# Programming Assignment 4: DCGAN, GCN, and DQN

Name: Ruyi Qu
Student Number: 1004849569

# Part 1: Deep Convolutional GAN (DCGAN)

## Generator Implementation

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

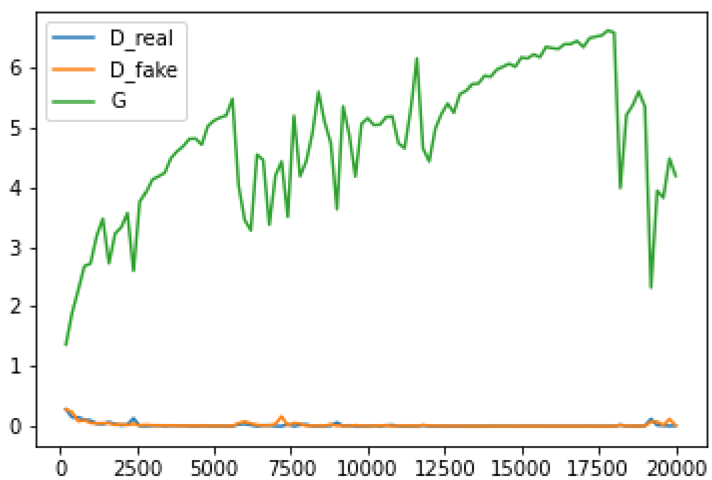
## Training Loop Implementation

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

    # --------------------------
    # 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(D(fake_images), 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)
```

## Experiment

### Q1




I trained the DCGAN to generate Windows emojis in the Training Section of the Notebook. Above is its corresponding loss graph. As we can see from this graph, result quality is overall increasing as the number of iteration increases. However, it does seem that the results are unstable. Sometimes there will be grayish result being generated after gaining a high quality result several iteractions eariler. One possible explanation to this situation is overfitting. However, regardless of the occasional bad results being produced, the quality of result we gained at the end (20000 iterations) is clearly better than what we got at the beginning.

Note that figure produced at 19800 iteration looks better than the one we got at 20000 iterations, this might be because of overfitting as well.

*2000 iteration -- early in the training*

*6200 iteration*

*13600 iteration*

*19400 iteration*

*19800 iteration -- satisfactory image quality*

*19400 iteration -- towards the end of training*

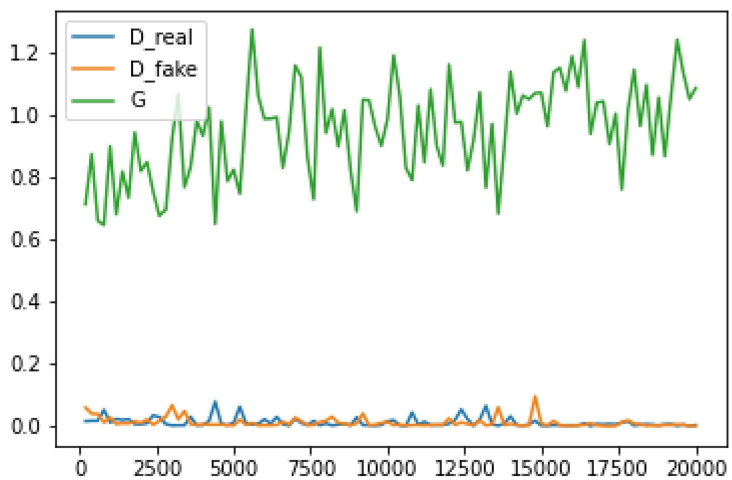
**Q2**

Below is the code I filled in in the gan_training_loop_leastsquares
function in the GAN section of the notebook

```python
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) ** 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()
```

```python
            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) / 2
        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()
```

We turn on the the least_squares_gan flag in the args_dict and
train the model again. The result loss vs iteration graph is shown below.




One obvious difference we can see from the regular GAN is that least-squares GAN seem to be producing images with a much lower loss than what's of regular GAN. Also, the results are more stable than what's of the regular GAN. There is no result that is extremely blurry in the middle of the training. For example, the results we obtained at iteration 13600 is shown below, which looks way better than what we got from regular GAN. To sum up, least-squares GAN is more stablized during the learning process and generates higher quality images than regular GANs

*regular GAN: 13600 iteration*

*least-squares GAN: 13600 iteration*

One problem with regular GAN is that its use of sigmoid cross entropy as loss function will lead to the vanishing gradient problem -- the discriminator gets too successful that the generator gradient vanishes and learns nothing -- as the fake data that are used for updating the generator can be sometimes too far away from the real data.

As pointed out by Xudong Mao et al. in "Least Squares Generative Adversarial Networks" in 2017, LSGANs solves this problem by pulling these fake data closer to the real data by applying panalities to those that are further away on the decision boundary.

# Part 2: Graph Convolution Networks

## Experiments

### Q1

```
[10] 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.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 ###################################

            output = torch.spmm(adj, self.layer(input))
            return output

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

**Q2**

```python
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.layer_1 = GraphConvolution(nfeat, n_hidden, bias)
    self.layer_2 = nn.Dropout(dropout)
    self.layer_3 = nn.ReLU()
    self.layer_4 = GraphConvolution(n_hidden, n_classes, bias)
```

```python
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.layer_2(self.layer_1(x, adj))
    x = self.layer_3(x)
    x = self.layer_4(x, adj)
    return x


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

**Q3**

```
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0095s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val: 0.6515 time: 0.0118s
Optimization Finished!
Total time elapsed: 1.2777s
Test set results: loss= 1.0643 accuracy= 0.6515
```

**Q4**

```python
    # 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)
```

```python
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).view(n_nodes, self.n_heads, self.n_hidden)
    sisj = torch.cat([s.repeat((n_nodes, 1, 1)),
                      s.repeat_interleave(n_nodes, dim=0)], dim=-1)
    sisj = sisj.view((n_nodes, n_nodes, self.n_heads, 2*self.n_hidden))
    attention = self.attention(sisj)
    e = self.activation(attention).squeeze(-1)
    e = e.masked_fill(adj_mat.unsqueeze(-1) == 0, float("-inf"))
    a = self.dropout_layer(self.softmax(e))

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

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

```python
# TODO: Concat or Mean
# Concatenate the heads
if self.is_concat:
    ############### Your code here ########################################
    return 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 ########################################
    return h_prime.mean(dim=1)
    #######################################################################
```

**Q5**

```
Epoch: 0099 loss_train: 0.9231 acc_train: 0.7929 loss_val: 1.0927 acc_val: 0.7321 time: 0.8442s
Epoch: 0100 loss_train: 0.9178 acc_train: 0.7857 loss_val: 1.0868 acc_val: 0.7325 time: 0.8465s
Optimization Finished!
Total time elapsed: 84.9473s
Test set results: loss= 1.0868 accuracy= 0.7325
```

**Q6**

As we can see from the evaluation results for Vanilla GCN and GAT, GAT has a relatively better test accuracy (about 0.72) but yet takes a lot longer to run (about 85s for GAT but 1.2s for GCT). GAT has a better accuracy because instead of making the coefficient solely dependent on the structure of the graph as what GCN does, GAT takes each nodes' features, which are passed into an attention function, into account. GAT requires more time because making the coefficient a learnable attention mechanism leads to more calculation and therefore more time to execute.

# Part 3: Deep Q-Learning Network (DQN)

## Experiments

**Q1**

```python
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, a = torch.max(Qp, axis=0)
    if torch.rand(1,).item() > epsilon:
        return a
    return torch.randint(0, action_space_len, (1,))
```

## Q2

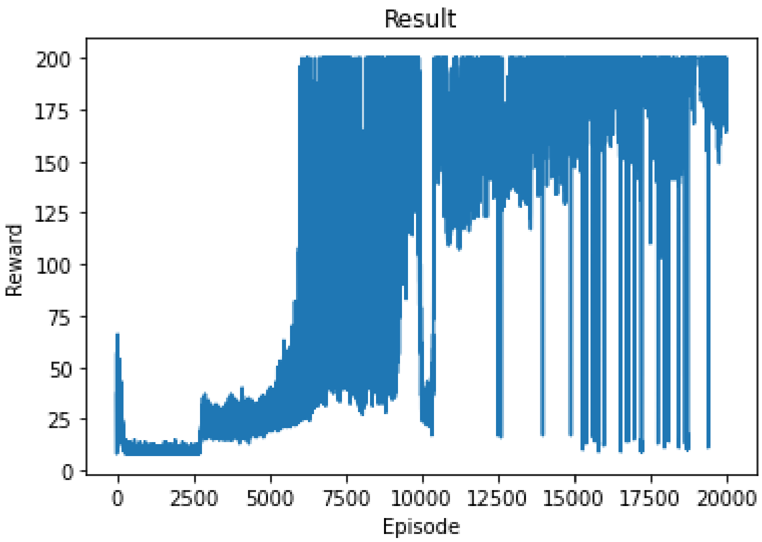
```python
def train(model, batch_size):
    state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)

    # TODO: predict expected return of current state using main network
    qp = model.policy_net(state)
    exp_ret = torch.max(qp, axis=1)[0]
    # TODO: get target return using target network
    tar_ret = torch.max(model.target_net(next_state), axis=1)[0] * model.gamma + reward


    # TODO: compute the loss
    loss = model.loss_fn(exp_ret, tar_ret)
    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

The parameters I chose are:

- exp_replay_size = 40
- episodes = 20000

  My epsilon decay rule is : epsilon = epsilon * 0.99




The final video I got is about 3 seconds long with a self-balancing pole moving to the right. The pole is a bit trembling though.