CSC413 PA4

Part 1:

Generator:

1.

```
[ ]  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   ##
             ########################################
             # TODO
             self.linear_bn = upconv(in_channels=noise_size, out_channels=conv_dim*64, kernel_size=1, stride=1, padding=0, spectral_norm=spectral_norm)
             self.upconv1 = upconv(in_channels=conv_dim*4, out_channels=conv_dim*2, kernel_size=5, stride=2, padding=2, spectral_norm=spectral_norm)
             self.upconv2 = upconv(in_channels=conv_dim*2, out_channels=conv_dim, kernel_size=5, stride=2, padding=2, spectral_norm=spectral_norm)
             self.upconv3 = upconv(in_channels=conv_dim, out_channels=3, kernel_size=5, stride=2, padding=2, batch_norm=False, 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
```

Training Loop:

1.

```python
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]
    # ones = Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0), requires_grad=False)

    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)
```

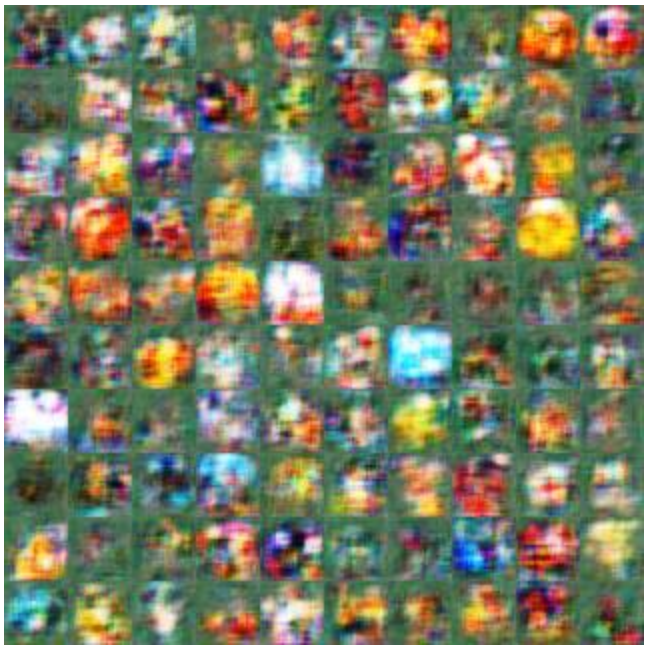
```python
            real_images, real_labels = train_iter.next()
            real_images, real_labels = to_var(real_images), to_var(real_labels).long().squeeze()

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

                # FILL THIS IN
                # 1. Compute the discriminator loss on real images
                ones = Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0), requires_grad=False)
                D_real_loss = adversarial_loss(D(real_images), 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
                zeros = Variable(torch.Tensor(fake_images.shape[0]).float().cuda().fill_(0.0), requires_grad=False)
                D_fake_loss = adversarial_loss(D(fake_images.detach()), zeros)

                # ---- 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
```

```
                gp = 0.0

            # ---------------------------
            # 5. Compute the total discriminator loss
            D_total_loss = D_real_loss + D_fake_loss + gp
```

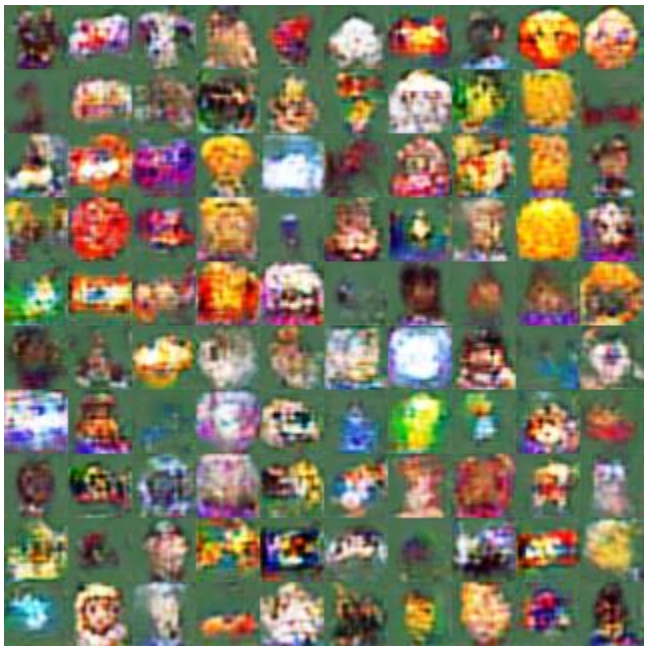
```
            D_total_loss.backward()
            d_optimizer.step()


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

        g_optimizer.zero_grad()

        # FILL THIS IN
        # 1. Sample noise
        noise = sample_noise(opts.batch_size, opts.noise_size)

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

        # 3. Compute the generator loss
        fake_ones = Variable(torch.Tensor(fake_images.shape[0]).float().cuda().fill_(1.0), requires_grad=False)
        G_loss = adversarial_loss(D(fake_images), fake_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
```

Experiment:

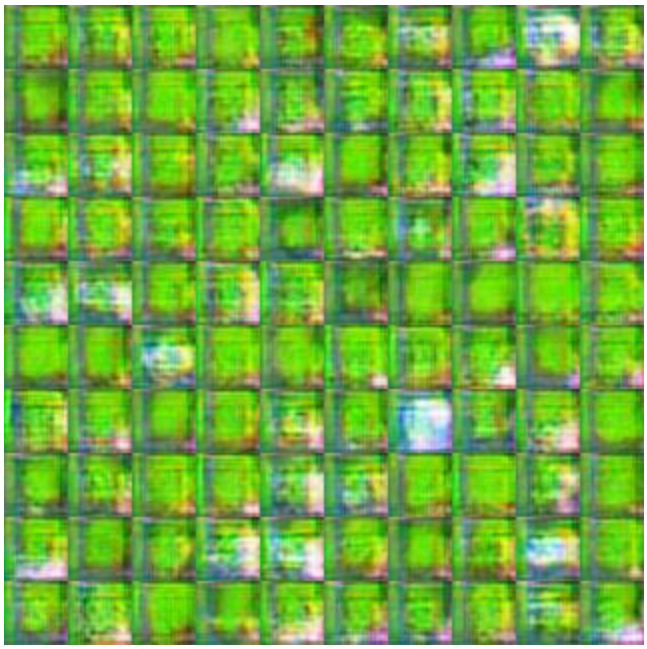
1.



Iteration 600 (poor)



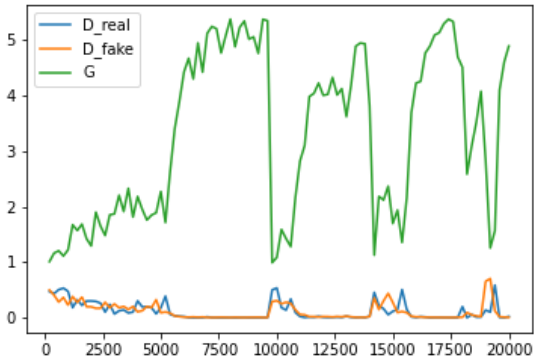
Iteration 4600 (satisfactory)



Iteration 8600

Iteration 15200 (satisfactory)

Iteration 19800 (end of training) (poor)



Regular training loop losses

The quality of the generated emojis at the beginning is very poor (blurry) but gradually improves later on in training. However, the quality oscillates wildly as training progresses. For example, we see that iteration 8600 is much worse than 4600 and 15200. This instability as a result of using the regular training loop can also be seen in the loss curves (large oscillations). During the last bit of training, the emoji quality dropped again dramatically (lots of red) perhaps due to mode collapse.

2.

```
[ ]
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": []}

    #adversarial_loss = torch.nn.BCEWithLogitsLoss()
    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()
```

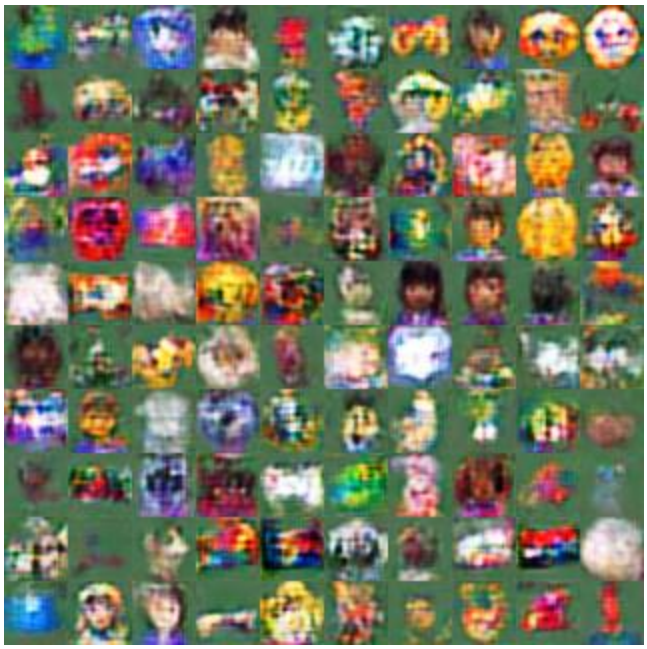
```
    for d_i in range(opts.d_train_iters):
        d_optimizer.zero_grad()

        # FILL THIS IN
        # 1. Compute the discriminator loss on real images
        D_real_loss = torch.mean((D(real_images) - 1) ** 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.detach()) ** 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()
        d_optimizer.step()

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

```
        g_optimizer.zero_grad()

        # FILL THIS IN
        # 1. Sample noise
        noise = sample_noise(opts.batch_size, 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()

        # 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
```

Iteration 600 (beginning, poor)

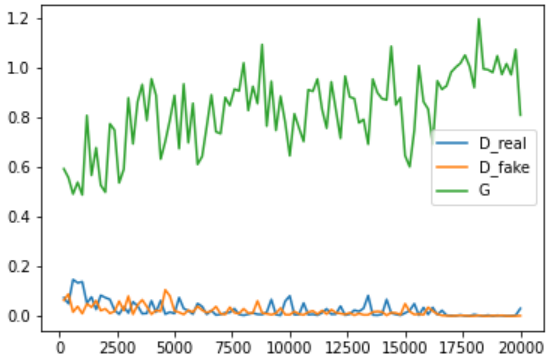
Iteration 4600 (getting better)



Iteration 8600

Iteration 15200 (satisfactory)

Iteration 19800 (poor)



Least squares GAN losses

The least squares GAN was able to stabilize training by a lot. The above sample iteration images show that the results are getting better gradually without significant oscillations. This is also consistent with the loss graph as there are no longer any large bumps. However, at the end of training, we still observe a sharp decrease in performance probably due to mode collapse.

The reason that the least squares GAN was able to stabilize training could be because this loss function penalizes samples that are far from the decision boundary hence remedying the vanishing gradient problem in regular GAN.

Part 2:

1.

```python
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.W = nn.Linear(in_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.W(input))


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

2.

```
[ ]  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.GraphConv1 = GraphConvolution(nfeat, n_hidden, bias=bias)
             self.GraphConv2 = GraphConvolution(n_hidden, n_classes, bias=bias)
             self.dropout = torch.nn.Dropout(dropout)
             self.ReLU = 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.GraphConv1(x, adj)
             x = self.ReLU(x)
             x = self.dropout(x)
             return self.GraphConv2(x, adj)
             ##############################################################
```

3. hyperparameters:
   training samples: 140
   epochs: 100
   lr: 0.01
   weight decay: 5e-4
   hidden: 16
   dropout: 0.5
   bias: True
   Test set results: loss= 1.0643 accuracy= 0.6515

```
Epoch: 0071 loss_train: 1.0473 acc_train: 0.6857 loss_val: 1.2939 acc_val: 0.5717 time: 0.0048s
Epoch: 0072 loss_train: 0.9992 acc_train: 0.7214 loss_val: 1.2845 acc_val: 0.5748 time: 0.0045s
Epoch: 0073 loss_train: 0.9883 acc_train: 0.7286 loss_val: 1.2754 acc_val: 0.5767 time: 0.0046s
Epoch: 0074 loss_train: 0.9673 acc_train: 0.7214 loss_val: 1.2660 acc_val: 0.5775 time: 0.0032s
Epoch: 0075 loss_train: 0.9550 acc_train: 0.7357 loss_val: 1.2560 acc_val: 0.5810 time: 0.0065s
Epoch: 0076 loss_train: 0.9518 acc_train: 0.7143 loss_val: 1.2468 acc_val: 0.5837 time: 0.0049s
Epoch: 0077 loss_train: 0.9441 acc_train: 0.7357 loss_val: 1.2378 acc_val: 0.5872 time: 0.0062s
Epoch: 0078 loss_train: 0.9319 acc_train: 0.7571 loss_val: 1.2292 acc_val: 0.5892 time: 0.0043s
Epoch: 0079 loss_train: 0.9036 acc_train: 0.7643 loss_val: 1.2215 acc_val: 0.5919 time: 0.0040s
Epoch: 0080 loss_train: 0.9106 acc_train: 0.7429 loss_val: 1.2132 acc_val: 0.5970 time: 0.0040s
Epoch: 0081 loss_train: 0.9149 acc_train: 0.7429 loss_val: 1.2047 acc_val: 0.6001 time: 0.0044s
Epoch: 0082 loss_train: 0.8926 acc_train: 0.7714 loss_val: 1.1965 acc_val: 0.6024 time: 0.0038s
Epoch: 0083 loss_train: 0.9165 acc_train: 0.7500 loss_val: 1.1878 acc_val: 0.6063 time: 0.0045s
Epoch: 0084 loss_train: 0.8798 acc_train: 0.7714 loss_val: 1.1790 acc_val: 0.6079 time: 0.0046s
Epoch: 0085 loss_train: 0.8747 acc_train: 0.7786 loss_val: 1.1702 acc_val: 0.6098 time: 0.0046s
Epoch: 0086 loss_train: 0.8627 acc_train: 0.7929 loss_val: 1.1616 acc_val: 0.6133 time: 0.0043s
Epoch: 0087 loss_train: 0.8612 acc_train: 0.7929 loss_val: 1.1537 acc_val: 0.6153 time: 0.0045s
Epoch: 0088 loss_train: 0.8698 acc_train: 0.8071 loss_val: 1.1461 acc_val: 0.6160 time: 0.0039s
Epoch: 0089 loss_train: 0.8098 acc_train: 0.8357 loss_val: 1.1387 acc_val: 0.6195 time: 0.0045s
Epoch: 0090 loss_train: 0.8286 acc_train: 0.8286 loss_val: 1.1313 acc_val: 0.6223 time: 0.0041s
Epoch: 0091 loss_train: 0.7919 acc_train: 0.8286 loss_val: 1.1238 acc_val: 0.6254 time: 0.0045s
Epoch: 0092 loss_train: 0.7797 acc_train: 0.8429 loss_val: 1.1160 acc_val: 0.6293 time: 0.0046s
Epoch: 0093 loss_train: 0.7539 acc_train: 0.8786 loss_val: 1.1086 acc_val: 0.6336 time: 0.0078s
Epoch: 0094 loss_train: 0.8075 acc_train: 0.8286 loss_val: 1.1020 acc_val: 0.6386 time: 0.0066s
Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val: 0.6421 time: 0.0095s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val: 0.6452 time: 0.0047s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val: 0.6491 time: 0.0045s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val: 0.6515 time: 0.0044s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0047s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val: 0.6515 time: 0.0045s
Optimization Finished!
Total time elapsed: 0.6852s
Test set results: loss= 1.0643 accuracy= 0.6515
```

4.

```
[ ]  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 = 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_hidden * n_heads, 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)
             ######################################################

         def forward(self, h: torch.Tensor, adj_mat: torch.Tensor):
```

```
[ ]
    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 ######################################
        s = self.W(h)
        s = s.view(n_nodes, self.n_heads, self.n_hidden)
        s_r = s.repeat(n_nodes, 1, 1)
        s_ri = s.repeat_interleave(n_nodes, dim=0)
        s_cat = torch.cat([s_ri, s_r], dim=-1)
        s_cat = s_cat.view(n_nodes, n_nodes, self.n_heads, 2 * self.n_hidden)
        s_att = self.attention(s_cat)
        e = self.activation(s_att)
        e = e.squeeze(-1)
        adj_mat = adj_mat.unsqueeze(-1)
        e = e.masked_fill(adj_mat == 0, float('-inf'))
        assert adj_mat.shape[0] == 1 or adj_mat.shape[0] == n_nodes
        assert adj_mat.shape[1] == 1 or adj_mat.shape[1] == n_nodes
        assert adj_mat.shape[2] == 1 or adj_mat.shape[2] == self.n_heads
        a = self.softmax(e)
        a = self.dropout_layer(a)
        ####################################################################

        # 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, self.n_hidden * self.n_heads)
```

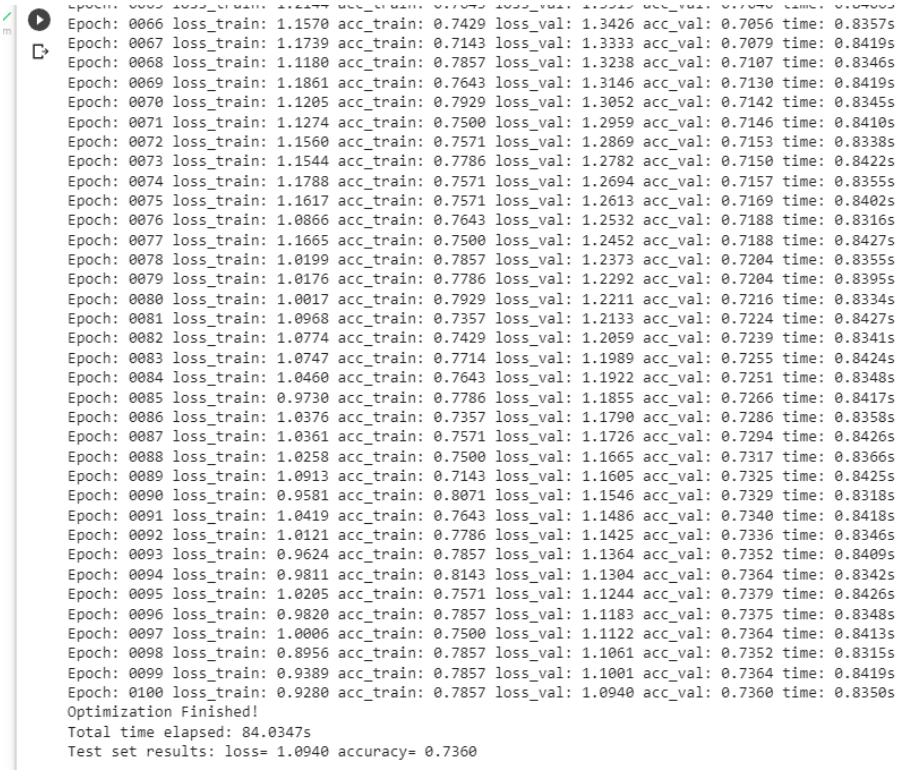
```
        # TODO: Concat or Mean
        # Concatenate the heads
        if self.is_concat:
            ############# Your code here ######################################
            return h_prime.reshape(n_nodes, self.n_hidden * self.n_heads)

            ####################################################################
        # Take the mean of the heads (for the last layer)
        else:
            ############# Your code here ######################################
            return h_prime.mean(dim=1)


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

5. Hyperparameters:
   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
Test set results: loss= 1.0940 accuracy= 0.7360



6. The GAT model performed better (test accuracy of 73.6%) compared to the vanilla GCN (test accuracy of 65.15%). This might be because the multi-head attention allows the GAT to focus on specific nodes instead of the entire neighborhood as in the Vanilla GCN. (Allows the model to use more information than just the graph structure when determining the coefficients.)

Part 3:

1.

```
[ ]  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
         if torch.rand(1) > epsilon:
           q, a = torch.max(Qp, axis=0)
         else:
           a = torch.randint(0, action_space_len, size=(1,))
         return a
```

2.

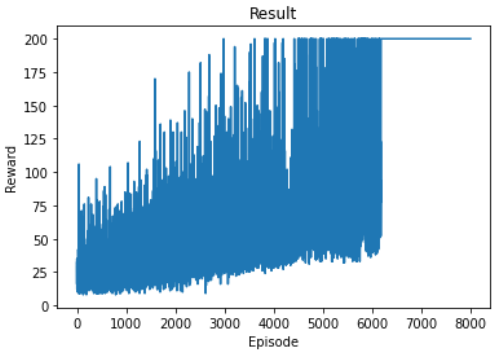
```
def train(model, batch_size):
    state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)

    # TODO: predict expected return of current state using main network
    q, a = torch.max( model.policy_net(state), axis=1)
    # TODO: get target return using target network
    q_prime, a_prime = torch.max(model.target_net(next_state), axis=1)
    target_return = reward + model.gamma * q_prime
    # TODO: compute the loss
    loss = model.loss_fn(q, 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()
```

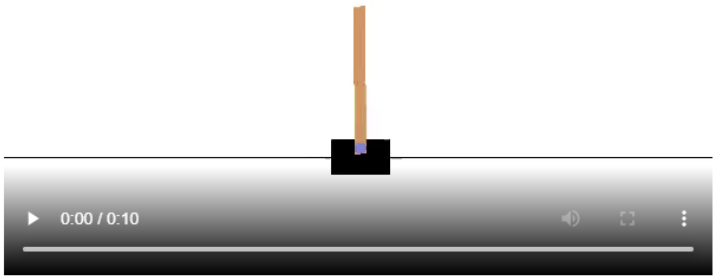
3. Hyperparameters:
   exp_replay_size = 256
   memory = ExperienceReplay(exp_replay_size)
   episodes = 8000
   epsilon = 1 # episilon start from 1 and decay gradually.

   Epsilon decay strategy:

```
if epsilon > 0.03:
    epsilon -= 1 / 4000
```

   results:





The agent was able to balance the pole for over 10 seconds. This is better than my own performance and also much better than the random agent's performance. However, finding the right hyperparameters was a bit difficult since hyperparameters that differed slightly gives very different results (fails to balance).