CS480/680, Spring 2023, Assignment 3

Designer: Chengjie Huang; Instructor: Hongyang Zhang

Released: June 26; Due: July 16, noon

Notes:

- Please save a copy of this notebook to avoid losing your changes.
- Debug your code and ensure that it can run.
- Save the output of each cell. Failure to do so may result in your coding questions not being graded.
- To accelerate the training time, you can use Google Colab and choose 'Runtime' -> 'Change runtime type' -> 'Hardware accelerator' and set 'Hardware accelerator' to 'GPU'. Question 1 may be more time-consuming, so you may need to plan ahead and leave enough time for model training.
- Your grade is independent of the test accuracy.

In [ ]:
```
# In addition to numpy, pytorch, and other standard libraries, you will nee
!pip install transformers datasets
```

Question 1: Large Language Model (? pts)

Large pre-trained language models such as GPT can be useful for many natural language tasks other than text generation. In this question, we will take a look at one of such tasks: question-answering (QA).

In QA task, the model is given some **context** text and a **question** related to the context. The model is tasked to generate the correct answer based on the context and question. For example, a context could be "Joe enjoys pizza but prefers pasta over anything else", and given a question "What's Joe's favorite food", the model should output "pasta".

In this question, we will extend and fine-tune a pre-trained large language model (GPT2) to perform question-answering task.

1.1 SQuAD Dataset (? pts)

A popular dataset for question-answering task is the Stanford Question-answering Dataset (SQuAD) ([Rajpurkar, Pranav, et al. "Squad: 100,000+ questions for machine comprehension of text." arXiv preprint arXiv:1606.05250 (2016). (https://arxiv.org/abs/1606.05250)](https://arxiv.org/abs/1606.05250)). The code below will automatically download and load the dataset. The training and validation split can be accessed with `squad_dataset['train']` and `squad_dataset['validation']` respectively.

First familiarize yourself with the format of the SQuAD dataset.
In the following cells, print the size of each split as well as one example from each split in the following format:

```
Train/validation split: 10000 samples
Sample id: 56de57394396321400ea2830
Context: Joe enjoys pizza but prefers pasta over anything else
Question: What's Joe's favorite food
Answer: pasta
```

In [ ]:
```
from datasets import load_dataset
squad_dataset = load_dataset("squad")

##########################################################################
# IMPLEMENT ME!
# raise NotImplementedError
data = squad_dataset['train'][10000]
print(f'Sample id: {data["id"]}')
print(f'Context: {data["context"]}')
print(f'Question: {data["question"]}')
print(f'Answer: {data["answers"]["text"][0]}')

data = squad_dataset['validation'][100]
print(f'Sample id: {data["id"]}')
print(f'Context: {data["context"]}')
print(f'Question: {data["question"]}')
print(f'Answer: {data["answers"]["text"][0]}')
##########################################################################
```

1.2 Extending GPT2 for question-answering task (? pts)

In this part, we will extend the GPT2 model to produce answers from the context based on the questions. To make use of the pre-trained GPT2 model, we will treat it as a feature extractor to compute token-wise feature vectors and add additional MLP layers to process the features for the QA task. These additional task-specific layers are sometimes called **"adapters"**.

Use the skeleton code below and implement the following three core components:

- [? pts] Add additional MLP layer(s) to predict the start and end location of the answer within the tokenized input.
- [? pts] Compute the location of the answer using features extracted from pre-trained GPT2.
- [? pts] Task-specific loss for question-answering.
  In this question, we formulate location prediction as a multi-class classification which can be optimized using cross-entropy loss.
  For example: given a input text of length 5 after tokenization, suppose the answer starts at token `2` and ends at token `4`, the model should predict `[0 0 1 0 0]` and `[0 0 0 0 1]` as the start and end location respectively.

Unlike image data, text input can have varying length, which makes batch training and loss computation more challenging. For simplicity, you can assume the batch size is 1 in this question. I.e., the `question` , `context` and `answer` belong to a single sample in the dataset.

```
In [3]:  import torch
         import torch.nn as nn
         from transformers import GPT2TokenizerFast, GPT2Model

         class GPT2QuestionAnswering(nn.Module):
             def __init__(self):
                 super().__init__()
                 self.tokenizer = GPT2TokenizerFast.from_pretrained('gpt2')
                 self.gpt2 = GPT2Model.from_pretrained('gpt2')
                 ###################################################################
                 # IMPLEMENT ME!
                 # Add additional layers for classifying the start and end location
                 # raise NotImplementedError
                 self.cls_start = nn.Linear(768, 1)
                 self.cls_end = nn.Linear(768, 1)
                 ###################################################################

             def forward(self, question, context, answer=None):
                 inputs = self.tokenizer(question, context, return_tensors='pt', re
                 self.inputs = inputs
                 input_ids = inputs.input_ids
                 attention_mask = inputs.attention_mask
                 if torch.cuda.is_available:
                     input_ids = input_ids.cuda()
                     attention_mask = attention_mask.cuda()
                 features = self.gpt2(input_ids=input_ids, attention_mask=attention_

                 ###################################################################
                 # IMPLEMENT ME!
                 # Compute location of the answer based on the hidden state features
                 # raise NotImplementedError
                 shape = features.shape[:2]
                 start_logits = self.cls_start( features.reshape(-1, 768) ).reshape
                 end_logits = self.cls_end( features.reshape(-1, 768) ).reshape(sha
                 ###################################################################

                 if self.training:
                     # In training mode, we want to return the loss based on the gr
                     return self.loss(start_logits, end_logits, answer, inputs.offs
                 else:
                     ###############################################################
                     # IMPLEMENT ME!
                     # In inference mode, we want to return the answer string based
                     # answer_start_index = ...
                     # answer_end_index = ...
                     answer_start_index = int(torch.argmax(start_logits, dim=-1)[0]
                     answer_end_index = int(torch.argmax(end_logits, dim=-1)[0])
                     ###############################################################
                     return self.tokenizer.decode(inputs.input_ids[0, answer_start_


             def loss(self, start_logits, end_logits, answer, offset_mapping):
                 ###################################################################
                 # IMPLEMENT ME!
                 # Compute the loss based on true answers
```

```
        # raise NotImplementedError

        # 1. Compute the start and end location of the answer
        #    Tip: use offset_mapping from the input tokenization.
        #         See https://huggingface.co/docs/transformers/main_classes
        answer_start = answer['answer_start'][0]
        answer_end = answer['answer_start'][0] + len(answer['text'][0])
        start_positions = -1
        end_positions = -1
        for i, offset in enumerate(offset_mapping[0]):
            if np.abs(offset[0]-answer_start) < 2:
                start_positions = i-1 if answer_start < offset[0] else i
            elif offset[1] >= answer_end:
                end_positions = i
                break
        if start_positions < 0:
            start_positions = 0
        start_positions = torch.tensor([start_positions]).to(start_logits.
        end_positions = torch.tensor([end_positions]).to(end_logits.device

        # print(answer['text'][0])
        # return self.tokenizer.decode(self.inputs.input_ids[0, start_posi

        # 2. Compute the cross-entropy loss between prediction and ground 
        loss_fct = nn.CrossEntropyLoss()
        start_loss = loss_fct(start_logits, start_positions)
        end_loss = loss_fct(end_logits, end_positions)
        total_loss = (start_loss + end_loss) / 2
        return total_loss

        ####################################################################
```

We can evaluate the pre-trained model's performance on the validation split. Since the model has not been adapted to the question-answering task yet, and additional untrained layers have been added, we expect the model to perform poorly.

In question-answering task, we use two evaluation metrics:

- **Exact match**: the percentage of predictions that match the ground truth answer exactly
- **F1 score**: the average overlap (in terms of tokens) between the prediction and ground truth answer

Higher values are better for both metrics. For reference, humans can achieve 77.0% exact match and 86.8% F1 score, while SOTA method achieves 90% exact match and over 95% F1 score.

In [4]:
```
def evaluate(model, dataset, metric):
    from tqdm.autonotebook import tqdm
    model = model.eval()
    preds = []
    for data in tqdm(dataset['validation']):
        preds.append(dict(id=data['id'], prediction_text=model(data['quest
    references = [dict(answers=data['answers'], id=data['id']) for data in
    return metric.compute(predictions=preds, references=references)
```

In [5]:
```python
from datasets import load_metric

squad_metric = load_metric('squad')
model = GPT2QuestionAnswering()
if torch.cuda.is_available:
    model = model.cuda()

evaluate(model, squad_dataset, squad_metric)
```

```
/tmp/ipykernel_292962/1889257040.py:3: FutureWarning: load_metric is depr
ecated and will be removed in the next major version of datasets. Use 'ev
aluate.load' instead, from the new library 🤗 Evaluate: https://huggingf
ace.co/docs/evaluate (https://huggingface.co/docs/evaluate)
  squad_metric = load_metric('squad')
```

```
100%                                            10570/10570 [01:47<00:00,

                                                102.63it/s]
```

```
2023-06-21 13:51:01.790799: I tensorflow/core/platform/cpu_feature_guard.
cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Netwo
rk Library (oneDNN) to use the following CPU instructions in performance-
critical operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropria
te compiler flags.
2023-06-21 13:51:02.599919: W tensorflow/compiler/xla/stream_executor/pla
tform/default/dso_loader.cc:64] Could not load dynamic library 'libnvinfe
r.so.7'; dlerror: libnvinfer.so.7: cannot open shared object file: No suc
h file or directory; LD_LIBRARY_PATH: /usr/local/nvidia/lib:/usr/local/nv
idia/lib64
2023-06-21 13:51:02.600011: W tensorflow/compiler/xla/stream_executor/pla
tform/default/dso_loader.cc:64] Could not load dynamic library 'libnvinfe
r_plugin.so.7'; dlerror: libnvinfer_plugin.so.7: cannot open shared objec
t file: No such file or directory; LD_LIBRARY_PATH: /usr/local/nvidia/li
b:/usr/local/nvidia/lib64
2023-06-21 13:51:02.600022: W tensorflow/compiler/tf2tensorrt/utils/py_ut
ils.cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you
would like to use Nvidia GPU with TensorRT, please make sure the missing
libraries mentioned above are installed properly.
```

Out[5]:
```
{'exact_match': 0.02838221381267739, 'f1': 3.4842183049907534}
```

1.3 Fine-tuning GPT2 on Squad (? pts)

To adapt a pre-trained model to a specific downstream task (in this case the question-answering task), a common technique is to **"fine-tune"** the model. Fine-tuning simply means training the model using task-specific data, typically with shorter epochs and smaller learning rates. Certain part of the model (e.g., pre-trained layers or early layers) can also be frozen, meaning the weights are not updated during training.

In this part, we will fine-tune the model on SQuAD dataset.
In the cell below, implement the training loop and record the loss values in a list to be plotted later.

Note: due to the size of the model and dataset, it is not required to train for too many iterations since this question will not be graded purely based on the performance of your model.

Optional: freezing GPT2 layers

Depending on the additional layers you added to the network in the previous part, you may choose to freeze the pre-trained GPT2 layers during fine-tuning. In PyTorch, this can be achieved by setting `requires_grad=False` for the layers of interest. You are encouraged to try both and note your observations.

Optional: gradient accumulation

The model from previous question is not suitable for batch training, which could increase the stochasiticy of the training process and make convergence slower. One way to circumvent this problem is via gradient accumulation, wherein the gradient is accumulated for multiple iterations before the weights are updated, which increases the effective batch size. In PyTorch, this can be implemented by accumulating the loss and performing `zero_grad()` , `barkward()` , and `step()` every few iterations instead of every iteration

In [7]:

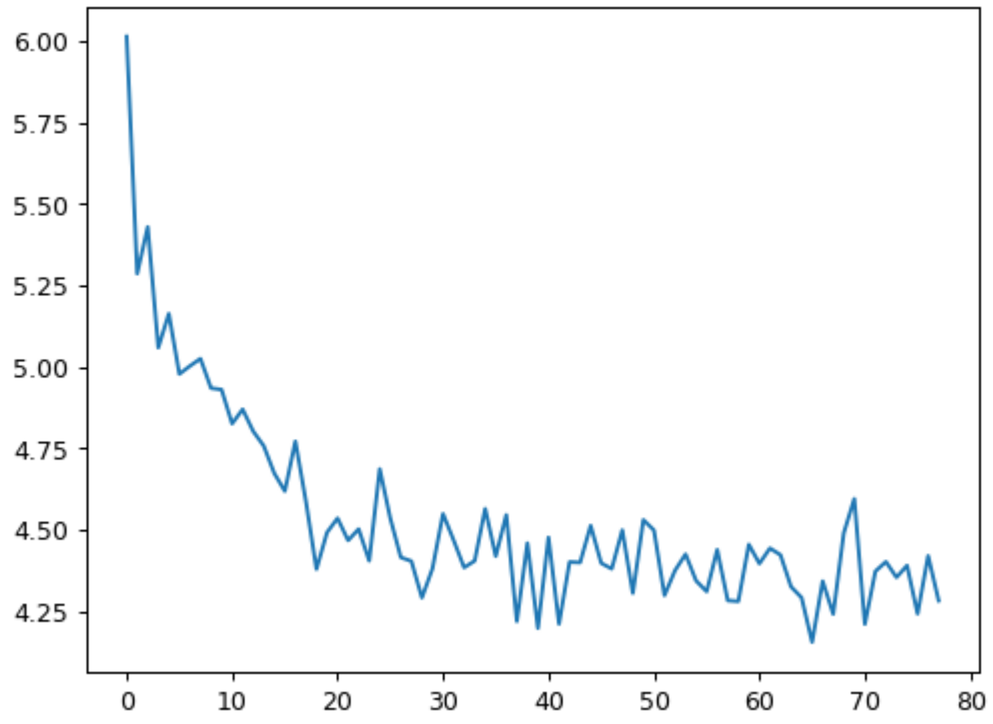
```
##########################################################################
# IMPLEMENT ME!
# raise NotImplementedError

import numpy as np
from tqdm.autonotebook import tqdm
num_epochs = 1
batch_size = 64
losses = []
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
model = model.train()
for e in range(num_epochs):
    indices = np.arange(len(squad_dataset['train']))
    np.random.shuffle(indices)
    pbar = tqdm(indices[:5000])
    for i, idx in enumerate(pbar):
        data = squad_dataset['train'][int(idx)]
        if i % batch_size == 0:
            if i > 0:
                loss = loss / batch_size
                losses.append(loss.item())
                pbar.set_postfix(loss=loss.item())
                loss.backward()
                optimizer.step()
            optimizer.zero_grad()
            loss = model(data['question'], data['context'], data['answers'
        else:
            loss += model(data['question'], data['context'], data['answers
```

```
100%                                        5000/5000 [02:13<00:00, 20.17it/s,
                                            loss=4.28]
```

Plot the loss values over the iterations.

In [8]:
```
import matplotlib.pyplot as plt
plt.figure(dpi=90)
plt.plot(losses)
plt.show()
```



Evaluate the model performance after fine-tuning. You should see a much higher score compared to the previous results.

Note: this question will not be graded based on the performance of your model. However, together with the code and the loss plot in the previous parts, the performance will be used to judge if the fine-tuning is implemented properly.

In [9]:
```
evaluate(model, squad_dataset, squad_metric)
```

```
100%                                      10570/10570 [01:48<00:00, 98.36it/s]
```

Out[9]:
```
{'exact_match': 2.5165562913907285, 'f1': 7.227697924708394}
```

Try the model using your own text and question. Can it give you the correct answer?

In [10]:
```
model('What\'s Joe\'s favorite food', 'Joe enjoys pizza but prefers pasta 
```

Out[10]:
```
"What's Joe's favorite foodJoe enjoys pizza"
```

Question 2: GAN (? pts)

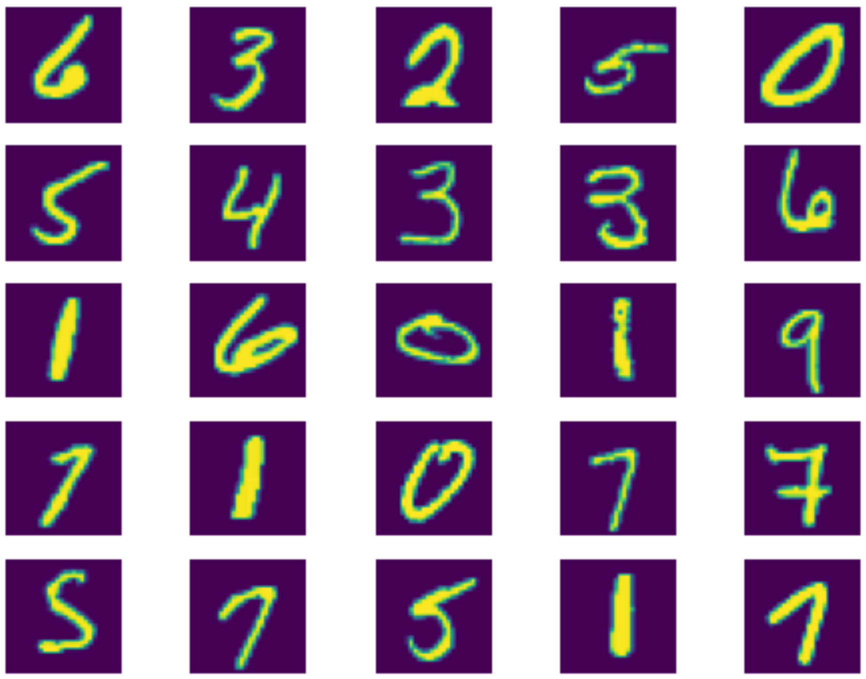
In this question, you will be designing and training a GAN to generate digits.

Training a GAN for image generation can be computationally demanding. Luckily, MNIST dataset provides 28x28 images of handwritten digits, allowing a GAN to be trained more

In [11]:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from torchvision.datasets import MNIST

mnist = MNIST(root='.', download=True)
images = np.stack([data[0] for data in mnist]).astype(np.float32)
images = images / 128 - 1    # normalize between -1 and 1
plt.figure(dpi=90)
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(images[np.random.choice(len(images))])
    plt.axis('off')
plt.show()
```



2.1 Generator and Discriminator (? pts)

In this part, you need to implement a generator and discriminator model using the skeleton code below. Recall that

- The **generator** takes a randomly sampled noise z as input and outputs an image with the same size as the dataset
- The **discriminator** takes an image as input and performs a binary classification

In this case, both the generator and discriminator should be convolutional neural network. You may not use a pretrained network, but other design decisions such as the depth and width of the network are up to you. Depending on the resources available to you, you may choose to implement a small network.

In [12]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

##########################################################################
# Part of the sample solution. Not provided to the student
def conv_layer(in_channels, out_channels, kernel_size=3, stride=1, padding
               dropout=False, output=False, deconv=False):
    layer = nn.Conv2d if not deconv else nn.ConvTranspose2d
    modules = [layer(in_channels, out_channels, kernel_size, stride, paddi
    if not output:
        modules.extend([nn.BatchNorm2d(out_channels), nn.LeakyReLU()])
    if dropout:
        modules.append(nn.Dropout(0.5))
    return nn.Sequential(*modules)
##########################################################################

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        ##################################################################
        # IMPLEMENT ME!
        # raise NotImplementedError
        layer_conf = [
            (1, 16, 3, 1, 0),
            (16, 32, 3, 1, 0),
            (32, 64, 3, 2, 0),
            (64, 32, 3, 2, 0),
            (32, 16, 3, 1, 0),
            (16, 1, 3, 1, 0, False, True)
        ]
        self.layers = nn.Sequential(*[conv_layer(*conf) for conf in layer_
        ##################################################################

    def forward(self, x):
        ##################################################################
        # IMPLEMENT ME!
        # raise NotImplementedError
        x = x.unsqueeze(1)
        x = self.layers(x)
        return x.reshape(x.shape[0], 1)
        ##################################################################


class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        ##################################################################
        # IMPLEMENT ME!
        # raise NotImplementedError
        layer_conf = [
            (1, 16, 3, 2, 0, True, False, True),
            (16, 32, 3, 2, 0, False, False, True),
            (32, 64, 3, 2, 0, False, False, True),
            (64, 32, 2, 2, 0, False, False, True),
            (32, 16, 4, 1, 1),
            (16, 1, 4, 1, 1, False, True)
        ]
        self.layers = nn.Sequential(*[conv_layer(*conf) for conf in layer_
        ###################################################################

    def forward(self, x):
        ###################################################################
        # IMPLEMENT ME!
        # raise NotImplementedError
        x = x.reshape(x.shape[0], 1, 1, 1)
        x = self.layers(x).squeeze(1)
        return torch.tanh(x)
        ###################################################################

gen = Generator()
disc = Discriminator()
if torch.cuda.is_available:
    gen = gen.cuda()
    disc = disc.cuda()
```

2.2 Generate image samples from generator (? pts)

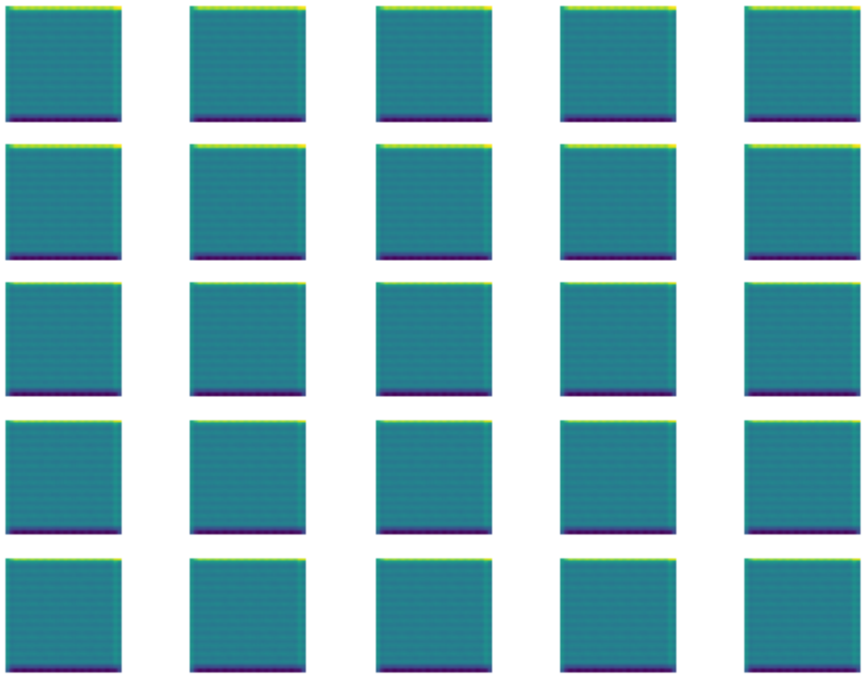
During the training and inference, the generator needs to generate batch of images from random noise. Implement the generation function below.

Note: This function will later be used in training, therefore you need to be careful and avoid cutting off the gradient accidentally

In [13]:
```
def generate_samples(model, num_samples):
    #####################################################################
    # IMPLEMENT ME!
    # raise NotImplementedError
    noise = torch.normal(torch.zeros(num_samples,1), torch.ones(num_sample
    if torch.cuda.is_available:
        noise = noise.cuda()
    return model(noise)
    #####################################################################
```

Without any training, the samples generated by the generator does not resemble any digit in the dataset.

In [14]:
```
gen.eval()
samples = generate_samples(gen, 25).detach().cpu().numpy()
plt.figure(dpi=90)
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.show()
```



2.3 GAN training algorithm (? pts)

In this part, you will implement the GAN training algorithm, which involves alternating the training of discriminator and generator. Complete the skeleton code below. For more information, refer to [Goodfellow, Ian, et al. "Generative adversarial networks." Communications of the ACM 63.11 (2020): 139-144. (https://arxiv.org/abs/1406.2661)](https://arxiv.org/abs/1406.2661).

In [15]:
```
def train_gan(gen, disc, images, num_epochs, batch_size):
    from tqdm.autonotebook import tqdm
    from torch.utils.data import DataLoader

    losses_gen = []
    losses_disc = []

    gen.train()
    disc.train()
    loader = DataLoader(images, batch_size=batch_size, shuffle=True)

    ##########################################################################
    # IMPLEMENT ME!

    # 1. [? pts] Build optimizer for each model and choose an appropriate
    #    You may specify different optimizers or learning rates for generat
    #    to balance the training loss and avoid one model overpowering the
    #    Ideally we would like to reach an equilibrium between the generato
    optimizer_gen = torch.optim.Adam(gen.parameters(), lr=1e-4)
    optimizer_disc = torch.optim.Adam(disc.parameters(), lr=1e-5)

    pbar = tqdm(range(num_epochs))
    for e in pbar:
        for i, data_real in enumerate(loader):
            if torch.cuda.is_available:
                data_real = data_real.cuda()

            # 2. Update discriminator
            # 2.1. [? pts] Unfreeze discriminator
            disc.train()
            gen.eval()
            for param in disc.parameters():
                param.requires_grad = True

            # 2.2. [? pts] Construct input for discriminator
            #      This should contain both real and fake samples
            data_fake = generate_samples(gen, batch_size).detach().clone()
            data = torch.cat([data_real, data_fake])

            # 2.3. [? pts] Construct training labels for discriminator
            labels = torch.cat([torch.ones(len(data_real), 1), torch.zeros

            if torch.cuda.is_available:
                labels = labels.cuda()

            # 2.4. [? pts] Discrminator training
            #      This should include loss computation and weight updates
            optimizer_disc.zero_grad()
            logits = disc(data)
#             loss_disc = F.binary_cross_entropy(torch.sigmoid(logits), lab
            loss_disc = F.mse_loss(logits, labels)
            loss_disc.backward()
            optimizer_disc.step()
            losses_disc.append(loss_disc.item())
            # 3. Update generator
            # 3.1. [? pts] Freeze discriminator
            disc.eval()
            gen.train()
            for param in disc.parameters():
                param.requires_grad = False

            # 3.2. [? pts] Construct input for the discriminator
            data_fake = generate_samples(gen, 2*batch_size)

            # 3.3. [? pts] Construct training labels for the discriminator
            labels = torch.ones(len(data_fake), 1)
            if torch.cuda.is_available:
                labels = labels.cuda()

            # 3.4. [? pts] Generator training
            #      This should include loss computation and weight updates
            optimizer_gen.zero_grad()
            logits = disc(data_fake)
            loss_gen = F.mse_loss(logits, labels)
#             loss_gen = F.binary_cross_entropy(torch.sigmoid(logits), labe
            loss_gen.backward()
            optimizer_gen.step()
            losses_gen.append(loss_gen.item())
        ########################################################################

            pbar.set_postfix(loss_gen=losses_gen[-1], loss_disc=losses_dis
    return losses_gen, losses_disc
```

You may change the number of epochs and batch size based on the time and resources available to you.

In [16]:
```
losses_gen, losses_disc = train_gan(gen, disc, images, num_epochs=10, batc
```

```
100%                                10/10 [01:25<00:00, 8.50s/it, loss_disc=0.288,
                                    loss_gen=0.222]
```

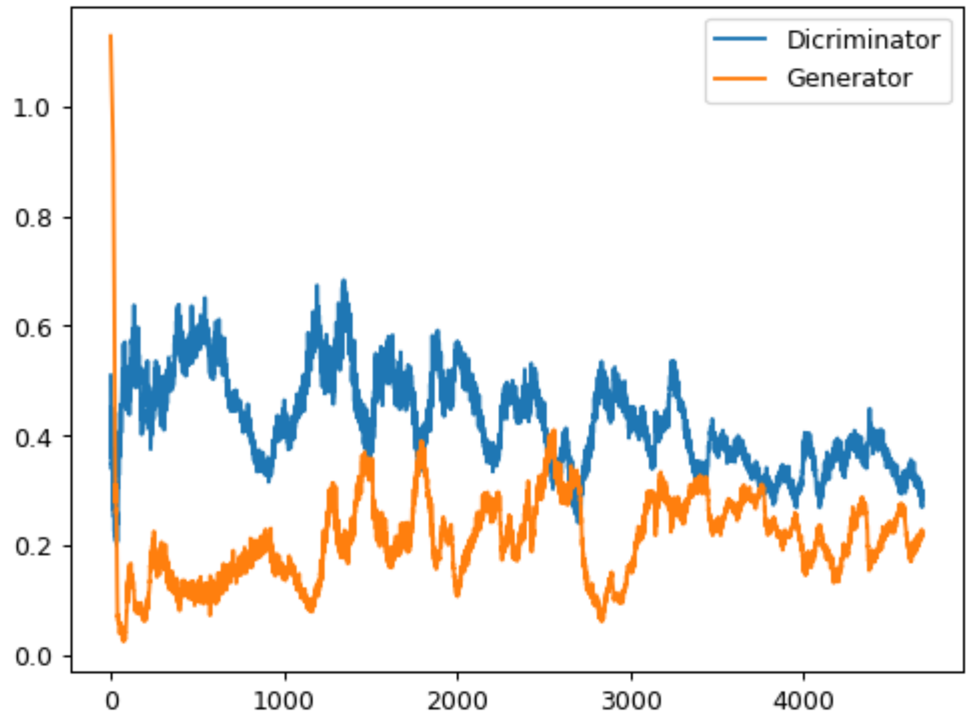
2.4 Observations and Analysis (? pts)

[? pts] Plot the losses for the generator and discriminator. Do you see any problem with the training process?
Do you think your GAN has converged? Why and why not?

Answer:

In [17]:

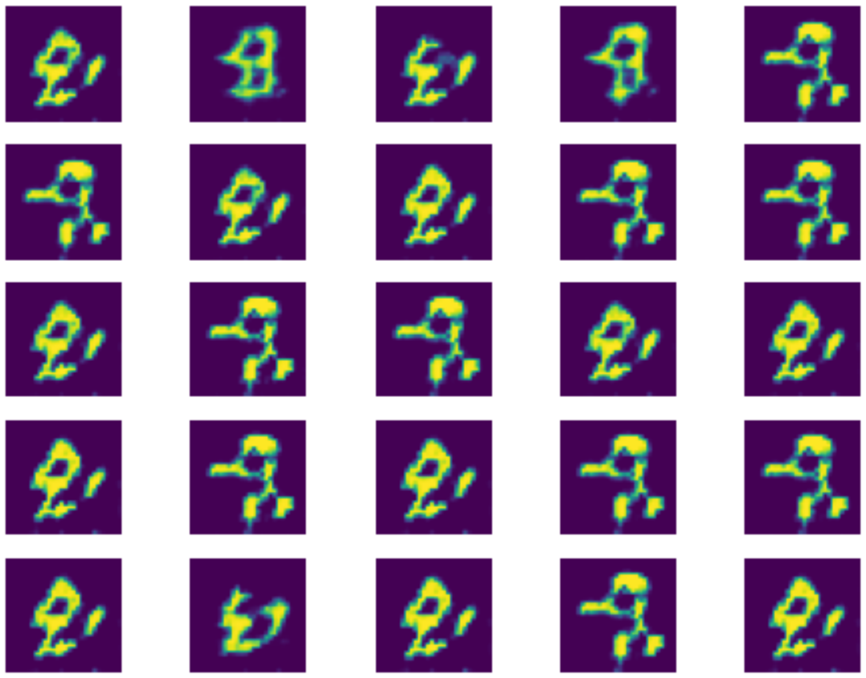
```
plt.figure(dpi=90)
plt.plot(losses_disc, label='Dicriminator')
plt.plot(losses_gen, label='Generator')
plt.legend()
plt.show()
```



[? pts] Visualize some images generated by your GAN. Do you see any problem with the samples generated by your GAN?
Do you think your GAN has mode collapse problem? Why and why not?

Answer: Likely yes, the student's GAN will likely repeatedly generate the same images

```
In [18]:  gen.eval()
          samples = generate_samples(gen, 25).detach().cpu().numpy()
          plt.figure(dpi=90)
          for i in range(25):
              plt.subplot(5, 5, i+1)
              plt.imshow(samples[i])
              plt.axis('off')
          plt.show()
```



```
In [ ]:
```