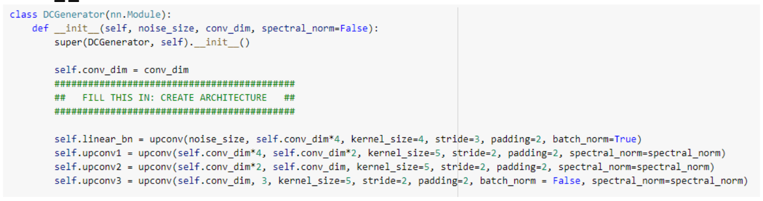
CSC413 Programming Assignment 4

TONGFEI ZHOU, Student #: 1004738448

March 15th, 2022

Part 1: Deep Convolutional GAN (DCGAN)

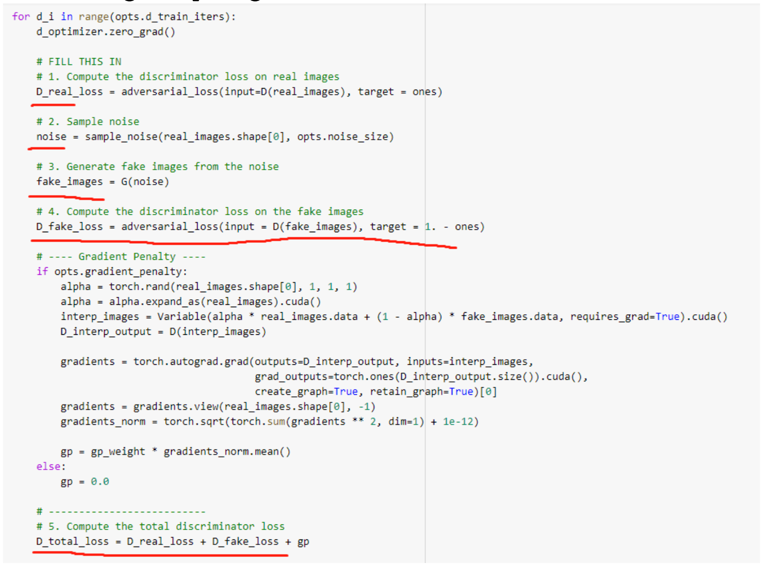
1. The code for the `__init__` method of the `DCGenerator` class is as follows:



```
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(noise_size, self.conv_dim*4, kernel_size=4, stride=3, padding=2, batch_norm=True)
        self.upconv1 = upconv(self.conv_dim*4, self.conv_dim*2, kernel_size=5, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upconv2 = upconv(self.conv_dim*2, self.conv_dim, kernel_size=5, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upconv3 = upconv(self.conv_dim, 3, kernel_size=5, stride=2, padding=2, batch_norm = False, spectral_norm=spectral_norm)
```

Training Loop

1. The code for the `gan_training_loop_regular` function is as follows:

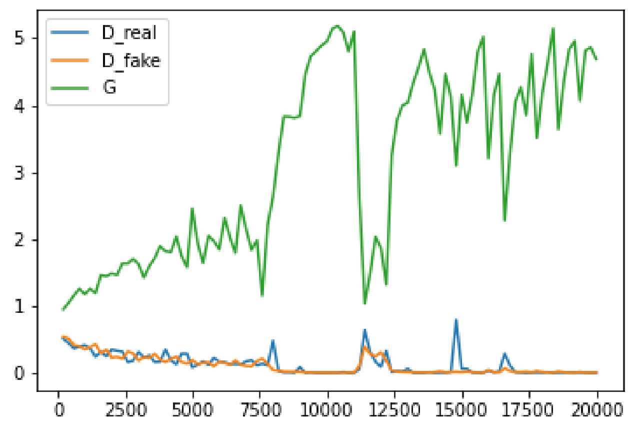


```
for d_i in range(opts.d_train_iters):
    d_optimizer.zero_grad()

    # FILL THIS IN
    # 1. Compute the discriminator loss on real images
    D_real_loss = adversarial_loss(input=D(real_images), target = 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(input = D(fake_images), target = 1. - 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 + gp
```

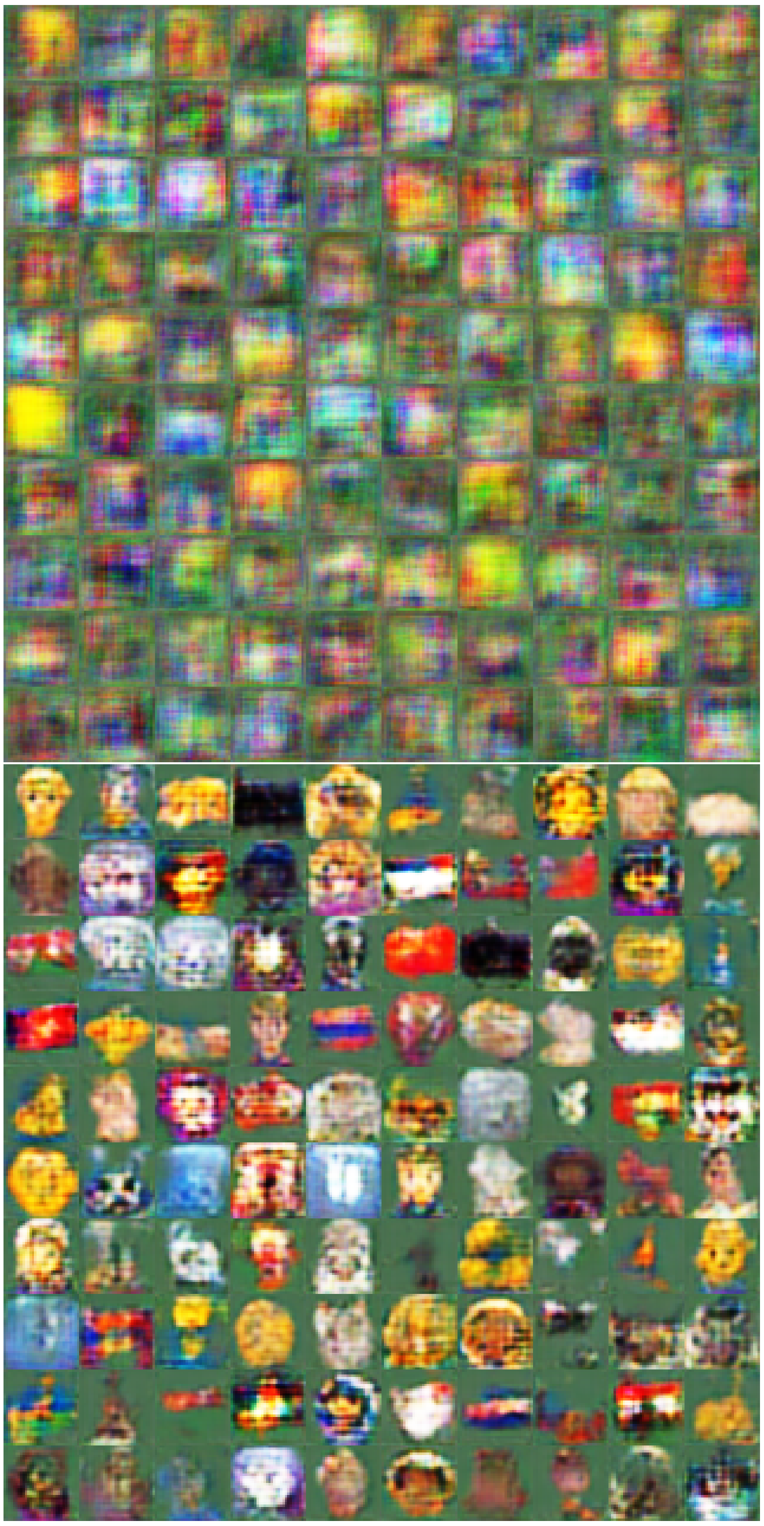
```
# FILL THIS IN
# 1. Sample noise
noise = sample_noise(real_images.shape[0], opts.noise_size)

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

# 3. Compute the generator loss
G_loss = adversarial_loss(input=D(fake_images), target = ones)

G_loss.backward()
g_optimizer.step()
```

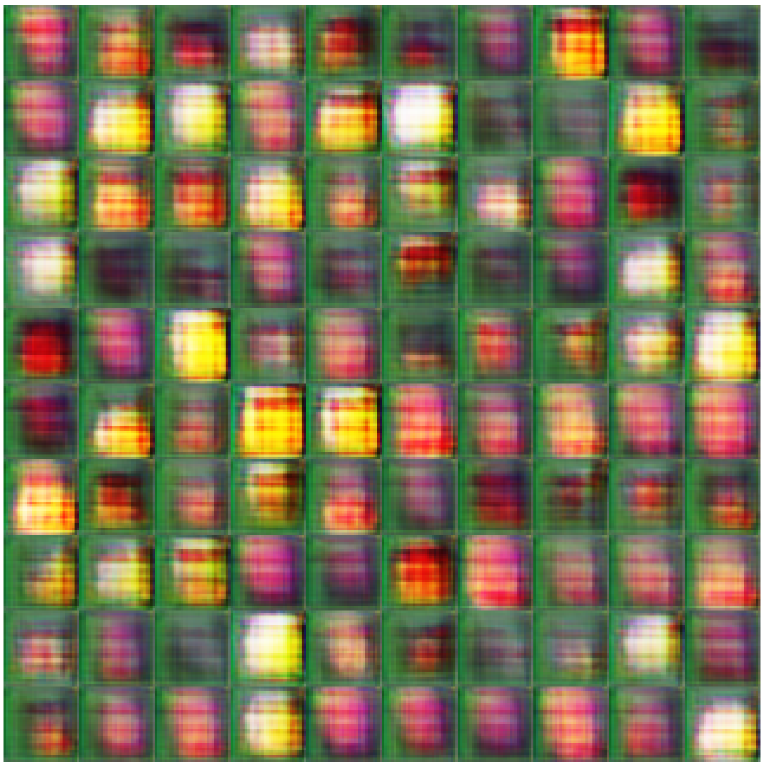
Experiment

1. The generator performs quite unstable overtime as the loss of training varies quite a lot.

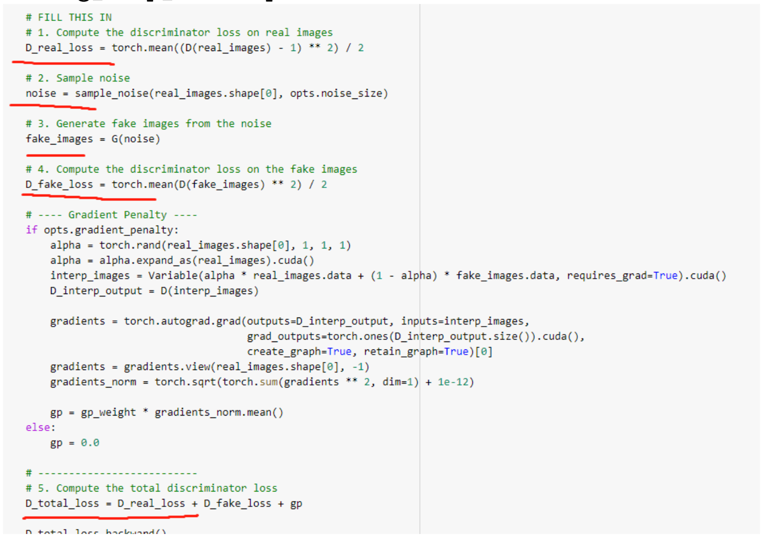


The following are the images generated at the number of iteration 200, 5600, and 19400, one can see both the early in the training and towards the end of training generated very bad output, while the model generated a relatively good output during the middle of the training. However, one can see that even the so-called ”good” output is quite vague, and so improvement in either model architecture or training procedure is required.

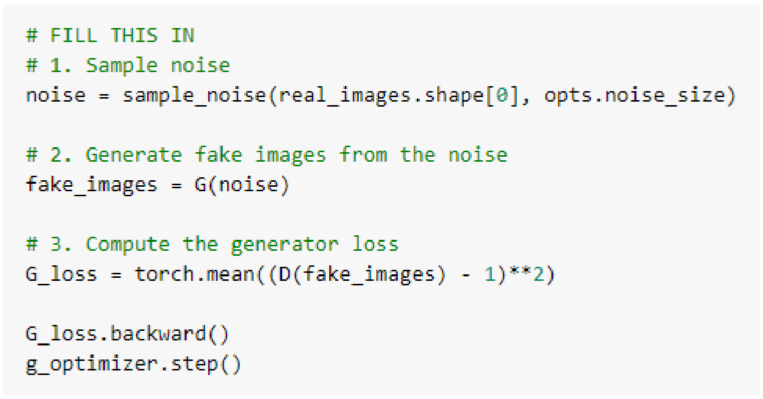




2. The code for `gan_training_loop_leastsquares` function is as follows:

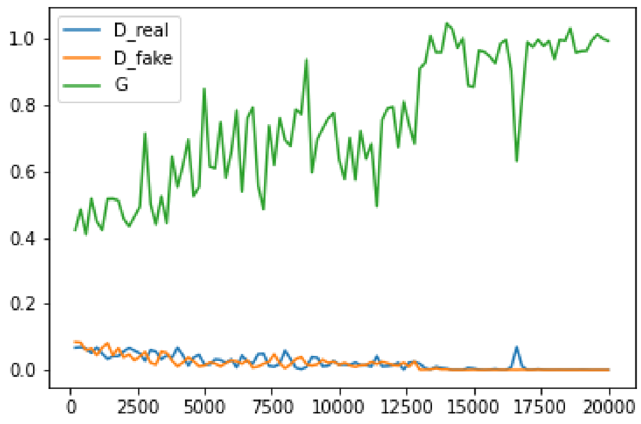


```
# FILL THIS IN
# 1. Compute the discriminator loss on real images
D_real_loss = torch.mean((D(real_images) - 1) ** 2) / 2

# 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 = torch.mean(D(fake_images) ** 2) / 2

# ---- 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 + gp

D_total_loss.backward()
```



```
# 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 = torch.mean((D(fake_images) - 1)**2)

G_loss.backward()
g_optimizer.step()
```

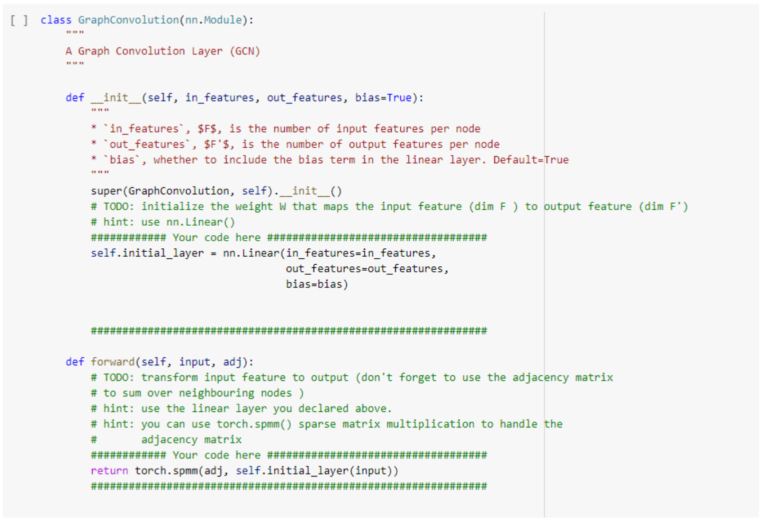
As one can see below, compared to the generator loss in the above, the generator loss with least square being applied is relatively more stable. However, the generator loss curve in general still vary up and down. The reason why least squares GAN can help is that the original Cross-Entropy loss function would lead to the situation that, when the generator produce a relatively good image that fools the discriminator, the gradient of loss would be nearly zero and so no further improvement. This would cause the GAN produces not too good images. Unlike regular GAN, LSGANs will penalize those samples even though they are correctly classified. Such changes are proven to be able to stablize the training process in the paper proposed LSGAN.



Part 2: Graph Convolution Networks

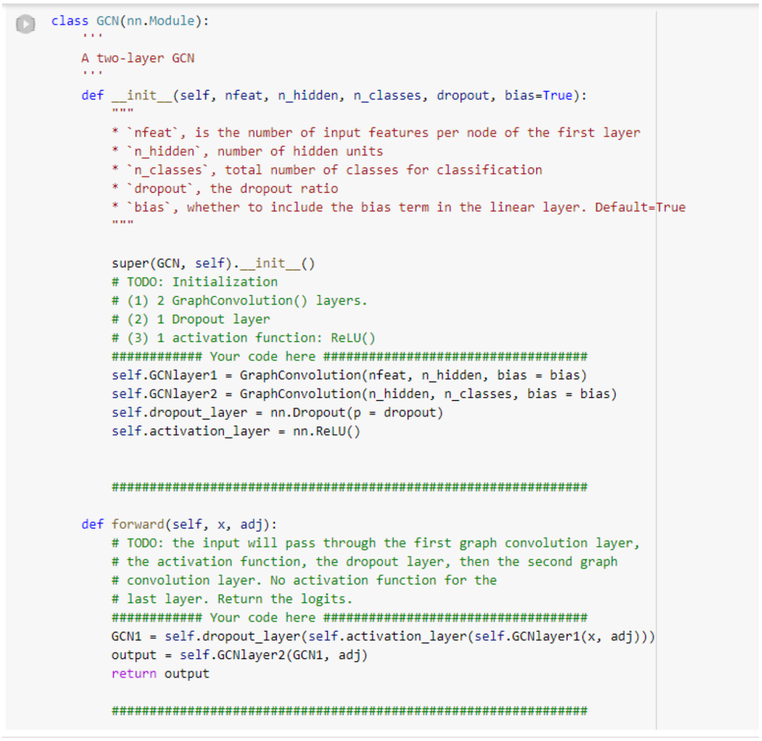
Experiments

1. The code for `GraphConvolution() Class` is as follows:



```
[ ] 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.initial_layer = nn.Linear(in_features=in_features,
                                           out_features=out_features,
                                           bias=bias)



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

        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 ###################################
            return torch.spmm(adj, self.initial_layer(input))
            ###############################################################
```

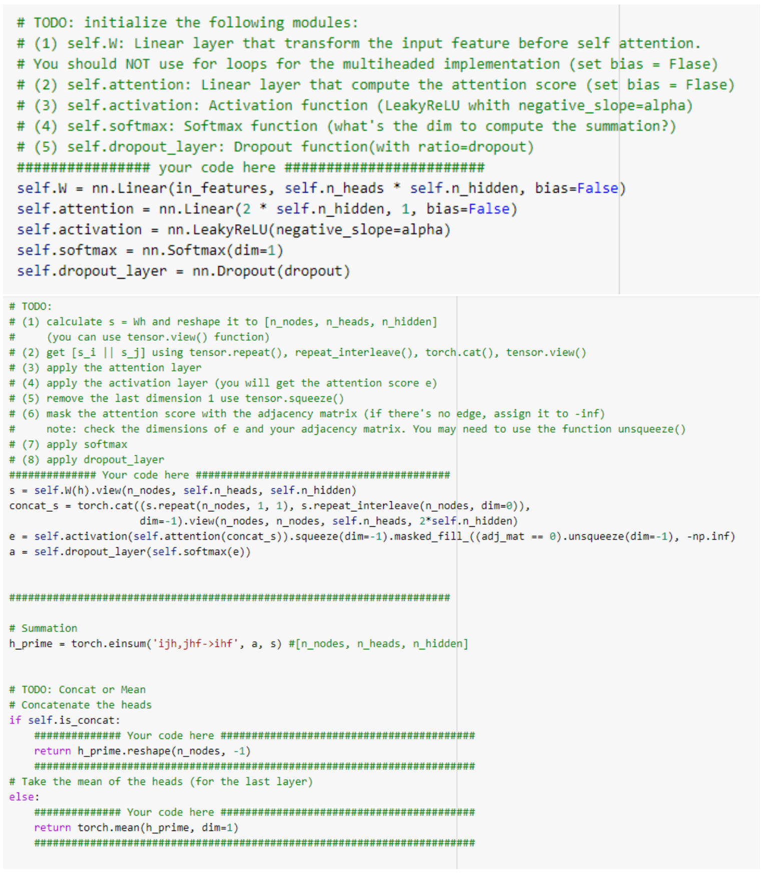
2. The code for `GCN() Class` is as follows:



```
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.GCNlayer1 = GraphConvolution(nfeat, n_hidden, bias = bias)
        self.GCNlayer2 = GraphConvolution(n_hidden, n_classes, bias = bias)
        self.dropout_layer = nn.Dropout(p = dropout)
        self.activation_layer = 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 ###################################
        GCN1 = self.dropout_layer(self.activation_layer(self.GCNlayer1(x, adj)))
        output = self.GCNlayer2(GCN1, adj)
        return output

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

3. The test set results for the GCN is that loss being 1.0727, and accuracy being 0.7558.

4. The code for `GraphAttentionLayer() Class` is as follows:



```
# 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 = Flase)
# (2) self.attention: Linear layer that compute the attention score (set bias = Flase)
# (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, self.n_heads * self.n_hidden, bias=False)
self.attention = nn.Linear(2 * self.n_hidden, 1, bias=False)
self.activation = nn.LeakyReLU(negative_slope=alpha)
self.softmax = nn.Softmax(dim=1)
self.dropout_layer = nn.Dropout(dropout)
```

```
# 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 #########################################
s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)
concat_s = torch.cat((s.repeat(n_nodes, 1, 1), s.repeat_interleave(n_nodes, dim=0)),
                     dim=-1).view(n_nodes, n_nodes, self.n_heads, 2*self.n_hidden)
e = self.activation(self.attention(concat_s)).squeeze(dim=-1).masked_fill_((adj_mat == 0).unsqueeze(dim=-1), -np.inf)
a = self.dropout_layer(self.softmax(e))


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

# 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 #########################################
    return h_prime.reshape(n_nodes, -1)
    #######################################################################
# Take the mean of the heads (for the last layer)
else:
    ############## Your code here #########################################
    return torch.mean(h_prime, dim=1)
    #######################################################################
```

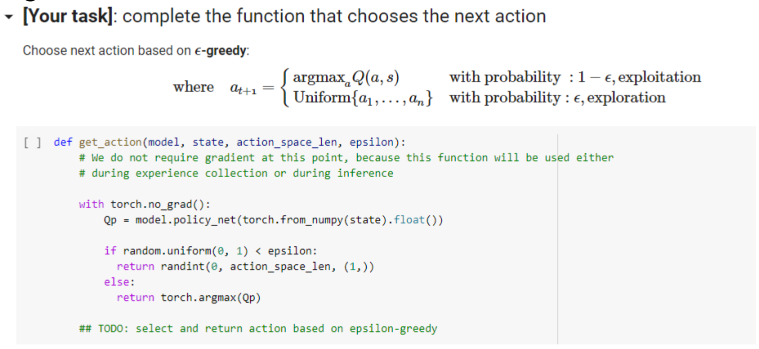
5. The test set results for the GAT is that loss being 1.1286, and accuracy being 0.7644, which is slightly better than the results for the vanilla GCN.

6. The reason for GAT performing slightly more accurate than the vanilla GCN is that in the GAT we are utilizing the attention mechanism so that we can gather more context information when we do the classification task on the node-level.

Part 3: Deep Q-Learning Network (DQN)

Experiments

1. The code for function `get_action` is as follows:



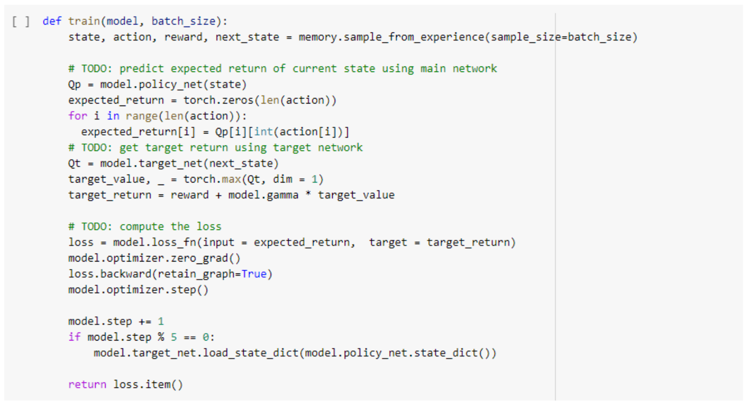
[Your task]: complete the function that chooses the next action

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

$$\text{where} \quad a_{t+1} = \begin{cases} \text{argmax}_a Q(a,s) & \text{with probability } : 1-\epsilon, \text{exploitation} \\ \text{Uniform}\{a_1,\ldots,a_n\} & \text{with probability} : \epsilon, \text{exploration} \end{cases}$$

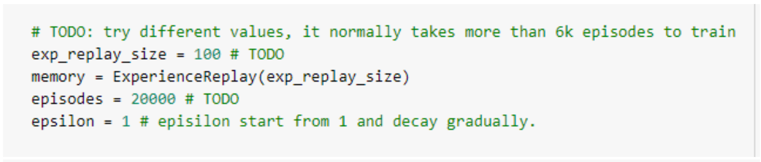
```
[ ]  def get_action(model, state, action_space_len, epsilon):
         # We do not require gradient at this point, because this function will be used either
         # during experience collection or during inference

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

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

         ## TODO: select and return action based on epsilon-greedy
```

2. The code for function `train` is as follows:

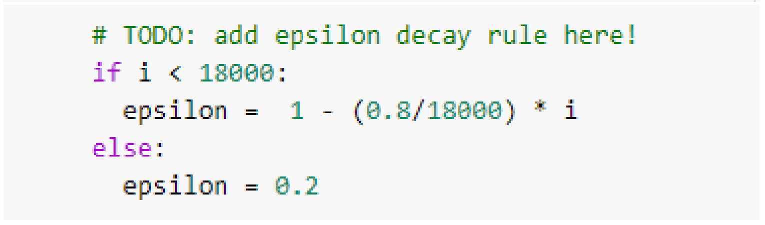


```
[ ] def train(model, batch_size):
        state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)

        # TODO: predict expected return of current state using main network
        Qp = model.policy_net(state)
        expected_return = torch.zeros(len(action))
        for i in range(len(action)):
          expected_return[i] = Qp[i][int(action[i])]
        # TODO: get target return using target network
        Qt = model.target_net(next_state)
        target_value, _ = torch.max(Qt, dim = 1)
        target_return = reward + model.gamma * target_value

        # TODO: compute the loss
        loss = model.loss_fn(input = expected_return,  target = 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()
```

3. The hyperparameters' value I choose, the epsilon decay rule and the agent trained results are as follows:

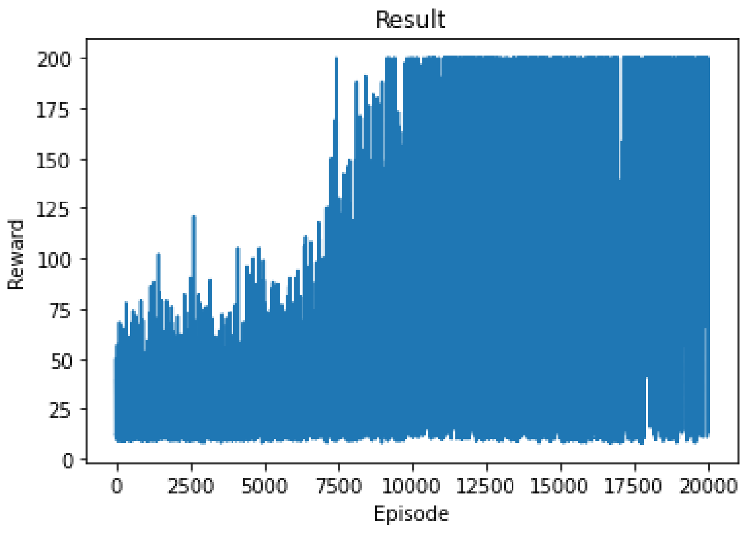


```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 100 # TODO
memory = ExperienceReplay(exp_replay_size)
episodes = 20000 # TODO
epsilon = 1 # episilon start from 1 and decay gradually.
```



```
# TODO: add epsilon decay rule here!
if i < 18000:
  epsilon =  1 - (0.8/18000) * i
else:
  epsilon = 0.2
```

The epsilon decay rule is that, the epsilon will anneal linearly to 0.2 after 18000 episodes, and then stay at this level to the end of episodes.



The video can be seen in the notebook file.