Reinforcement Learning Course: WiSe 2020/21

Marin Bukov
Faculty of Physics, Sofia University, 5 James Bourchier Blvd., 1164 Sofia, Bulgaria
(Dated: December 12, 2020)

I. DEEP POLICY GRADIENT (PG)

In this notebook, our goal is to implement the REINFORCE algorithm for policy gradient using JAX. We will apply this RL algorithm to control a single quantum bit of information (qubit).

A. The REINFROCE Algorithm

The reinforcement learning objective J is the expected total return, following the policy π. If the transition probability is denoted by $p(s'|s,a)$, and the initial state distribution is $p(s_0)$, the probability for a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T)$ to occur can be written as

$$P_\pi(\tau) = p(s_0)\prod_{t=1}^{T}\pi(a_t|s_t)p(s_{t+1}|s_t,a_t).$$

The RL ojbective then takes the form

$$J = \mathrm{E}_{\tau\sim P_\pi}\left[G(\tau)|S_{t=0}=s_0\right], \quad G(\tau) = \sum_{t=1}^{T} r(s_t,a_t).$$

Policy gradient methods in RL approximate directly the policy $\pi \approx \pi_\theta$ using a variational ansatz, parametrized by the unknown parameters θ. The goal is then to find those optimal parameters θ, which optimize the RL objective $J(\theta)$. To define an update rule for θ, we may use gradient ascent. This requires us to evaluate the gradient of the RL objective w.r.t. the parameters θ:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathrm{E}_{\tau\sim P_\pi}\left[\sum_{t=1}^{T} r(s_t,a_t)|S_{t=0}=s_0\right] = \int \mathrm{d}\tau \nabla_\theta P_{\pi_\theta}(\tau)G(\tau).$$

In a model-free setting, we don't have access to the transition probabilities $p(s'|s,a)$ and this requires us to be able to estimate the gradients from samples. This can be accomplished by noticing that $\nabla_\theta P_{\pi_\theta} = P_{\pi_\theta}\nabla_\theta \log P_{\pi_\theta}$ (almost everywhere, i.e. up to a set of measure zero):

$$\nabla_\theta J(\theta) = \int \mathrm{d}\tau \nabla_\theta P_{\pi_\theta}(\tau)G(\tau) = \int \mathrm{d}\tau P_{\pi_\theta}(\tau)\nabla_\theta \log P_{\pi_\theta}(\tau)G(\tau) = \mathrm{E}_{\tau\sim P_\pi}\left[\nabla_\theta \log P_{\pi_\theta}(\tau)G(\tau)\right].$$

Since the initial state distribution and the transition proabilities are independent of θ, using the definition of P_{π_θ}, we see that $\nabla_\theta P_{\pi_\theta}(\tau) = \nabla_\theta \pi_\theta(\tau)$ where $\pi_\theta(\tau) = \prod_{t=1}^{T}\pi(a_t|s_t)$.

We can now use MC to estimate the gradients directly from a sample of trajectories $\{\tau_j\}_{j=1}^{N}$:

$$\nabla_\theta J(\theta) = \mathrm{E}_{\tau\sim P_\pi}\left[\nabla_\theta \log P_{\pi_\theta}(\tau)G(\tau)\right] \approx \frac{1}{N}\sum_{j=1}^{N}\nabla_\theta \log \pi_\theta(\tau_j)G(\tau_j).$$

In class, we discussed problems that arise due to large the variance of the gradient estimate. In particular, we showed that one can use causality and a baseline to reduce variance. The PG update then rakes the form

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{j=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t^j|s_t^j)\left[\sum_{t'=t}^{T} r(a_{t'}^j|s_{t'}^j)) - b\right].$$

The corresponding gradient ascent update rule reads as

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta),$$

for some step size (or learning rate) α.

B. Qubit Environment

Let us recall the qubit environment we defined in Notebook 2.

1. *Basic Definitions*

The state of a qubit $|\psi\rangle \in \mathbb{C}^2$ is modeled by a two-dimensional complex-valued vector with unit norm: $\langle\psi|\psi\rangle := \sqrt{|\psi_1|^2 + |\psi_2|^2} = 1$. Every qubit state is uniquely described by two angles $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi)$:

$$|\psi\rangle = \begin{pmatrix}\psi_1\\ \psi_2\end{pmatrix} = \mathrm{e}^{i\alpha}\begin{pmatrix}\cos\frac{\theta}{2}\\ \mathrm{e}^{i\varphi}\sin\frac{\theta}{2}\end{pmatrix} \tag{1}$$

The overall phase α of a single quantum state has no physical meaning. Thus, any qubit state can be pictured as an arrow on the unit sphere (called the Bloch sphere) with coordinates (θ, φ).

To operate on qubits, we use quantum gates. Quantum gates are represented as unitary transformations $U \in \mathrm{U}(2)$, where $\mathrm{U}(2)$ is the unitary group. Gates act on qubit states by matrix multiplication to transform an input state $|\psi\rangle$ to the output state $|\psi'\rangle$: $|\psi'\rangle = U|\psi\rangle$. For this problem, we consider four gates

$$U_0 = \mathbf{1}, \qquad U_x = \exp(-i\delta t\sigma^x/2), \qquad U_y = \exp(-i\delta t\sigma^y/2), \qquad U_z = \exp(-i\delta t\sigma^z/2), \tag{2}$$

where δt is a fixed time step, $\exp(\cdot)$ is the matrix exponential, $\mathbf{1}$ is the identity, and the Pauli matrices are defined as

$$\mathbf{1} = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \qquad \sigma^x = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, \qquad \sigma^y = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, \qquad \sigma^z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \tag{3}$$

To determine if a qubit, described by the state $|\psi\rangle$, is in a desired target state $|\psi_{\text{target}}\rangle$, we compute the fidelity

$$F = |\langle\psi_{\text{target}}|\psi\rangle|^2 = |(\psi_{\text{target}})_1^*\psi_1 + (\psi_{\text{target}})_2^*\psi_2|^2, \qquad F \in [0, 1] \tag{4}$$

where $*$ stands for complex conjugation. Physically, the fidelity corresponds to the angle between the arrows representing the qubit state on the Bloch sphere (we want to maximize the fidelity but minimize the angle between the states).

2. Constructing the Qubit Environment

Now, let us define an episodic RL environment, which contains the laws of physics that govern the dynamics of the qubit (i.e. the application of the gate operations to the qubit state). Our RL agent will later interact with this environment to learn how to control the qubit to bring it from an initial state to a prescribed target state.

We define the RL states $s = (\theta, \varphi)$ as an array containing the Bloch sphere angles of the quantum state. Each step within an episode, the agent can choose to apply one out of the actions, corresponding to the four gates $(\mathbf{1}, U_x, U_y, U_z)$. We use the instantaneous fidelity w.r.t. the target state as a reward: $r_t = F = |\langle\psi_*|\psi(t)\rangle|^2$:

state space: $\mathcal{S} = \{(\theta, \varphi)|\theta \in [0, \pi], \varphi \in [0, 2\pi)\}$. Unlike in Notebook 2, there are no terminal states here. Instead, we consider a fixed number of time steps, after which the episode terminates deterministically. The target state (i.e. the qubit state we want to prepare) is $|\psi_{\text{target}}\rangle = (1, 0)^t$: it has the Bloch sphere coordinates $s_{\text{target}} = (0, 0)$.

action space: $\mathcal{A} = \{\mathbf{1}, U_x, U_y, U_z\}$. Actions act on RL states as follows: 1. if the current state is $s = (\theta, \varphi)$, we first create the quantums state $|\psi(s)\rangle$; 2. we apply the gate U_a corresponding to action a to the quantum state, and obtain the new quantum state $|\psi(s')\rangle = U_a|\psi(s)\rangle$. 3. last, we compute the Bloch sphere coordinates which define the next state $s' = (\theta', \varphi')$, using the Bloch sphere parametrization for qubits given above. Note that all actions are allowed from every state.

reward space: $\mathcal{R} = [0, 1]$. We use the fidelity between the next state s' and the terminal state s_{target} as a reward at every episode step:

$$r(s, s', a) = F = |\langle\psi_{\text{target}}|U_a|\psi(s)\rangle|^2 = |\langle\psi_{\text{target}}|\psi(s')\rangle|^2$$

for all states $s, s' \in \mathcal{S}$ and actions $a \in \mathcal{A}$.

```
In [19]: import numpy as np
         import import_ipynb
         from Notebook_2_RL_environments import QubitEnv2 # import environment,␣
↪notebooks must be
         in same directory
```

```
In [2]: # set seed of rng (for reproducibility of the results)
        n_time_steps = 60
        seed=0
        np.random.seed(seed)

        # create environment class
        env=QubitEnv2(n_time_steps, seed=seed)
```

C. Policy Gradient Implementation

The implementation of PG follows similar steps as the MNIST problem from Notebook 7:

1. Define the a SoftMax model for the discrete policy π_θ.
2. Define the pseudo loss function to easily compute $\nabla_\theta J(\theta)$.
3. Define generalized gradient descent optimizer.
4. Define the PG training loop and train the policy.

1. Define the a SoftMax model for the discrete policy π_θ

Use JAX to construct a feed-forward fully-connected deep neural network with neuron acrchitecture $(M_s, 512, 256, |\mathcal{A}|)$, where there are 512 (256) neurons in the first (second) hidden layer, respectively, and M_s and $|\mathcal{A}|$ define the input and output sizes.

The input data into the neural network should have the shape `input_shape = (-1, n_time_steps, M_s)`, where `M_s` is the number of features/components in the RL state $s = (\theta, \varphi)$. The output data should have the shape `output_shape = (-1, n_time_steps, abs_A)`, where `abs_A`$= |\mathcal{A}|$. In this way, we can use the neural network to process simultaneously all time steps and MC samples, generated in a single training iteration.

Check explicitly the output shape and test that the network runs on some fake data (e.g. a small batch of vectors of ones with the appropriate shape).

```
In [3]: import jax.numpy as jnp # jax's numpy version with GPU support
        from jax import random # used to define a RNG key to control the random input in␣
↪JAX
        from jax.experimental import stax # neural network library
        from jax.experimental.stax import Dense, Relu, LogSoftmax # neural network layers

        # set key for the RNG (see JAX docs)
        rng = random.PRNGKey(seed)

        # define functions which initialize the parameters and evaluate the model
        initialize_params, predict = stax.serial(
                                                    ### fully connected DNN
                                                    Dense(512), # 512 hidden neurons
                                                    Relu,
                                                    Dense(256), # 256 hidden neurons
                                                    Relu,
                                                    Dense(env.n_actions), # 4 output␣
↪neurons
                                                    LogSoftmax # NB: computes the log-
        probability
                                                )

        # initialize the model parameters
        input_shape = (-1,env.n_time_steps,2) # -1: number of MC points, number of time␣
↪steps,
        size of state vector
        output_shape, inital_params = initialize_params(rng, input_shape) # fcc layer␣
↪28x28
        pixes in each image

        print('\noutput shape of the policy network is {}.\n'.format(output_shape))


        # test network
        states=np.ones((3,env.n_time_steps,2), dtype=np.float32)

        predictions = predict(inital_params, states)
        # check the output shape
        print(predictions.shape)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)



output shape of the policy network is (-1, 60, 4).

(3, 60, 4)
```

2. Define the pseudo loss function to easily compute $\nabla_\theta J(\theta)$

In class we can defined a scalar pseudoloss function, whose gradients give $\nabla_\theta J(\theta)$. Note that this pseudoloss does ***NOT*** correspond to the RL objective $J(\theta)$: the difference stems from the fact that the two operations of taking the derivative and performing the MC approximation are not interchangeable (do you see why?).

$$J_{\text{pseudo}}(\theta) = \frac{1}{N}\sum_{j=1}^{N}\sum_{t=1}^{T}\log \pi_\theta(a_t^j|s_t^j)\left[\sum_{t'=t}^{T} r(a_{t'}^j|s_{t'}^j)) - b_t\right], \qquad b_t = \frac{1}{N}\sum_{j=1}^{N} G_t(\tau_j).$$

The baseline is a sample average of the reward-to-go (return) from time step t onwards: $G_t(\tau_j) = \sum_{t'=t}^{N} r(s_{t'}^j, s_{t'}^j)$.

Because we will be doing gradient **a**scent, do **NOT** forget to add an extra minus sign to the output ot the pseudoloss (or else your agent will end up minimizing the return).

Below, we also add an L2 regularizer to the pseudoloss function to prevent overfitting.

```
In [4]: ### define loss and accuracy functions

        from jax import grad
        from jax.tree_util import tree_flatten # jax params are stored as nested tuples; use
        this to manipulate tuples


        def l2_regularizer(params, lmbda):
            """
            Define l2 regularizer: $\lambda \ sum_j ||theta_j||^2 $ for every parameter in the
        model $\theta_j$

            """
            return lmbda*jnp.sum(jnp.array([jnp.sum(jnp.abs(theta)**2) for theta in
        tree_flatten(params)[0] ]))


        def pseudo_loss(params, trajectory_batch):
            """
            Define the pseudo loss function for policy gradient.

            params: object(jax pytree):
                parameters of the deep policy network.
            trajectory_batch: tuple (states, actions, returns) containing the RL states, actions
        and returns (not the rewards!):
                states: np.array of size (N_MC, env.n_time_steps,2)
                actions: np.array of size (N_MC, env.n_time_steps)
                returns: np.array of size (N_MC, env.n_time_steps)

            Returns:
                -J_{pseudo}(\theta)

            """
            # extract data from the batch
            states, actions, returns = trajectory_batch
            # compute policy predictions
            preds = predict(params, states)
```

```
    # combute the baseline
    baseline = jnp.mean(rewards, axis=0)
    # select those values of the policy along the action trajectory
    preds_select = jnp.take_along_axis(preds, jnp.expand_dims(actions, axis=2),
axis=2).squeeze()
    # return negative pseudo loss function (want to maximize reward with gradient
DEscent)
    return -jnp.mean(jnp.sum(preds_select * (returns - baseline) )) +
l2_regularizer(params, 0.001)
```

3. Define generalized gradient descent optimizer

Define the optimizer and the `update` function which computes the gradient o the pseudo-loss function and performs the update.

We use the Adam optimizer here with `step_size = 0.001` and the rest of the parameters have default values.

```
In [5]: ### define generalized gradient descent optimizer and a function to update model
        parameters

        from jax.experimental import optimizers # gradient descent optimizers
        from jax import jit

        step_size = 0.001 # step size or learning rate

        # compute optimizer functions
        opt_init, opt_update, get_params = optimizers.adam(step_size)


        # define function which updates the parameters using the change computed by the
        optimizer
        @jit # Just In Time compilation speeds up the code; requires to use jnp␣
↪everywhere;
        remove when debugging
        def update(i, opt_state, batch):
            """
            i: int,
                counter to count how many update steps we have performed
            opt_state: object,
                the state of the optimizer
            batch: np.array
                batch containing the data used to update the model

            Returns:
            opt_state: object,
                the new state of the optimizer

            """
            # get current parameters of the model
            current_params = get_params(opt_state)
            # compute gradients
            grad_params = grad(pseudo_loss)(current_params, batch)
            # use the optimizer to perform the update using opt_update
            return opt_update(i, grad_params, opt_state)
```

4. Define the PG training loop and train the policy

Finally, we implement the REINFORCE algorithm for policy gradient. Follow the steps below

1. Preallocate variables

   - Define the number of episodes `N_episodes`, and the batch size `N_MC`.
   - Preallocate arrays for the current `state`, and the `states`, `actions`, `returns` triple which defines the trajectory batch.
   - Preallocate arrays to compute the `mean_final_reward`, `std_final_reward`, `min_final_reward`, and , `max_final_reward`.

2. Initialize the optimizer using the `opt_init` function.

3. Loop over the episodes; for every episode:

   3.1 get the current Network parameters

   3.2 loop to collect MC samples

   `3.2.1 reset the \`env\` and roll out the policy until the episode is over; collect the trajectory da`
   `3.2.2 compute the returns (rewards to go)`

   3.3 compile the PG data into a trajctory batch

   3.4 use the `update` function to update the network parameters

   3.5 print instantaneous performance

In [13]:
```
### Train model

import time

# define number of training episodes
N_episodes = 201
N_MC = 64 #128


# preallocate data using arrays initialized with zeros

state=np.zeros((2,), dtype=np.float32)

states = np.zeros((N_MC, env.n_time_steps,2), dtype=np.float32)
actions = np.zeros((N_MC, env.n_time_steps), dtype=np.int)
returns = np.zeros((N_MC, env.n_time_steps), dtype=np.float32)

# mean reward at the end of the episode
mean_final_reward = np.zeros(N_episodes, dtype=np.float32)
# standard deviation of the reward at the end of the episode
std_final_reward = np.zeros_like(mean_final_reward)
# batch minimum at the end of the episode
min_final_reward = np.zeros_like(mean_final_reward)
# batch maximum at the end of the episode
max_final_reward = np.zeros_like(mean_final_reward)


print("\nStarting training...\n")

# set the initial model parameters in the optimizer
```

```
        opt_state = opt_init(inital_params)

        # loop over the number of training episodes
        for episode in range(N_episodes):

            ### record time
            start_time = time.time()

            # get current policy  network params
            current_params = get_params(opt_state)

            # MC sample
            for j in range(N_MC):

                # reset environment to a random initial state
                #env.reset(random=False) # fixed initial state
                env.reset(random=True) # Haar-random initial state (i.e. uniformly␣
↪sampled on
        the sphere)

                # zero rewards array (auxiliary array to store the rewards, and help␣
↪compute the
        returns)
                rewards = np.zeros((env.n_time_steps, ), dtype=np.float32)

                # loop over steps in an episode
                for time_step in range(env.n_time_steps):

                    # select state
                    state[:] = env.state[:]
                    states[j,time_step,:] = state

                    # select an action according to current policy
                    pi_s = np.exp( predict(current_params, state) )
                    action = np.random.choice(env.actions, p = pi_s)
                    actions[j,time_step] = action

                    # take an environment step
                    state[:], reward, _ = env.step(action)

                    # store reward
                    rewards[time_step] = reward


                # compute reward-to-go
                returns[j,:] = jnp.cumsum(rewards[::-1])[::-1]



            # define batch of data
            trajectory_batch = (states, actions, returns)

            # update model
            opt_state = update(episode, opt_state, trajectory_batch)

            ### record time needed for a single epoch
```

```
            episode_time = time.time() - start_time

            # check performance
            mean_final_reward[episode]=jnp.mean(returns[:,-1])
            std_final_reward[episode] =jnp.std(returns[:,-1])
            min_final_reward[episode], max_final_reward[episode] = np.min(returns[:
↪,-1]),
        np.max(returns[:,-1])


            # print results every 10 epochs
            #if episode % 5 == 0:
            print("episode {} in {:0.2f} sec".format(episode, episode_time))
            print("mean reward: {:0.4f}".format(mean_final_reward[episode]) )
            print("return standard deviation: {:0.4f}".
↪format(std_final_reward[episode]) )
            print("min return: {:0.4f}; max return: {:0.4f}\n".
↪format(min_final_reward[episode],
        max_final_reward[episode]) )
```

```
Starting training...

episode 0 in 8.90 sec
mean reward: 0.4562
return standard deviation: 0.2593
min return: 0.0051; max return: 0.9465

episode 1 in 9.14 sec
mean reward: 0.4051
return standard deviation: 0.2847
min return: 0.0071; max return: 0.9868

episode 2 in 8.67 sec
mean reward: 0.4554
return standard deviation: 0.2654
min return: 0.0070; max return: 0.9843

episode 3 in 8.77 sec
mean reward: 0.5347
return standard deviation: 0.2965
min return: 0.0125; max return: 0.9942

episode 4 in 9.10 sec
mean reward: 0.4993
return standard deviation: 0.2915
min return: 0.0036; max return: 0.9948

episode 5 in 9.29 sec
mean reward: 0.5093
return standard deviation: 0.2557
min return: 0.0265; max return: 0.9842

episode 6 in 8.86 sec
mean reward: 0.5573
return standard deviation: 0.2985
```

```
min return: 0.0192; max return: 0.9903

episode 7 in 9.31 sec
mean reward: 0.4865
return standard deviation: 0.2842
min return: 0.0125; max return: 0.9935

episode 8 in 9.03 sec
mean reward: 0.5378
return standard deviation: 0.2851
min return: 0.0060; max return: 0.9813

episode 9 in 9.46 sec
mean reward: 0.5306
return standard deviation: 0.3091
min return: 0.0541; max return: 0.9978

episode 10 in 9.03 sec
mean reward: 0.5294
return standard deviation: 0.2730
min return: 0.0009; max return: 0.9724

episode 11 in 9.41 sec
mean reward: 0.5583
return standard deviation: 0.2846
min return: 0.0234; max return: 0.9857

episode 12 in 8.96 sec
mean reward: 0.6407
return standard deviation: 0.2760
min return: 0.0545; max return: 0.9864

episode 13 in 9.20 sec
mean reward: 0.6151
return standard deviation: 0.2683
min return: 0.0156; max return: 0.9967

episode 14 in 9.36 sec
mean reward: 0.5790
return standard deviation: 0.2913
min return: 0.0216; max return: 0.9975

episode 15 in 8.87 sec
mean reward: 0.5868
return standard deviation: 0.2821
min return: 0.0052; max return: 0.9910

episode 16 in 8.99 sec
mean reward: 0.7151
return standard deviation: 0.2415
min return: 0.1630; max return: 0.9982

episode 17 in 9.41 sec
mean reward: 0.6150
return standard deviation: 0.2886
min return: 0.0168; max return: 0.9989
```

```
episode 18 in 9.68 sec
mean reward: 0.6341
return standard deviation: 0.2463
min return: 0.0259; max return: 0.9963

episode 19 in 9.30 sec
mean reward: 0.6562
return standard deviation: 0.2485
min return: 0.0988; max return: 0.9996

episode 20 in 9.23 sec
mean reward: 0.6304
return standard deviation: 0.2602
min return: 0.0964; max return: 0.9991

episode 21 in 9.55 sec
mean reward: 0.7116
return standard deviation: 0.2353
min return: 0.0394; max return: 0.9997

episode 22 in 9.52 sec
mean reward: 0.6820
return standard deviation: 0.2720
min return: 0.0979; max return: 0.9998

episode 23 in 9.68 sec
mean reward: 0.6908
return standard deviation: 0.2357
min return: 0.1425; max return: 0.9970

episode 24 in 11.33 sec
mean reward: 0.7206
return standard deviation: 0.2125
min return: 0.0277; max return: 0.9913

episode 25 in 9.63 sec
mean reward: 0.7112
return standard deviation: 0.2254
min return: 0.0384; max return: 0.9940

episode 26 in 9.24 sec
mean reward: 0.7421
return standard deviation: 0.2210
min return: 0.0871; max return: 0.9838

episode 27 in 9.04 sec
mean reward: 0.7187
return standard deviation: 0.2057
min return: 0.2195; max return: 0.9878

episode 28 in 9.19 sec
mean reward: 0.7141
return standard deviation: 0.2373
min return: 0.2032; max return: 0.9948
```

```
episode 29 in 8.96 sec
mean reward: 0.7788
return standard deviation: 0.1877
min return: 0.2063; max return: 0.9973

episode 30 in 9.16 sec
mean reward: 0.7236
return standard deviation: 0.2281
min return: 0.0355; max return: 0.9875

episode 31 in 9.15 sec
mean reward: 0.7176
return standard deviation: 0.1989
min return: 0.2246; max return: 0.9769

episode 32 in 8.86 sec
mean reward: 0.7703
return standard deviation: 0.1802
min return: 0.3438; max return: 0.9994

episode 33 in 9.12 sec
mean reward: 0.7312
return standard deviation: 0.2225
min return: 0.1938; max return: 0.9923

episode 34 in 8.97 sec
mean reward: 0.7701
return standard deviation: 0.1705
min return: 0.3199; max return: 0.9985

episode 35 in 9.50 sec
mean reward: 0.7752
return standard deviation: 0.1667
min return: 0.3696; max return: 0.9938

episode 36 in 9.07 sec
mean reward: 0.7569
return standard deviation: 0.1618
min return: 0.2729; max return: 0.9893

episode 37 in 9.03 sec
mean reward: 0.7762
return standard deviation: 0.1813
min return: 0.2023; max return: 0.9927

episode 38 in 9.56 sec
mean reward: 0.7589
return standard deviation: 0.1825
min return: 0.1899; max return: 0.9992

episode 39 in 8.75 sec
mean reward: 0.8207
return standard deviation: 0.1517
min return: 0.1449; max return: 0.9992

episode 40 in 8.80 sec
```

```
mean reward: 0.7550
return standard deviation: 0.1653
min return: 0.2631; max return: 0.9950

episode 41 in 8.78 sec
mean reward: 0.7910
return standard deviation: 0.1739
min return: 0.1389; max return: 0.9999

episode 42 in 8.79 sec
mean reward: 0.7976
return standard deviation: 0.1599
min return: 0.2042; max return: 0.9865

episode 43 in 8.83 sec
mean reward: 0.8185
return standard deviation: 0.1551
min return: 0.2409; max return: 0.9972

episode 44 in 8.75 sec
mean reward: 0.7915
return standard deviation: 0.2075
min return: 0.1613; max return: 0.9962

episode 45 in 8.79 sec
mean reward: 0.7909
return standard deviation: 0.2015
min return: 0.1553; max return: 0.9999

episode 46 in 9.26 sec
mean reward: 0.8140
return standard deviation: 0.1590
min return: 0.2167; max return: 0.9990

episode 47 in 9.72 sec
mean reward: 0.8372
return standard deviation: 0.1374
min return: 0.2924; max return: 0.9997

episode 48 in 9.65 sec
mean reward: 0.8210
return standard deviation: 0.1420
min return: 0.2590; max return: 0.9910

episode 49 in 9.23 sec
mean reward: 0.8402
return standard deviation: 0.1232
min return: 0.5231; max return: 0.9986

episode 50 in 9.14 sec
mean reward: 0.8365
return standard deviation: 0.1377
min return: 0.4923; max return: 0.9972

episode 51 in 8.36 sec
mean reward: 0.8628
```

```
return standard deviation: 0.1150
min return: 0.5252; max return: 0.9988

episode 52 in 8.26 sec
mean reward: 0.8809
return standard deviation: 0.0942
min return: 0.5694; max return: 0.9916

episode 53 in 8.20 sec
mean reward: 0.8907
return standard deviation: 0.0919
min return: 0.5806; max return: 0.9984

episode 54 in 8.24 sec
mean reward: 0.8687
return standard deviation: 0.1057
min return: 0.5968; max return: 0.9998

episode 55 in 8.21 sec
mean reward: 0.8456
return standard deviation: 0.1166
min return: 0.5156; max return: 0.9999

episode 56 in 8.20 sec
mean reward: 0.8457
return standard deviation: 0.1084
min return: 0.5797; max return: 0.9954

episode 57 in 8.18 sec
mean reward: 0.8819
return standard deviation: 0.0976
min return: 0.5934; max return: 0.9997

episode 58 in 8.20 sec
mean reward: 0.8878
return standard deviation: 0.0924
min return: 0.6578; max return: 0.9996

episode 59 in 8.20 sec
mean reward: 0.8851
return standard deviation: 0.1024
min return: 0.6078; max return: 0.9981

episode 60 in 8.32 sec
mean reward: 0.8834
return standard deviation: 0.1038
min return: 0.4849; max return: 0.9997

episode 61 in 8.21 sec
mean reward: 0.8864
return standard deviation: 0.0946
min return: 0.5364; max return: 0.9995

episode 62 in 8.19 sec
mean reward: 0.8756
return standard deviation: 0.0889
```

```
min return: 0.6705; max return: 0.9987

episode 63 in 8.20 sec
mean reward: 0.8888
return standard deviation: 0.1026
min return: 0.6283; max return: 0.9999

episode 64 in 8.20 sec
mean reward: 0.9136
return standard deviation: 0.0786
min return: 0.5912; max return: 0.9992

episode 65 in 8.36 sec
mean reward: 0.8897
return standard deviation: 0.0774
min return: 0.6997; max return: 0.9962

episode 66 in 8.22 sec
mean reward: 0.9129
return standard deviation: 0.0653
min return: 0.7374; max return: 0.9991

episode 67 in 8.24 sec
mean reward: 0.9091
return standard deviation: 0.0766
min return: 0.5948; max return: 0.9991

episode 68 in 8.21 sec
mean reward: 0.9218
return standard deviation: 0.0653
min return: 0.6746; max return: 0.9991

episode 69 in 8.22 sec
mean reward: 0.9284
return standard deviation: 0.0715
min return: 0.6303; max return: 0.9999

episode 70 in 8.22 sec
mean reward: 0.9421
return standard deviation: 0.0653
min return: 0.6942; max return: 0.9998

episode 71 in 8.25 sec
mean reward: 0.9335
return standard deviation: 0.0581
min return: 0.7803; max return: 0.9989

episode 72 in 8.29 sec
mean reward: 0.9331
return standard deviation: 0.0591
min return: 0.7460; max return: 0.9996

episode 73 in 8.22 sec
mean reward: 0.9406
return standard deviation: 0.0512
min return: 0.7864; max return: 0.9997
```

```
episode 74 in 8.28 sec
mean reward: 0.9360
return standard deviation: 0.0658
min return: 0.6442; max return: 0.9969

episode 75 in 8.21 sec
mean reward: 0.9306
return standard deviation: 0.1241
min return: 0.0649; max return: 0.9997

episode 76 in 8.20 sec
mean reward: 0.9414
return standard deviation: 0.0589
min return: 0.7616; max return: 0.9992

episode 77 in 8.17 sec
mean reward: 0.9313
return standard deviation: 0.0571
min return: 0.7284; max return: 0.9997

episode 78 in 8.16 sec
mean reward: 0.9351
return standard deviation: 0.0800
min return: 0.4576; max return: 0.9999

episode 79 in 8.20 sec
mean reward: 0.9483
return standard deviation: 0.0439
min return: 0.8036; max return: 0.9996

episode 80 in 8.20 sec
mean reward: 0.9413
return standard deviation: 0.0550
min return: 0.6899; max return: 0.9982

episode 81 in 8.19 sec
mean reward: 0.9439
return standard deviation: 0.0487
min return: 0.7783; max return: 0.9989

episode 82 in 8.17 sec
mean reward: 0.9374
return standard deviation: 0.0513
min return: 0.8131; max return: 0.9997

episode 83 in 8.16 sec
mean reward: 0.9390
return standard deviation: 0.0580
min return: 0.7053; max return: 0.9979

episode 84 in 8.15 sec
mean reward: 0.9378
return standard deviation: 0.0516
min return: 0.7748; max return: 0.9993
```

```
episode 85 in 8.20 sec
mean reward: 0.9363
return standard deviation: 0.0452
min return: 0.7753; max return: 0.9994

episode 86 in 8.13 sec
mean reward: 0.9408
return standard deviation: 0.0506
min return: 0.7392; max return: 0.9998

episode 87 in 8.17 sec
mean reward: 0.9413
return standard deviation: 0.0510
min return: 0.8213; max return: 1.0000

episode 88 in 8.14 sec
mean reward: 0.9276
return standard deviation: 0.0665
min return: 0.7419; max return: 0.9997

episode 89 in 8.17 sec
mean reward: 0.9267
return standard deviation: 0.0626
min return: 0.7629; max return: 0.9995

episode 90 in 8.16 sec
mean reward: 0.9204
return standard deviation: 0.1075
min return: 0.2549; max return: 0.9995

episode 91 in 8.29 sec
mean reward: 0.9205
return standard deviation: 0.0975
min return: 0.3478; max return: 0.9961

episode 92 in 8.14 sec
mean reward: 0.9312
return standard deviation: 0.0591
min return: 0.7153; max return: 0.9989

episode 93 in 8.14 sec
mean reward: 0.9343
return standard deviation: 0.0664
min return: 0.7087; max return: 0.9991

episode 94 in 8.22 sec
mean reward: 0.9189
return standard deviation: 0.1261
min return: 0.2715; max return: 0.9995

episode 95 in 8.18 sec
mean reward: 0.8925
return standard deviation: 0.1021
min return: 0.5659; max return: 0.9997

episode 96 in 8.13 sec
```

```
mean reward: 0.8824
return standard deviation: 0.1471
min return: 0.3693; max return: 0.9996

episode 97 in 8.18 sec
mean reward: 0.8992
return standard deviation: 0.1243
min return: 0.2676; max return: 0.9989

episode 98 in 8.14 sec
mean reward: 0.8647
return standard deviation: 0.1670
min return: 0.2451; max return: 0.9995

episode 99 in 8.14 sec
mean reward: 0.9135
return standard deviation: 0.1198
min return: 0.4102; max return: 0.9997

episode 100 in 8.15 sec
mean reward: 0.8907
return standard deviation: 0.1430
min return: 0.3313; max return: 0.9998

episode 101 in 8.16 sec
mean reward: 0.9076
return standard deviation: 0.1043
min return: 0.4901; max return: 0.9993

episode 102 in 8.15 sec
mean reward: 0.9323
return standard deviation: 0.0698
min return: 0.6613; max return: 0.9984

episode 103 in 8.17 sec
mean reward: 0.9249
return standard deviation: 0.0901
min return: 0.3862; max return: 0.9999

episode 104 in 8.13 sec
mean reward: 0.9185
return standard deviation: 0.0726
min return: 0.7143; max return: 0.9975

episode 105 in 8.14 sec
mean reward: 0.9379
return standard deviation: 0.1014
min return: 0.2571; max return: 0.9998

episode 106 in 8.16 sec
mean reward: 0.9465
return standard deviation: 0.0448
min return: 0.8116; max return: 0.9991

episode 107 in 8.14 sec
mean reward: 0.9408
```

```
return standard deviation: 0.0590
min return: 0.7569; max return: 0.9990

episode 108 in 8.17 sec
mean reward: 0.9483
return standard deviation: 0.0427
min return: 0.8300; max return: 1.0000

episode 109 in 8.21 sec
mean reward: 0.9419
return standard deviation: 0.0681
min return: 0.4777; max return: 0.9979

episode 110 in 8.19 sec
mean reward: 0.9614
return standard deviation: 0.0410
min return: 0.8126; max return: 0.9999

episode 111 in 8.16 sec
mean reward: 0.9611
return standard deviation: 0.0412
min return: 0.8131; max return: 0.9991

episode 112 in 8.18 sec
mean reward: 0.9590
return standard deviation: 0.0425
min return: 0.8133; max return: 0.9999

episode 113 in 8.19 sec
mean reward: 0.9672
return standard deviation: 0.0345
min return: 0.8631; max return: 0.9998

episode 114 in 8.17 sec
mean reward: 0.9613
return standard deviation: 0.0362
min return: 0.8465; max return: 0.9993

episode 115 in 8.16 sec
mean reward: 0.9525
return standard deviation: 0.0498
min return: 0.8003; max return: 0.9998

episode 116 in 8.17 sec
mean reward: 0.9564
return standard deviation: 0.0440
min return: 0.7952; max return: 0.9996

episode 117 in 8.17 sec
mean reward: 0.9609
return standard deviation: 0.0384
min return: 0.8364; max return: 0.9998

episode 118 in 8.15 sec
mean reward: 0.9631
return standard deviation: 0.0474
```

```
min return: 0.7872; max return: 0.9998

episode 119 in 8.15 sec
mean reward: 0.9661
return standard deviation: 0.0302
min return: 0.8693; max return: 0.9993

episode 120 in 8.23 sec
mean reward: 0.9635
return standard deviation: 0.0347
min return: 0.8605; max return: 0.9999

episode 121 in 8.16 sec
mean reward: 0.9666
return standard deviation: 0.0334
min return: 0.8362; max return: 0.9990

episode 122 in 8.16 sec
mean reward: 0.9650
return standard deviation: 0.0318
min return: 0.8783; max return: 0.9997

episode 123 in 8.17 sec
mean reward: 0.9689
return standard deviation: 0.0277
min return: 0.8868; max return: 0.9998

episode 124 in 8.21 sec
mean reward: 0.9685
return standard deviation: 0.0282
min return: 0.8813; max return: 0.9999

episode 125 in 8.19 sec
mean reward: 0.9623
return standard deviation: 0.0416
min return: 0.8181; max return: 0.9999

episode 126 in 8.16 sec
mean reward: 0.9767
return standard deviation: 0.0244
min return: 0.8786; max return: 0.9985

episode 127 in 8.18 sec
mean reward: 0.9755
return standard deviation: 0.0247
min return: 0.8597; max return: 0.9997

episode 128 in 8.15 sec
mean reward: 0.9712
return standard deviation: 0.0274
min return: 0.8736; max return: 0.9999

episode 129 in 8.16 sec
mean reward: 0.9723
return standard deviation: 0.0294
min return: 0.8423; max return: 0.9996
```

```
episode 130 in 8.13 sec
mean reward: 0.9757
return standard deviation: 0.0273
min return: 0.8886; max return: 0.9996

episode 131 in 8.17 sec
mean reward: 0.9706
return standard deviation: 0.0350
min return: 0.8081; max return: 1.0000

episode 132 in 8.14 sec
mean reward: 0.9736
return standard deviation: 0.0254
min return: 0.8619; max return: 0.9998

episode 133 in 8.17 sec
mean reward: 0.9690
return standard deviation: 0.0399
min return: 0.8256; max return: 1.0000

episode 134 in 8.17 sec
mean reward: 0.9744
return standard deviation: 0.0267
min return: 0.8564; max return: 0.9996

episode 135 in 8.13 sec
mean reward: 0.9696
return standard deviation: 0.0338
min return: 0.8445; max return: 0.9999

episode 136 in 8.16 sec
mean reward: 0.9718
return standard deviation: 0.0385
min return: 0.7722; max return: 0.9997

episode 137 in 8.18 sec
mean reward: 0.9750
return standard deviation: 0.0268
min return: 0.8717; max return: 0.9993

episode 138 in 8.28 sec
mean reward: 0.9754
return standard deviation: 0.0258
min return: 0.8755; max return: 0.9997

episode 139 in 8.17 sec
mean reward: 0.9771
return standard deviation: 0.0261
min return: 0.8484; max return: 0.9997

episode 140 in 8.16 sec
mean reward: 0.9729
return standard deviation: 0.0268
min return: 0.8683; max return: 1.0000
```

```
episode 141 in 8.16 sec
mean reward: 0.9796
return standard deviation: 0.0192
min return: 0.9099; max return: 0.9999

episode 142 in 8.15 sec
mean reward: 0.9816
return standard deviation: 0.0174
min return: 0.9369; max return: 1.0000

episode 143 in 8.19 sec
mean reward: 0.9739
return standard deviation: 0.0353
min return: 0.7853; max return: 1.0000

episode 144 in 8.15 sec
mean reward: 0.9788
return standard deviation: 0.0255
min return: 0.8590; max return: 0.9996

episode 145 in 8.21 sec
mean reward: 0.9688
return standard deviation: 0.0327
min return: 0.8534; max return: 0.9994

episode 146 in 8.15 sec
mean reward: 0.9747
return standard deviation: 0.0378
min return: 0.8046; max return: 0.9998

episode 147 in 8.17 sec
mean reward: 0.9778
return standard deviation: 0.0311
min return: 0.8614; max return: 0.9995

episode 148 in 8.26 sec
mean reward: 0.9813
return standard deviation: 0.0194
min return: 0.9101; max return: 0.9996

episode 149 in 8.15 sec
mean reward: 0.9755
return standard deviation: 0.0321
min return: 0.8205; max return: 0.9999

episode 150 in 8.13 sec
mean reward: 0.9775
return standard deviation: 0.0290
min return: 0.8489; max return: 0.9994

episode 151 in 8.17 sec
mean reward: 0.9790
return standard deviation: 0.0251
min return: 0.8608; max return: 0.9998

episode 152 in 8.12 sec
```

```
mean reward: 0.9820
return standard deviation: 0.0201
min return: 0.9171; max return: 1.0000

episode 153 in 8.25 sec
mean reward: 0.9851
return standard deviation: 0.0158
min return: 0.9317; max return: 0.9999

episode 154 in 8.18 sec
mean reward: 0.9770
return standard deviation: 0.0280
min return: 0.8563; max return: 0.9996

episode 155 in 8.16 sec
mean reward: 0.9808
return standard deviation: 0.0204
min return: 0.8870; max return: 0.9998

episode 156 in 8.15 sec
mean reward: 0.9844
return standard deviation: 0.0164
min return: 0.9389; max return: 1.0000

episode 157 in 8.13 sec
mean reward: 0.9798
return standard deviation: 0.0264
min return: 0.8693; max return: 0.9999

episode 158 in 8.16 sec
mean reward: 0.9783
return standard deviation: 0.0234
min return: 0.8940; max return: 0.9998

episode 159 in 8.14 sec
mean reward: 0.9814
return standard deviation: 0.0238
min return: 0.8897; max return: 0.9999

episode 160 in 8.17 sec
mean reward: 0.9800
return standard deviation: 0.0224
min return: 0.8786; max return: 0.9991

episode 161 in 8.14 sec
mean reward: 0.9790
return standard deviation: 0.0211
min return: 0.9156; max return: 0.9999

episode 162 in 8.15 sec
mean reward: 0.9778
return standard deviation: 0.0257
min return: 0.8528; max return: 0.9996

episode 163 in 8.17 sec
mean reward: 0.9633
```

```
return standard deviation: 0.0779
min return: 0.4841; max return: 0.9996

episode 164 in 8.14 sec
mean reward: 0.9678
return standard deviation: 0.0782
min return: 0.3886; max return: 1.0000

episode 165 in 8.14 sec
mean reward: 0.9769
return standard deviation: 0.0449
min return: 0.6764; max return: 0.9998

episode 166 in 8.12 sec
mean reward: 0.9745
return standard deviation: 0.0553
min return: 0.5742; max return: 1.0000

episode 167 in 8.22 sec
mean reward: 0.9651
return standard deviation: 0.0811
min return: 0.3754; max return: 0.9999

episode 168 in 8.15 sec
mean reward: 0.9653
return standard deviation: 0.0790
min return: 0.4658; max return: 0.9999

episode 169 in 8.17 sec
mean reward: 0.9793
return standard deviation: 0.0238
min return: 0.9017; max return: 0.9998

episode 170 in 8.15 sec
mean reward: 0.9607
return standard deviation: 0.0882
min return: 0.4643; max return: 0.9996

episode 171 in 8.16 sec
mean reward: 0.9719
return standard deviation: 0.0503
min return: 0.6354; max return: 0.9998

episode 172 in 8.15 sec
mean reward: 0.9753
return standard deviation: 0.0244
min return: 0.8730; max return: 0.9992

episode 173 in 8.14 sec
mean reward: 0.9751
return standard deviation: 0.0289
min return: 0.8815; max return: 0.9995

episode 174 in 8.14 sec
mean reward: 0.9746
return standard deviation: 0.0725
```

```
min return: 0.4192; max return: 0.9997

episode 175 in 8.15 sec
mean reward: 0.9752
return standard deviation: 0.0336
min return: 0.8464; max return: 0.9996

episode 176 in 8.12 sec
mean reward: 0.9769
return standard deviation: 0.0291
min return: 0.8393; max return: 0.9997

episode 177 in 8.14 sec
mean reward: 0.9810
return standard deviation: 0.0224
min return: 0.8959; max return: 1.0000

episode 178 in 8.15 sec
mean reward: 0.9842
return standard deviation: 0.0177
min return: 0.9100; max return: 0.9998

episode 179 in 8.15 sec
mean reward: 0.9833
return standard deviation: 0.0210
min return: 0.9133; max return: 0.9999

episode 180 in 8.16 sec
mean reward: 0.9871
return standard deviation: 0.0184
min return: 0.8865; max return: 0.9997

episode 181 in 8.16 sec
mean reward: 0.9891
return standard deviation: 0.0114
min return: 0.9421; max return: 1.0000

episode 182 in 8.25 sec
mean reward: 0.9845
return standard deviation: 0.0163
min return: 0.9198; max return: 0.9998

episode 183 in 8.20 sec
mean reward: 0.9901
return standard deviation: 0.0115
min return: 0.9499; max return: 1.0000

episode 184 in 8.15 sec
mean reward: 0.9832
return standard deviation: 0.0188
min return: 0.9049; max return: 0.9997

episode 185 in 8.17 sec
mean reward: 0.9858
return standard deviation: 0.0168
min return: 0.9231; max return: 0.9999
```

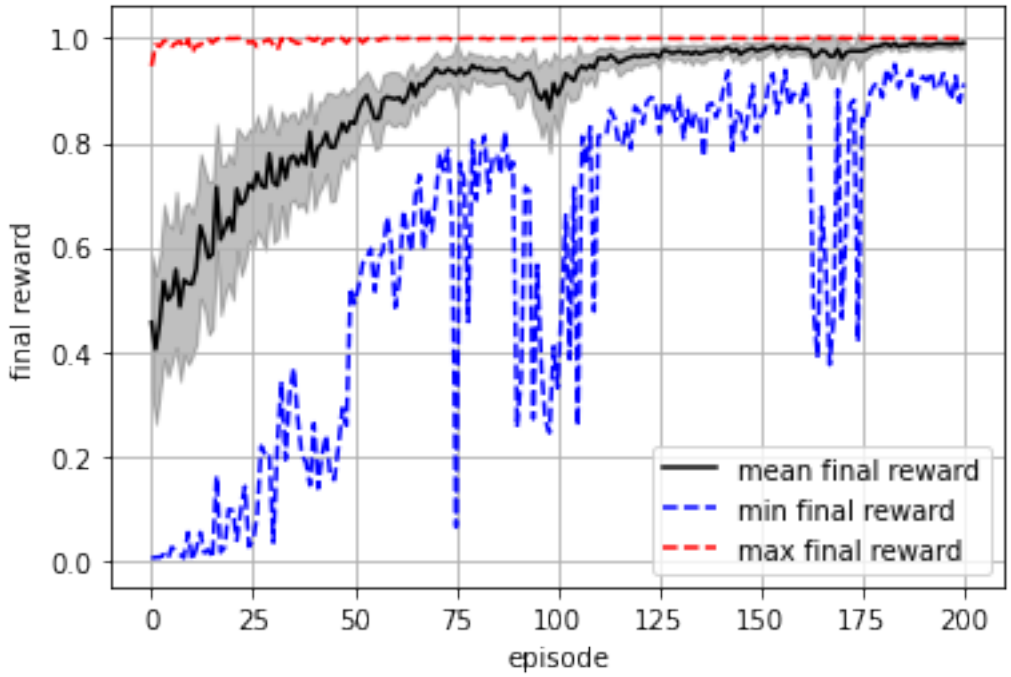
```
episode 186 in 8.16 sec
mean reward: 0.9849
return standard deviation: 0.0179
min return: 0.9244; max return: 0.9995

episode 187 in 8.16 sec
mean reward: 0.9906
return standard deviation: 0.0108
min return: 0.9386; max return: 0.9996

episode 188 in 8.15 sec
mean reward: 0.9878
return standard deviation: 0.0175
min return: 0.8778; max return: 0.9998

episode 189 in 8.16 sec
mean reward: 0.9866
return standard deviation: 0.0163
min return: 0.9165; max return: 0.9997

episode 190 in 8.15 sec
mean reward: 0.9884
return standard deviation: 0.0148
min return: 0.9264; max return: 0.9999

episode 191 in 8.16 sec
mean reward: 0.9861
return standard deviation: 0.0172
min return: 0.9188; max return: 0.9999

episode 192 in 8.15 sec
mean reward: 0.9863
return standard deviation: 0.0138
min return: 0.9350; max return: 0.9998

episode 193 in 8.14 sec
mean reward: 0.9887
return standard deviation: 0.0153
min return: 0.9008; max return: 1.0000

episode 194 in 8.15 sec
mean reward: 0.9891
return standard deviation: 0.0157
min return: 0.9090; max return: 0.9998

episode 195 in 8.16 sec
mean reward: 0.9887
return standard deviation: 0.0140
min return: 0.9323; max return: 1.0000

episode 196 in 8.17 sec
mean reward: 0.9880
return standard deviation: 0.0172
min return: 0.9048; max return: 0.9999
```

```
episode 197 in 8.24 sec
mean reward: 0.9873
return standard deviation: 0.0189
min return: 0.8821; max return: 0.9997

episode 198 in 8.17 sec
mean reward: 0.9903
return standard deviation: 0.0118
min return: 0.9272; max return: 0.9997

episode 199 in 8.17 sec
mean reward: 0.9881
return standard deviation: 0.0186
min return: 0.8778; max return: 0.9999

episode 200 in 8.15 sec
mean reward: 0.9904
return standard deviation: 0.0134
min return: 0.9127; max return: 0.9998
```

D. Plot the training curves

Plot the mean final reward at each episode, and its variance. What do you observe?

```
In [18]: import matplotlib
         from matplotlib import pyplot as plt
         # static plots
         %matplotlib inline

         ### plot and examine learning curves

         episodes=list(range(N_episodes))

         plt.plot(episodes, mean_final_reward, '-k', label='mean final reward' )
         plt.fill_between(episodes,
                          mean_final_reward-0.5*std_final_reward,
                          mean_final_reward+0.5*std_final_reward,
                          color='k',
                          alpha=0.25)

         plt.plot(episodes, min_final_reward, '--b' , label='min final reward' )
         plt.plot(episodes, max_final_reward, '--r' , label='max final reward' )

         plt.xlabel('episode')
         plt.ylabel('final reward')

         plt.legend(loc='lower right')
         plt.grid()
         plt.show()
```



E. Questions

0. Try out different batch sizes and hyperparameters (including different network architectures). Can you improve the performance?

1. Explore the final batch of trajectories. Check the sequence of actions. Can you make sense of the solution found by the agent? Hint: think of the dynamics on the Bloch sphere and try to visualize the trajectory there.

2. Compare the Policy Gradient method to conventional optimal control: can optimal control give you a control protocol that works for all states? Why or why not?

3. Take one of the high-reward tranjectories in the final batch of data. Now perturb it manually at a few time steps in the first half of the protocol such that it no longer produces an optimal reward (you would have to add a function to the environment which evalues a given trajectory). Last, use the policy to see how it would react to those perturbations in real time. Will it correct on-line for the introduced mistakes (i.e. before the opisode is over)?

4. Find ways to visualize the policy. What is a meaningful way to do that?

F. Advanced Problems

1. What is the initial state distribution $p(s_0)$ in the impelementation above? Check the performance of the PG algorith if $p(s_0)$ is

   - a delta distribution
   - a compactly-supported uniform distribution over some sector of the sphere (say a cap around the south pole)
   - a Gaussian distribution with non-compact support

2. Introduce small Gaussian noise to the rewards, e.g. $r(s,a) \to r(s,a) + \delta r$ where $\delta r \sim \mathcal{N}(0,\delta)$ for some noise strength δ. Does this lead to a serious performance drop as you vary $\delta \in [0, 0.5]$? Why or why not?

3. The loop over the N_{MC} trajectories slows down the algorithm significantly. Consider ways to speed up the evaluation of a single PG iteration. This may include a modification of the environment `QubitEnv2` or the use of parallelization software (see JAX's function `vmap` and `pmap`).

4. Change the environment `QubitEnv2` to define a nonepisodic task. Additionally, introduce a "stop" action so that when the agent bring the RL state close to s_{target} the episode comes to an end and the environment is reset. This would require you to also modify the Policy Gradient implementation above because episodes now can have different length.

In [ ]: