8810 Homework 3

Song Liao
Department of Computing
Clemson University
Clemson, USA
liao5@g.clemson.edu

I. Task Description

In this homework, we need to train a DCGAN and Wasserstein GAN on CIFAR10 dataset. After that, use inception score and FID score to evaluate the two models result.

II. Dataset Preprocess

In this task, we need to use the CIFAR 10 dataset, which contains 50000 images in 10 classes. These images are saved in 5 batch files and the order are 1024 red channel value, 1024 green value and 1024 blue value, which can be reshape as (3,32,32). After that, I resize the image size from 32*32 to 64*64 and reshape shape to (50000,64,64,3) as the input to the model. The value is set in range(0,1). The dataset preprocess part is shown in Fig.1.

```
[ ] def read_image():
        f=open('gan_dataset/data_batch_'+str(1),'rb')
        data=pickle.load(f,encoding='bytes')
        image_input=data[b'data']

        for i in range(2,6):
            f=open('gan_dataset/data_batch_'+str(i),'rb')
            data=pickle.load(f,encoding='bytes')
            image_input=np.concatenate((image_input,data[b'data']),axis=0)
            f.close()

        return image_input
```

```
[ ] image_input=read_image()
```

```
[ ] image_input2=[]
    for i in image_input:
        image_input2.append(np.reshape(i.reshape(3,32,32),3072,order='F'))

    image_input3=np.array(image_input2).reshape(50000,32,32,3)

    session = tf.Session()
    image_input4=session.run(tf.image.resize_images(image_input3,(64,64)))

    image_input=image_input4/255
```

Fig. 1. Dataset Preprocess

III. DCGAN

Deep Convolutional Generative Adversarial Networks (DCGAN) is an architecture for learning to generate new content. There are mainly two parts in the DCGAN, a generator to create new image and tries to fool the discriminator, and a discriminator which learns to distinguish fake images and real images. Two parts are both learning and competing with each other, then they both become stronger.

First is the generator. In this function, a 100 dim noise is used as input and then processed with a dense layer, which has 4*4*512 units. After that, the net has four transposed 2D convolution layers to increase the image size and decrease dimension. Each layer halves the filter number. Each part also contains a dropout layer and batch norm layer. At last, a tanh function is used to limit the output range. The output of this part is images with shape (64,64,3). The generator function is shown in Fig.2.

```
def generator(x):
    activation = tf.nn.relu
    with tf.variable_scope("generator",reuse=None):
        net = tf.layers.dense(inputs=x,units=4*4*512, activation=activation)
        net = tf.layers.dropout(net, 0.8)
        net = tf.contrib.layers.batch_norm(net, decay=0.9)

        net = tf.reshape(net, shape=[-1,4,4,512])

        net = tf.layers.conv2d_transpose(inputs=net,strides=2,padding='same',
                                filters=256, kernel_size=5, activation=activation)
        net = tf.layers.dropout(net, 0.8)
        net = tf.contrib.layers.batch_norm(net, decay=0.9)

        net = tf.layers.conv2d_transpose(inputs=net,strides=2,padding='same',
                                filters=128, kernel_size=5, activation=activation)
        net = tf.layers.dropout(net, 0.8)
        net = tf.contrib.layers.batch_norm(net, decay=0.9)

        net = tf.layers.conv2d_transpose(inputs=net,strides=2,padding='same',
                                filters=64, kernel_size=5, activation=activation)
        net = tf.layers.dropout(net, 0.8)
        net = tf.contrib.layers.batch_norm(net, decay=0.9)

        net = tf.layers.conv2d_transpose(inputs=net,strides=2,padding='same',
                                filters=3, kernel_size=5)

        net = tf.nn.tanh(net)

        return net
```

Fig. 2. Generator

Second part is the discriminator. Discriminator learn to distinguish the real images and fake images. The input is images and output is the probability that the images are real images. The input images with shape (64,64,3) are processed with five convolution layers and a flatter layer. At last, a dense layer with sigmod activation is used to ensure the range of output. The code of discriminator is shown in Fig.3.

Next is to build the graph. First I define the loss function for gan, which is based on the formula in paper.

Two placeholder for image and noise are defined. The generator can generate images with noise, and discriminator can analyze real images and fake images. Between them, the parameters in discriminator are reused. All parameters of generator and discriminator are obtained for optimizer. Here we define the loss function of real images as gan loss function from discriminator result to 1, which means real images. The fake images loss function is from result to 0. The loss of discriminator is average of real images loss and fake images loss. The loss of generator is from fake images to 1, which

```python
def discriminator(image_input,reuse=None):

    with tf.variable_scope("discriminator", reuse=reuse):

        image_input=tf.reshape(image_input,shape=[-1,64,64,3])

        net = tf.layers.conv2d(inputs=image_input, strides=2,padding='same',
                               filters=64, kernel_size=5, activation=tf.nn.leaky_relu)

        net = tf.layers.conv2d(inputs=net, strides=2,padding='same',
                               filters=128, kernel_size=5, activation=tf.nn.relu)

        net=tf.contrib.layers.batch_norm(net, decay=0.9,activation_fn=tf.nn.leaky_relu)

        net = tf.layers.conv2d(inputs=net, strides=2,padding='same',
                               filters=256, kernel_size=5, activation=tf.nn.relu)

        net=tf.contrib.layers.batch_norm(net, decay=0.9,activation_fn=tf.nn.leaky_relu)

        net = tf.layers.conv2d(inputs=net, strides=2,padding='same',
                               filters=512, kernel_size=5, activation=tf.nn.relu)

        net=tf.contrib.layers.batch_norm(net, decay=0.9)

        net = tf.layers.conv2d_transpose(inputs=net,strides=(1,1),
                               filters=512, kernel_size=1, activation=tf.nn.relu)

        net=tf.layers.flatten(net)

        net = tf.layers.dense(inputs=net,units=1, activation='sigmoid')

        return net
```

Fig. 3. Discriminator

```python
def gan_loss_function(x,y):
    eps=1e-12
    return (-(x*tf.log(y+eps)+(1-x)*tf.log(1-y+eps)))
```

Fig. 4. Gan Loss Function

means generated images should be close to real images. The graph build part is shown in Fig.6.

Then we can start to train this GAN. We set batch size to 64 and run 100 epoches. The training code is in Fig.7.

```python
noise = tf.placeholder(tf.float32, shape=[None, 100])

image = tf.placeholder(tf.float32, shape=[None,64,64,3])
```

Fig. 5. Placeholder

```python
g = generator(noise)

d_real=discriminator(image)
d_fake=discriminator(g,reuse=True)

vars_g = [var for var in tf.trainable_variables() if var.name.startswith("generator")]
vars_d = [var for var in tf.trainable_variables() if var.name.startswith("discriminator")]
d_reg = tf.contrib.layers.apply_regularization(tf.contrib.layers.l2_regularizer(1e-6), vars_d)
g_reg = tf.contrib.layers.apply_regularization(tf.contrib.layers.l2_regularizer(1e-6), vars_g)


loss_d_real=gan_loss_function(tf.ones_like(d_real),d_real)
loss_d_fake=gan_loss_function(tf.zeros_like(d_fake),d_fake)
loss_g=tf.reduce_mean(gan_loss_function(tf.ones_like(d_fake),d_fake))
loss_d=tf.reduce_mean(0.5*(loss_d_real+loss_d_fake))


update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
    optimizer_d = tf.train.AdamOptimizer(learning_rate=0.0002, beta1=0.5).minimize(loss_d , var_list=vars_d)
    optimizer_g = tf.train.AdamOptimizer(learning_rate=0.0002, beta1=0.5).minimize(loss_g , var_list=vars_g)
```

Fig. 6. Graph

IV. WGAN

The Wasserstein Generative Adversarial Network (WGAN) is an extension of the GAN that training model to better approximate the distribution of data in dataset. Insteat of using a discriminator to classify the probability of generated images being real or fake, WGAN replaces discriminator with a critic that scores the realness or fakeness of a give image. There are mainly six differences with GAN.

```python
j=0
epoch=100
saver = tf.train.Saver()
d_loss_list=[]
g_loss_list=[]

for i in range(0,epoch * 781):
    train_g = True
    train_d = True

    if j+batch_size > dataset_size:
        j=0
        print("epoch:"+str(int(i/781))+", d_loss:"+str(d_loss)+", g_loss:"+str(g_loss))
        save_path = saver.save(session, "dcgan_result/model_100.ckpt")
        pickle.dump(fake_image[0],open('dcgan_result/epoch_'+str(int(i/781))+'.p','wb'))
        np.random.shuffle(image_input)
        continue
    else:
        batch=image_input[j:j+batch_size]
        j=j+batch_size



    noise_input=np.random.random_sample((len(batch),100))

    d_real_loss, d_fake_loss, g_loss, d_loss=session.run([loss_d_real, loss_d_fake, loss_g, loss_d],
                                                         feed_dict={noise: noise_input, image:batch})
    fake_image=session.run([g],feed_dict={noise: noise_input, image:batch})

    d_loss_list.append(d_loss)
    g_loss_list.append(g_loss)

    d_real_loss=np.mean(d_real_loss)
    d_fake_loss=np.mean(d_fake_loss)

    if g_loss * 1.5< d_loss:
        train_g = False

    if d_loss * 2< g_loss:
        train_d = False

    if train_d:
        session.run(optimizer_d, feed_dict={noise: noise_input, image:batch})

    if train_g:
        session.run(optimizer_g, feed_dict={noise: noise_input, image:batch})
```

Fig. 7. Train

First is that WGAN use a linear activation function in the output layer of the critic mode, instead of the sigmod in the discriminator. Since the linear is default activation, I set it as None there. This function is shown in Fig.8.

```python
def critic_conv(img, reuse=False):
    with tf.variable_scope('critic') as scope:
        if reuse:
            scope.reuse_variables()
        size = 64

        img = ly.conv2d(img, num_outputs=size, kernel_size=5,
                        stride=2, activation_fn=lrelu)

        img = ly.conv2d(img, num_outputs=size * 2, kernel_size=5,
                        stride=2, activation_fn=lrelu, normalizer_fn=ly.batch_norm)

        img = ly.conv2d(img, num_outputs=size * 4, kernel_size=5,
                        stride=2, activation_fn=lrelu, normalizer_fn=ly.batch_norm)

        img = ly.conv2d(img, num_outputs=size * 8, kernel_size=5,
                        stride=2, activation_fn=lrelu, normalizer_fn=ly.batch_norm)

        logit = ly.fully_connected(tf.reshape(
            img, [batch_size, -1]), 1, activation_fn=None)

    return logit
```

Fig. 8. Critic

Second is the loss function of WGAN. It uses a new loss function to encourages the discriminator to predict a score of how real images looks. It means the discriminator is not a classifier but a critic. At last, the loss of critic is set as average of fake image score subtract true image score, and loss of generator is set as minus fake image score. When building graph, the RMSPropOptimizer is used to replace the AdamOptimizer because it is better recommended for WGAN. This part of code is shown in Fig.9.

Third is that updating critic more time than generator while training. In DCGAN, generator and discriminator are trained for same times, but for WGAN the critic should be more. In

```
c_loss = tf.reduce_mean(fake_logit - true_logit)
g_loss = tf.reduce_mean(-fake_logit)
g_loss_sum = tf.summary.scalar("g_loss", g_loss)
c_loss_sum = tf.summary.scalar("c_loss", c_loss)
img_sum = tf.summary.image("img", train, max_outputs=10)
theta_g = tf.get_collection(
    tf.GraphKeys.TRAINABLE_VARIABLES, scope='generator')
theta_c = tf.get_collection(
    tf.GraphKeys.TRAINABLE_VARIABLES, scope='critic')
counter_g = tf.Variable(trainable=False, initial_value=0, dtype=tf.int32)
opt_g = ly.optimize_loss(loss=g_loss, learning_rate=learning_rate_ger,
                optimizer=partial(tf.train.RMSPropOptimizer,
                variables=theta_g, global_step=counter_g,
                summaries = ['gradient_norm'])
counter_c = tf.Variable(trainable=False, initial_value=0, dtype=tf.int32)
opt_c = ly.optimize_loss(loss=c_loss, learning_rate=learning_rate_dis,
                optimizer=partial(tf.train.RMSPropOptimizer,
                variables=theta_c, global_step=counter_c,
                summaries = ['gradient_norm'])
```

Fig. 9. Building Graph

my training, the critic would be trained 5 times while generator is trained 1 time in an iteration. The training code is in Fig.10.

```
for j in range(0,5):
    batch=image_input[m:m+batch_size]
    m=m+batch_size
    if i % 100 == 99 and j == 0:
        run_options = tf.RunOptions(
            trace_level=tf.RunOptions.FULL_TRACE)
        run_metadata = tf.RunMetadata()
        _, merged = sess.run([opt_c, merged_all], feed_dict={real_data:batch},
                             options=run_options, run_metadata=run_metadata)
        summary_writer.add_summary(merged, i)
        summary_writer.add_run_metadata(
            run_metadata, 'critic_metadata {}'.format(i), i)
    else:
        sess.run(opt_c, feed_dict={real_data:batch})

batch=image_input[m:m+batch_size]
m=m+batch_size
if i % 100 == 99:
    _, merged = sess.run([opt_g, merged_all], feed_dict={real_data:batch},
          options=run_options, run_metadata=run_metadata)
    summary_writer.add_summary(merged, i)
    summary_writer.add_run_metadata(
        run_metadata, 'generator_metadata {}'.format(i), i)
else:
    sess.run(opt_g, feed_dict={real_data:batch})
if i % 1000 == 999:
    saver.save(sess, os.path.join(
        ckpt_dir, "model.ckpt"), global_step=i)
```

Fig. 10. WGAN Training

V. Evaluation

We use the Inception Score and Frechet Inception Distance (FID) to evaluate the result.

The Inception Score uses a pre-trined DNN to classify the generated images. The score captures two properties of generatid images, image quality, whether images look like a specific object, and image diversity, whether a wide range of objects generated. For the CIFAR-10 dataset, it has 10 classes of objects, and the inception score is about 11.

The inception score is calculated by first using a pre-trianed inception v3 model to predict the class probabilities for each generated images. These are conditional probabilities. Images that are classified strongly as one class over all other classes indicate a hith quality. The code of inception score is shown in Fig.11

The Frechet Inception Distance (FID) is a metric for evaluating quality of generated images. Like the inception score, the FID uses the inception v3 mode. But the FID can also capture the mean and covariance of images, and they are then

```
inception_images = tf.compat.v1.placeholder(tf.float32, [None, 3, None, None])
def inception_logits(images = inception_images, num_splits = 1):
    images = tf.transpose(images, [0, 2, 3, 1])
    size = 299
    images = tf.compat.v1.image.resize_bilinear(images, [size, size])
    generated_images_list = array_ops.split(images, num_or_size_splits = num_splits)
    logits = tf.map_fn(
        fn = functools.partial(
            tfgan.eval.run_inception,
            default_graph_def_fn = functools.partial(
            tfgan.eval.get_graph_def_from_url_tarball,
            INCEPTION_URL,
            INCEPTION_FROZEN_GRAPH,
            os.path.basename(INCEPTION_URL)),
            output_tensor = 'logits:0'),
        elems = array_ops.stack(generated_images_list),
        parallel_iterations = 8,
        back_prop = False,
        swap_memory = True,
        name = 'RunClassifier')
    logits = array_ops.concat(array_ops.unstack(logits), 0)
    return logits

logits=inception_logits()

def get_inception_probs(inps):
    n_batches = int(np.ceil(float(inps.shape[0]) / BATCH_SIZE))
    preds = np.zeros([inps.shape[0], 1000], dtype = np.float32)
    for i in range(n_batches):
        inp = inps[i * BATCH_SIZE:(i + 1) * BATCH_SIZE] / 255. * 2 - 1
        preds[i * BATCH_SIZE : i * BATCH_SIZE + min(BATCH_SIZE, inp.shape[0])] = session.run(logits,{inception_images: inp})[:, :1000]
    preds = np.exp(preds) / np.sum(np.exp(preds), 1, keepdims=True)
    return preds

def preds2score(preds, splits=10):
    scores = []
    for i in range(splits):
        part = preds[(i * preds.shape[0] // splits):((i + 1) * preds.shape[0] // splits), :]
        kl = part * (np.log(part) - np.log(np.expand_dims(np.mean(part, 0), 0)))
        kl = np.mean(np.sum(kl, 1))
        scores.append(np.exp(kl))
    return np.mean(scores), np.std(scores)
```

Fig. 11. Inception Score

```
def inception_activations(images = inception_images, num_splits = 1):
    images = tf.transpose(images, [0, 2, 3, 1])
    size = 299
    images = tf.compat.v1.image.resize_bilinear(images, [size, size])
    generated_images_list = array_ops.split(images, num_or_size_splits = num_splits)
    activations = tf.map_fn(
        fn = functools.partial(tfgan.eval.run_inception, output_tensor = 'pool_3:0'),
        elems = array_ops.stack(generated_images_list),
        parallel_iterations = 8,
        back_prop = False,
        swap_memory = True,
        name = 'RunClassifier')
    activations = array_ops.concat(array_ops.unstack(activations), 0)
    return activations

activations =inception_activations()

def get_inception_activations(inps):
    n_batches = int(np.ceil(float(inps.shape[0]) / BATCH_SIZE))
    act = np.zeros([inps.shape[0], 2048], dtype = np.float32)
    for i in range(n_batches):
        inp = inps[i * BATCH_SIZE : (i + 1) * BATCH_SIZE] / 255. * 2 - 1
        act[i * BATCH_SIZE : i * BATCH_SIZE + min(BATCH_SIZE, inp.shape[0])] = session.run(activations, feed_dict = {inception_images: inp})
    return act

def activations2distance(act1, act2):
    return session.run(fcd, feed_dict = {activations1: act1, activations2: act2})

def get_fid(images1, images2):
    assert(type(images1) == np.ndarray)
    assert(len(images1.shape) == 4)
    assert(images1.shape[1] == 3)
    assert(np.min(images1[0]) >= 0 and np.max(images1[0]) > 10), 'Image values should be in the range [0, 255]'
    assert(type(images2) == np.ndarray)
    assert(len(images2.shape) == 4)
    assert(images2.shape[1] == 3)
    assert(np.min(images2[0]) >= 0 and np.max(images2[0]) > 10), 'Image values should be in the range [0, 255]'
    assert(images1.shape == images2.shape), 'The two numpy arrays must have the same shape'
    print('Calculating FID with %i images from each distribution' % (images1.shape[0]))
    start_time = time.time()
    act1 = get_inception_activations(images1)
    act2 = get_inception_activations(images2)
    fid = activations2distance(act1, act2)
    print('FID calculation time: %f s' % (time.time() - start_time))
    return fid
```

Fig. 12. FID

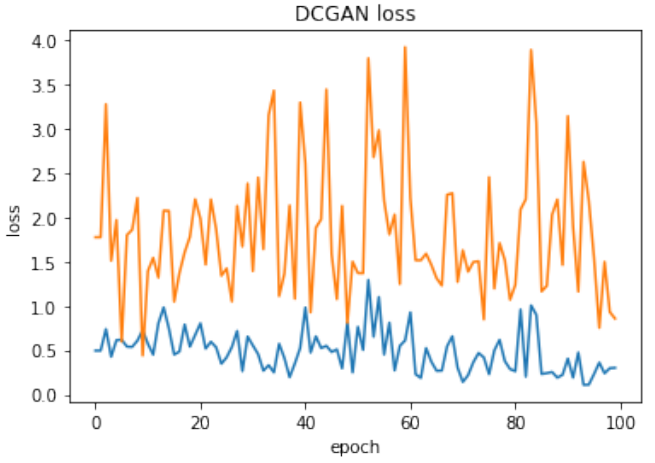

Fig. 13. DCGAN loss

calculated for the activations acrros the real images and fake images. Then the distance between these two distribution is calculated using the Frechet distance. The code of FID is shown in Fig.12.

As a result, the loss function of DCGAN is shown in Fig.13 and the loss of WGAN is shown in Fig.

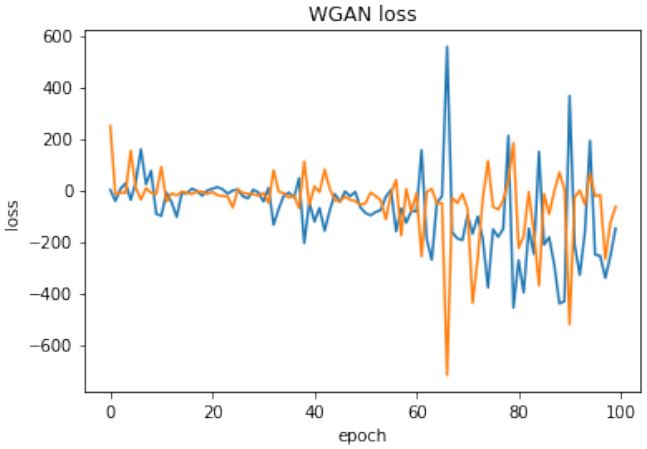

Fig. 14. WGAN loss

In my result, the best inception score is 2.8 in DCGAN and most are around 2.4, and the best inception score in WGAN is only 2.1, when most are 1.8. The best FID in DCGAN is about 270, while most are about 290. The best in WGAN is about 260, while most are about 270.

The WGAN seems not better than DCGAN. This may because of the loss function of WGAN is straightly set as reduce mean of critic and this also cause the loss value of WGAN pretty huge.

The best ten images are shown in Fig.15.

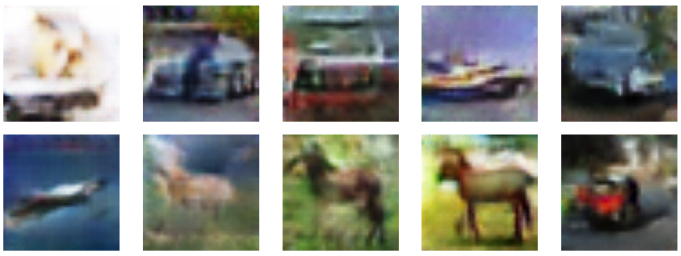

Fig. 15. result