CSC413 Programming Assignment 4

Part 1: Deep Convolutional GAN (DCGAN)

Generator

1. Implement the DCGenerator class.

```
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, conv_dim*4*4*4, kernel_size=1, stride=1, padding=0, spectral_norm=spectral_norm)
        self.upconv1 = upconv(conv_dim*4, conv_dim*2, kernel_size=5, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upconv2 = upconv(conv_dim*2, conv_dim, kernel_size=5, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upconv3 = upconv(conv_dim, 3, kernel_size=5, stride=2, padding=2, batch_norm=False, spectral_norm=spectral_norm)
```

Training Loop

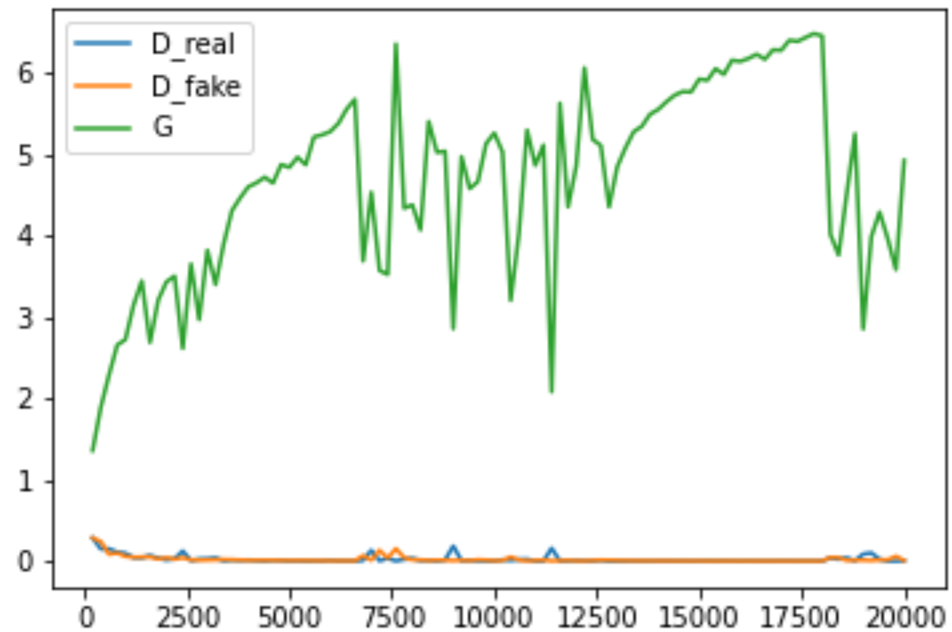
1. Implement gan_training_loop_regular function.

```
            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)
```

```
                # --------------------------
                # 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(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)
```

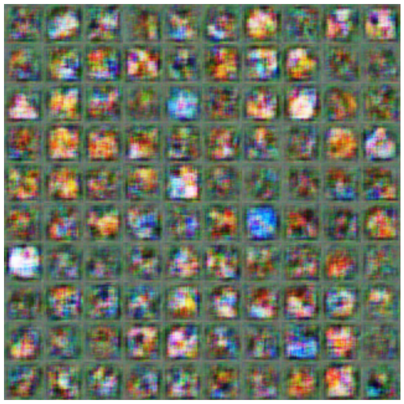
Experiment

1. Train a DCGAN to generate Windows emojis in the Training.
Here is the plot of loss per iteration.



The generator performance is unstable since the loss of training fluctuates over iterations.

The following three images were generated at the iteration numbers of 200, 7600, and 20000. One early in the training, one with satisfactory image quality, and one towards the end of training. The latter two outputs have better quality than the first one, but still not perfectly clear.

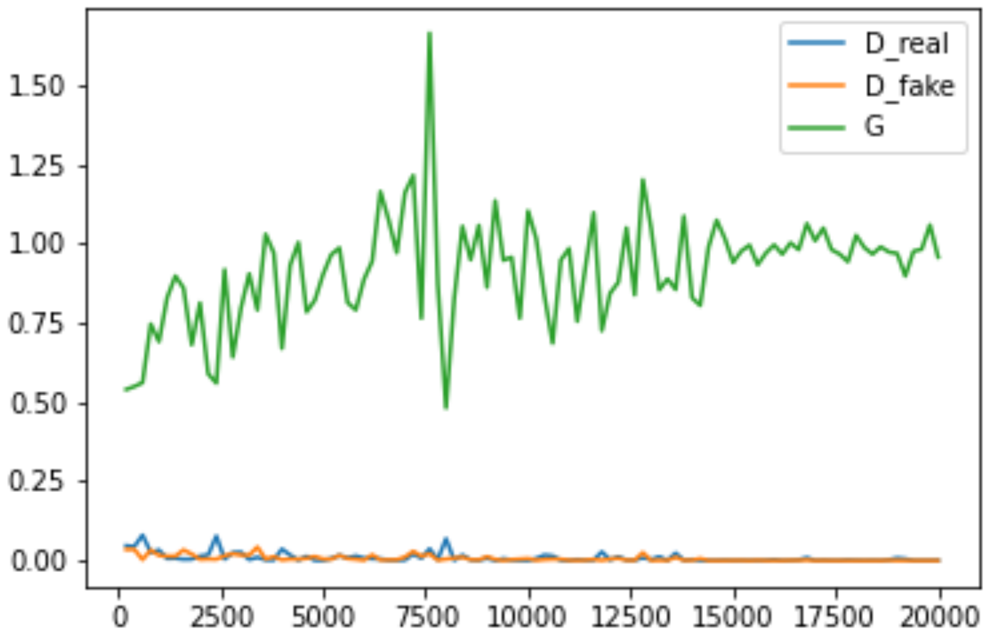
  

2. Implement gan_training_loop_leastsquares function.

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

```
            # -------------------------
            # 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(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)
```

Here is the plot of loss per iteration.



From the above plot, we could see that the training performance of least-squares GAN is relatively more stable over iterations and with smaller loss values than the regular GAN.

The reason why least-squares GAN can help is that with the original Cross-Entropy loss function, the gradient of loss decreases close to zero without further improvement when the generator produces a relatively good image. However, the least-squares GAN could still apply penalty on the samples even if they are correctly classified, which could reduce the loss.

Part 2: Graph Convolution Networks

Experiments:

1. Implementation of Graph Convolution Layer. GraphConvolution() Class.

```
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=
        """
        super(GraphConvolution, self).__init__()
        # TODO: initialize the weight W that maps the input feature (dim F ) to
        # hint: use nn.Linear()
        ############ Your code here ###################################
        self.linear_layer = nn.Linear(in_features, out_features, bias)
        ###############################################################

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

        return out
        ###############################################################
```

2. Implementation of Graph Convolution Network.

```
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=
        """

        super(GCN, self).__init__()
        # TODO: Initialization
        ############ Your code here ###################################
        # (1) 2 GraphConvolution() layers.
        self.GCN_layer1 = GraphConvolution(nfeat, n_hidden, bias)
        self.GCN_layer2 = GraphConvolution(n_hidden, n_classes, bias)

        # (2) 1 Dropout layer
        self.dropout_layer = nn.Dropout(dropout)

        # (3) 1 activation function: ReLU()
        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 ###################################
        out = self.GCN_layer1(x, adj)
        out = self.activation_layer(out)
        out = self.dropout_layer(out)
        out = self.GCN_layer2(out, adj)

        return out
        ###############################################################
```

3. Train your Graph Convolution Network.

```
Optimization Finished!
Total time elapsed: 0.6792s
Test set results: loss= 1.0643 accuracy= 0.6515
```

4. Implementation of Graph Attention Layer.

```
        # TODO: initialize the following modules:
        ################ your code here ########################
        # (1) Linear layer that transform the input feature before self attention.
        # You should NOT use for loops for the multiheaded implementation (set bias = Flase)
        self.W = nn.Linear(in_features, self.n_heads * self.n_hidden, bias=False)

        # (2) Linear layer that compute the attention score (set bias = Flase)
        self.attention = nn.Linear(self.n_hidden * 2, 1, bias=False)

        # (3) Activation function (LeakyReLU whith negative_slope=alpha)
        self.activation = nn.LeakyReLU(negative_slope=alpha)

        # (4) Softmax function (what's the dim to compute the summation?)
        self.softmax = nn.Softmax(dim=1)

        # (5) Dropout function(with ratio=dropout)
        self.dropout_layer = nn.Dropout(dropout)
        ########################################################
```

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

        # TODO:
        ############# Your code here #########################################
        # (1) calculate s = Wh and reshape it to [n_nodes, n_heads, n_hidden]
        #     (you can use tensor.view() function)
        s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)

        # (2) get [s_i || s_j] using tensor.repeat(), repeat_interleave(), torch.cat(), tensor.view()
        concat_s_ij = torch.cat((s.repeat(n_nodes, 1, 1),
                                 s.repeat_interleave(n_nodes, dim=0)),
                                 dim=-1)
        # Reshape s_i || s_j has shape of [n_nodes, n_nodes, n_heads, 2 * n_hidden]
        concat_s_ij = concat_s_ij.view(n_nodes, n_nodes,
                                       self.n_heads, 2 * self.n_hidden)

        # (3) apply the attention layer
        attention = self.attention(concat_s_ij)

        # (4) apply the activation layer (you will get the attention score e)
        e = self.activation(attention)

        # (5) remove the last dimension 1 use tensor.squeeze()
        e = e.squeeze(dim=-1)

        # (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
        e = e.masked_fill((adj_mat.unsqueeze(dim=-1) == 0), -np.inf)

        # (7) apply softmax
        soft_max = self.softmax(e)
```

```
        # (8) apply dropout_layer
        a = self.dropout_layer(soft_max)
        #######################################################################

        # 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 #########################################
            out = h_prime.reshape(n_nodes, self.n_heads * self.n_hidden)
            #######################################################################
        # Take the mean of the heads (for the last layer)
        else:
            ############# Your code here #########################################
            out = torch.mean(h_prime, dim=1)

        return out
            #######################################################################
```

5. Train your Graph Convolution Network.

```
[36] Optimization Finished!
     Total time elapsed: 26.3678s
     Test set results: loss= 1.0351 accuracy= 0.7847
```

6. Compare your models.

Compare the evaluation results for Vanilla GCN and GAT.

Vanilla GCN evaluation result

```
Optimization Finished!
Total time elapsed: 0.6792s
Test set results: loss= 1.0643 accuracy= 0.6515
```

GAT evaluation result

```
[36] Optimization Finished!
     Total time elapsed: 26.3678s
     Test set results: loss= 1.0351 accuracy= 0.7847
```

GAT performs better than Vanilla GCN as the GAT evaluation result shows a smaller loss and a greater accuracy level. This is because that GAT leverages the attention layer to take in more information into final layer.

Part 3: Deep Q-Learning Network (DQN)

Q1. Implementation of ε – greedy.

```
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
    Q, max_action = torch.max(Qp, axis=0)

    if torch.rand(1, ).item() > epsilon:
      action = max_action
    else:
      action = torch.randint(0, action_space_len, (1,))  # random action

    return action
```

Q2. Implementation of DQN training step.

```
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
    expected_return = torch.max(model.policy_net(state), axis=1)[0]

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

    # TODO: compute the loss
    loss = model.loss_fn(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()
```

Q3. Train your DQN Agent.

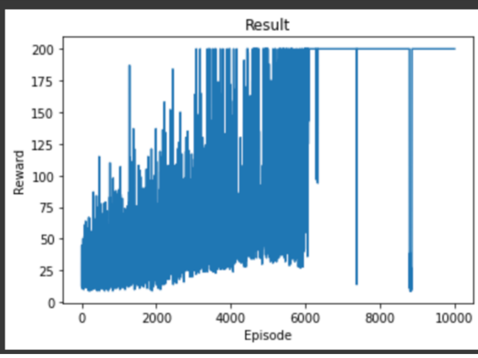
```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 260
memory = ExperienceReplay(exp_replay_size)
episodes = 10000
epsilon = 1 # episilon start from 1 and decay gradually.
```

```
    # TODO: add epsilon decay rule here!
    if epsilon > 0.05:
        epsilon -= (1 / 4000)

    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%|██████████| 10000/10000 [01:55<00:00, 86.68it/s]Saving trained model
```

The training time for the latter episodes is longer than the early episodes, as the agent is getting better and better at playing the game and thus each episode takes longer.

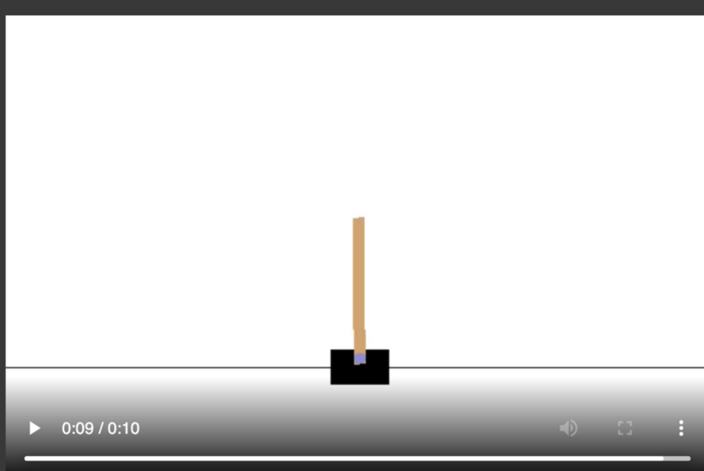
```
plot_reward(reward_list)
```



From the above plot of reward vs. episode, we could see that the reward tends to stabilize at 200 after running about 6000 episodes.

```
env.close()
show_video()
```

```
100%|██████████| 200/200 [00:34<00:00,  5.88it/s]average reward per episode : 500.0
```



From the above simulated video, we could see a much longer video (10 seconds) with a self-balancing pole. The agent plays the game very well with an average reward of 500 per episode.