Table of Contents

[Exercise 1 1](#_Toc165065963)

[Exercise 2 3](#_Toc165065964)

[Exercise 3 6](#_Toc165065965)

[Exercise 4 11](#_Toc165065966)

# Exercise 1

Answer:

The method **id\_to\_fruit** is supposed to return the fruit name corresponding to the given id. However, it is not working as expected due to a bug.

1. The returned result is wrong because the method is not returning the fruit at the correct index. In the for loop, the **idx** variable is incremented before the comparison with **fruit\_id**. Therefore, the method will always return the first fruit in the set.
2. To fix this, the **idx** variable should be incremented after the comparison with **fruit\_id**. Here's the corrected method:
3. This means that the function's logic of iterating over the set and comparing the index to the fruit\_id will not work as expected because sets do not maintain a consistent order of their elements.To fix this issue, you can convert the set to a list before iterating over it. This way, you can access elements by their index. Here's how you can modify the id\_to\_fruit function to fix the bug:

import numpy as np

from typing import Set

def id\_to\_fruit(fruit\_id: int, fruits: Set[str]) -> str:

"""

This method returns the fruit name by getting the string at a specific index of the set.

:param fruit\_id: The id of the fruit to get

:param fruits: The set of fruits to choose the id from

:return: The string corrosponding to the index ``fruit\_id``

\*\*This method is part of a series of debugging exercises.\*\*

\*\*Each Python method of this series contains bug that needs to be found.\*\*

| ``1 It does not print the fruit at the correct index, why is the returned result wrong?``

| ``2 How could this be fixed?``

This example demonstrates the issue:

name1, name3 and name4 are expected to correspond to the strings at the indices 1, 3, and 4:

'orange', 'kiwi' and 'strawberry'..

>>> name1 = id\_to\_fruit(1, {"apple", "orange", "melon", "kiwi", "strawberry"})

>>> name3 = id\_to\_fruit(3, {"apple", "orange", "melon", "kiwi", "strawberry"})

>>> name4 = id\_to\_fruit(4, {"apple", "orange", "melon", "kiwi", "strawberry"})

"""

fruits\_list = list(fruits)

idx = 0

for fruit in fruits:

if fruit\_id == idx:

return fruit

idx += 1

raise RuntimeError(f"Fruit with id {fruit\_id} does not exist")

name1 = id\_to\_fruit(1, {"apple", "orange", "melon", "kiwi", "strawberry"})

name3 = id\_to\_fruit(4, {"apple", "orange", "melon", "kiwi", "strawberry"})

name4 = id\_to\_fruit(3, {"apple", "orange", "melon", "kiwi", "strawberry"})

print(name1)

print(name3)

print(name4)

# Exercise 2

answer:

The provided swap function aims to swap the x and y coordinates in a numpy array of bounding box coordinates. However, there are two issues with the current implementation:

1. Obvious Error: The function attempts to swap the coordinates by directly assigning values from one index to another. However, the assignment is incorrect because it mistakenly assigns coords[:, 1] to both coords[:, 0] and coords[:, 1], and coords[:, 3] to both coords[:, 2] and coords[:, 3]. This results in no actual swapping of the x and y coordinates.
2. Fixing the Swap: To correctly swap the x and y coordinates, you need to assign the x coordinates to the y positions and the y coordinates to the x positions. The correct assignment should be coords[:, 0] = coords[:, 1] and coords[:, 1] = coords[:, 0]for the x and y coordinates, and similarly for the second pair of coordinates.

# You can copy this code to your personal pipeline project or execute it here.

def swap(coords: np.ndarray):

"""

This method will flip the x and y coordinates in the coords array.

:param coords: A numpy array of bounding box coordinates with shape [n,5] in format:

::

[[x11, y11, x12, y12, classid1],

[x21, y21, x22, y22, classid2],

...

[xn1, yn1, xn2, yn2, classid3]]

:return: The new numpy array where the x and y coordinates are flipped.

\*\*This method is part of a series of debugging exercises.\*\*

\*\*Each Python method of this series contains bug that needs to be found.\*\*

| ``1 Can you spot the obvious error?``

| ``2 After fixing the obvious error it is still wrong, how can this be fixed?``

>>> import numpy as np

>>> coords = np.array([[10, 5, 15, 6, 0],

... [11, 3, 13, 6, 0],

... [5, 3, 13, 6, 1],

... [4, 4, 13, 6, 1],

... [6, 5, 13, 16, 1]])

>>> swapped\_coords = swap(coords)

The example demonstrates the issue. The returned swapped\_coords are expected to have swapped

x and y coordinates in each of the rows.

"""

coords[:, 0], coords[:, 1] = coords[:, 1], coords[:, 0]

coords[:, 2], coords[:, 3] = coords[:, 3], coords[:, 2]

return coords

import numpy as np

coords = np.array([[10, 5, 15, 6, 0],

[11, 3, 13, 6, 0],

[5, 3, 13, 6, 1],

[4, 4, 13, 6, 1],

[6, 5, 13, 16, 1]])

swapped\_coords = swap(coords)

print(swapped\_coords)

# Exercise 3

Answer:

To fix these issues:

1. Convert the string values to floats when reading the CSV file.
2. Correct the order of the axes in the plot function.

import csv

import numpy as np

import matplotlib.pyplot as plt

def plot\_data(csv\_file\_path: str):

"""

This code plots the precision-recall curve based on data from a .csv file,

where precision is on the x-axis and recall is on the y-axis.

It it not so important right now what precision and recall means.

:param csv\_file\_path: The CSV file containing the data to plot.

"""

# load data

results = []

with open(csv\_file\_path) as result\_csv:

csv\_reader = csv.reader(result\_csv, delimiter=',')

next(csv\_reader) # Skip the header

for row in csv\_reader:

results.append([float(val) for val in row]) # Convert strings to floats

results = np.array(results) # Convert list of lists to numpy array

# plot precision-recall curve

plt.plot(results[:, 1], results[:, 0]) # Corrected: recall on x-axis, precision on y-axis

plt.ylim([-0.05, 1.05])

plt.xlim([-0.05, 1.05])

plt.xlabel('Recall')

plt.ylabel('Precision')

plt.show()

# Example usage

f = open("data\_file.csv", "w")

w = csv.writer(f)

\_ = w.writerow(["precision", "recall"])

w.writerows([[0.013,0.951],

[0.376,0.851],

[0.441,0.839],

[0.570,0.758],

[0.635,0.674],

[0.721,0.604],

[0.837,0.531],

[0.860,0.453],

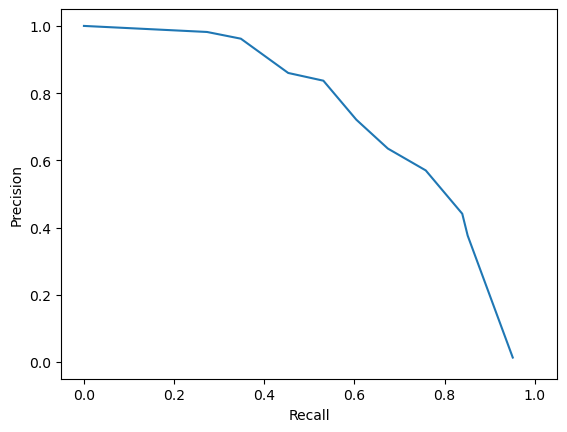
[0.962,0.348],

[0.982,0.273],

[1.0,0.0]])

f.close()

plot\_data('data\_file.csv')



# **Exercise 4**

answer:

The structural bug is caused by the mismatch between the target size and the input size when the batch size is changed to 64. This can be fixed by adjusting the size of the labels accordingly.

The cosmetic bug is that the loss values are not displayed correctly in the title of the generated images figure. This can be fixed by converting the loss values to numpy arrays and then to floats before formatting them as strings.

Here is the updated code with the bugs fixed:

import torch

import torch.nn as nn

import torchvision

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

from IPython.display import clear\_output

from IPython.display import display

import time

class Generator(nn.Module):

"""

Generator class for the GAN

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.model = nn.Sequential(

nn.Linear(100, 256),

nn.ReLU(),

nn.Linear(256, 512),

nn.ReLU(),

nn.Linear(512, 1024),

nn.ReLU(),

nn.Linear(1024, 784),

nn.Tanh(),

)

def forward(self, x):

output = self.model(x)

output = output.view(x.size(0), 1, 28, 28)

return output

class Discriminator(nn.Module):

"""

Discriminator class for the GAN

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.model = nn.Sequential(

nn.Linear(784, 1024),

nn.ReLU(),

nn.Dropout(0.3),

nn.Linear(1024, 512),

nn.ReLU(),

nn.Dropout(0.3),

nn.Linear(512, 256),

nn.ReLU(),

nn.Dropout(0.3),

nn.Linear(256, 1),

nn.Sigmoid(),

)

def forward(self, x):

x = x.view(x.size(0), 784)

output = self.model(x)

return output

# Assuming Generator and Discriminator classes are defined as provided

def train\_gan(batch\_size: int = 32, num\_epochs: int = 100, device: str = "cuda:0" if torch.cuda.is\_available() else "cpu"):

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

try:

train\_set = torchvision.datasets.MNIST(root=".", train=True, download=True, transform=transform)

except:

print("Failed to download MNIST, retrying with different URL")

torchvision.datasets.MNIST.resources = [

('https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz',

'f68b3c2dcbeaaa9fbdd348bbdeb94873'),

('https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz',

'd53e105ee54ea40749a09fcbcd1e9432'),

('https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz',

'9fb629c4189551a2d022fa330f9573f3'),

('https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz',

'ec29112dd5afa0611ce80d1b7f02629c')

]

train\_set = torchvision.datasets.MNIST(root=".", train=True, download=True, transform=transform)

train\_loader = torch.utils.data.DataLoader(train\_set, batch\_size=batch\_size, shuffle=True)

discriminator = Discriminator().to(device)

generator = Generator().to(device)

lr = 0.0001

loss\_function = nn.BCELoss()

optimizer\_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr)

optimizer\_generator = torch.optim.Adam(generator.parameters(), lr=lr)

for epoch in range(num\_epochs):

for n, (real\_samples, mnist\_labels) in enumerate(train\_loader):

real\_samples = real\_samples.to(device=device)

real\_samples\_labels = torch.ones((real\_samples.size(0), 1)).to(device=device)

latent\_space\_samples = torch.randn((real\_samples.size(0), 100)).to(device=device)

generated\_samples = generator(latent\_space\_samples)

generated\_samples\_labels = torch.zeros((generated\_samples.size(0), 1)).to(device=device)

all\_samples = torch.cat((real\_samples, generated\_samples))

all\_samples\_labels = torch.cat((real\_samples\_labels, generated\_samples\_labels))

discriminator.zero\_grad()

output\_discriminator = discriminator(all\_samples)

loss\_discriminator = loss\_function(output\_discriminator, all\_samples\_labels)

loss\_discriminator.backward()

optimizer\_discriminator.step()

latent\_space\_samples = torch.randn((real\_samples.size(0), 100)).to(device=device)

generator.zero\_grad()

generated\_samples = generator(latent\_space\_samples)

output\_discriminator\_generated = discriminator(generated\_samples)

loss\_generator = loss\_function(output\_discriminator\_generated, real\_samples\_labels)

loss\_generator.backward()

optimizer\_generator.step()

if n == batch\_size - 1:

name = f"Generate images\n Epoch: {epoch} Loss D.: {loss\_discriminator:.2f} Loss G.: {loss\_generator:.2f}"

generated\_samples = generated\_samples.detach().cpu().numpy()

fig = plt.figure()

for i in range(16):

sub = fig.add\_subplot(4, 4, 1 + i)

sub.imshow(generated\_samples[i].reshape(28, 28), cmap="gray\_r")

sub.axis('off')

fig.suptitle(name)

fig.tight\_layout()

clear\_output(wait=False)

display(fig)

train\_gan(batch\_size=32, num\_epochs=100)