As an application example, the author chose the newly emerged nanobeam photonic crystal nanocavity (hereinafter referred to as nanobeam) in recent years, and nanobeam has gradually become one of the popular choices of nano-laser structure. The author lists step-by-step procedure to implement the code here and explains them, and the same content will not be repeated.

Specific implementation method of the DQN algorithm for optimizing nanobeam photonic crystal nanocavity laser:

1) The initial nanobeam structure is designed in FDTD simulation software, then we run FDTD and ensure that we obtain the base mode. The simulation result of the initial structure is: $Q\ factor = 8 \times 10^4$ under the base mode. This code aims to optimize Q factor to over 3 million (3×10^6).

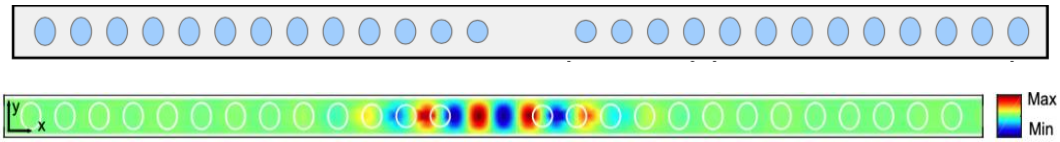
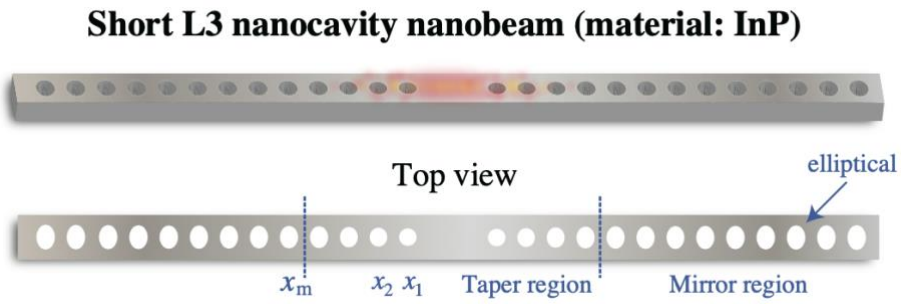


Figure 1. The initial structure of a nanobeam photonic crystal nanocavity designed in FDTD (top: the top view of the structure, bottom: the electric field diagram of base mode calculated by FDTD)

2) Create a Gym environment, defining each environment variable as follows:
   - States: the cumulative variation of the nanobeam's air hole locations (i.e. x coordinate and semi-major axis r2, semi-minor axis r) and the number of cavity defects. The maximum allowable variation of the coordinates is plus or minus 30 or 50 nm, the radius is plus or minus 20 nm, and the cavity number belongs to [1,3,5]. Beyond this scope is considered the termination of this episode.



According to the figure above, x1-x4 represent the coordinates of the four medial tapered holes, xm represents the coordinates of the nine lateral mirror holes, cavity (abbreviated as cav) represents the cavity defect, r represents the semi-minor axis, and r2 represents the semi-major axis. Eight states are set as:

Index	State	Min	Max
0	net x1 change	-30 nm	30 nm
1	net x2 change	-30 nm	30 nm
2	net x3 change	-30 nm	30 nm
3	net x4 change	-30 nm	30 nm
4	net r change	-30 nm	30 nm
5	net r2 change	-20 nm	20 nm
6	num of cavs	[1 3 5]	

7	net xm change	-50 nm	50 nm

- Actions: change the coordinate and radius of the hole (Note: in order to maintain the structural symmetry, only change the hole on the right side, and change the left side according to the mirror image principle, see Figure 2). There are 16 actions in total which are respectively set to increase and decrease the coordinates, radius, and the number of cavs:

Index	Action
0	increase x1 by 2.5 nm
1	decrease x1 by 2.5 nm
2	increase x2 by 2.5 nm
3	decrease x2 by 2.5 nm
4	increase x3 by 2.5 nm
5	decrease x3 by 2.5 nm
6	increase x4 by 2.5 nm
7	decrease x4 by 2.5 nm
8	increase r by 2.5 nm
9	decrease r by 2.5 nm
10	increase r2 by 2.5 nm
11	decrease r2 by 2.5 nm
12	increase num of cavs
13	decrease num of cavs
14	increase xm by 2.5 nm
15	decrease xm by 2.5 nm

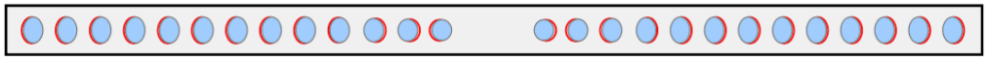


Figure 2. The comparison before and after the change of the coordinates and radius of the hole (blue circle is the original structure before change, red is after change). The amount of change on the left and right sides is symmetrical (i.e. a mirror flip of each other).

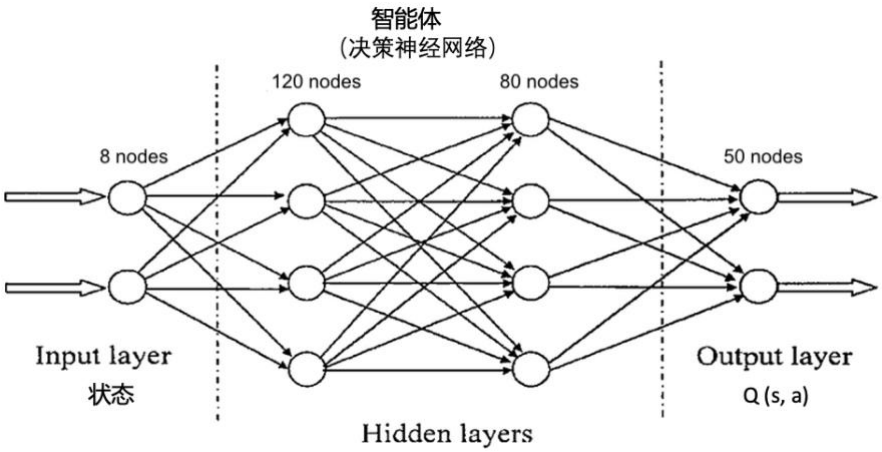
- Reward: the reward is positively correlated with the magnitude of the Q factor, that is, the larger the Q factor at each step, the greater the corresponding reward. The specific reward value is calculated from the formula below, where 3×10^6 represents the optimization target value.

$$reward = (30 - (3 \times 10^6 - Q) \times 10^{-5})$$

- Termination conditions: make a termination judgment at each step and output a Boolean value representing whether the episode is terminated or not. The episode is terminated if any of the following conditions is met:
  1. Episode length is more than 500,
  2. net x1-x4 change is over $\pm$ 30 nm,
  3. net r change is over $\pm$ 30 nm,
  4. net r2 change is over $\pm$ 20 nm,
  5. net xm change is over ± 50 nm,
  6. Solved requirement: considered solved when the reward ≥ 30 (i.e. Q factor ≥ 3E+6).

3) Write the DQN-based reinforcement learning code in Python, and define the policy neural networks (four-layer fully-connected neural networks):

The input of the neural network is the state, and the output is the Q value (i.e. the action value function), with two hidden layers (80 and 120 neurons each). The agent uses this neural network to make the best action decision, choosing the best action from the 16 actions above.



✧ Define the action selection function (e-greedy sampling method):

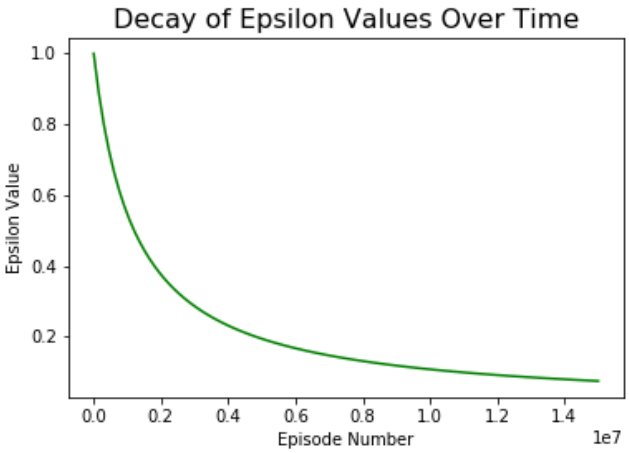


Figure 3. e-greedy sampling method. As the number of iterations decreased exponentially, with an initial value of 0.9 and a termination value of 0.05. At each step, the agent has the probability to do random action choices (exploration mode).

✧ Define the function to train neural networks:

➢ The Pytorch components and functions which have been called includes:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

➢ The definition of loss function:

The loss function used by DQN is typically Mean Squared Error (MSE), but when the Q value becomes noisy or has singular values, the MSE becomes unstable. To overcome this shortcoming, the author chose Smooth L1 Loss (similar to Huber Loss), which is defined as:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

δ represents the difference between the predicted Q value and the expected Q value:

$$\delta = Q(s,a) - (r + \gamma \max_a Q(s',a))$$

- The selection of the optimizer:
  The optimizer used for training the neural network is RMSProp, which addresses the too aggressively changing problem of the rprop algorithm's updated weights, and overcomes the fast and monotonic drop of learning rates of Adagrad algorithm.
  The optimizer will gradually drop the loss function to minimize the loss:

```
optimizer.zero_grad()
loss.backward()
for param in policy_net.parameters():
    param.grad.data.clamp_(-1,1)
optimizer.step()
```

4) Start training the reinforcement learning DQN model: see Algorithm 1 for pseudo-code. The overall algorithm flow is defined in the paper.

The results of nanobeam's optimization by DQN algorithm: after training the DQN model for certain hours, the reward reaches the convergence. At the xx hour of the training, nanobeam's Q factor increases to a super-high xx (i.e. three orders of magnitude increases), exceeding the highest Q of its kind in other literature. This optimization speed is far faster than the manual optimization speed of any experienced scientific researcher, which fully reflects the super-high intelligent level of the deep reinforcement learning algorithm designed by this project.