q.get_weights()
    target_weights=target_Q.get_weights()
    updated_target_weights= [rho * tw + (1 - rho) * mw for mw, tw in zip(main_weights, target_weights)]
    target_Q.set_weights(updated_target_weights)


def Update_alpha(log_alpha, logpi, Htarget=-2): #from the paper: Htarget=-dim(action space)

    with tf.GradientTape() as tape:
        alpha=tf.exp(log_alpha)
        loss=-tf.reduce_mean(alpha*(logpi+Htarget))
    grad =tape.gradient(loss, [log_alpha])
    optimizer_alpha.apply_gradients(zip(grad, [log_alpha]))
    return loss
```

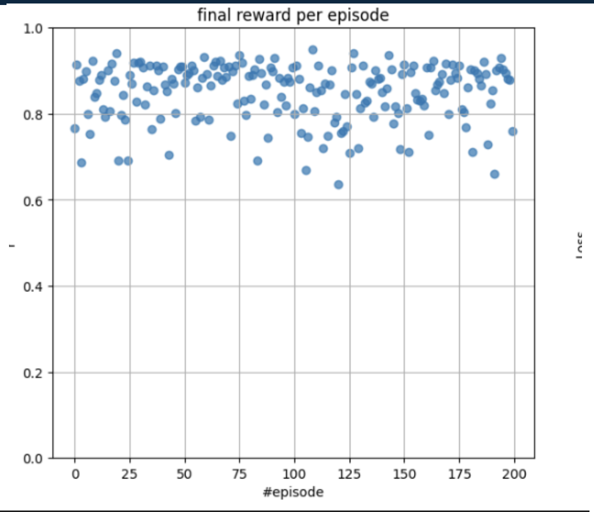
Training loop

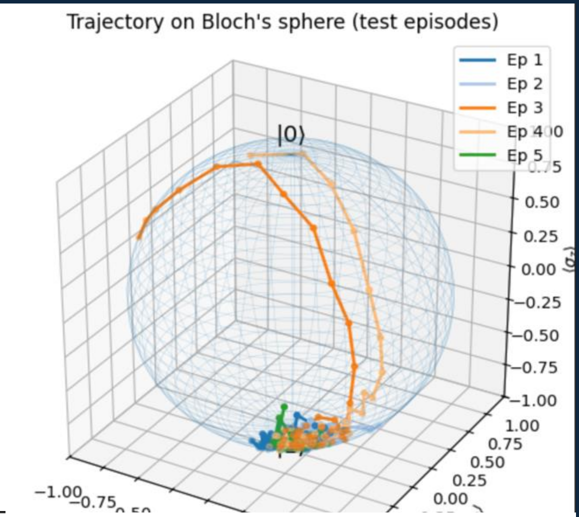


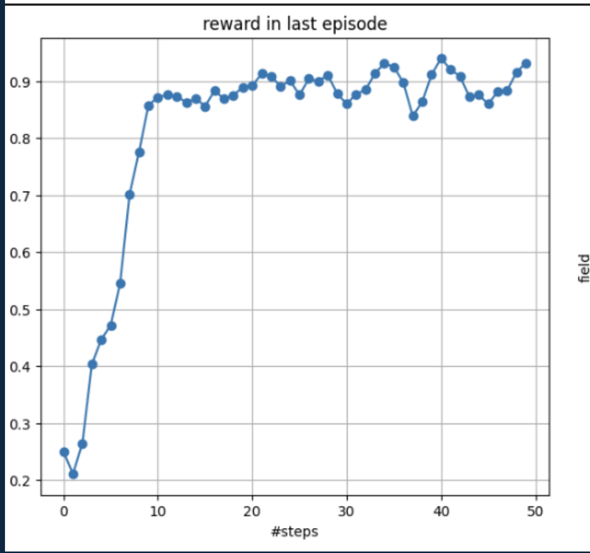
```
for episode in range(episodes):
    state, info= env.reset()
    terminated= False
    truncated = False
    episode_reward=0.0
    steps=0
    while not (terminated or truncated):
        mu, logsigma=policy(state[np.newaxis], training=False)
        action, _=sample_from_policy(mu,logsigma)
        next_state, reward, terminated, truncated, _=env.step(action.numpy()[0])
        done = terminated or truncated
        replay_buffer.append((state, action, reward, next_state, done) )
        state=next_state
        episode_reward+=reward
        steps+=1
        #training
        if(len(replay_buffer)>=batch_size):
            #random sample a batch of transitions from replay buffer
            minibatch= random.sample(replay_buffer, batch_size)
            #targets yi:
            state_mb, action_mb, reward_mb, next_state_mb, done_mb=map(np.array, zip(*minibatch))
            reward_mb = tf.reshape(tf.convert_to_tensor(reward_mb, dtype=tf.float32), (-1,1))
            done_mb   = tf.reshape(tf.convert_to_tensor(done_mb, dtype=tf.float32), (-1,1))
            state_mb  = tf.convert_to_tensor(state_mb, dtype=tf.float32)
            action_mb = tf.convert_to_tensor(action_mb, dtype=tf.float32)
            action_mb = tf.squeeze(action_mb, axis=1)  # (64,1,2) → (64,2)

            next_state_mb = tf.convert_to_tensor(next_state_mb, dtype=tf.float32)

            #evaluate target Qs predictions and take the min
            mutrial, logsigmatrial = policy(next_state_mb, training= False)
            trial_action, log_pi = sample_from_policy(mutrial, logsigmatrial)
            target_qs1=target_Q1([next_state_mb, trial_action], training= False)
            target_qs2=target_Q2([next_state_mb, trial_action], training=False)
            #evaluate targets
            y=reward_mb+gamma*(1-done_mb)*(tf.minimum(target_qs1, target_qs2)- tf.exp(log_alpha)*log_pi)
            #update Q networks
            mse1, mse2= Train_Qs(Q1, Q2, state_mb, action_mb, y)
            critic_losses1.append(mse1)
            critic_losses2.append(mse2)
            #update policy network (and temperature)
            actor_loss, entropy_loss=Train_policy(state_mb, Q1, Q2, policy, log_alpha)
            actor_losses.append(actor_loss)
            entropy_losses.append(entropy_loss)
            #update target networks
            if(steps)%update_target_every==0:
                update_target_network(Q1, target_Q1)
                update_target_network(Q2, target_Q2)


    reward_history.append(episode_reward)
    if episode >= 10:
        print(f"Ep {episode:4d} | R={episode_reward:.3f}| AvgR(10)={np.mean(reward_history[-10:]):.3f}| steps={steps}")
    else:
        print(f"Ep {episode:4d} | R={episode_reward:.3f}")
```

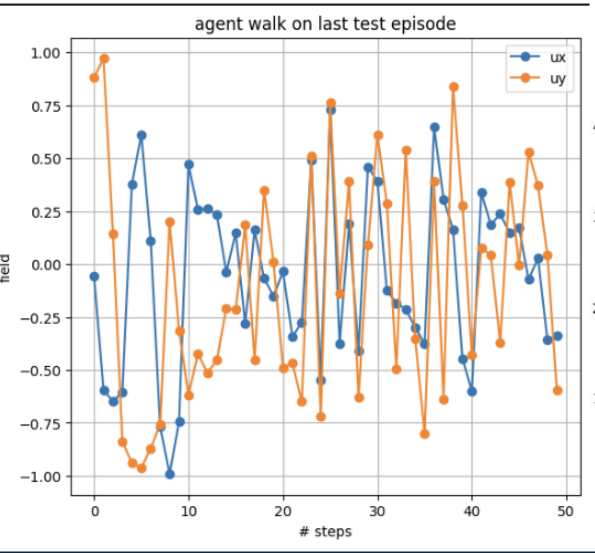
Results

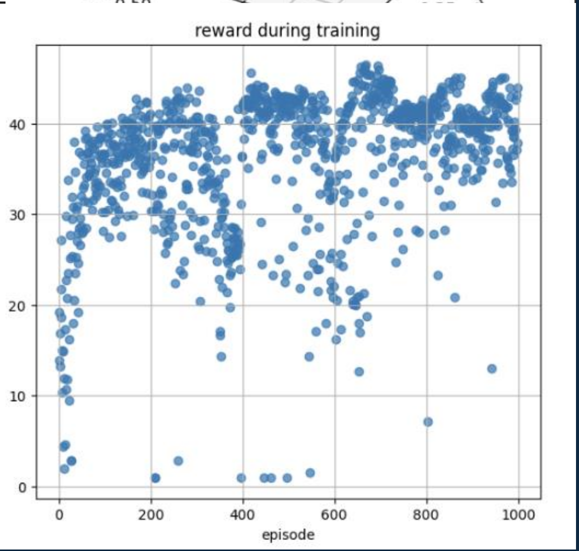

total reward on test episodes
r
episodes


final reward per episode
r
#episode


Trajectory on Bloch's sphere (test episodes)
|0⟩
Ep 1
Ep 2
Ep 3
Ep 400
Ep 5
Loss
⟨σz⟩


reward in last episode
#steps


agent walk on last test episode
ux
uy
field
# steps


reward during training
episode

Possible improvements

- Extend the system to more than one Qbit
- Extend to different Hamiltonians
- Introduce noise models

Citations

- DQN: ’’Playing Atari with deep reinforcement learning’’- DeepMind- https://arxiv.org/pdf/1312.5602
- Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor- Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine. https://arxiv.org/abs/1801.01290
- ’’Modern’’ version of the SAC algorithm by openAI: https://spinningup.openai.com/en/latest/algorithms/sac.html#soft-actor-critic
- Furtherly improved version of the SAC algorithm, including automatic temperature regularization: Soft Actor-Critic Algorithms and Applications- T.Haarnoja et al. https://arxiv.org/abs/1801.01290
- Gymnasium custom environments: https://gymnasium.farama.org/introduction/create_custom_env/