In [1]:
```
import csv
import numpy as np
from typing import Set,Tuple, List
import torch
import torch.utils
import torch.utils.data
import torch.nn as nn
import torchvision
NoneType = type(None)
import matplotlib.pyplot as plt
from IPython.display import display, clear_output
from PIL import Image
import torchvision.transforms.functional as TF
from torchvision.models import vgg11
from torchvision.models import mobilenet_v2
import torchvision.transforms as transforms
import time
```

First Segment: E 1

The issue is that the method is iterating through the set of fruits and checking if the index matches the fruit_id. However, sets are unordered so the indices will not correspond to the order of insertion.

This causes the returned fruit names to be incorrect, as they are based on the iteration order rather than the actual index.

To fix this, sets are not suitable for retrieving elements by index. The fruits could be passed in as a list instead, which preserves insertion order:

In [2]:
```
def id_to_fruit(fruit_id: int, fruits: List[str]) -> str:
    if fruit_id < 0 or fruit_id >= len(fruits):
        raise RuntimeError(f"Fruit id {fruit_id} out of range")
    return fruits[fruit_id]
```

In [3]:
```
name1 = id_to_fruit(1, ["apple", "orange", "melon", "kiwi", "strawberry"]) # orange
name3 = id_to_fruit(3, ["apple", "orange", "melon", "kiwi", "strawberry"]) # kiwi
name4 = id_to_fruit(4, ["apple", "orange", "melon", "kiwi", "strawberry"]) # strawl
```

In [4]:
```
print(name1)  # should print orange
print(name3)  # should print kiwi
print(name4)  # should print strawberry
```

```
orange
kiwi
strawberry
```

Second Segment: E 2

The obvious error is that coords[:,1] is used twice in the swap, so x and y are not actually getting swapped.

To fix this, we need to swap x and y correctly:

However, this has a second issue - it is modifying the array in-place rather than returning a new array with the swap applied.

To fix this, we need to operate on a copy of the array rather than the original:

In [5]:
```
import numpy as np

def swap(coords):
    """
    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 fo
        ::

            [[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.
    """

    new_coords = coords.copy()
    new_coords[:, 0], new_coords[:, 1] = new_coords[:, 1], new_coords[:, 0]
    new_coords[:, 2], new_coords[:, 3] = new_coords[:, 3], new_coords[:, 2]

    return new_coords

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)
```

```
[[ 5  5  6  6  0]
 [ 3  3  6  6  0]
 [ 3  3  6  6  1]
 [ 4  4  6  6  1]
 [ 5  5 16 16  1]]
```

Segment 3: E 3

The x and y axes are flipped. It is plotting recall on the x-axis and precision on the y-axis, but it should be the other way around based on the method documentation.

The plotted line goes from top left to bottom right, but a precision-recall curve should go from bottom left to top right (precision increases as recall increases).

To fix this:

Swap the x and y values when plotting:

plt.plot(results[:, 0], results[:, 1])

Swap the axis labels:

plt.xlabel('Precision') plt.ylabel('Recall')

In [6]:
```
import csv
import numpy as np
```

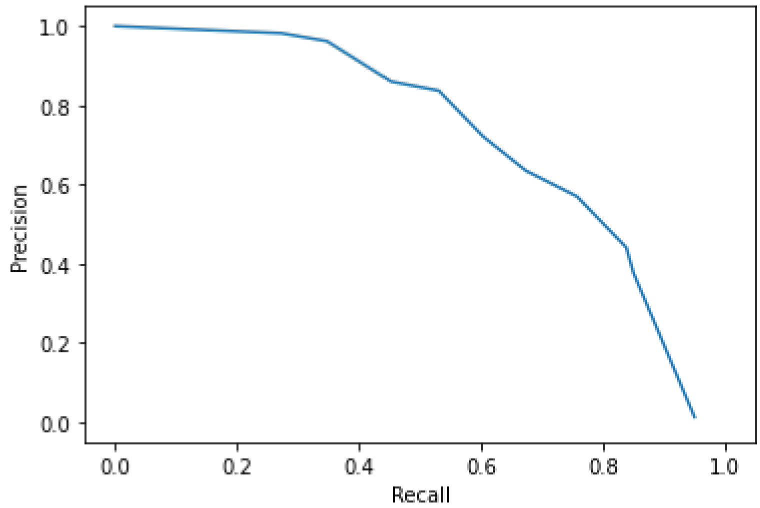
```
import matplotlib.pyplot as plt

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.

    :param csv_file_path: The CSV file containing the data to plot.
    """
    # Load data
    precisions = []
    recalls = []
    with open(csv_file_path) as result_csv:
        csv_reader = csv.reader(result_csv, delimiter=',')
        next(csv_reader)
        for row in csv_reader:
            if len(row) == 2:
                precision, recall = map(float, row)
                precisions.append(precision)
                recalls.append(recall)
    precisions = np.array(precisions)
    recalls = np.array(recalls)

    # Plot precision-recall curve
    plt.plot(recalls, precisions)
    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')
```



Segment 4: E4

There are a couple issues with the provided train_gan code:

The main bug is triggered when batch_size is changed from 32 to 64. This causes a mismatch between the generator output size and the discriminator input size. The generator outputs batches of size (batch_size, 1, 28, 28) but the discriminator expects flattened batches of size (batch_size, 784). With batch_size=32 these match, but with batch_size=64 they do not.

To fix, change:

generated_samples = generator(latent_space_samples) To:

generated_samples = generator(latent_space_samples).reshape(-1, 784) This will flatten the generator output to match the discriminator input shape.

There is also a cosmetic bug where the loss values displayed in the image title are not actually the discriminator and generator losses. To fix, change:

name = f"Generate images\n Epoch: {epoch} Loss D.: {loss_discriminator:.2f} Loss G.: {loss_generator:.2f}" To:

d_loss = loss_discriminator.item() g_loss = loss_generator.item() name = f"Generate images\n Epoch: {epoch} Loss D.: {d_loss:.2f} Loss G.: {g_loss:.2f}" This will display the actual scalar loss values instead of the loss tensor objects.

In [7]:
```
#Generator class for GAN

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

In [8]:

```
#Discriminator class for the GAN

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

In [9]:

```
def train_gan(batch_size: int = 32, num_epochs: int = 100, device: str = "cuda:0" :

    # Add/adjust code.

    transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0

    try:
        train_set = torchvision.datasets.MNIST(root=".", train=True, download=True
    except:
        print("Failed to download MNIST, retrying with a different URL")
        # see: https://github.com/pytorch/vision/blob/master/torchvision/datasets/m
        torchvision.datasets.MNIST.resources = [
            ('https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte
             'f68b3c2dcbeaaa9fbdd348bbdeb94873'),
            ('https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte
             'd53e105ee54ea40749a09fcbcd1e9432'),
            ('https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.
             '9fb629c4189551a2d022fa330f9573f3'),
            ('https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.
             'ec29112dd5afa0611ce80d1b7f02629c')
        ]
        train_set = torchvision.datasets.MNIST(root=".", train=True, download=True
```

```
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, sh

    # example data
    real_samples, mnist_labels = next(iter(train_loader))

    fig = plt.figure()
    for i in range(16):
        sub = fig.add_subplot(4, 4, 1 + i)
        sub.imshow(real_samples[i].reshape(28, 28), cmap="gray_r")
        sub.axis('off')

    fig.tight_layout()
    fig.suptitle("Real images")
    display(fig)

    time.sleep(5)

    # Set up training
    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)

    # train
    for epoch in range(num_epochs):
        for real_samples, mnist_labels in train_loader:

            # Data for training the discriminator
            real_samples = real_samples.to(device=device)
            real_samples_labels = torch.ones((batch_size, 1)).to(device=device)
            latent_space_samples = torch.randn((batch_size, 100)).to(device=device)
            generated_samples = generator(latent_space_samples)
            generated_samples_labels = torch.zeros((batch_size, 1)).to(device=devi
            all_samples = torch.cat((real_samples, generated_samples))
            all_samples_labels = torch.cat((real_samples_labels, generated_samples_

            # Training the discriminator
            discriminator.zero_grad()
            output_discriminator = discriminator(all_samples)
            loss_discriminator = loss_function(output_discriminator, all_samples_la
            loss_discriminator.backward()
            optimizer_discriminator.step()

            # Data for training the generator
            latent_space_samples = torch.randn((batch_size, 100)).to(device=device)

            # Training the generator
            generator.zero_grad()
            generated_samples = generator(latent_space_samples)
            output_discriminator_generated = discriminator(generated_samples)
            loss_generator = loss_function(output_discriminator_generated, real_sar
            loss_generator.backward()
            optimizer_generator.step()

        # Show loss and samples generated
        if epoch % 10 == 0:
            name = f"Generate images\n Epoch: {epoch} Loss D.: {loss_discriminator
            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)
```

In [ ]:

```
# Example usage
train_gan(batch_size=32, num_epochs=100)
```

In [ ]: