Part 1: Deep Convolutional GAN (DCGAN)

Implementation of DCGenerator's __init__ method:

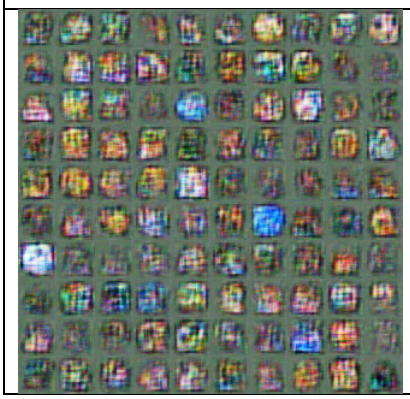
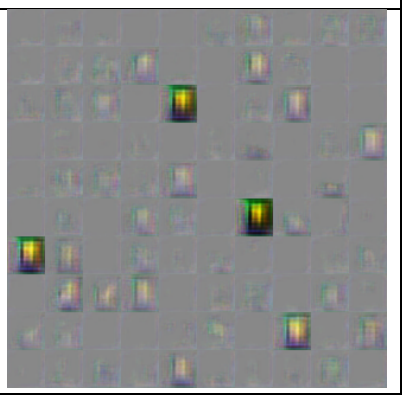
```
self.linear_bn = upconv(noise_size, self.conv_dim*4*4*4, kernel_size=1, stride=1, padding=0,
                        batch_norm=True, spectral_norm=spectral_norm)
self.upconv1 = upconv(conv_dim*4, conv_dim*2, kernel_size=5, stride=2, padding=2,
                      batch_norm=True, spectral_norm=spectral_norm)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

Implementation of training loop – Regular

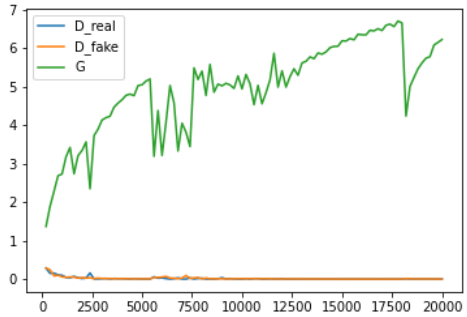
```
# Reset data_iter for each epoch
if iteration % iter_per_epoch == 0:
    train_iter = iter(dataloader)

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

ones_real = Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0),
                     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_real)


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

    # 3. Generate fake images from the noise
    fake_images = G(noise)
    zeros_fake = Variable(torch.Tensor(fake_images.shape[0]).float().cuda().fill_(0.0),
                          requires_grad=False)

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

    # ---- 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
```

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

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

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

Experiments and answers

Iteration num = 400	Iteration num = 10000	Iteration num = 20000
		

The loss curve:



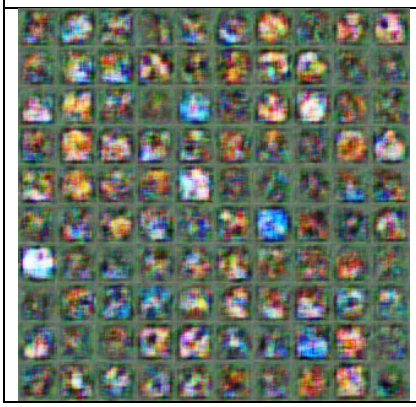
The quality of images improve but until a point. As you can see in the table iteration 10000 shows improvement. After some point because of the increase in the generative loss (also showed in the loss curve) vanishing gradient and loss saturation happen, the training becomes unstable, and the images are ruined. We use the least square loss to stabilize the GAN training and solving this issue.

Implementation of training loop – Least-squares

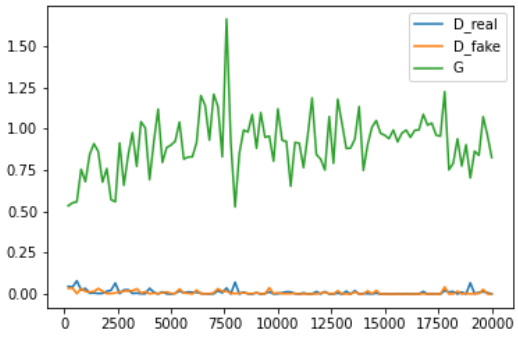
```
# Reset data_iter for each epoch
if iteration % iter_per_epoch == 0:
    train_iter = iter(dataloader)

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

    # 2. Sample noise
    ### batch size is real_images.shape[0]
    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) /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
```

```
########################################
###          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)
```

Experiments and answers

Iteration num = 400	Iteration num = 10000	Iteration num = 20000
		

The loss curve:



The loss curve shows that the training is more stable than the regular loss. There is no huge increase in generative loss and the quality of images are better. There is no ruined image at the final iterations either.

LSGAN helps because:

The least squares loss will penalize generated images based on their distance from the decision boundary. This will provide a strong gradient signal for generated images that are very different or far from the existing data and address the problem of saturated loss.

From: https://machinelearningmastery.com/least-squares-generative-adversarial-network/

Part 2: Graph Convolution Networks

1. Implementation of Graph Convolution Layer

```
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=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 ##################################
        out = self.W(input)
        return torch.spmm(adj, 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=True
        """

        super(GCN, self).__init__()
        # TODO: Initialization
        # (1) 2 GraphConvolution() layers.
        # (2) 1 Dropout layer
        # (3) 1 activation function: ReLU()
        ############ Your code here ##################################
        self.conv1 = GraphConvolution(nfeat, n_hidden, bias=bias)
        self.conv2 = GraphConvolution(n_hidden, n_classes, bias=bias)
        self.dropout = nn.Dropout(dropout)
        self.relu = 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.conv1(x, adj)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.conv2(x, adj)

        return x
```

3. Train your Graph Convolution Network

Results with default hyperparameters

```
Epoch: 0090 loss_train: 0.8969 acc_train: 0.8357 loss_val: 1.2217 acc_val: 0.6667 time: 0.0025s
Epoch: 0091 loss_train: 0.8162 acc_train: 0.8929 loss_val: 1.2297 acc_val: 0.6608 time: 0.0026s
Epoch: 0092 loss_train: 0.9124 acc_train: 0.8429 loss_val: 1.2022 acc_val: 0.6624 time: 0.0033s
Epoch: 0093 loss_train: 0.8378 acc_train: 0.8571 loss_val: 1.1941 acc_val: 0.6671 time: 0.0034s
Epoch: 0094 loss_train: 0.8411 acc_train: 0.8429 loss_val: 1.1969 acc_val: 0.6667 time: 0.0035s
Epoch: 0095 loss_train: 0.8258 acc_train: 0.8786 loss_val: 1.1897 acc_val: 0.6530 time: 0.0029s
Epoch: 0096 loss_train: 0.7872 acc_train: 0.9143 loss_val: 1.1887 acc_val: 0.6589 time: 0.0025s
Epoch: 0097 loss_train: 0.7441 acc_train: 0.8857 loss_val: 1.1821 acc_val: 0.6581 time: 0.0027s
Epoch: 0098 loss_train: 0.8019 acc_train: 0.8643 loss_val: 1.1773 acc_val: 0.6651 time: 0.0024s
Epoch: 0099 loss_train: 0.7884 acc_train: 0.8643 loss_val: 1.1690 acc_val: 0.6682 time: 0.0028s
Epoch: 0100 loss_train: 0.7571 acc_train: 0.8786 loss_val: 1.1729 acc_val: 0.6565 time: 0.0027s
Optimization Finished!
Total time elapsed: 0.3120s
Test set results: loss= 1.1568 accuracy= 0.6760
```

4. Implementation of Graph Attention Layer

```python
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, n_heads*self.n_hidden, bias=False)
    self.attention = nn.Linear(self.n_hidden*2, 1, bias=False)
    self.activation = nn.LeakyReLU(alpha)
    self.softmax = nn.Softmax(dim=1)
    self.dropout = nn.Dropout(dropout)
```

```
############## Your code here ########################################
#(1)
s = self.W(h)
s = s.view((n_nodes, self.n_heads, self.n_hidden)) #torch.Size([2708, 8, 2])

#(2)
s_i = s.repeat((n_nodes,1,1)) #torch.Size([7333264, 8, 2])
s_j = s.repeat_interleave(n_nodes, dim=0) #torch.Size([7333264, 8, 2])

cat_s = torch.cat((s_i, s_j), dim=-1)
cat_s = cat_s.view((n_nodes, n_nodes, self.n_heads, 2 * self.n_hidden)) #torch.Size([2708, 2708, 8, 4])

#(3) attention layer
a = self.attention(cat_s) #torch.Size([2708, 2708, 8, 1])

#(4) activation layer and (5)
e = self.activation(a).squeeze(-1) #torch.Size([2708, 2708, 8])

#(6)
mask = -np.inf*torch.ones_like(e) #torch.Size([2708, 2708, 8])
masked_e = torch.where(adj_mat.unsqueeze(-1) > 0, e, mask) #torch.Size([2708, 2708, 8])

# (7) softmax and (8)dropout
masked_e = self.softmax(masked_e)
masked_e = self.dropout(masked_e) #torch.Size([2708, 2708, 8])

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

# Summation
h_prime = torch.einsum('ijh,jhf->ihf', masked_e, s) #[n_nodes, n_heads, n_hidden] #torch.Size([2708, 8, 2]


# TODO: Concat or Mean
# Concatenate the heads
if self.is_concat:
    ############## Your code here ########################################
    #In the code, we only need to reshape the tensor to shape of [n_nodes, n_heads * n_hidden]
    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 ########################################
    #instead, we sum over the n_heads dimension:
    return torch.sum(h_prime, dim=1)

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

5. Train your Graph Attention Network

Results with default hyperparameters

```
Epoch: 0090 loss_train: 0.8828 acc_train: 0.8929 loss_val: 1.0664 acc_val: 0.8139 time: 0.1068s
Epoch: 0091 loss_train: 0.9551 acc_train: 0.8000 loss_val: 1.0589 acc_val: 0.8135 time: 0.1075s
Epoch: 0092 loss_train: 0.8869 acc_train: 0.8357 loss_val: 1.0517 acc_val: 0.8139 time: 0.1074s
Epoch: 0093 loss_train: 0.8949 acc_train: 0.8143 loss_val: 1.0444 acc_val: 0.8150 time: 0.1066s
Epoch: 0094 loss_train: 0.8864 acc_train: 0.8286 loss_val: 1.0374 acc_val: 0.8178 time: 0.1066s
Epoch: 0095 loss_train: 0.8668 acc_train: 0.8143 loss_val: 1.0306 acc_val: 0.8185 time: 0.1069s
Epoch: 0096 loss_train: 0.8360 acc_train: 0.8571 loss_val: 1.0236 acc_val: 0.8193 time: 0.1069s
Epoch: 0097 loss_train: 0.9084 acc_train: 0.7929 loss_val: 1.0170 acc_val: 0.8197 time: 0.1073s
Epoch: 0098 loss_train: 0.8461 acc_train: 0.8643 loss_val: 1.0107 acc_val: 0.8220 time: 0.1072s
Epoch: 0099 loss_train: 0.8349 acc_train: 0.8143 loss_val: 1.0045 acc_val: 0.8217 time: 0.1071s
Epoch: 0100 loss_train: 0.8310 acc_train: 0.8357 loss_val: 0.9985 acc_val: 0.8224 time: 0.1068s
Optimization Finished!
Total time elapsed: 10.7039s
Test set results: loss= 0.9985 accuracy= 0.8224
```

6. Compare your models

The test accuracy is 0.82 for GAT and 0.67 for Vanilla GCN. We can easily see the improvement that comes with using multi head attention. Because we leverage the masked self-attention layers to help the short coming of vanilla GCN. We focus of the most relevant parts of the input to make decisions. So that's why the results are better with GAT.

Part 3: Deep Q-Learning Network (DQN)

1. Implementation of epsilon-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
    if (random.uniform(0, 1) > epsilon):
        _, a = torch.max(Qp, axis=0)
        return a
    else:
        return torch.randint(0, action_space_len, (1,))
```

2. Implementation of DQN training step

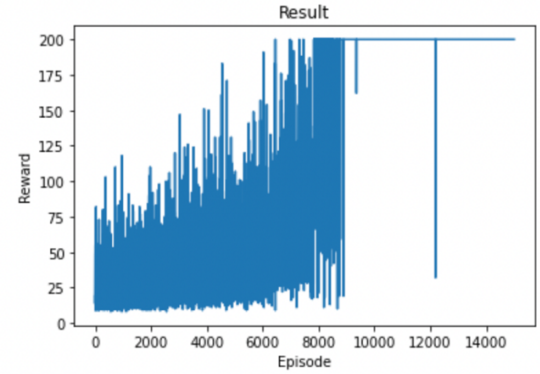
```
def train(model, batch_size):
    state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)

    #### Based on equation 3 and 4 in the handout
    # TODO: predict expected return of current state using main network
    qp = model.policy_net(state)
    pred, _ = torch.max(qp, axis=1)

    # TODO: get target return using target network
    next_qp = model.target_net(next_state)
    next_q, _ = torch.max(next_qp, axis=1)
    tar = reward + model.gamma * next_q

    # TODO: compute the loss
    loss = model.loss_fn(pred, tar)
    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. Train your DQN Agent

```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 128
memory = ExperienceReplay(exp_replay_size)
episodes = 15000
epsilon = 1 # episilon start from 1 and decay gradually.
```

```
# TODO: add epsilon decay rule
if epsilon > 0.01:
    epsilon = epsilon - 0.0001
```

Visualization result



The reward is perfectly converged to 200 around episode 9000.
The video too is stable for 10 seconds