# Part 1:

Generator init:

```
self.linear_bn = upconv(in_channels=100, out_channels=self.conv_dim*4, kernel_size=3, stride=2, padding=2, spectral_norm=spectral_norm)
self.upconv1 = upconv(in_channels=self.conv_dim*4, out_channels=self.conv_dim*2, kernel_size=5, stride=2, spectral_norm=spectral_norm)
self.upconv2 = upconv(in_channels=self.conv_dim*2, out_channels=self.conv_dim, kernel_size=5, stride=2, spectral_norm=spectral_norm)
self.upconv3 = upconv(in_channels=self.conv_dim, out_channels=3, kernel_size=5, stride=2, spectral_norm=spectral_norm)
```

gan_training_loop_regular code:

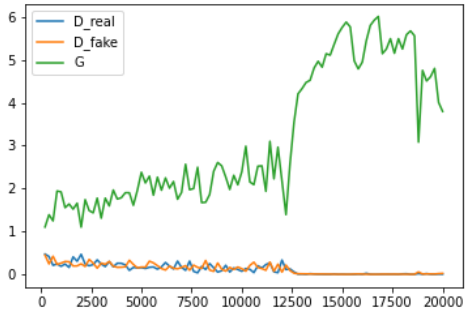
```
# FILL THIS IN
# 1. Compute the discriminator loss on real images
ones = Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0), requires_grad=False)
D_real_loss = adversarial_loss(D(real_images), ones)

# 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(real_images.shape[0]).float().cuda().fill_(0.0), requires_grad=False)
D_fake_loss = adversarial_loss(D(fake_images), zeros)
```

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

    G_loss.backward()
    g_optimizer.step()

    # Print the log info
```

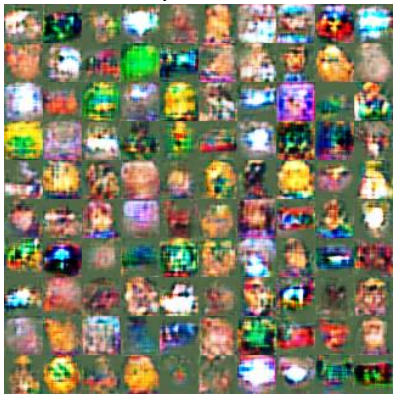
**Experiment results:**

Q1: training loss graph is displayed at right.

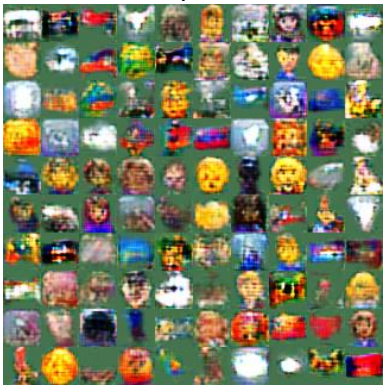
The loss function of generator has a tendency of increasing, and the loss became quite high around epoches 12000 to 18000. Three chosen images below shows image quality improvement.



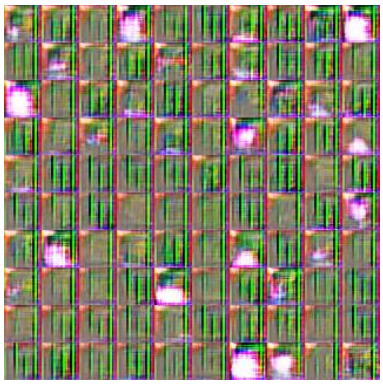
3200 epoches



9800 epoches



16800 epoches



At the early stage(3200), the emojis are not well-formed, but the general shape is obvious. In the middle stage however, the quality of images are the best, with details also being addressed clearly. However during the final stage, might because of overfitting, even the general shapes of emojis are gone.

Q2:

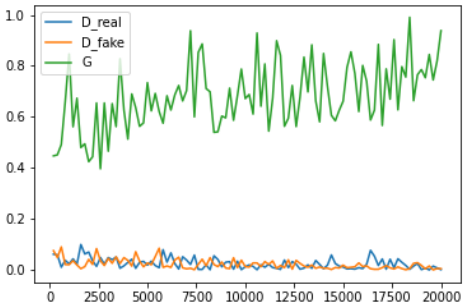
Least squares implementation

```
# 1. Compute the discriminator loss on real images
image_amount = real_images.shape[0]
D_real_loss = 1/2 * torch.mean((D(real_images) - 1)**2)

# 2. Sample noise
noise = sample_noise(image_amount, opts.noise_size)

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

# 4. Compute the discriminator loss on the fake image
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()
```

```
# 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()
```

Applying least squares method helps lowering G training loss to no higher than 1.3; Also by observing the generated images, the quality is also better. (loss function shown right)




Possible reasons for such improvement are: least squares resolves the problem of vanishing gradients by having a high gradient calculated for those samples far from desired image decision boundary.

# Part 2:

## class GraphConvolution

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

forward(self, input, adj):
# TODO: transform input feature to output (don't forget
# to sum over neighbouring nodes )
# hint: use the linear layer you declared above.
# hint: you can use torch.spmm() sparse matrix multiplic
#             adjacency matrix
############ Your code here ##################################
s = self.W(input)
output = torch.spmm(adj, s)
return output
```

## class GCN

### init

```
############ Your code here ##################################
self.gc1 = GraphConvolution(nfeat, n_hidden, bias)
self.gc2 = GraphConvolution(n_hidden, n_classes, bias)
self.drop_out = nn.Dropout(dropout)
self.act = nn.ReLU()
############################################################
```

### forward

```
############ Your code here ##########################
gc1_out = self.gc1.forward(x, adj)
relu_out = self.act.forward(gc1_out)
dropout_out = self.drop_out.forward(relu_out)
gc2_out = self.gc2.forward(dropout_out, adj)
# return nn.Softmax().forward(gc2_out)
return gc2_out
```

## GCN training results (100 epoches)

```
Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val: 0.6421 time: 0.0032s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val: 0.6452 time: 0.0023s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val: 0.6491 time: 0.0032s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val: 0.6515 time: 0.0070s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0038s
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: 0.5694s
Test set results: loss= 1.0643 accuracy= 0.6515
```

## class GraphAttentionLayer

### init

```
############ your code here ######################
self.W = nn.Linear(in_features, self.n_hidden * self.n_heads, bias=False)
self.attention = nn.Linear(self.n_hidden * 2, 1, bias=False)
self.activation = nn.LeakyReLU(alpha)
self.softmax = nn.Softmax(dim=1) # do softmax on the dimension of n_nde
self.dropout_layer = nn.Dropout(dropout)
```

### forward

```
############ Your code here ##################################
s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)
si_repeat = torch.repeat_interleave(s, n_nodes, dim=0)
sj_repeat = s.repeat(n_nodes, 1, 1)
si_sj = torch.cat([si_repeat, sj_repeat], dim=-1).view(n_nodes,
                                    n_nodes, self.n_heads, 2 * self.n_hidden)
attention_out = self.attention(si_sj)
activation_out = self.activation(attention_out).squeeze(-1)

mask = torch.where(adj_mat > 0, False, True)
mask = mask.unsqueeze(2)
mask_out = activation_out.masked_fill_(mask, float('-inf'))

softmax_out = self.softmax(mask_out)
a = self.dropout_layer(softmax_out)
```

## GAN training result:

```
Epoch: 0095 loss_train: 0.9759 acc_train: 0.8143 loss_val: 1.0773 acc_val: 0.7484 time: 0.2524s
Epoch: 0096 loss_train: 0.8995 acc_train: 0.7786 loss_val: 1.0707 acc_val: 0.7496 time: 0.2519s
Epoch: 0097 loss_train: 0.8695 acc_train: 0.8000 loss_val: 1.0640 acc_val: 0.7508 time: 0.2521s
Epoch: 0098 loss_train: 0.9250 acc_train: 0.7571 loss_val: 1.0578 acc_val: 0.7516 time: 0.2511s
Epoch: 0099 loss_train: 0.9572 acc_train: 0.7786 loss_val: 1.0514 acc_val: 0.7527 time: 0.2518s
Epoch: 0100 loss_train: 0.9285 acc_train: 0.7786 loss_val: 1.0451 acc_val: 0.7535 time: 0.2519s
Optimization Finished!
Total time elapsed: 25.2415s
Test set results: loss= 1.0451 accuracy= 0.7535
```

```
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 torch.mean(h_prime, dim=1)
```

Q6: compare your models:

By comparing the results acquired, GAN has a higher accuracy and lower training loss compared with Vanilla GCN, with the cost of increased training time.

As GAN involves attention layers, associations of other input features allows more comprehensive analysis for classification task, with the cost of more computations during training to analyze each input features' corresponding associations.

# Part 3:

def get_action:

```
## TODO: select and return action based on epsilon-greedy
Q, A = torch.max(Qp, axis=0)
## TODO:
if random.random() > epsilon:
    return A
else:
    return torch.randint(0, action_space_len, (1,))
```

def train:

```
# TODO: predict expected return of current state using
qp = model.policy_net(state)
pred_return, _ = torch.max(qp, axis=1)
# TODO: get target return using target network
qp_next = model.target_net(next_state)
q_next, _ = torch.max(qp_next, axis=1)
target_return = reward + model.gamma * q_next

# TODO: compute the loss
loss = model.loss_fn(pred_return, target_return)
model.optimizer.zero_grad()
```

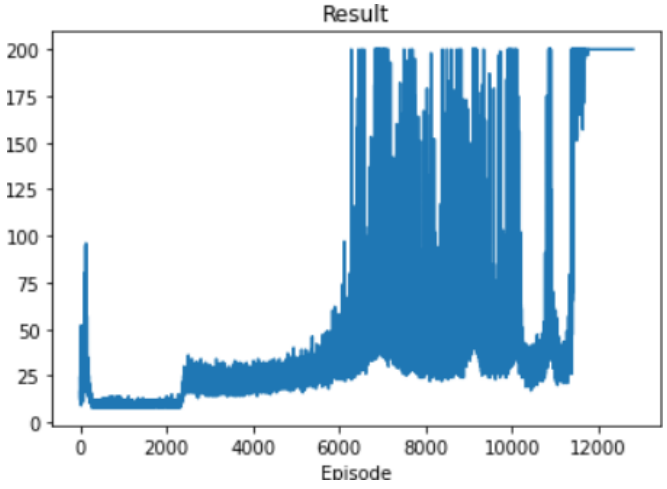
training loop hyperparameters:

```
exp_replay_size = 256
memory = ExperienceReplay(exp_replay_size)
episodes = 12800
epsilon = 1 # episilon start from 1
```

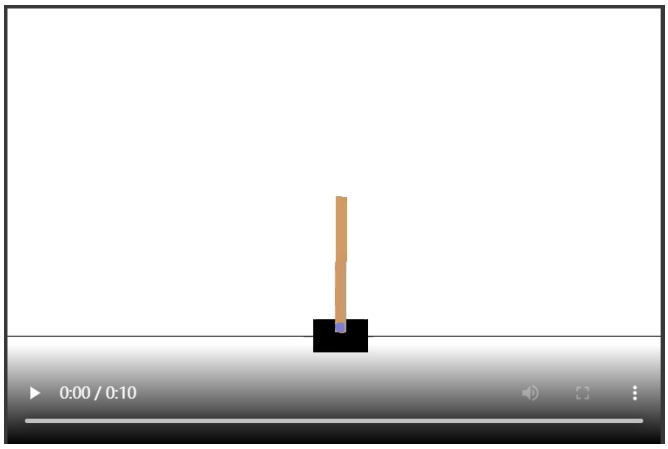
epsilon_decay_rule:

```
# TODO: add epsilon decay rule here!
if epsilon > 0.05:
    epsilon = epsilon * 0.99
```

training reward results:




demo_video: has 10 seconds




Result summarization:

As epsilon gets smaller and as more actions are explored, a clear standard for action choice is formed after 6500+ episodes, and finally converges after around 11000 episodes. Visualization also indicates the agent can balance the pole brilliantly that only tiny movements can be observed.