Colab FAQ

For some basic overview and features offered in Colab notebooks, check out: Overview of Colaboratory Features

You need to use the colab GPU for this assignmentby selecting:

Runtime → **Change runtime type** → **Hardware Accelerator: GPU**

Setup PyTorch

All files are stored at /content/csc421/a4/ folder

```
######################################################################
# Setup python environment and change the current working directory
######################################################################
!pip install torch torchvision
!pip install imageio

!pip install matplotlib

%mkdir -p /content/csc413/a4/
%cd /content/csc413/a4
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.10.0+cu111)
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-packages (0.11.1+cu111)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from torch) (3.10.0.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from torchvision) (1.21.5)
Requirement already satisfied: pillow!=8.3.0,>=5.3.0 in /usr/local/lib/python3.7/dist-packages (from torchvision) (7.1.2)
Requirement already satisfied: imageio in /usr/local/lib/python3.7/dist-packages (2.4.1)
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages (from imageio) (7.1.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from imageio) (1.21.5)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (3.2.2)
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (1.21.5)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib) (1.4.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!
```

```
=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from
matplotlib) (3.0.7)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (0.11.0)
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from kiwisolver>=1.0.1-
>matplotlib) (3.10.0.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.1-
>matplotlib) (1.15.0)
/content/csc413/a4
```

Helper code

Utility functions

```
import os

import numpy as np
import matplotlib.pyplot as plt

import torch
from torch import nn
from torch.nn import Parameter
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import transforms

from six.moves.urllib.request import urlretrieve
import tarfile

import imageio
from urllib.error import URLError
from urllib.error import HTTPError


def get_file(fname,
             origin,
             untar=False,
             extract=False,
             archive_format='auto',
             cache_dir='data'):
    datadir = os.path.join(cache_dir)
    if not os.path.exists(datadir):
```

```
        os.makedirs(datadir)

    if untar:
        untar_fpath = os.path.join(datadir, fname)
        fpath = untar_fpath + '.tar.gz'
    else:
        fpath = os.path.join(datadir, fname)

    print(fpath)
    if not os.path.exists(fpath):
        print('Downloading data from', origin)

        error_msg = 'URL fetch failure on {}: {} -- {}'
        try:
            try:
                urlretrieve(origin, fpath)
            except URLError as e:
                raise Exception(error_msg.format(origin, e.errno,
e.reason))
            except HTTPError as e:
                raise Exception(error_msg.format(origin, e.code,
e.msg))
        except (Exception, KeyboardInterrupt) as e:
            if os.path.exists(fpath):
                os.remove(fpath)
            raise

    if untar:
        if not os.path.exists(untar_fpath):
            print('Extracting file.')
            with tarfile.open(fpath) as archive:
                archive.extractall(datadir)
        return untar_fpath

    return fpath


class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self


def to_var(tensor, cuda=True):
    """Wraps a Tensor in a Variable, optionally placing it on the GPU.

        Arguments:
            tensor: A Tensor object.
            cuda: A boolean flag indicating whether to use the GPU.
```

```python
        Returns:
            A Variable object, on the GPU if cuda==True.
    """
    if cuda:
        return Variable(tensor.cuda())
    else:
        return Variable(tensor)


def to_data(x):
    """Converts variable to numpy."""
    if torch.cuda.is_available():
        x = x.cpu()
    return x.data.numpy()


def create_dir(directory):
    """Creates a directory if it doesn't already exist.
    """
    if not os.path.exists(directory):
        os.makedirs(directory)


def gan_checkpoint(iteration, G, D, opts):
    """Saves the parameters of the generator G and discriminator D.
    """
    G_path = os.path.join(opts.checkpoint_dir, 'G.pkl')
    D_path = os.path.join(opts.checkpoint_dir, 'D.pkl')
    torch.save(G.state_dict(), G_path)
    torch.save(D.state_dict(), D_path)

def load_checkpoint(opts):
    """Loads the generator and discriminator models from checkpoints.
    """
    G_path = os.path.join(opts.load, 'G.pkl')
    D_path = os.path.join(opts.load, 'D_.pkl')

    G = DCGenerator(noise_size=opts.noise_size,
conv_dim=opts.g_conv_dim, spectral_norm=opts.spectral_norm)
    D = DCDiscriminator(conv_dim=opts.d_conv_dim)

    G.load_state_dict(torch.load(G_path, map_location=lambda storage,
loc: storage))
    D.load_state_dict(torch.load(D_path, map_location=lambda storage,
loc: storage))

    if torch.cuda.is_available():
        G.cuda()
```

```python
        D.cuda()
        print('Models moved to GPU.')

    return G, D


def merge_images(sources, targets, opts):
    """Creates a grid consisting of pairs of columns, where the first
column in
    each pair contains images source images and the second column in
each pair
    contains images generated by the CycleGAN from the corresponding
images in
    the first column.
    """
    _, _, h, w = sources.shape
    row = int(np.sqrt(opts.batch_size))
    merged = np.zeros([3, row * h, row * w * 2])
    for (idx, s, t) in (zip(range(row ** 2), sources, targets, )):
        i = idx // row
        j = idx % row
        merged[:, i * h:(i + 1) * h, (j * 2) * h:(j * 2 + 1) * h] = s
        merged[:, i * h:(i + 1) * h, (j * 2 + 1) * h:(j * 2 + 2) * h]
= t
    return merged.transpose(1, 2, 0)


def generate_gif(directory_path, keyword=None):
    images = []
    for filename in sorted(os.listdir(directory_path)):
        if filename.endswith(".png") and (keyword is None or keyword
in filename):
            img_path = os.path.join(directory_path, filename)
            print("adding image {}".format(img_path))
            images.append(imageio.imread(img_path))

    if keyword:
        imageio.mimsave(
            os.path.join(directory_path,
'anim_{}.gif'.format(keyword)), images)
    else:
        imageio.mimsave(os.path.join(directory_path, 'anim.gif'),
images)


def create_image_grid(array, ncols=None):
    """
    """
    num_images, channels, cell_h, cell_w = array.shape
```

```
    if not ncols:
        ncols = int(np.sqrt(num_images))
    nrows = int(np.math.floor(num_images / float(ncols)))
    result = np.zeros((cell_h * nrows, cell_w * ncols, channels),
dtype=array.dtype)
    for i in range(0, nrows):
        for j in range(0, ncols):
            result[i * cell_h:(i + 1) * cell_h, j * cell_w:(j + 1) *
cell_w, :] = array[i * ncols + j].transpose(1, 2,

0)

    if channels == 1:
        result = result.squeeze()
    return result


def gan_save_samples(G, fixed_noise, iteration, opts):
    generated_images = G(fixed_noise)
    generated_images = to_data(generated_images)

    grid = create_image_grid(generated_images)

    # merged = merge_images(X, fake_Y, opts)
    path = os.path.join(opts.sample_dir, 'sample-
{:06d}.png'.format(iteration))
    imageio.imwrite(path, grid)
    print('Saved {}'.format(path))
```

Data loader

```
def get_emoji_loader(emoji_type, opts):
    """Creates training and test data loaders.
    """
    transform = transforms.Compose([
                    transforms.Scale(opts.image_size),
                    transforms.ToTensor(),
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))
                ])

    train_path = os.path.join('data/emojis', emoji_type)
    test_path = os.path.join('data/emojis',
'Test_{}'.format(emoji_type))

    train_dataset = datasets.ImageFolder(train_path, transform)
    test_dataset = datasets.ImageFolder(test_path, transform)

    train_dloader = DataLoader(dataset=train_dataset,
batch_size=opts.batch_size, shuffle=True,
```

```
num_workers=opts.num_workers)
    test_dloader = DataLoader(dataset=test_dataset,
batch_size=opts.batch_size, shuffle=False,
num_workers=opts.num_workers)

    return train_dloader, test_dloader
```

Training and evaluation code

```
def print_models(G_XtoY, G_YtoX, D_X, D_Y):
    """Prints model information for the generators and discriminators.
    """
    print("                 G                     ")
    print("---------------------------------------")
    print(G_XtoY)
    print("---------------------------------------")

    print("                  D                    ")
    print("---------------------------------------")
    print(D_X)
    print("---------------------------------------")


def create_model(opts):
    """Builds the generators and discriminators.
    """
    ### GAN
    G = DCGenerator(noise_size=opts.noise_size,
conv_dim=opts.g_conv_dim, spectral_norm=opts.spectral_norm)
    D = DCDiscriminator(conv_dim=opts.d_conv_dim,
spectral_norm=opts.spectral_norm)

    print_models(G, None, D, None)

    if torch.cuda.is_available():
        G.cuda()
        D.cuda()
        print('Models moved to GPU.')
    return G, D

def train(opts):
    """Loads the data, creates checkpoint and sample directories, and
starts the training loop.
    """

    # Create train and test dataloaders for images from the two
domains X and Y
    dataloader_X, test_dataloader_X =
get_emoji_loader(emoji_type=opts.X, opts=opts)
```

```
    # Create checkpoint and sample directories
    create_dir(opts.checkpoint_dir)
    create_dir(opts.sample_dir)

    # Start training
    if opts.least_squares_gan:
        G, D = gan_training_loop_leastsquares(dataloader_X,
test_dataloader_X, opts)
    else:
        G, D = gan_training_loop_regular(dataloader_X,
test_dataloader_X, opts)

    return G, D

def print_opts(opts):
    """Prints the values of all command-line arguments.
    """
    print('=' * 80)
    print('Opts'.center(80))
    print('-' * 80)
    for key in opts.__dict__:
        if opts.__dict__[key]:
            print('{:>30}: {:<30}'.format(key,
opts.__dict__[key]).center(80))
    print('=' * 80)
```

Your code for generators and discriminators

Helper modules

```
def sample_noise(batch_size, dim):
    """
    Generate a PyTorch Tensor of uniform random noise.

    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.

    Output:
    - A PyTorch Tensor of shape (batch_size, dim, 1, 1) containing
uniform
      random noise in the range (-1, 1).
    """
    return to_var(torch.rand(batch_size, dim) * 2 -
1).unsqueeze(2).unsqueeze(3)


def upconv(in_channels, out_channels, kernel_size, stride=2,
padding=2, batch_norm=True, spectral_norm=False):
```

```
    """Creates a upsample-and-convolution layer, with optional batch
normalization.
    """
    layers = []
    if stride>1:
        layers.append(nn.Upsample(scale_factor=stride))
    conv_layer = nn.Conv2d(in_channels=in_channels,
out_channels=out_channels, kernel_size=kernel_size, stride=1,
padding=padding, bias=False)
    if spectral_norm:
        layers.append(SpectralNorm(conv_layer))
    else:
        layers.append(conv_layer)
    if batch_norm:
        layers.append(nn.BatchNorm2d(out_channels))
    return nn.Sequential(*layers)


def conv(in_channels, out_channels, kernel_size, stride=2, padding=2,
batch_norm=True, init_zero_weights=False, spectral_norm=False):
    """Creates a convolutional layer, with optional batch
normalization.
    """
    layers = []
    conv_layer = nn.Conv2d(in_channels=in_channels,
out_channels=out_channels, kernel_size=kernel_size, stride=stride,
padding=padding, bias=False)
    if init_zero_weights:
        conv_layer.weight.data = torch.randn(out_channels,
in_channels, kernel_size, kernel_size) * 0.001

    if spectral_norm:
        layers.append(SpectralNorm(conv_layer))
    else:
        layers.append(conv_layer)

    if batch_norm:
        layers.append(nn.BatchNorm2d(out_channels))
    return nn.Sequential(*layers)


class ResnetBlock(nn.Module):
    def __init__(self, conv_dim):
        super(ResnetBlock, self).__init__()
        self.conv_layer = conv(in_channels=conv_dim,
out_channels=conv_dim, kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        out = x + self.conv_layer(x)
        return out
```

DCGAN

Spectral Norm class

```
def l2normalize(v, eps=1e-12):
    return v / (v.norm() + eps)


class SpectralNorm(nn.Module):
    def __init__(self, module, name='weight', power_iterations=1):
        super(SpectralNorm, self).__init__()
        self.module = module
        self.name = name
        self.power_iterations = power_iterations
        if not self._made_params():
            self._make_params()

    def _update_u_v(self):
        u = getattr(self.module, self.name + "_u")
        v = getattr(self.module, self.name + "_v")
        w = getattr(self.module, self.name + "_bar")

        height = w.data.shape[0]
        for _ in range(self.power_iterations):
            v.data = l2normalize(torch.mv(torch.t(w.view(height,-
1).data), u.data))
            u.data = l2normalize(torch.mv(w.view(height,-1).data,
v.data))

        # sigma = torch.dot(u.data, torch.mv(w.view(height,-1).data,
v.data))
        sigma = u.dot(w.view(height, -1).mv(v))
        setattr(self.module, self.name, w / sigma.expand_as(w))

    def _made_params(self):
        try:
            u = getattr(self.module, self.name + "_u")
            v = getattr(self.module, self.name + "_v")
            w = getattr(self.module, self.name + "_bar")
            return True
        except AttributeError:
            return False

    def _make_params(self):
        w = getattr(self.module, self.name)

        height = w.data.shape[0]
        width = w.view(height, -1).data.shape[1]

        u = Parameter(w.data.new(height).normal_(0, 1),
```

```
requires_grad=False)
        v = Parameter(w.data.new(width).normal_(0, 1),
requires_grad=False)
        u.data = l2normalize(u.data)
        v.data = l2normalize(v.data)
        w_bar = Parameter(w.data)

        del self.module._parameters[self.name]

        self.module.register_parameter(self.name + "_u", u)
        self.module.register_parameter(self.name + "_v", v)
        self.module.register_parameter(self.name + "_bar", w_bar)

    def forward(self, *args):
        self._update_u_v()
        return self.module.forward(*args)
```

[Your Task] GAN generator

```
class DCGenerator(nn.Module):
    def __init__(self, noise_size, conv_dim, spectral_norm=False):
        super(DCGenerator, self).__init__()

        self.conv_dim = conv_dim
        ###########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ###########################################

        self.linear_bn = upconv(in_channels=noise_size,
out_channels=conv_dim * 4, kernel_size=5, stride=4)
        self.upconv1 = upconv(in_channels=conv_dim * 4,
out_channels=conv_dim * 2, kernel_size=5)
        self.upconv2 = upconv(in_channels=conv_dim * 2,
out_channels=conv_dim, kernel_size=5)
        self.upconv3 = upconv(in_channels=conv_dim, out_channels=3,
kernel_size=5, stride=2, spectral_norm=spectral_norm)

    def forward(self, z):
        """Generates an image given a sample of random noise.

            Input
            -----
                z: BS x noise_size x 1 x 1   -->  BSx100x1x1 (during
training)

            Output
            ------
                out: BS x channels x image_width x image_height  -->
BSx3x32x32 (during training)
        """
```

```
        batch_size = z.size(0)

        out = F.relu(self.linear_bn(z)).view(-1, self.conv_dim*4, 4,
4)    # BS x 128 x 4 x 4
        out = F.relu(self.upconv1(out))  # BS x 64 x 8 x 8
        out = F.relu(self.upconv2(out))  # BS x 32 x 16 x 16
        out = F.tanh(self.upconv3(out))  # BS x 3 x 32 x 32

        out_size = out.size()
        if out_size != torch.Size([batch_size, 3, 32, 32]):
            raise ValueError("expect {} x 3 x 32 x 32, but get
{}".format(batch_size, out_size))
        return out
```

GAN discriminator

```
class DCDiscriminator(nn.Module):
    """Defines the architecture of the discriminator network.
       Note: Both discriminators D_X and D_Y have the same
architecture in this assignment.
    """
    def __init__(self, conv_dim=64, spectral_norm=False):
        super(DCDiscriminator, self).__init__()

        self.conv1 = conv(in_channels=3, out_channels=conv_dim,
kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.conv2 = conv(in_channels=conv_dim,
out_channels=conv_dim*2, kernel_size=5, stride=2,
spectral_norm=spectral_norm)
        self.conv3 = conv(in_channels=conv_dim*2,
out_channels=conv_dim*4, kernel_size=5, stride=2,
spectral_norm=spectral_norm)
        self.conv4 = conv(in_channels=conv_dim*4, out_channels=1,
kernel_size=5, stride=2, padding=1, batch_norm=False,
spectral_norm=spectral_norm)

    def forward(self, x):
        batch_size = x.size(0)

        out = F.relu(self.conv1(x))    # BS x 64 x 16 x 16
        out = F.relu(self.conv2(out))    # BS x 64 x 8 x 8
        out = F.relu(self.conv3(out))    # BS x 64 x 4 x 4

        out = self.conv4(out).squeeze()
        out_size = out.size()
        if out_size != torch.Size([batch_size,]):
            raise ValueError("expect {} x 1, but get
{}".format(batch_size, out_size))
        return out
```

[Your Task] GAN training loop

- Regular GAN
- Least Squares GAN

```
def gan_training_loop_regular(dataloader, test_dataloader, opts):
    """Runs the training loop.
        * Saves checkpoint every opts.checkpoint_every iterations
        * Saves generated samples every opts.sample_every iterations
    """

    # Create generators and discriminators
    G, D = create_model(opts)

    g_params = G.parameters()  # Get generator parameters
    d_params = D.parameters()  # Get discriminator parameters

    # Create optimizers for the generators and discriminators
    g_optimizer = optim.Adam(g_params, opts.lr, [opts.beta1,
opts.beta2])
    d_optimizer = optim.Adam(d_params, opts.lr * 2., [opts.beta1,
opts.beta2])

    train_iter = iter(dataloader)

    test_iter = iter(test_dataloader)

    # Get some fixed data from domains X and Y for sampling. These are
images that are held
    # constant throughout training, that allow us to inspect the
model's performance.
    fixed_noise = sample_noise(100, opts.noise_size)  # # 100 x
noise_size x 1 x 1

    iter_per_epoch = len(train_iter)
    total_train_iters = opts.train_iters

    losses = {"iteration": [], "D_fake_loss": [], "D_real_loss": [],
"G_loss": []}

    gp_weight = 1

    adversarial_loss = torch.nn.BCEWithLogitsLoss() # Use this loss
    # [Hint: you may find the folowing code helpful]


    try:
        for iteration in range(1, opts.train_iters + 1):

            # Reset data_iter for each epoch
```

```
            if iteration % iter_per_epoch == 0:
                train_iter = iter(dataloader)

            real_images, real_labels = train_iter.next()
            real_images, real_labels = to_var(real_images),
to_var(real_labels).long().squeeze()

            valid_ones =
Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0),
requires_grad=False)
            fake_ones =
Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(0.0),
requires_grad=False)

            for d_i in range(opts.d_train_iters):
                d_optimizer.zero_grad()

                # FILL THIS IN
                # 1. Compute the discriminator loss on real images
                D_real_loss = adversarial_loss(D(real_images),
valid_ones)

                # 2. Sample noise
                noise = sample_noise(real_images.shape[0],
opts.noise_size)

                # 3. Generate fake images from the noise
                fake_images = G(noise)

                # 4. Compute the discriminator loss on the fake images
                D_fake_loss = adversarial_loss(D(fake_images),
fake_ones)

                # ---- Gradient Penalty ----
                if opts.gradient_penalty:
                    alpha = torch.rand(real_images.shape[0], 1, 1, 1)
                    alpha = alpha.expand_as(real_images).cuda()
                    interp_images = Variable(alpha * real_images.data
+ (1 - alpha) * fake_images.data, requires_grad=True).cuda()
                    D_interp_output = D(interp_images)

                    gradients =
torch.autograd.grad(outputs=D_interp_output, inputs=interp_images,

grad_outputs=torch.ones(D_interp_output.size()).cuda(),
                                                    create_graph=True,
retain_graph=True)[0]
                    gradients = gradients.view(real_images.shape[0], -
1)
```

```
                    gradients_norm = torch.sqrt(torch.sum(gradients **
2, dim=1) + 1e-12)

                    gp = gp_weight * gradients_norm.mean()
                else:
                    gp = 0.0

                # --------------------------
                # 5. Compute the total discriminator loss
                D_total_loss = (D_real_loss + D_fake_loss) / 2

                D_total_loss.backward()
                d_optimizer.step()

            ###########################################
            ###          TRAIN THE GENERATOR        ###
            ###########################################

            g_optimizer.zero_grad()

            # FILL THIS IN
            # 1. Sample noise
            noise = sample_noise(real_images.shape[0],
opts.noise_size)

            # 2. Generate fake images from the noise
            fake_images = G(noise)

            # 3. Compute the generator loss
            G_loss = adversarial_loss(D(fake_images), valid_ones)

            G_loss.backward()
            g_optimizer.step()

            # Print the log info
            if iteration % opts.log_step == 0:
                losses['iteration'].append(iteration)
                losses['D_real_loss'].append(D_real_loss.item())
                losses['D_fake_loss'].append(D_fake_loss.item())
                losses['G_loss'].append(G_loss.item())
                print('Iteration [{:4d}/{:4d}] | D_real_loss: {:6.4f}
| D_fake_loss: {:6.4f} | G_loss: {:6.4f}'.format(
                    iteration, total_train_iters, D_real_loss.item(),
D_fake_loss.item(), G_loss.item()))

            # Save the generated samples
            if iteration % opts.sample_every == 0:
                gan_save_samples(G, fixed_noise, iteration, opts)
```

```
            # Save the model parameters
            if iteration % opts.checkpoint_every == 0:
                gan_checkpoint(iteration, G, D, opts)

    except KeyboardInterrupt:
        print('Exiting early from training.')
        return G, D

    plt.figure()
    plt.plot(losses['iteration'], losses['D_real_loss'],
label='D_real')
    plt.plot(losses['iteration'], losses['D_fake_loss'],
label='D_fake')
    plt.plot(losses['iteration'], losses['G_loss'], label='G')
    plt.legend()
    plt.savefig(os.path.join(opts.sample_dir, 'losses.png'))
    plt.close()
    return G, D

def gan_training_loop_leastsquares(dataloader, test_dataloader, opts):
    """Runs the training loop.
        * Saves checkpoint every opts.checkpoint_every iterations
        * Saves generated samples every opts.sample_every iterations
    """

    # Create generators and discriminators
    G, D = create_model(opts)

    g_params = G.parameters()  # Get generator parameters
    d_params = D.parameters()  # Get discriminator parameters

    # Create optimizers for the generators and discriminators
    g_optimizer = optim.Adam(g_params, opts.lr, [opts.beta1,
opts.beta2])
    d_optimizer = optim.Adam(d_params, opts.lr * 2., [opts.beta1,
opts.beta2])

    train_iter = iter(dataloader)

    test_iter = iter(test_dataloader)

    # Get some fixed data from domains X and Y for sampling. These are
images that are held
    # constant throughout training, that allow us to inspect the
model's performance.
    fixed_noise = sample_noise(100, opts.noise_size)  # # 100 x
noise_size x 1 x 1

    iter_per_epoch = len(train_iter)
```

```
    total_train_iters = opts.train_iters

    losses = {"iteration": [], "D_fake_loss": [], "D_real_loss": [],
"G_loss": []}

    # minimizes MSE loss instead of BCE
    adversarial_loss = torch.nn.MSELoss()
    gp_weight = 1

    try:
        for iteration in range(1, opts.train_iters + 1):

            # Reset data_iter for each epoch
            if iteration % iter_per_epoch == 0:
                train_iter = iter(dataloader)

            real_images, real_labels = train_iter.next()
            real_images, real_labels = to_var(real_images),
to_var(real_labels).long().squeeze()

            valid_ones =
Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0),
requires_grad=False)
            fake_ones =
Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(0.0),
requires_grad=False)
            for d_i in range(opts.d_train_iters):
                d_optimizer.zero_grad()

                # FILL THIS IN
                # 1. Compute the discriminator loss on real images
                D_real_loss = adversarial_loss(D(real_images),
valid_ones)

                # 2. Sample noise
                noise = sample_noise(real_images.shape[0],
opts.noise_size)

                # 3. Generate fake images from the noise
                fake_images = G(noise)

                # 4. Compute the discriminator loss on the fake images
                D_fake_loss = adversarial_loss(D(fake_images),
fake_ones)

                # ---- Gradient Penalty ----
                if opts.gradient_penalty:
                    alpha = torch.rand(real_images.shape[0], 1, 1, 1)
                    alpha = alpha.expand_as(real_images).cuda()
```

```
                    interp_images = Variable(alpha * real_images.data
+ (1 - alpha) * fake_images.data, requires_grad=True).cuda()
                    D_interp_output = D(interp_images)

                    gradients =
torch.autograd.grad(outputs=D_interp_output, inputs=interp_images,

grad_outputs=torch.ones(D_interp_output.size()).cuda(),
                                                    create_graph=True,
retain_graph=True)[0]
                    gradients = gradients.view(real_images.shape[0], -
1)
                    gradients_norm = torch.sqrt(torch.sum(gradients **
2, dim=1) + 1e-12)

                    gp = gp_weight * gradients_norm.mean()
                else:
                    gp = 0.0

                # --------------------------
                # 5. Compute the total discriminator loss
                D_total_loss = (D_real_loss + D_fake_loss) / 2

                D_total_loss.backward()
                d_optimizer.step()

            ###########################################
            ###          TRAIN THE GENERATOR        ###
            ###########################################

            g_optimizer.zero_grad()

            # FILL THIS IN
            # 1. Sample noise
            noise = sample_noise(real_images.shape[0],
opts.noise_size)

            # 2. Generate fake images from the noise
            fake_images = G(noise)

            # 3. Compute the generator loss
            G_loss = adversarial_loss(D(fake_images), valid_ones)

            G_loss.backward()
            g_optimizer.step()

            # Print the log info
            if iteration % opts.log_step == 0:
                losses['iteration'].append(iteration)
```

```
                losses['D_real_loss'].append(D_real_loss.item())
                losses['D_fake_loss'].append(D_fake_loss.item())
                losses['G_loss'].append(G_loss.item())
                print('Iteration [{:4d}/{:4d}] | D_real_loss: {:6.4f}
| D_fake_loss: {:6.4f} | G_loss: {:6.4f}'.format(
                    iteration, total_train_iters, D_real_loss.item(),
D_fake_loss.item(), G_loss.item()))

            # Save the generated samples
            if iteration % opts.sample_every == 0:
                gan_save_samples(G, fixed_noise, iteration, opts)

            # Save the model parameters
            if iteration % opts.checkpoint_every == 0:
                gan_checkpoint(iteration, G, D, opts)

    except KeyboardInterrupt:
        print('Exiting early from training.')
        return G, D

    plt.figure()
    plt.plot(losses['iteration'], losses['D_real_loss'],
label='D_real')
    plt.plot(losses['iteration'], losses['D_fake_loss'],
label='D_fake')
    plt.plot(losses['iteration'], losses['G_loss'], label='G')
    plt.legend()
    plt.savefig(os.path.join(opts.sample_dir, 'losses.png'))
    plt.close()
    return G, D
```

[Your Task] Training

Download dataset

```
######################################################################
# Download Translation datasets
######################################################################
data_fpath = get_file(fname='emojis',

origin='http://www.cs.toronto.edu/~jba/emojis.tar.gz',
                          untar=True)
```

```
data/emojis.tar.gz
```

Train DCGAN

```
SEED = 11

# Set the random seed manually for reproducibility.
```

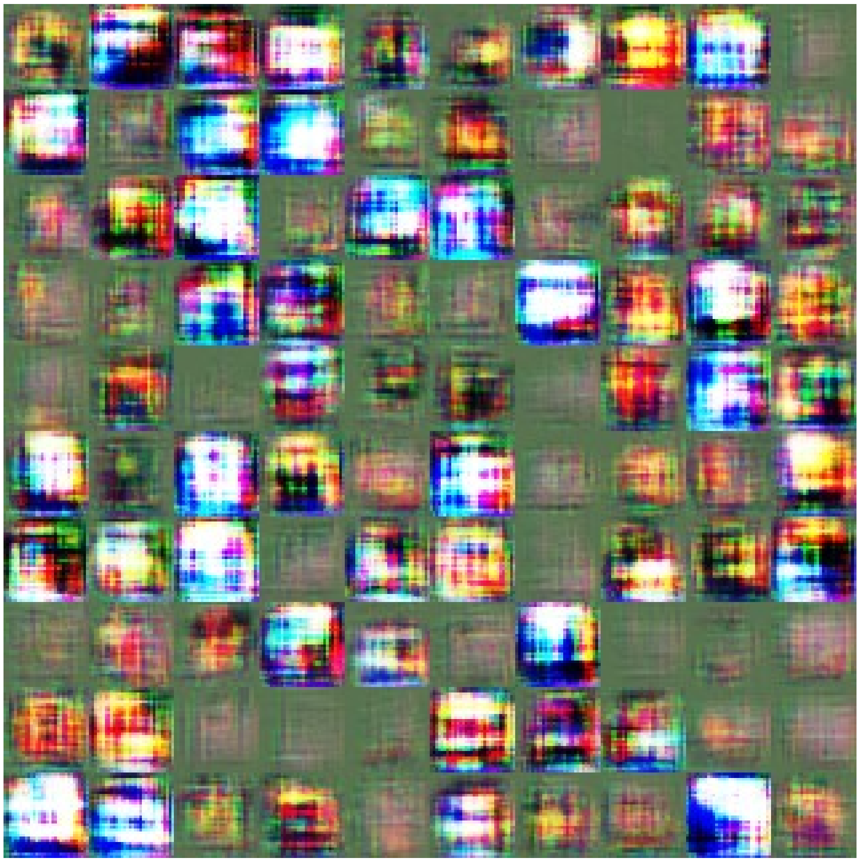
```
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed(SEED)


args = AttrDict()
args_dict = {
              'image_size':32,
              'g_conv_dim':32,
              'd_conv_dim':64,
              'noise_size':100,
              'num_workers': 0,
              'train_iters':20000,
              'X':'Apple',  # options: 'Windows' / 'Apple'
              'Y': None,
              'lr':0.00003,
              'beta1':0.5,
              'beta2':0.999,
              'batch_size':32,
              'checkpoint_dir': 'results/checkpoints_gan_gp1_lr3e-5',
              'sample_dir': 'results/samples_gan_gp1_lr3e-5',
              'load': None,
              'log_step':200,
              'sample_every':200,
              'checkpoint_every':1000,
              'spectral_norm': False,
              'gradient_penalty': True,
              'least_squares_gan': True,
              'd_train_iters': 1
}
args.update(args_dict)

print_opts(args)
G, D = train(args)

generate_gif("results/samples_gan_gp1_lr3e-5")
```

```
======================================================================
==========
                                      Opts

----------------------------------------------------------------------
----------
                             image_size: 32

                             g_conv_dim: 32

                             d_conv_dim: 64
```

```
                     noise_size: 100

                    train_iters: 20000

                              X: Apple

                             lr: 3e-05

                          beta1: 0.5

                          beta2: 0.999

                     batch_size: 32

                 checkpoint_dir: 
results/checkpoints_gan_gp1_lr3e-5
                     sample_dir: results/samples_gan_gp1_lr3e-
5
                       log_step: 200

                   sample_every: 200

               checkpoint_every: 1000

               gradient_penalty: 1

              least_squares_gan: 1

                  d_train_iters: 1

======================================================================
==========
                 G
---------------------------------------
DCGenerator(
  (linear_bn): Sequential(
    (0): Upsample(scale_factor=4.0, mode=nearest)
    (1): Conv2d(100, 128, kernel_size=(5, 5), stride=(1, 1), 
padding=(2, 2), bias=False)
    (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
  )
  (upconv1): Sequential(
    (0): Upsample(scale_factor=2.0, mode=nearest)
    (1): Conv2d(128, 64, kernel_size=(5, 5), stride=(1, 1), 
padding=(2, 2), bias=False)
    (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
  )
```

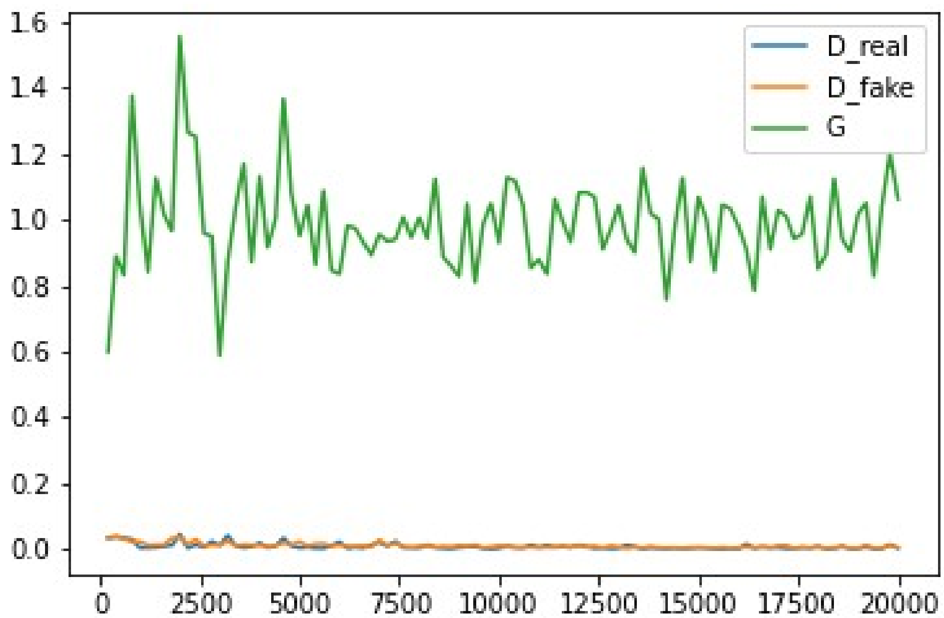
Discussion

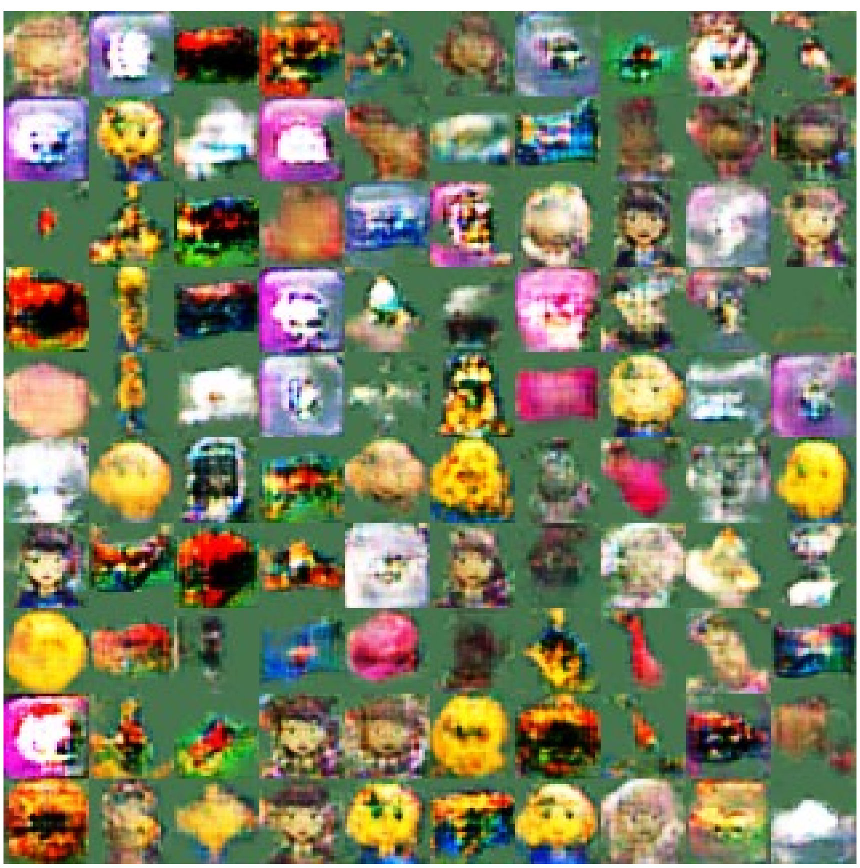
Question 1



The first image is the output at epoch 200 and the second image is the output at epoch 19600. The first image is blurry and contains no notable information. It just seems like a random organization of pixels. However, in the second image, you could see faces of some emojis. Although other emojis are unclear, there are patterns in their colours, which means that GAN is improving.

Question 2


D_real
D_fake
G
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2500
5000
7500
10000
12500
15000
17500
20000



Becase Least Squares uses least squares loss, penalizing the samples lying a long way to the decision boundary can generate more gradients when updating the generator. This resolves the vanishing gradient problem. From the loss plot, Least Square GAN stabliizes trainig as the loss of G fluctuates less. Additionally, the outputs from the code shows that there are more distinctions between every emoji, which means that LS GAN is more stable and performs better than GAN.

Download the Cora data

```
! wget https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz
! tar -zxvf cora.tgz
```

```
--2022-04-04 15:18:44--
https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz
Resolving linqs-data.soe.ucsc.edu (linqs-data.soe.ucsc.edu)...
128.114.47.74
Connecting to linqs-data.soe.ucsc.edu (linqs-data.soe.ucsc.edu)|
128.114.47.74|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 168052 (164K) [application/x-gzip]
Saving to: ‘cora.tgz’

cora.tgz            100%[===================>] 164.11K   972KB/s    in
0.2s

2022-04-04 15:18:44 (972 KB/s) - ‘cora.tgz’ saved [168052/168052]

cora/
cora/README
cora/cora.cites
cora/cora.content
```

import modules and set random seed

```
import numpy as np
import scipy.sparse as sp
import torch
import pandas as pd
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import time

seed = 0

np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
```

Loading and preprocessing the data

```
def encode_onehot(labels):
    # The classes must be sorted before encoding to enable static
class encoding.
    # In other words, make sure the first class always maps to index
0.
    classes = sorted(list(set(labels)))
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in
                    enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                             dtype=np.int32)
    return labels_onehot


def load_data(path="/content/cora/", dataset="cora",
training_samples=140):
    """Load citation network dataset (cora only for now)"""
    print('Loading {} dataset...'.format(dataset))

    idx_features_labels = np.genfromtxt("{}{}.content".format(path,
dataset),
                                        dtype=np.dtype(str))
    features = sp.csr_matrix(idx_features_labels[:, 1:-1],
dtype=np.float32)
    labels = encode_onehot(idx_features_labels[:, -1])

    # build graph
    idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
    idx_map = {j: i for i, j in enumerate(idx)}
    edges_unordered = np.genfromtxt("{}{}.cites".format(path,
dataset),
                                    dtype=np.int32)
    edges = np.array(list(map(idx_map.get,
edges_unordered.flatten())),
                     dtype=np.int32).reshape(edges_unordered.shape)
    adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0],
edges[:, 1])),
                        shape=(labels.shape[0], labels.shape[0]),
                        dtype=np.float32)

    # build symmetric adjacency matrix
    adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T >
adj)

    features = normalize(features)
    adj = adj + sp.eye(adj.shape[0])
    adj = normalize_adj(adj)

    # Random indexes
```

```python
    idx_rand = torch.randperm(len(labels))
    # Nodes for training
    idx_train = idx_rand[:training_samples]
    # Nodes for validation
    idx_val= idx_rand[training_samples:]

    adj = torch.FloatTensor(np.array(adj.todense()))
    features = torch.FloatTensor(np.array(features.todense()))
    labels = torch.LongTensor(np.where(labels)[1])

    idx_train = torch.LongTensor(idx_train)
    idx_val = torch.LongTensor(idx_val)

    return adj, features, labels, idx_train, idx_val

def normalize_adj(mx):
    """symmetric normalization"""
    rowsum = np.array(mx.sum(1))
    r_inv_sqrt = np.power(rowsum, -0.5).flatten()
    r_inv_sqrt[np.isinf(r_inv_sqrt)] = 0.
    r_mat_inv_sqrt = sp.diags(r_inv_sqrt)
    return mx.dot(r_mat_inv_sqrt).transpose().dot(r_mat_inv_sqrt)

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx


def accuracy(output, labels):
    preds = output.max(1)[1].type_as(labels)
    correct = preds.eq(labels).double()
    correct = correct.sum()
    return correct / len(labels)
```

check the data

```python
adj, features, labels, idx_train, idx_val = load_data()
```

```
Loading cora dataset...
```

```python
print(adj)
print(adj.shape)
```

```
tensor([[0.1667, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.5000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.2000,  ..., 0.0000, 0.0000, 0.0000],
```

```
        ...,
        [0.0000, 0.0000, 0.0000,  ..., 0.2000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.2000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.2500]])
torch.Size([2708, 2708])
```

```
print(features)
print(features.shape)
```

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
torch.Size([2708, 1433])
```

```
print(labels)
print(labels.unique())
print(len(labels))
```

```
tensor([2, 5, 4,  ..., 1, 0, 2])
tensor([0, 1, 2, 3, 4, 5, 6])
2708
```

```
print(len(idx_train))
print(len(idx_val))
```

```
140
2568
```

Vanilla GCN for node classification

Define Graph Convolution layer (Your Task)

This module takes $h=\{\overrightarrow{h_1},\overrightarrow{h_2},\ldots,\overrightarrow{h_N}\}$ where $\overrightarrow{h_i} \in R^F$ as input and outputs $h'=\{\overrightarrow{h'_1},\overrightarrow{h'_2},\ldots,\overrightarrow{h'_N}\}$, where $\overrightarrow{h'_i} \in R^{F'}$.

1. perform initial transformation: $s=W \times h^{(l)}$
2. multiply s by normalized adjacency matrix: $h'=A \times s$

```
class GraphConvolution(nn.Module):
    """
    A Graph Convolution Layer (GCN)
    """

    def __init__(self, in_features, out_features, bias=True):
        """
        * `in_features`, $F$, is the number of input features per node
```

```
        * `out_features`, $F'$, is the number of output features per
node
        * `bias`, whether to include the bias term in the linear
layer. Default=True
        """
        super(GraphConvolution, self).__init__()
        # TODO: initialize the weight W that maps the input feature
(dim F ) to output feature (dim F')
        # hint: use nn.Linear()
        ############ Your code here
###################################
        self.linear = nn.Linear(in_features = in_features,
out_features = out_features, bias = True)

###############################################################

    def forward(self, input, adj):
        # TODO: transform input feature to output (don't forget to use
the adjacency matrix
        # to sum over neighbouring nodes )
        # hint: use the linear layer you declared above.
        # hint: you can use torch.spmm() sparse matrix multiplication
to handle the
        #       adjacency matrix
        ############ Your code here
###################################
        s = self.linear(input)
        h = torch.sparse.mm(adj, s)
        return h

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

Define GCN (Your Task)

you will implement a two-layer GCN with ReLU activation function and Dropout after the first Conv layer.

```
class GCN(nn.Module):
    '''
    A two-layer GCN
    '''
    def __init__(self, nfeat, n_hidden, n_classes, dropout,
bias=True):
        """
        * `nfeat`, is the number of input features per node of the
first layer
        * `n_hidden`, number of hidden units
        * `n_classes`, total number of classes for classification
        * `dropout`, the dropout ratio
        * `bias`, whether to include the bias term in the linear
```

```
layer. Default=True
        """

        super(GCN, self).__init__()
        # TODO: Initialization
        # (1) 2 GraphConvolution() layers.
        # (2) 1 Dropout layer
        # (3) 1 activation function: ReLU()
        ############ Your code here
###################################
        self.graph_conv1 = GraphConvolution(in_features = nfeat,
out_features = n_hidden)
        self.graph_conv2 = GraphConvolution(in_features = n_hidden,
out_features = n_classes)
        self.dropout = torch.nn.Dropout(p = dropout)
        self.activation = torch.nn.ReLU()

###############################################################

    def forward(self, x, adj):
        # TODO: the input will pass through the first graph
convolution layer,
        # the activation function, the dropout layer, then the second
graph
        # convolution layer. No activation function for the
        # last layer. Return the logits.
        ############ Your code here
###################################
        x = self.graph_conv1(x, adj)
        x = self.dropout(self.activation(x))
        x = self.graph_conv2(x, adj)
        return x

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

define loss function

```
criterion = nn.CrossEntropyLoss()
```

training loop

```
args = {"training_samples": 140,
        "epochs": 100,
        "lr": 0.01,
        "weight_decay": 5e-4,
        "hidden": 16,
        "dropout": 0.5,
        "bias": True,
        }

def train(epoch):
    t = time.time()
```

```
    model.train()
    optimizer.zero_grad()
    output = model(features, adj)
    loss_train = criterion(output[idx_train], labels[idx_train])
    acc_train = accuracy(output[idx_train], labels[idx_train])
    loss_train.backward()
    optimizer.step()

    model.eval()
    output = model(features, adj)

    loss_val = criterion(output[idx_val], labels[idx_val])
    acc_val = accuracy(output[idx_val], labels[idx_val])
    print('Epoch: {:04d}'.format(epoch+1),
          'loss_train: {:.4f}'.format(loss_train.item()),
          'acc_train: {:.4f}'.format(acc_train.item()),
          'loss_val: {:.4f}'.format(loss_val.item()),
          'acc_val: {:.4f}'.format(acc_val.item()),
          'time: {:.4f}s'.format(time.time() - t))


def test():
    model.eval()
    output = model(features, adj)
    loss_test = criterion(output[idx_val], labels[idx_val])
    acc_test = accuracy(output[idx_val], labels[idx_val])
    print("Test set results:",
          "loss= {:.4f}".format(loss_test.item()),
          "accuracy= {:.4f}".format(acc_test.item()))



model = GCN(nfeat=features.shape[1],
            n_hidden=args["hidden"],
            n_classes=labels.max().item() + 1,
            dropout=args["dropout"]).to(device)
optimizer = optim.Adam(model.parameters(),
                       lr=args["lr"],
weight_decay=args["weight_decay"])


adj, features, labels, idx_train, idx_val =
load_data(training_samples=args["training_samples"])
adj, features, labels, idx_train, idx_val = adj.to(device),
features.to(device), labels.to(device), idx_train.to(device),
idx_val.to(device)
```

```
Loading cora dataset...
```

training Vanilla GCN

```
# Train model
t_total = time.time()
for epoch in range(args["epochs"]):
    train(epoch)
print("Optimization Finished!")
print("Total time elapsed: {:.4f}s".format(time.time() - t_total))

# evaluating
test()
```

```
Epoch: 0001 loss_train: 1.9387 acc_train: 0.1143 loss_val: 1.9272
acc_val: 0.1597 time: 0.1681s
Epoch: 0002 loss_train: 1.9319 acc_train: 0.1143 loss_val: 1.9223
acc_val: 0.1597 time: 0.0103s
Epoch: 0003 loss_train: 1.9245 acc_train: 0.1143 loss_val: 1.9165
acc_val: 0.1663 time: 0.0112s
Epoch: 0004 loss_train: 1.9154 acc_train: 0.1786 loss_val: 1.9103
acc_val: 0.2134 time: 0.0125s
Epoch: 0005 loss_train: 1.9070 acc_train: 0.2000 loss_val: 1.9037
acc_val: 0.1340 time: 0.0103s
Epoch: 0006 loss_train: 1.8966 acc_train: 0.2071 loss_val: 1.8968
acc_val: 0.1464 time: 0.0107s
Epoch: 0007 loss_train: 1.8865 acc_train: 0.2429 loss_val: 1.8893
acc_val: 0.3470 time: 0.0099s
Epoch: 0008 loss_train: 1.8768 acc_train: 0.4000 loss_val: 1.8815
acc_val: 0.3520 time: 0.0098s
Epoch: 0009 loss_train: 1.8699 acc_train: 0.3857 loss_val: 1.8735
acc_val: 0.3104 time: 0.0102s
Epoch: 0010 loss_train: 1.8577 acc_train: 0.3786 loss_val: 1.8654
acc_val: 0.3018 time: 0.0095s
Epoch: 0011 loss_train: 1.8397 acc_train: 0.3500 loss_val: 1.8571
acc_val: 0.3014 time: 0.0108s
Epoch: 0012 loss_train: 1.8268 acc_train: 0.3429 loss_val: 1.8487
acc_val: 0.3014 time: 0.0092s
Epoch: 0013 loss_train: 1.8119 acc_train: 0.3286 loss_val: 1.8403
acc_val: 0.3014 time: 0.0114s
Epoch: 0014 loss_train: 1.8050 acc_train: 0.3286 loss_val: 1.8320
acc_val: 0.3010 time: 0.0088s
Epoch: 0015 loss_train: 1.7883 acc_train: 0.3286 loss_val: 1.8240
acc_val: 0.3010 time: 0.0109s
Epoch: 0016 loss_train: 1.7659 acc_train: 0.3214 loss_val: 1.8162
acc_val: 0.3010 time: 0.0097s
Epoch: 0017 loss_train: 1.7582 acc_train: 0.3214 loss_val: 1.8088
acc_val: 0.3010 time: 0.0100s
Epoch: 0018 loss_train: 1.7511 acc_train: 0.3214 loss_val: 1.8020
acc_val: 0.3010 time: 0.0109s
Epoch: 0019 loss_train: 1.7432 acc_train: 0.3214 loss_val: 1.7958
acc_val: 0.3010 time: 0.0113s
Epoch: 0020 loss_train: 1.7293 acc_train: 0.3286 loss_val: 1.7901
```

```
acc_val: 0.6421 time: 0.0096s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892
acc_val: 0.6452 time: 0.0094s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826
acc_val: 0.6491 time: 0.0104s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760
acc_val: 0.6515 time: 0.0092s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703
acc_val: 0.6488 time: 0.0098s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643
acc_val: 0.6515 time: 0.0097s
Optimization Finished!
Total time elapsed: 1.2954s
Test set results: loss= 1.0643 accuracy= 0.6515
```

Graph Attention Networks

Graph attention layer (Your task)

A GAT is made up of multiple such layers. In this section, you will implement a single graph attention layer. Similar to the `GraphConvolution()`, this `GraphAttentionLayer()` module takes $h=\{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$ where $\vec{h}_i \in R^F$ as input and outputs $h'=\{\vec{h'}_1, \vec{h'}_2, \ldots, \vec{h'}_N\}$, where $\vec{h'}_i \in R^{F'}$. However, instead of weighing each neighbouring node based on the adjacency matrix, we will use self attention to learn the relative importance of each neighbouring node. Recall from HW4 where you are asked to write out the equation for single headed attention, here we will implement multi-headed attention, which involves the following steps:

The initial transformation

In GCN above, you have completed similar transformation. But here, we need to define a weight matrix and perform this transformation for each head: $\vec{s}_i^k = W^k \vec{h}_i$. We will perform a single linear transformation and then split it up for each head later. Note the input $\vec{h}$ has shape `[n_nodes, in_features]` and $\vec{s}$ has shape of `[n_nodes, n_heads * n_hidden]`. Remember to reshape $\vec{s}$ has shape of `[n_nodes, n_heads, n_hidden]` for later uses. Note: set `bias=False` for this linear transformation.

attention score

We calculate these for each head k. Here for simplicity of the notation, we omit k in the following equations. The attention scores are defined as the follows:

$$e_{ij} = a\left(W\vec{h}_i, W\vec{h}_j\right) = a\left(\vec{s}_i, \vec{s}_j\right)$$

, where e_{ij} is the attention score (importance) of node j to node i. We will have to calculate this for each head. a is the attention mechanism, that calculates the attention score. The paper concatenates $\vec{s}_i$, $\vec{s}_j$ and does a linear transformation with a weight vector $a \in R^{2F'}$ followed by a LeakyReLU.

$$e_{ij} = \text{LeakyReLU}\left(a^\top \left[\vec{s}_i \| \vec{s}_j\right]\right)$$

How to vectorize this? Some hints:

1. `tensor.repeat()` gives you $\{\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_N, \vec{s}_1, \vec{s}_2, \ldots, \vec{s}_N, \ldots\}$.

2. `tensor.repeat_interleave()` gives you $\{\vec{s}_1, \vec{s}_1, \ldots, \vec{s}_1, \vec{s}_2, \vec{s}_2, \ldots, \vec{s}_2, \ldots\}$.

3. concatenate to get $[\vec{s}_i \| \vec{s}_j]$ for all pairs of i, j. Reshape $\vec{s}_i \| \vec{s}_j$ has shape of `[n_nodes, n_nodes, n_heads, 2 * n_hidden]`

4. apply the attention layer and non-linear activation function to get $e_{ij} = \text{LeakyReLU}\left(a^\top [\vec{s}_i \| \vec{s}_j]\right)$, where $a^\top$ is a single linear transformation that maps from dimension `n_hidden * 2` to 1. Note: set the `bias=False` for this linear transformation. e is of shape `[n_nodes, n_nodes, n_heads, 1]`. Remove the last dimension 1 using `squeeze()`.

Perform softmax

First, we need to mask e_{ij} based on adjacency matrix. We only need to sum over the neighbouring nodes for the attention calculation. Set the elements in e_{ij} to $-\infty$ if there is no edge from i to j for the softmax calculation. We need to do this for all heads and the adjacency matrix is the same for each head. Use `tensor.masked_fill()` to mask e_{ij} based on adjacency matrix for all heads. Hint: reshape the adjacency matrix to `[n_nodes, n_nodes, 1]` using `unsqueeze()`. Now we are ready to normalize attention scores (or coefficients)

$$\alpha_{ij} = \text{softmax}_j\left(e_{ij}\right) = \frac{\exp\left(e_{ij}\right)}{\sum_{k \in N_i} \exp\left(e_{ik}\right)}$$

Apply dropout

Apply the dropout layer. (this step is easy)

Calculate final output for each head

$$\vec{h'}_i^k = \sum_{j \in N_i} \alpha_{ij}^k \vec{s}_j^k$$

Concat or Mean

Finally we concateneate the transformed features: $\vec{h'}_i = \|_{k=1}^K \vec{h'}_i^k$. In the code, we only need to reshape the tensor to shape of `[n_nodes, n_heads * n_hidden]`. Note that if it is the final layer, then it doesn't make sense to do concatenation anymore. Instead, we sum over the `n_heads` dimension: $\overrightarrow{h'}_i = \frac{1}{K}\sum_{k=1}^{K}\overrightarrow{h'}_i^k$.

```
class GraphAttentionLayer(nn.Module):

    def __init__(self, in_features: int, out_features: int, n_heads:
int,
                 is_concat: bool = True,
                 dropout: float = 0.6,
                 alpha: float = 0.2):
        """
        in_features: F, the number of input features per node
        out_features: F', the number of output features per node
        n_heads: K, the number of attention heads
        is_concat: whether the multi-head results should be
concatenated or averaged
        dropout: the dropout probability
        alpha: the negative slope for leaky relu activation
        """
        super(GraphAttentionLayer, self).__init__()

        self.is_concat = is_concat
        self.n_heads = n_heads

        if is_concat:
            assert out_features % n_heads == 0
            self.n_hidden = out_features // n_heads
        else:
            self.n_hidden = out_features

        # TODO: initialize the following modules:
        # (1) self.W: Linear layer that transform the input feature
before self attention.
        # You should NOT use for loops for the multiheaded
implementation (set bias = False)
        # (2) self.attention: Linear layer that compute the attention
score (set bias = False)
        # (3) self.activation: Activation function (LeakyReLU whith
negative_slope=alpha)
        # (4) self.softmax: Softmax function (what's the dim to
compute the summation?)
        # (5) self.dropout_layer: Dropout function(with ratio=dropout)
        ################ your code here ########################
        self.W = nn.Linear(in_features = in_features, out_features =
self.n_heads * self.n_hidden, bias = False)
        self.attention = nn.Linear(in_features = 2 * self.n_hidden,
out_features = 1, bias = False)
        self.activation = nn.LeakyReLU(negative_slope = alpha)
```

```
        self.softmax = nn.Softmax(dim = 0)
        self.dropout = nn.Dropout(p = dropout)
        ########################################################

    def forward(self, h: torch.Tensor, adj_mat: torch.Tensor):
        # Number of nodes
        n_nodes = h.shape[0]

        # TODO:
        # (1) calculate s = Wh and reshape it to [n_nodes, n_heads,
n_hidden]
        #     (you can use tensor.view() function)
        # (2) get [s_i || s_j] using tensor.repeat(),
repeat_interleave(), torch.cat(), tensor.view()
        # (3) apply the attention layer
        # (4) apply the activation layer (you will get the attention
score e)
        # (5) remove the last dimension 1 use tensor.squeeze()
        # (6) mask the attention score with the adjacency matrix (if
there's no edge, assign it to -inf)
        #     note: check the dimensions of e and your adjacency
matrix. You may need to use the function unsqueeze()
        # (7) apply softmax
        # (8) apply dropout_layer
        ############## Your code here
#########################################

        # print(n_nodes) 2708
        # print('n_heads') 8
        # print('n_hidden') 2
        s = self.W(h)
        s = torch.reshape(s, (n_nodes, self.n_heads, self.n_hidden))

        s_prime = torch.cat((s.repeat(n_nodes, 1, 1),
s.repeat_interleave(n_nodes, dim = 0)), dim = -1) #
torch.Size([7333264, 8, 4])
        s_prime = torch.reshape(s_prime, (n_nodes, n_nodes,
self.n_heads, self.n_hidden * 2)) # torch.Size([2708, 2708, 8, 4])

        e = self.attention(s_prime)
        e = torch.squeeze(e, dim = -1)
        e = self.activation(e)

        adj_mat = torch.unsqueeze(adj_mat, dim = -1)
        e = e.masked_fill_(adj_mat == 0, -np.inf)
        a = self.softmax(e)
        a = self.dropout(a) # torch.Size([2708, 2708, 8])

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

```
        # Summation
        h_prime = torch.einsum('ijh,jhf->ihf', a, s) #[n_nodes,
n_heads, n_hidden]

        # TODO: Concat or Mean
        # Concatenate the heads
        if self.is_concat:
            ############## Your code here
#########################################
          h_prime = torch.reshape(h_prime, (n_nodes, self.n_heads *
self.n_hidden))

######################################################################
#
        # Take the mean of the heads (for the last layer)
        else:
            ############## Your code here
#########################################
          h_prime = torch.sum(h_prime, dim = 1) / h_prime.size()[1]

######################################################################
#

        return h_prime
```

Define GAT network

it's really similar to how we defined GCN. We followed the paper to use two attention layers and ELU() activation function.

```
class GAT(nn.Module):

    def __init__(self, nfeat: int, n_hidden: int, n_classes: int,
n_heads: int, dropout: float, alpha: float):
        """
        in_features: the number of features per node
        n_hidden: the number of features in the first graph attention
layer
        n_classes: the number of classes
        n_heads: the number of heads in the graph attention layers
        dropout: the dropout probability
        alpha: the negative input slope for leaky ReLU of the
attention layer
        """
        super().__init__()

        # First graph attention layer where we concatenate the heads
        self.gc1 = GraphAttentionLayer(nfeat, n_hidden, n_heads,
is_concat=True, dropout=dropout, alpha=alpha)
```

```
        self.gc2 = GraphAttentionLayer(n_hidden, n_classes, 1,
is_concat=False, dropout=dropout, alpha=alpha)
        self.activation = nn.ELU()
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor, adj_mat: torch.Tensor):
        """
        x: the features vectors
        adj_mat: the adjacency matrix
        """
        x = self.dropout(x)
        x = self.gc1(x, adj_mat)
        x = self.activation(x)
        x = self.dropout(x)
        x = self.gc2(x, adj_mat)
        return x
```

training GAT

```
args = {"training_samples": 140,
        "epochs": 100,
        "lr": 0.01,
        "weight_decay": 5e-4,
        "hidden": 16,
        "dropout": 0.5,
        "bias": True,
        "alpha": 0.2,
        "n_heads": 8
        }

model = GAT(nfeat=features.shape[1],
            n_hidden=args["hidden"],
            n_classes=labels.max().item() + 1,
            dropout=args["dropout"],
            alpha=args["alpha"],
            n_heads=args["n_heads"]).to(device)
optimizer = optim.Adam(model.parameters(),
                       lr=args["lr"],
weight_decay=args["weight_decay"])

adj, features, labels, idx_train, idx_val =
load_data(training_samples=args["training_samples"])
adj, features, labels, idx_train, idx_val = adj.to(device),
features.to(device), labels.to(device), idx_train.to(device),
idx_val.to(device)
```

```
Loading cora dataset...
```

```
# Train model
t_total = time.time()
for epoch in range(args["epochs"]):
```

```
acc_val: 0.7262 time: 16.5824s
Epoch: 0098 loss_train: 0.9726 acc_train: 0.7429 loss_val: 1.1507
acc_val: 0.7274 time: 16.5545s
Epoch: 0099 loss_train: 0.9083 acc_train: 0.7857 loss_val: 1.1458
acc_val: 0.7301 time: 16.4303s
Epoch: 0100 loss_train: 0.9175 acc_train: 0.7643 loss_val: 1.1409
acc_val: 0.7309 time: 16.4859s
Optimization Finished!
Total time elapsed: 1653.8945s
Test set results: loss= 1.1409 accuracy= 0.7309
```

Question: (Your task)

Compare the evaluation results for Vanilla GCN and GAT. Comment on the discrepancy in their performance (if any) and briefly explain why you think it's the case (in 1-2 sentences).

Without making further adjustments to the hyperparameters, GAT has a higher testing accuracy than Vanilla GCN. GAT performs better than GCN because it calculates the coefficient implicitly rather than explicitly (like GCN). Therefore, this coefficient utilizes more information besides the graph setup/structure from the graph when determining the importance of every node.

Enable rendering OpenAI Gym environments from CoLab

In this assignemnt, We will use OpenAI Gym for rendering game envionment for our agent to play and learn. It is possible and important to visualize the game your agent is playing, even on Colab. This section imports the necessary package and functions needed to generate a video in Colab. The video processing steps credit to here.

```
# You will need to run this block twice to make it effective
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
!pip install box2d-py > /dev/null 2>&1
!pip install gym[Box_2D] > /dev/null 2>&1
```

```
Requirement already satisfied: setuptools in
/usr/local/lib/python3.7/dist-packages (62.0.0)
```

```
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
```

Import openAI gym and define the functions used to show the video.

```
import gym
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
Utility functions to enable video recording of gym environment
and displaying it.
To enable video, just do "env = wrap_env(env)""
"""

def show_video():
  mp4list = glob.glob('video/*.mp4')
  if len(mp4list) > 0:
    mp4 = mp4list[0]
    video = io.open(mp4, 'r+b').read()
    encoded = base64.b64encode(video)
    ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                loop controls style="height: 400px;">
```

```
                <source src="data:video/mp4;base64,{0}"
type="video/mp4" />
             </video>'''.format(encoded.decode('ascii'))))
  else:
    print("Could not find video")


def wrap_env(env):
  env = Monitor(env, './video', force=True)
  return env
```

Import other packages:

We will use Pytorch for building and learning our DQN network.

```
import torch
from torch import nn
import copy
from collections import deque
import random
from tqdm import tqdm
import matplotlib.pyplot as plt

random.seed(42)
```

Run the game with random agent.

```
from torch import randint
from time import sleep

env = wrap_env(gym.make('CartPole-v1'))
reward_arr = []
episode_count = 20
for i in tqdm(range(episode_count)):
    obs, done, rew = env.reset(), False, 0
    env.render()
    while not done:
        A = randint(0, env.action_space.n, (1,))
        obs, reward, done, info = env.step(A.item())
        rew += reward
        sleep(0.01)
    reward_arr.append(rew)
print("average reward per episode :", sum(reward_arr) /
len(reward_arr))
env.close()
show_video()
```

```
100%|██████████| 20/20 [00:07<00:00,  2.54it/s]

average reward per episode : 21.15

<IPython.core.display.HTML object>
```

The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The video is short (< 1s) because the pole loses balance immediately.

You can see that a random agent is having trouble balancing the CartPole, just like you. However, a difficult game for human may be very simple to a computer. Let's see how we can use DQN to train a agent.

Experience Replay

The technique of experience replay was first proposed in to resolve temporal correlation in the input data by mixing recent experiences as well past experiences, essentially forcing the input to become independent and identically distributed (i.i.d.). It has been shown that this greatly stabilizes and improves the DQN training procedure.

```
class ExperienceReplay(object):
      def __init__(self, length):
        self.experience_replay = deque(maxlen=length)

      def collect(self, experience):
        self.experience_replay.append(experience)
        return

      def sample_from_experience(self, sample_size):
        if len(self.experience_replay) < sample_size:
            sample_size = len(self.experience_replay)
        sample = random.sample(self.experience_replay, sample_size)
        state = torch.tensor([exp[0] for exp in sample]).float()
        action = torch.tensor([exp[1] for exp in sample]).float()
        reward = torch.tensor([exp[2] for exp in sample]).float()
        next_state = torch.tensor([exp[3] for exp in sample]).float()
        return state, action, reward, next_state
```

Build our DQN Network

We will use a simple multi-layer neural network to learn the optimal actions. We will use Adam Optimizor and MSE loss for training. **Notice that the loss function and gamma is given to you in the class attribute.**

```
class DQN_Network:

    def __init__(self, layer_size_list, lr, seed=1423):
        torch.manual_seed(seed)
        self.policy_net = self.create_network(layer_size_list)
        self.target_net = copy.deepcopy(self.policy_net)

        self.loss_fn = torch.nn.MSELoss() # the loss function
        self.optimizer =
torch.optim.Adam(self.policy_net.parameters(), lr=lr)
```

```
        self.step = 0
        self.gamma = torch.tensor(0.95).float()
        return

    def create_network(self, layer_size_list):
        assert len(layer_size_list) > 1

        layers = []
        for i in range(len(layer_size_list) - 1):
            linear = nn.Linear(layer_size_list[i], layer_size_list[i +
1])

            if i < len(layer_size_list) - 2:
              activation = nn.Tanh()
            else:
              activation = nn.Identity()

            layers += (linear, activation)
        return nn.Sequential(*layers)

    def load_pretrained_model(self, model_path):
        self.policy_net.load_state_dict(torch.load(model_path))

    def save_trained_model(self, model_path="cartpole-dqn.pth"):
        torch.save(self.policy_net.state_dict(), model_path)
```

[Your task]: complete the function that chooses the next action

Choose next action based on ϵ-**greedy**:

\begin{align}\text{where} \quad \mathcal{a_{t+1}} = \begin{cases} \text{argmax}*{a}Q(a, s) & \text{with probability }: 1 - \epsilon, \text{exploitation}\ \text{Uniform}{a*{1},...,a_{n}} & \text{with probability}: \epsilon, \text{exploration} \ \end{cases}\end{align}

```
def get_action(model, state, action_space_len, epsilon):
    # We do not require gradient at this point, because this function
will be used either
    # during experience collection or during inference

    with torch.no_grad():
        Qp = model.policy_net(torch.from_numpy(state).float())

    ## TODO: select and return action based on epsilon-greedy
    p = random.random()

    if p < epsilon:
      return torch.randint(0, action_space_len, (1, ))
    else:
      return torch.argmax(Qp)
```

[Your task]: complete the function that train the network for one step

Here, you can find an `train` function that performs a single step of the optimization.

For our training update rule, the loss you are trying to minimize is:

\begin{align}\text{loss} = Q(s, a) - (r + \gamma \max_a Q(s', a))\end{align}

```
def train(model, batch_size):
    state, action, reward, next_state =
memory.sample_from_experience(sample_size=batch_size)

    # TODO: predict expected return of current state using main
network
    # model.policy_net(state)[action.long()].size() [16, 2]
    pred_expected_return = torch.max(model.policy_net(state), axis =
1)[0]

    # TODO: get target return using target network
    # reward.size() [16]
    target_return = reward + model.gamma *
torch.max(model.target_net(next_state), axis = 1)[0]

    # TODO: compute the loss
    loss = model.loss_fn(pred_expected_return, target_return)
    model.optimizer.zero_grad()
    loss.backward(retain_graph=True)
    model.optimizer.step()

    model.step += 1
    if model.step % 5 == 0:

model.target_net.load_state_dict(model.policy_net.state_dict())

    return loss.item()
```

[Your task]: Finish the training loop

In this part, you can play around with `exp_replay_size`, `episode`, `epsilon` and the "episodo decay" logic to train your model. **If you have done correctly, you will observe that the training time for the latter episodes is longer than the early episodes. This is because your agent is getting better and better at playing the game and thus each episode takes longer**

```
# Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n # 2
agent = DQN_Network(layer_size_list=[input_dim, 64, output_dim],
lr=1e-3)
```

```python
# Main training loop
losses_list, reward_list, episode_len_list, epsilon_list = [], [], [],
[]

# TODO: try different values, it normally takes more than 6k episodes
to train
exp_replay_size = 1000
memory = ExperienceReplay(exp_replay_size)
episodes = 15000
epsilon = 1 # episilon start from 1 and decay gradually.

# initiliaze experiance replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()
    done = False
    while not done:
        A = get_action(agent, obs, env.action_space.n, epsilon=1)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])
        obs = obs_next
        index += 1
        if index > exp_replay_size:
            break

index = 128
for i in tqdm(range(episodes)):
    obs, done, losses, ep_len, rew = env.reset(), False, 0, 0, 0
    while not done:
        ep_len += 1
        A = get_action(agent, obs, env.action_space.n, epsilon)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])

        obs = obs_next
        rew += reward
        index += 1

        if index > 128:
            index = 0
            for j in range(4):
                loss = train(agent, batch_size=16)
                losses += loss

    # TODO: add epsilon decay rule here!
    epsilon = (episodes - i) / episodes #* epsilon

    losses_list.append(losses / ep_len), reward_list.append(rew)
```

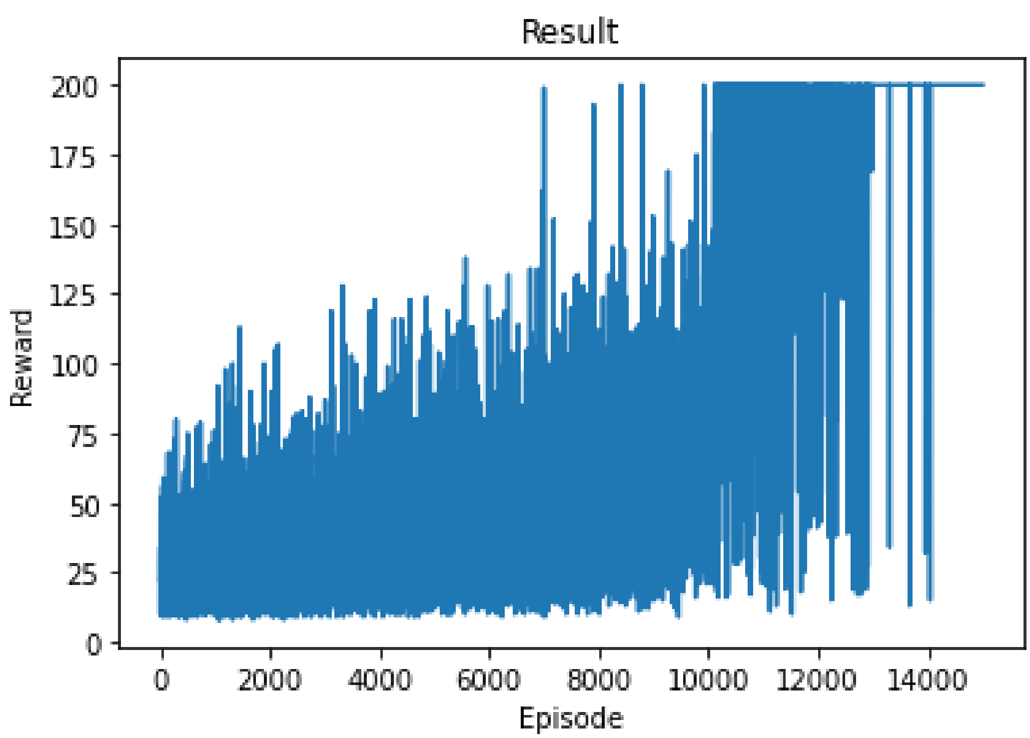
```
    episode_len_list.append(ep_len), epsilon_list.append(epsilon)

print("Saving trained model")
agent.save_trained_model("cartpole-dqn.pth")
```

```
100%|██████████| 15000/15000 [03:54<00:00, 63.84it/s]

Saving trained model
```

Last Step: evaluate your trained model! Make sure to include your visualizations (plot+video) in the notebook for your submission!

First we can plot the reward vs. episode. **If you have done correctly, you should see the reward can stabilize at 200 in later episodes**

```
def plot_reward(r):
    plt.figure(2)
    plt.clf()
    plt.title('Result')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.plot(r)

plot_reward(reward_list)
```



Next let check out how well your agent plays the game. **If you have done correctly, you should see a relatively longer video (> 3~4s) with a self-balancing pole.**

```
env = wrap_env(gym.make('CartPole-v1'))

input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
model_validate = DQN_Network(layer_size_list=[input_dim, 64,
output_dim], lr=1e-3)
model_validate.load_pretrained_model("cartpole-dqn.pth")

reward_arr = []
for i in tqdm(range(200)):
    obs, done, rew = env.reset(), False, 0
    env.render()
    while not done:
        A = get_action(model_validate, obs, env.action_space.n,
epsilon=0)
        obs, reward, done, info = env.step(A.item())
        rew += reward
        # sleep(0.01)

    reward_arr.append(rew)
print("average reward per episode :", sum(reward_arr) /
len(reward_arr))
env.close()
show_video()
```

```
100%|██████████| 200/200 [00:43<00:00,  4.61it/s]

average reward per episode : 500.0
```

```
<IPython.core.display.HTML object>
```

epsilon decay rule: `(espoids - i) / espoids`

`exp_replay_size = 1000`

`episodes = 15000`

```
100%|██████████| 200/200 [00:43<00:00,  4.61it/s]average reward per episode : 500.0
```

1.00