[CS 376] Machine Learning

Assignment #4: CNN and Q-Learning

Joyce Jiyoung Whang
KAIST School of Computing

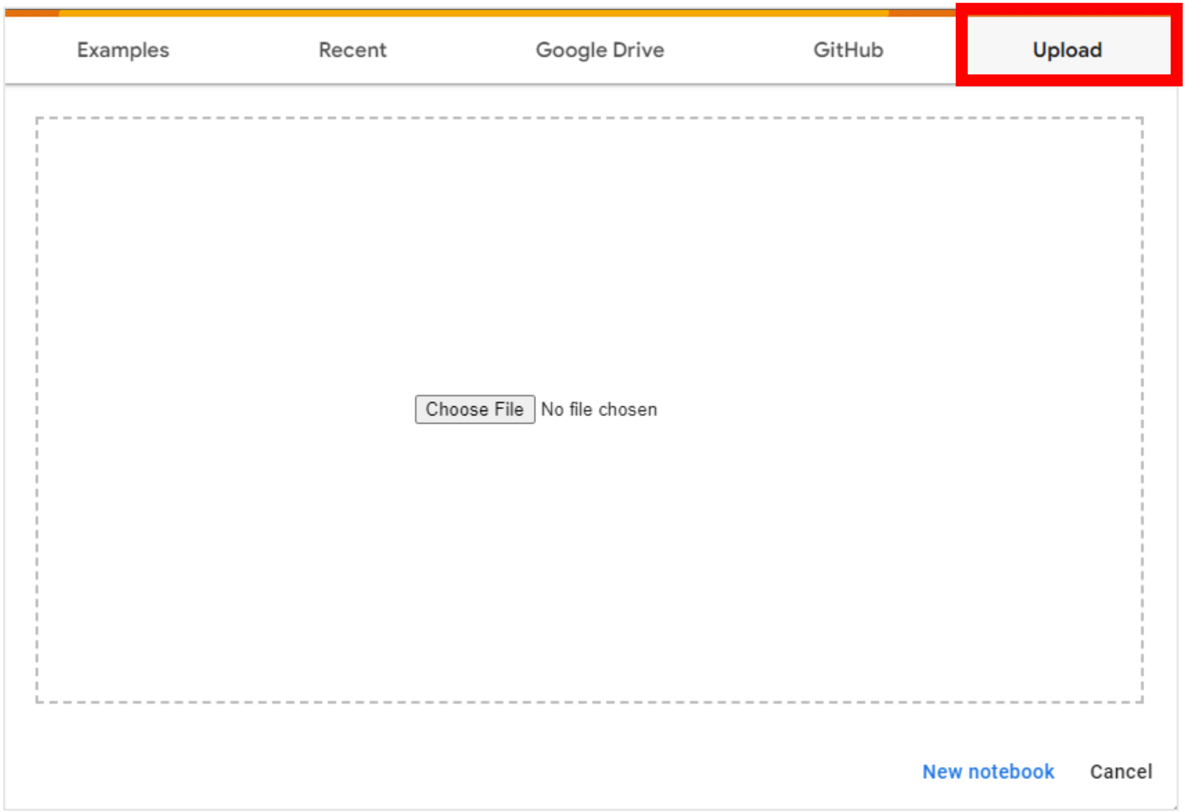
Preliminaries

Google Colab

In this assignment, we will provide files in ipynb (Jupyter Notebook) format. You can open this file with Google Colab, which you can use freely with a Google account.

First, go to https://colab.research.google.com. Then, press the “Upload” tab and upload the provided ipynb file. Now, you can read and make changes to it.

A folder named Colab Notebooks will be created in your drive, so you can access uploaded notebooks here.



Preliminaries

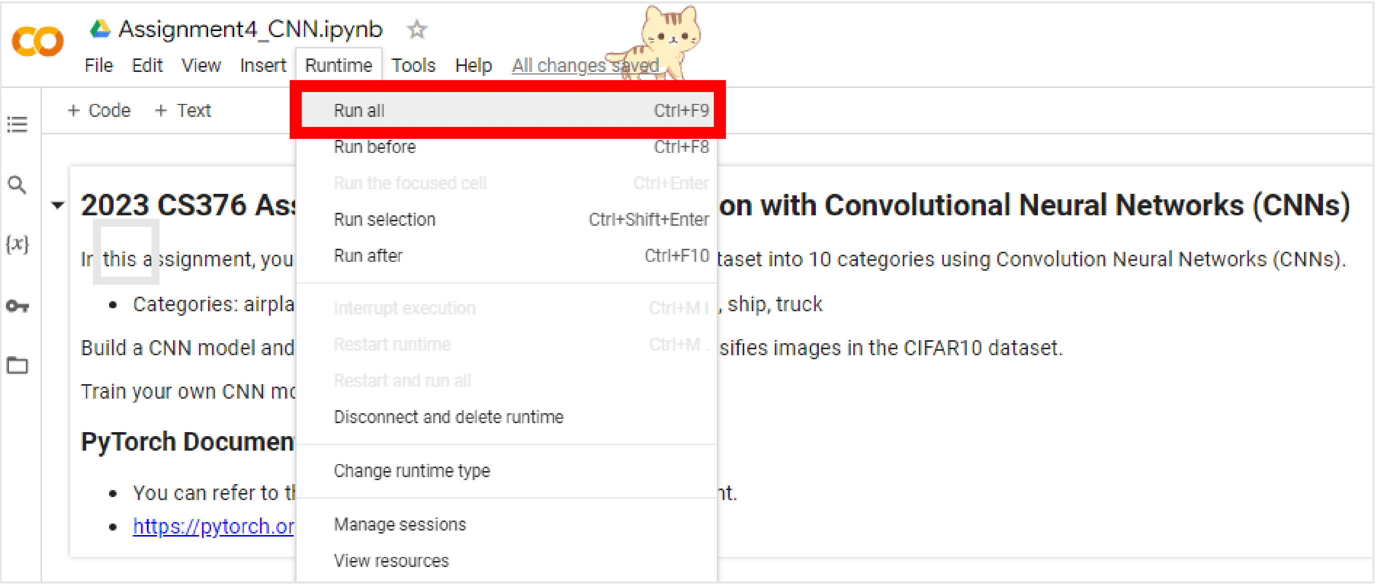
Google Colab

Here are some basic tips for running notebooks in Google Colab.

If you want to run the entire code in the notebook, you can either press Ctrl+F9 or go through the tabs “Runtime” -> “Run all.”

Blocks in notebooks are called cells. If you want to run a single cell, press Ctrl+Enter, or press the Play button, which is located at the top-left corner of each cell.

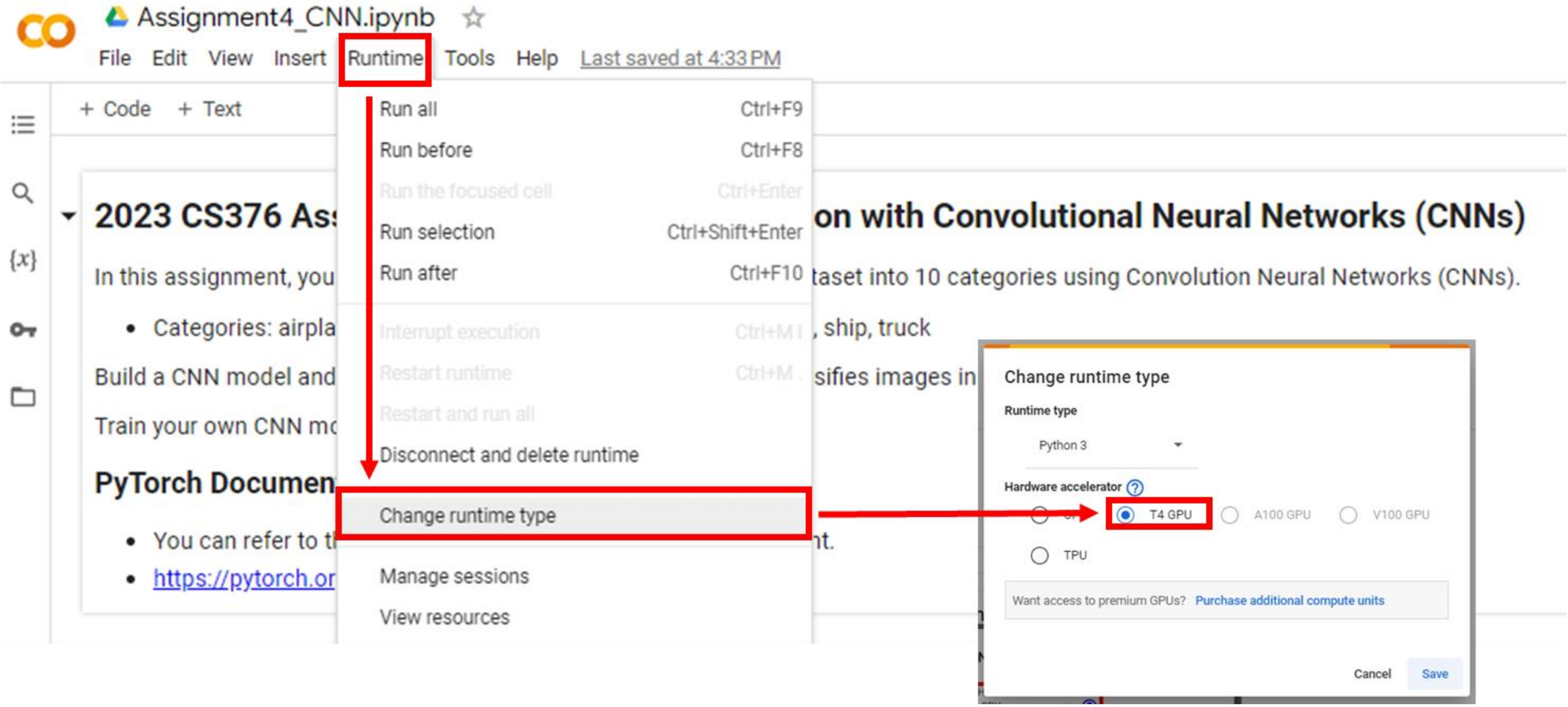
Variables in previous cells are preserved once you run them, so you do not have to rerun all cells every time you change and run a single cell.



Preliminaries

Google Colab

For this assignment, you should change the runtime type to GPU in Google Colab to use **GPU**.



Preliminaries

NumPy

In this assignment, you will need to handle multi-dimensional data. NumPy is a Python library for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. Here are some basic functions in NumPy that you may find useful for this assignment. For more details on these functions, check the official NumPy documentation.

Notations
a, **b**: NumPy arrays. **n**, **m**, **k**: Integers. **condition**: Boolean array, **np**: NumPy (after running the line "import numpy as np").

np.max(a, axis=None): It returns the maximum of an array or maximum along an axis.
np.mean(a, axis=None): It computes the arithmetic mean along the specified axis.
np.transpose(a, axis=None): It returns an array with axes transposed.
np.random.choice(a): It generates a random sample from a given 1-D array.
np.random.rand(): It generates a random sample from a uniform distribution over [0, 1).
np.random.randint(k): It returns a random integer from the "discrete" uniform distribution in [0, k).
np.random.seed(k): It reseeds the singleton RandomState instance.
np.where(condition): It returns elements chosen from x or y depending on **condition**.
np.zeros((n,m)): It returns a new array of given shape and type, filled with zeros (n: row shape, m: column shape).

Preliminaries

PyTorch

In this assignment, you will need to handle multi-dimensional data. PyTorch is a library for large, multi-dimensional arrays and matrices like NumPy. One of the main difference is that PyTorch accelerates GPU operation. Here are some functions in PyTorch that you may find useful for this environment. For more details on these functions, check the official PyTorch documentation.

Notations

a, **c**: PyTorch tensor. **b**: Boolean. **n**, **m**, **k**: Integers. **nn**: Neural network library for PyTorch (valid after running the line “import torch.nn as nn”).
s, **t**: Tuple of ints. **F**: Sub-library of **nn** for immediate calculation instead of creating a layer(valid after running the line “import torch.nn.functional as F)

a.backward(): It computes the gradient of the tensor a.
a.transpose(n,m): It outputs a tensor equal to a but with n-th dimension and m-th dimension swapped.
a.zero_grad(): It resets the gradients of the tensor a.
torch.as_strided(a, size = s, stride = t): It outputs a tensor with the size s, where the tensor a is viewed with the given stride t for each dimension.
torch.manual_seed(s): It sets the seed for generating random numbers.
torch.cuda.manual_seed(s): It sets the seed for generating random numbers for the current GPU.
torch.optim.SGD(params): Stochastic gradient descent optimizer with the model parameter params.
torch.tensordot(a,c,dims = ([n1, n2, …, nk],[m1, m2, …,mk])): It performs element-wise dot product for ni-th dimention in a and mi-th dimension in c.
torch.zeros((n1,n2,…,nk), requires_grad = b): It outputs a zero-valued tensor with size (n1,n2,…,nk). It will be trained with backpropagation if b is true .
F.cross_entropy(a, c): It outputs the cross entropy loss, where a is the logits and c is the ground truth labels.
F.pad(a, pad = (n1, n2, n3, n4)): It ouputs a padded version of a, with padding size n1 for left side, n2 for right side, n3 for top side and n4 for bottom side.

Preliminaries

PyTorch

In this assignment, you will need to handle multi-dimensional data. PyTorch is a library for large, multi-dimensional arrays and matrices like NumPy. One of the main difference is that PyTorch accelerates GPU operation. Here are some functions in PyTorch that you may find useful for this environment. For more details on these functions, check the official PyTorch documentation.

Notations
a, **c**: PyTorch tensor. **b**: Boolean. **n**, **m**, **k**: Integers. **nn**: Neural network library for PyTorch (valid after running the line “import torch.nn as nn”).
s, **t**: Tuple of ints. **F**: Sub-library of **nn** for immediate calculation instead of creating a layer(valid after running the line “import torch.nn.functional as F)

nn.AvgPool2d(k,n): It outputs a nn layer with filter size k and stride n.
nn.Dropout(): It outputs a dropout layer.
nn.init.xavier_uniform_(a): It fills the input tensor with values according to xavier weight initialization.
nn.Linear(n,m): It outputs a linear layer with input dimension n and output dimension m.
nn.MaxPool2d(k, n): It outputs a nn layer with filter size k and stride n.
nn.Parameter(a): It outputs a parameter for model that can be trained with backpropagation (when the requires_grad of a is true).
nn.ReLU(): It outputs a ReLU(Rectified Linear Unit) layer.
Optimizer.step(): It performs a parameter update based on the current gradient.

Preliminaries

Zero Padding

Zero padding is a technique used to preserve the spatial dimensions of the input image after convolution operations on a feature map. Padding involves adding extra pixels around the border of the input feature map before convolution. For more details on the padding technique, check the official PyTorch documentation.

- During the convolution operation, the size of the image decreases. In this process, meaningful data around the edges may be lost. For small images, the output can become almost meaningless.

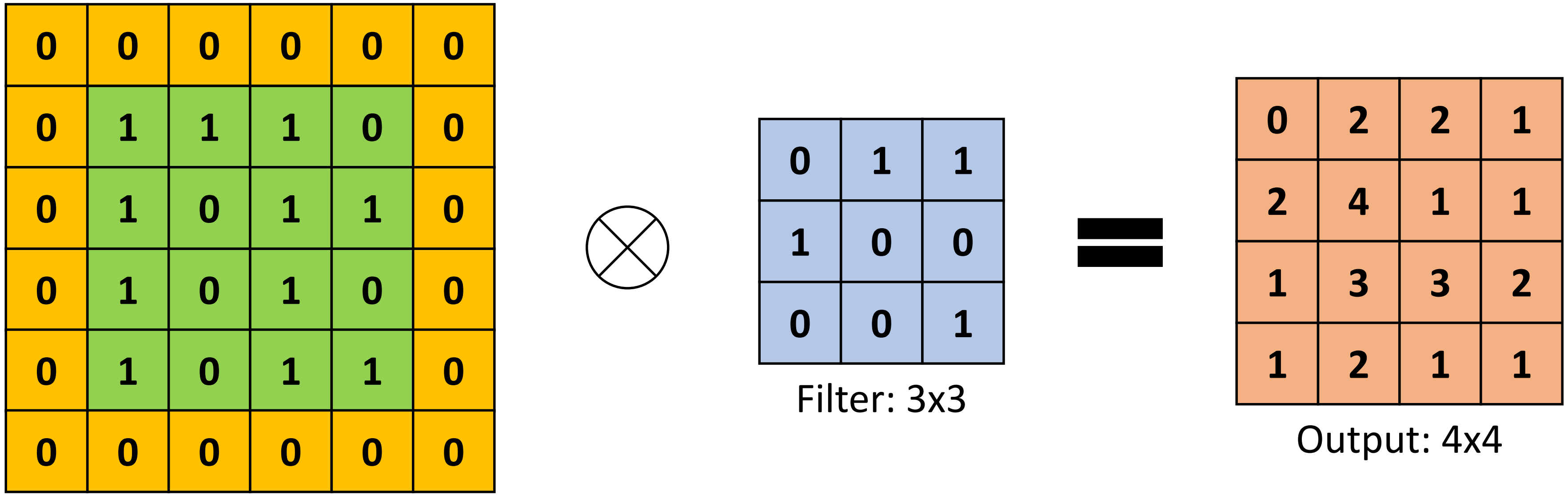
$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}}_{\text{Input: 4x4}} \otimes \underbrace{\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Filter: 3x3}} = \underbrace{\begin{bmatrix} 4 & 1 \\ 3 & 3 \end{bmatrix}}_{\text{Output: 2x2}}$$

Preliminaries

Zero Padding

Zero padding is a technique used to preserve the spatial dimensions of the input image after convolution operations on a feature map. Padding involves adding extra pixels around the border of the input feature map before convolution. For more details on the padding technique, check the official PyTorch documentation.

- With each convolutional layer, just as we define how many filters to have and the size of the filters, we can also specify whether or not to use padding.



Input: 4x4   Filter: 3x3   Output: 4x4

Part 1: CNN-based Image Classification

In Part 1, you should create a CNN-based model to classify images in the CIFAR10 dataset into 10 categories.

Step 1. Construct the data pipeline

Step 2. Implement the CNN layer

Step 3. Build your own CNN-based model

Step 4. Train and test your own CNN-based model

Step 5. Visualize the loss and accuracy changes in training and test set

Part 1: CNN-based Image Classification

[Preparation] Packages and experiment configuration

First, let's import the required libraries. We also set up the hyperparameters for the experiment. You may change **max_epoch**, **learning_rate** and **batch_size**.

```
# Import the required packages.
import os
import random

import numpy as  np

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
from torchvision import transforms

import matplotlib.pyplot as plt

# Fix the random seed.
np.random.seed(0)
random.seed(0)
torch.manual_seed(0)
torch.cuda.manual_seed(0)
```

```
# Training & optimization hyper-parameters.
max_epoch = 100
learning_rate = 0.1
batch_size = 512
device = 'cuda'
```

Part 1: CNN-based Image Classification

[Step 1] Construct the data pipeline

Download the training set and test set of **CIFAR-10** from **torchvision**. We set the dataloader using **torch.utils.DataLoader**. As preprocessing, we perform normalization for each channel of the image. [0.4914, 0.4822, 0.4465] are the mean of each channel, and 0.247, 0.243, 0.261 are the standard deviation of each channel for images in **CIFAR-10**.

```python
data_dir = "./my_data"

# torchvision.transforms is a module provided by PyTorch for preprocessing and augmenting image data.
transform = transforms.Compose(
    [transforms.ToTensor(), # Transform the data type of images into Tensor.
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))]) # Normalize images.

# Define the dataset and dataloader.
# torchvision.datasets is a package that aggregates datasets provided by PyTorch.
train_dataset = torchvision.datasets.CIFAR10(root=data_dir, train=True, download=True, transform=transform)
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)

test_dataset = torchvision.datasets.CIFAR10(root=data_dir, train=False, download=True, transform=transform)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=2)

# The classes of the CIFAR10 dataset
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
output_dim = len(classes)
```

Part 1: CNN-based Image Classification

[Step 2] Implement the CNN layer

Implement your own Convolutional Neural Networks (CNNs). For simplicity, we assume that all filters are square filters.

```
# Let's build your own CNN.
class My_Conv2d(nn.Module):
    def __init__(self, in_feature, n_filter, filter_size, stride, pad):
        super(My_Conv2d, self).__init__()
        # Create your own convolutional layer.
        """
        Your convolutional layer will be initialized with

        - in_feature(int): input feature dimension (in_feature(int))
        - n_filter(int): the number of filters (n_filter(int))
        - filter_size(int): the size of filter
        - stride(int): stride
        - pad(int): number of paddings

        For simplicity, we assume that all filters are square filters
        The shape of your filter should be [n_filter x in_feature x filter_size x filter_size]
        """
        ################## IMPLEMENT HERE ##################
        # Define the parameters for CNN.
        self.filter_size = NotImplemented
        self.in_feature = NotImplemented
        self.n_filter = NotImplemented

        # [n_filter x in_feature x filter_size x filter_size]
        self.filters = NotImplemented
        self.bias = NotImplemented
        torch.nn.init.xavier_uniform_(self.filters)

        self.stride = NotImplemented
        self.pad = NotImplemented
        ####################################################
```

```
def forward(self,x):
    # The size of the input will be [n_batch x in_feature x H x W]
    # The size of the output should be [n_batch x n_filter x H' x W']
    # where H', W' are the image size after convolution

    # Calculate the size of feature map after a convolution operation.
    ################## IMPLEMENT HERE ##################
    new_h = NotImplemented
    new_w = NotImplemented
    ####################################################

    # Add the padding to the input x.
    padded_x = F.pad(x, pad = (self.pad, self.pad, self.pad, self.pad))

    ################## IMPLEMENT HERE ##################
    # Precalculate the strided verison of the padded input.
    strided = torch.as_strided(padded_x, (NotImplemented), (NotImplemented))

    # Calculate the CNN result.
    result_x = torch.tensordot(NotImplemented).transpose(0,1) + self.bias
    ####################################################
    return result_x
```

Part 1: CNN-based Image Classification

[Step 3] Build your own CNN-based model

Based on the operations we learned on class, build your own CNN model. You may use the functions such as **nn.MaxPool2d** or **nn.Linear**, as long as you use **My_Conv2d** for convolution.

```python
# Let's build your own model.

class MyOwnClassifier(nn.Module):
    def __init__(self):
        super(MyOwnClassifier, self).__init__()
        ################## IMPLEMENT HERE ##################
        # Define your own layers!
        raise NotImplementedError
        ####################################################

    def forward(self, x):
        ################## IMPLEMENT HERE ##################
        # Pass the input through the layer.
        raise NotImplementedError
        ####################################################
        return output
```

Part 1: CNN-based Image Classification

[Step 3] Build your own CNN-based model

After building your own CNN model, initialize the network and optimizer. Print your neural network architecture.

```
[ ]  my_classifier = MyOwnClassifier()
     my_classifier = my_classifier.to(device)

     # Print the architecture of your convolution neural network.
     print(my_classifier)

     optimizer = optim.SGD(my_classifier.parameters(), lr=learning_rate)
```

```
MyOwnClassifier(
  (conv1): My_Conv2d()
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout(p=0.5, inplace=False)
  (conv2): My_Conv2d()
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout2): Dropout(p=0.5, inplace=False)
  (conv3): My_Conv2d()
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout3): Dropout(p=0.5, inplace=False)
  (fcn1): Linear(in_features=4096, out_features=256, bias=True)
  (dropout4): Dropout(p=0.5, inplace=False)
  (fcn2): Linear(in_features=256, out_features=10, bias=True)
  (relu): ReLU()
)
```

Part 1: CNN-based Image Classification

[Step 4] Model training & testing

Let's train your model on the training set and test it on the test set. To train the model, the loss should be backpropagated.

```python
train_losses = []
train_accs = []
test_losses = []
test_accs = []
for epoch in range(max_epoch):
    # Train phase.
    my_classifier.train()
    epoch_loss = 0
    epoch_acc = 0
    n_train = 0
    for inputs, labels in train_dataloader:
        # Load data to with GPU.
        # Send 'inputs' and 'labels' to either cpu or gpu using the 'device' variable.
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Feed data into the network and get outputs.
        # Feed `inputs` into the network, get an output, and keep it in a variable called `logits`.
        logits = my_classifier(inputs)

        # Calculate loss.
        # Note: `F.cross_entropy` function receives logits, or pre-softmax outputs, rather than final probability scores.
        # Compute loss using `logits` and `labels`, and keep it in a variable called `loss`.
        ################## IMPLEMENT HERE ##################
        loss = NotImplemented
        ####################################################

        # Note: You should flush out gradients computed in the previous step before computing gradients in the current step.
        #       Otherwise, gradients will accumulate.

        # Flush out the previously computed gradient.
        # write your code here (one-liner).
        ################## IMPLEMENT HERE ##################
        raise NotImplementedError
        ####################################################
```

```python
        # Backpropagate loss.
        # Backward the computed loss.
        # write your code here (one-liner).
        ################## IMPLEMENT HERE ##################
        raise NotImplementedError
        ####################################################

        # Update the network weights.
        # Write your code here (one-liner).
        ################## IMPLEMENT HERE ##################
        raise NotImplementedError
        ####################################################

        # Gather loss and accuracy for visualization.
        epoch_loss += loss.item()*inputs.size(0)
        epoch_acc += (logits.argmax(dim=1) == labels).float().sum().item()
        n_train += inputs.size(0)

    # Save losses and accuracies in a list so that we can visualize them later.
    epoch_loss /= n_train
    epoch_acc /= n_train
    train_losses.append(epoch_loss)
    train_accs.append(epoch_acc)


    # Test phase
    n_test = 0.
    test_loss = 0.
    test_acc = 0.
```

Part 1: CNN-based Image Classification

[Step 4] Model training & testing

Let's train your model on the training set and test it on the test set. To train the model, the loss should be backpropagated.

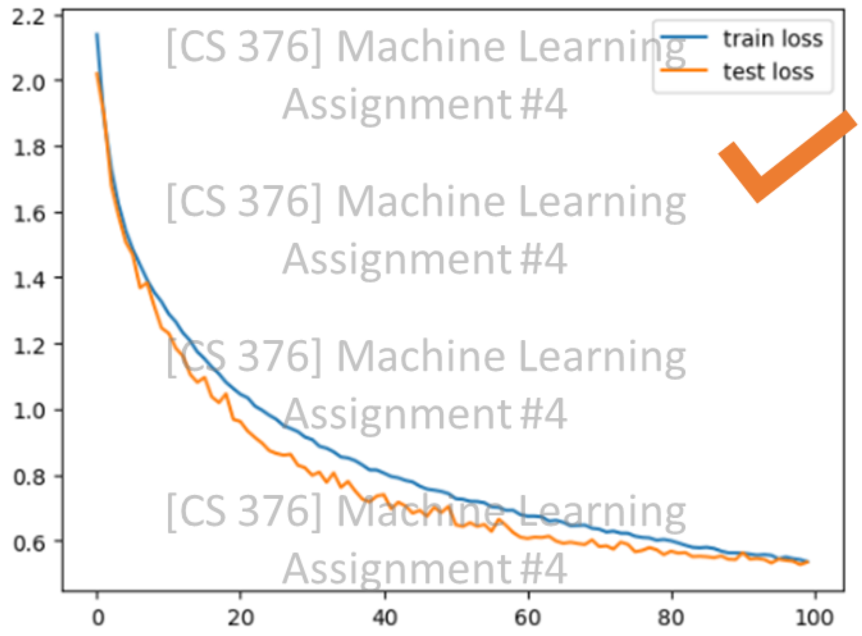
```
my_classifier.eval()
for test_inputs, test_labels in test_dataloader:
    # Send 'test_inputs' and 'test_labels' to either cpu or gpu using the 'device' variable.
    test_inputs = test_inputs.to(device)
    test_labels = test_labels.to(device)

    # Feed `inputs` into the network, get an output, and keep it in a variable called `logits`.
    ################## IMPLEMENT HERE ##################
    logits = NotImplemented
    ####################################################

    # Calculate loss
    # Note: `F.cross_entropy` function receives logits, or pre-softmax outputs, rather than final probability scores.
    # Compute loss using `logits` and `labels`, and keep it in a variable called `tmp_loss`.
    ################## IMPLEMENT HERE ##################
    tmp_loss = NotImplemented
    ####################################################

    # Gather loss and accuracy for visualization.
    test_loss += tmp_loss.item()
    test_acc += (logits.argmax(dim=1) == test_labels).float().sum().item()
    n_test += test_inputs.size(0)

test_loss /= n_test
test_acc /= n_test
test_losses.append(test_loss)
test_accs.append(test_acc)

print('[epoch:{}] train loss : {:.4f} train accuracy : {:.4f} test_loss : {:.4f} test accuracy : {:.4f}'.format(epoch+1, epoch_loss, epoch_acc, test_loss, test_acc))
```

Part 1: CNN-based Image Classification

[Step 5] Visualize the loss and accuracy changes in training and test set

Let’s visualize the changes of the train and the test loss.

```
# Visualize the losses for the train and test set.
plt.plot(train_losses, label='train loss')
plt.plot(test_losses, label='test loss')
plt.legend()
plt.show()
```

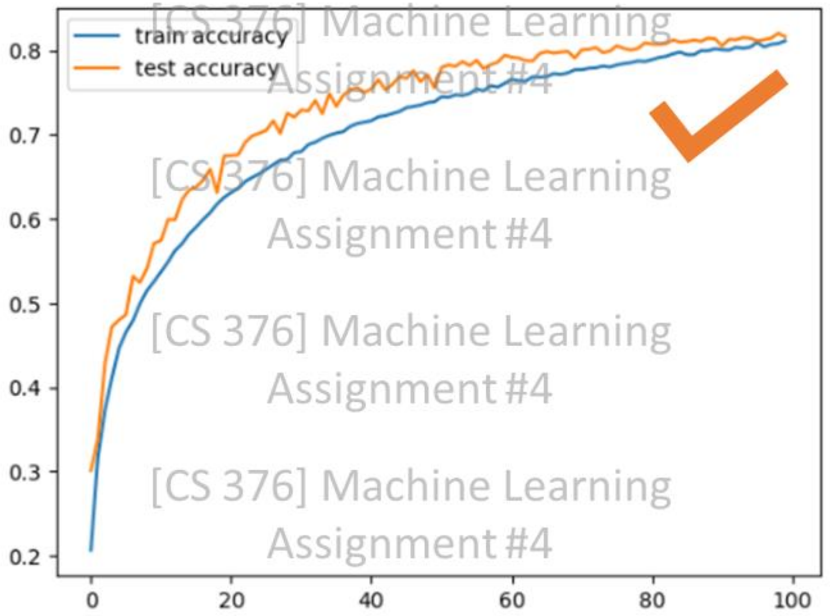


Part 1: CNN-based Image Classification

[Step 5] Visualize the loss and accuracy changes in training and test set

Let's visualize the accuracy changes on the training set and the test set.

```
# Visualize the accruacy for the train and test set.
plt.clf()
plt.plot(train_accs, label='train accuracy')
plt.plot(test_accs, label='test accuracy')
plt.legend()
plt.show()
```



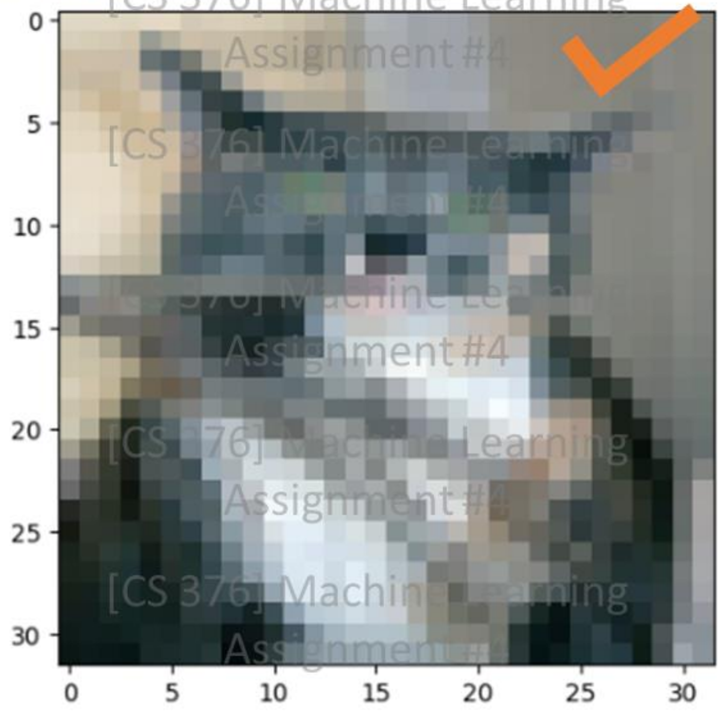
Part 1: CNN-based Image Classification

[Step 5] Visualize the loss and accuracy changes in training and test set

Check the prediction result of the model given an image.

```
my_classifier.eval()

num_test_samples = len(test_dataset)
random_idx = random.randint(0, num_test_samples)
test_input, test_label = test_dataset.__getitem__(random_idx)
test_prediction = F.softmax(my_classifier(test_input.unsqueeze(0).to(device)), dim=1).argmax().item()
print('label : %s' % classes[test_label])
print('prediction : %s' % classes[test_prediction])

# Functions to show an image
def imshow(img):
    img[0] = img[0] * 0.247 + 0.4914
    img[1] = img[1] * 0.243 + 0.4822
    img[2] = img[2] * 0.261 + 0.4465
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
# Show images
imshow(torchvision.utils.make_grid(test_input))
```

```
label : cat
prediction : cat
```



Part 2: Q-Learning

For part 2 of this assignment, you will implement the Q-learning algorithm.

Here are the steps you should follow.

Step 1. Implement the environment.

Step 2. Initialize Q-table.

Step 3. Implement the epsilon-greedy strategy.

Step 4. Train the agent.

Step 5. Test the agent.

Step 6. Visualize the agent’s behavior based on the learned Q-table.

Part 2: Q-Learning

[Step 0] Preparation

Before we start, we will import some useful libraries (e.g., NumPy).

Do not use other libraries to implement Part 2.

```
[ ]  !pip install numpy
     !pip install imageio
     !pip install pygame
     !pip install tqdm
```

```
[ ]  import numpy as np
     from tqdm import tqdm

     # To visualize
     import pygame
     import imageio
     from PIL import Image
     import IPython
     import os

     os.environ["SDL_VIDEODRIVER"] = "dummy"
```

Part 2: Q-Learning

[Step 0] Preparation

Also, we will upload some assets for visualization.

Assets are from the following references:

- Elf and stool from https://franuka.itch.io/rpg-snow-tileset
- Rock from https://toppng.com/show_download/226268/rock-rock-pixel-art/large
- All other assets by Mel Tillery http://www.cyaneus.com/

```
[ ]  # Upload img.zip
     from google.colab import files

     uploaded = files.upload()
```

```
[ ]  !unzip img.zip
```

Part 2: Q-Learning

[Step 0] Preparation

We provide a helper function to visualize the environment.

```
[ ]  # Set parameters for visualization
     window_size = (512, 512)

     # Load images
     hole_image = pygame.image.load("img/cracked_hole.png")
     rock_image = pygame.image.load("img/rock.png")
     ice_image = pygame.image.load("img/ice.png")
     goal_image = pygame.image.load("img/goal.png")
     start_image = pygame.image.load("img/stool.png")
     elfs = [
         "img/elf_left.png",
         "img/elf_down.png",
         "img/elf_right.png",
         "img/elf_up.png",
     ]
     elf_images = [pygame.image.load(f_name) for f_name in elfs]

     # Set display
     pygame.init()
     pygame.display.init()
     pygame.display.set_caption("SlipperyFrozenLake")

     window_surface = pygame.Surface(window_size)
     cell_width = 64
     cell_height = 64
     smaller_cell_scale = 1
     small_cell_width = int(cell_width * smaller_cell_scale)
     small_cell_height = int(cell_height * smaller_cell_scale)

     def _center_small_rect(big_rect, small_dims):
         offset_w = (big_rect[2] - small_dims[0]) / 2
         offset_h = (big_rect[3] - small_dims[1]) / 2
         return (
             big_rect[0] + offset_w,
             big_rect[1] + offset_h,
         )
```

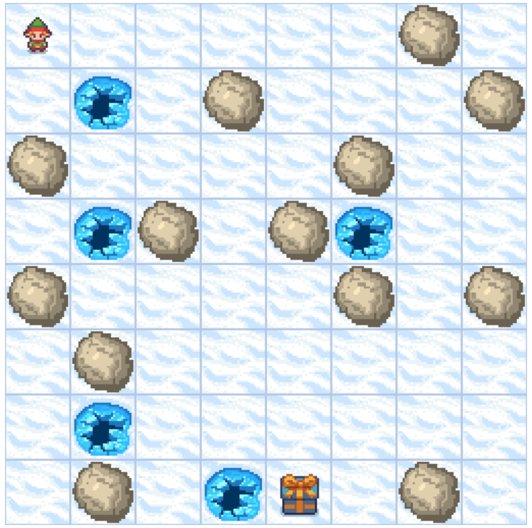
```
def render(lake, row, col, a_prev):
    # Prepare images
    #elf_img = elf_images[a_prev]
    elf_img = pygame.transform.scale(elf_images[a_prev], (cell_width, cell_height))
    hole_img = pygame.transform.scale(hole_image, (cell_width, cell_height))
    rock_img = pygame.transform.scale(rock_image, (cell_width, cell_height))
    ice_img = pygame.transform.scale(ice_image, (cell_width, cell_height))
    goal_img = pygame.transform.scale(goal_image, (cell_width, cell_height))
    start_img = pygame.transform.scale(start_image, (small_cell_width, small_cell_height))

    for y in range(8):
        for x in range(8):
            rect = (x * cell_width, y * cell_height, cell_width, cell_height)
            if lake[y][x] == "H":
                window_surface.blit(hole_img, (rect[0], rect[1]))
            elif lake[y][x] == "R":
                window_surface.blit(rock_img, (rect[0], rect[1]))
            elif lake[y][x] == "G":
                window_surface.blit(ice_img, (rect[0], rect[1]))
                goal_rect = _center_small_rect(rect, goal_img.get_size())
                window_surface.blit(goal_img, goal_rect)
            elif lake[y][x] == "S":
                window_surface.blit(ice_img, (rect[0], rect[1]))
                stool_rect = _center_small_rect(rect, start_img.get_size())
                window_surface.blit(start_img, stool_rect)
            else:
                window_surface.blit(ice_img, (rect[0], rect[1]))

            pygame.draw.rect(window_surface, (180, 200, 230), rect, 1)

    cell_rect = (
        col * cell_width,
        row * cell_height,
        cell_width,
        cell_height,
    )

    elf_rect = _center_small_rect(cell_rect, elf_img.get_size())
    window_surface.blit(elf_img, elf_rect)
    return np.transpose(np.array(pygame.surfarray.pixels3d(window_surface)), axes=(1, 0, 2))
```

Part 2: Q-Learning

[Step 1] Implement the environment.

We provide the information on the environment below:

- The agent ( ) starts at the starting point ( ), and the goal is to reach the goal ( ) while avoiding the holes ( ).
- The agent can choose four actions: LEFT (0), DOWN (1), RIGHT (2), UP (3)
- If the agent chooses one action, the agent moves toward the selected direction until one of the conditions is satisfied.
  - The agent stops if the rock ( ) blocks the agent.
  - The agent stops if the agent reaches the boundary of the lake.
  - The agent stops if the agent meets the hole ( ).
    This is the case that we should avoid.
  - The agent stops if the agent reaches the goal ( ).
    This is the case that we want.
- The reward is only given when the agent reaches the goal (+10).



Part 2: Q-Learning

[Step 1] Implement the environment.

Using the helper function, we can visualize the environment as follows:

```
rgb_array = render(lake, 0, 0, DOWN)
img = Image.fromarray(rgb_array)
img
```

Part 2: Q-Learning

[Step 1] Implement the environment.

We have to implement next_step(), which determines the next state, reward, and terminate condition for the given (state, action) pair.

```
[ ] def next_step(row, col, action):
        reward = 0 # Calculated reward
        terminate = False # Boolean value indicating whether the episode is terminated or not

        while True:
            # 1-1. Implement the case when the agent meets a hole (which terminates the episode).
            # At the end, the agent must stand on the hole.
            ################################ IMPLEMENT HERE ################################
            raise NotImplementedError
            ################################################################################

            # 1-2. Implement the case when the agent meets the goal (which terminates the episode with positive reward of 10).
            # At the end, the agent must stand on the goal.
            ################################ IMPLEMENT HERE ################################
            raise NotImplementedError
            ################################################################################

            # 1-3. Implement the case when the agent faces a rock.
            # Note that the agent cannot reach to the cell containing rock.
            ################################ IMPLEMENT HERE ################################
            raise NotImplementedError
            ################################################################################

            # 1-4. Implement the case when the agent reaches the boundary of the lake.
            # Note that the agent cannot move out of the boundary.
            ################################ IMPLEMENT HERE ################################
            raise NotImplementedError
            ################################################################################

            # 1-5. If no condition is satisfied, prepare next iteration (change the position of the agent)
            ################################ IMPLEMENT HERE ################################
            raise NotImplementedError
            ################################################################################

        return (row, col), reward, terminate
```

Part 2: Q-Learning

[Step 1] Implement the environment.

To check that next_step() is well implemented, we provide some test cases.

You need to submit a screenshot of the test result.

```
[10] assert next_step(0, 0, RIGHT) == ((0, 5), 0, False)
     assert next_step(0, 0, UP) == ((0, 0), 0, False)
     assert next_step(3, 7, LEFT) == ((3, 5), 0, True)
     assert next_step(6, 4, DOWN) == ((7, 4), 10, True)
     print("Sanity check done!")

Sanity check done!
```

Part 2: Q-Learning

[Step 2] Initialize Q-Table

To perform Q-learning, we first need to initialize the Q-table.

Q-table contains the state-action values for each (state, action) pair.

```
[ ]  # 2. Initialize Q-table.
     # Q-table contains the state-action values for each (state, action) pair
     ################################ IMPLEMENT HERE ################################
     q_table = NotImplemented
     raise NotImplementedError
     ################################################################################
```

Part 2: Q-Learning

[Step 3] Implement the epsilon-greedy strategy

Epsilon-greedy strategy is a policy that handles the exploration/exploitation trade-off. For a given epsilon, the agent chooses an action based on the following strategy:

- With probability epsilon, we randomly select the possible actions.
- With probability 1 - epsilon, we greedily choose the next action based on the Q-table.

```
[ ] # to_s converts each (row, col) pair into an integer (row * 8 + col), which is the index representing the state (row, col)
    def to_s(row, col):
        return row * 8 + col
```

```
[ ] def epsilon_greedy(row, col, q_table, epsilon):
        if np.random.rand() < epsilon:
            # 3-1. Implement exploration.
            ############################### IMPLEMENT HERE ###############################
            a = NotImplemented
            raise NotImplementedError
            ##############################################################################
        else:
            # 3-2. Implement exploitation.
            # Note that if there are multiple candidates (due to same values), then the agent must randomly choose the action among them.
            ############################### IMPLEMENT HERE ###############################
            a = NotImplemented
            raise NotImplementedError
            ##############################################################################
        return a
```

Part 2: Q-Learning

[Step 4] Train the agent

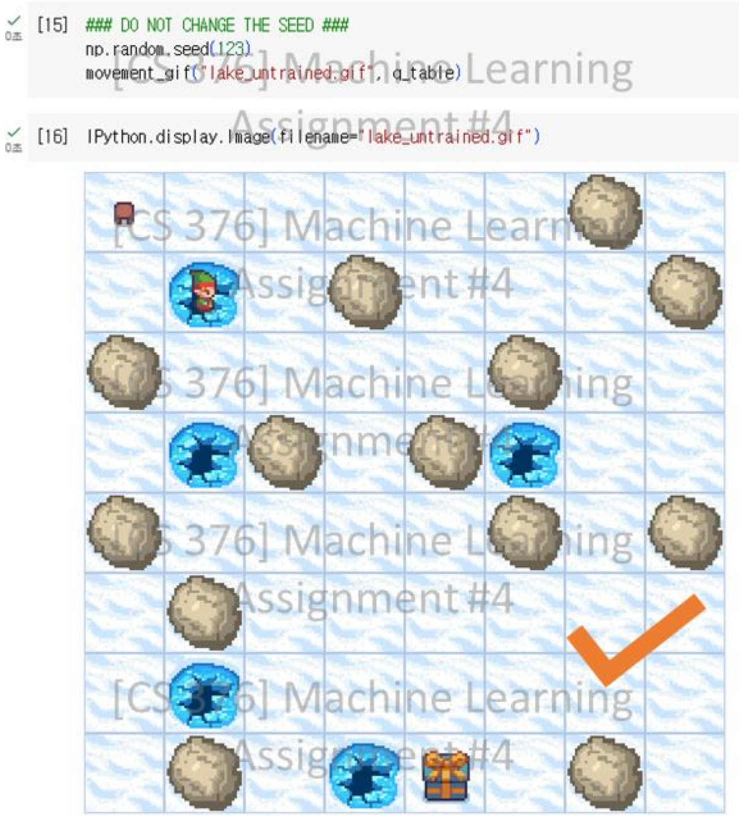
In this step, we are going to train the agent. Before training, we first watch the behavior of the untrained agent.

```
[ ]  # Generates gif file of the agent's movement based on the input q_table
     def movement_gif(file, q_table):
         row = 0
         col = 0
         done = False
         images = []

         rgb_array = render(lake, 0, 0, DOWN)
         img = Image.fromarray(rgb_array)
         for _ in range(4):
             images.append(img)

         while not done:
             a = epsilon_greedy(row, col, q_table, 0)

             s = to_s(row, col)
             (row, col), r, done = next_step(row, col, a)

             rgb_array = render(lake, row, col, a)
             img = Image.fromarray(rgb_array)
             images.append(img)

         for _ in range(7):
             images.append(img)

         imageio.mimsave(file, [np.array(img) for i, img in enumerate(images)], fps=2)
```

Part 2: Q-Learning

[Step 4] Train the agent

In this step, we are going to train the agent. Before training, we first watch the behavior of the untrained agent.



As we can see, the agent fails to reach the goal.

You should take a screenshot when the episode terminates. Otherwise, the screenshot will not be counted.

Part 2: Q-Learning

[Step 4] Train the agent

Now we are going to train the agent.

First, we set some parameters (learning rate and discount rate).

```
[ ]  learning_rate = 0.1
     discount_rate = 0.8
```

Part 2: Q-Learning

[Step 4] Train the agent

Then we implement the overall training.

```python
def train_episode(epsilon):
    row = 0
    col = 0
    done = False

    while not done:
        # 4-1. Choose the action using epsilon greedy algorithm
        ############################### IMPLEMENT HERE ###############################
        a = NotImplemented
        raise NotImplementedError
        ##############################################################################

        # 4-2. Calculate the next state and the reward
        ############################### IMPLEMENT HERE ###############################
        (row_new, col_new), r, done = NotImplemented
        raise NotImplementedError
        ##############################################################################

        # 4-3. Update Q-table
        ############################### IMPLEMENT HERE ###############################
        raise NotImplementedError
        ##############################################################################

        # 4-4. Update current state
        ############################### IMPLEMENT HERE ###############################
        row = NotImplemented
        col = NotImplemented
        raise NotImplementedError
        ##############################################################################

def train(num_epoch=10000):
    # 4-5. Train the agent.
    # We use epsilon = 1 / (i + 1) in i-th episode. (Note that i=0 in the first episode)
    for i in tqdm(range(num_epoch)):
        ############################### IMPLEMENT HERE ###############################
        raise NotImplementedError
        ##############################################################################
```

Part 2: Q-Learning

[Step 4] Train the agent

After training, we can check the trained Q-table. Your screenshot should contain **more than ten rows** of the Q-table.

```
[20] q_table

array([[8.58993459e-02, 1.92413213e-02, 1.07374182e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [4.78035596e-05, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [3.68185595e-07, 1.34217728e+00, 0.00000000e+00, 2.13698080e-01],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 2.32786484e-01],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.55013683e-01, 1.67772160e+00, 0.00000000e+00],
       [0.00000000e+00, 2.09715200e+00, 1.67771880e-01, 3.18767104e-01],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
```

Part 2: Q-Learning

[Step 5] Test the agent

Now we test our agent. To do so, we first implement test_episode() and test().

```
[ ] def test_episode():
        row = 0
        col = 0
        reward_total = 0
        done = False

        while not done:
            # 5-1. Choose the action using greedy algorithm (Note: This step is different from epsilon greedy!)
            ################################ IMPLEMENT HERE ################################
            a = NotImplemented
            raise NotImplementedError
            ################################################################################

            # 5-2. Calculate the next state and the reward
            ################################ IMPLEMENT HERE ################################
            (row_new, col_new), r, done = NotImplemented
            raise NotImplementedError
            ################################################################################

            # 5-3. Update current state and total reward
            ################################ IMPLEMENT HERE ################################
            row = NotImplemented
            col = NotImplemented
            reward_total = NotImplemented
            raise NotImplementedError
            ################################################################################

        return reward_total
```

```
def test(num_epoch=10000):
    list_reward = []
    # 5-4. Test the agent for num_epoch episodes.
    # We should save the resulting reward in list_reward.

    for i in tqdm(range(num_epoch)):
        ################################ IMPLEMENT HERE ################################
        raise NotImplementedError
        ################################################################################

    # 5-5. Calculate the average reward.
    ################################ IMPLEMENT HERE ################################
    reward_average = NotImplemented
    raise NotImplementedError
    ################################################################################

    return reward_average
```

Part 2: Q-Learning

[Step 5] Test the agent

To evaluate our agent, we run 10,000 episodes using the trained Q-table and report the average reward.

```
[22] test()

100%|██████████| 10000/10000 [00:04<00:00, 2382.06it/s]
10.0
```

Part 2: Q-Learning

[Step 6] Visualize the agent’s behavior based on the learned Q-table

Also, we visualize the behavior of our trained agent.

```
[23] # 6-1. Make the gif file of the trained agent's movement.
     movement_gif("lake_trained.gif", q_table)

     # 6-2. Visualize the gif file.
     IPython.display.Image(filename="lake_trained.gif")
```

Now, the agent succeeds to reach the goal.

You should take a screenshot when the episode terminates. Otherwise, the screenshot will not be counted.

Submission

- Take screenshots of all results for both part 1 and part 2. You should submit the results with the check mark:
- You should reproduce the result by yourself. You should not submit the pictures provided in this material.
- **Make one PDF document containing all the screenshots.**
- **Deadline: Dec. 8th (Friday) PM 11:59. We do not accept late submissions.**
- If you have any questions, please post them on the KLMS Q&A Board.
- Good luck, and have fun!

References

- https://huggingface.co/blog/deep-rl-q-part2
- https://deeplizard.com/learn/video/qSTv_m-KFk0
- https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html