Names: Justin Kieu, Louis Yang, Sheng-Hung Hung

1. **Explain what you have implemented during this week.**

Code:

```
def make_generator_model():
  model = tf.keras.Sequential()
  model.add(layers.Dense((256 * 8 * 8), input_shape=(100,)))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())

  model.add(layers.Reshape((8, 8, 256)))
  assert model.output_shape == (None, 8, 8, 256)  # Note: None is the
batch size

  model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())

  model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())

  model.add(layers.Conv2DTranspose(32, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())

  model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2),
padding='same', activation='tanh', use_bias=False))

  return model

def make_discriminator_model():
  model = tf.keras.Sequential()
  model.add(layers.Input(shape=(128, 128, 3)))
  model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same'))
```

```
  model.add(layers.LeakyReLU())

  model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.5))

  model.add(layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.5))

  model.add(layers.Flatten())
  model.add(layers.Dense(1))

  return model
```

The above code creates the model for both our discriminator and our generator. The generator requires 4 deconvolution layers with a 5x5 kernel, while the discriminator requires 3 convolution layers. They both require 2x2 strides as well.

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
generator_optimizer = tf.keras.optimizers.Adam(0.0002)
discriminator_optimizer = tf.keras.optimizers.Adam(0.0002)

def generator_loss(fake_output):
  return cross_entropy(tf.ones_like(fake_output), fake_output)

def discriminator_loss(real_output, fake_output):
  real_loss = cross_entropy(tf.ones_like(real_output), real_output)
  fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
  total_loss = real_loss + fake_loss
  return total_loss
```

We implement Adam optimizer to train deep learning models and the Binary Cross Entropy for the losses. We set the learning rate to 0.0002 as we believed this would produce the best results for us.

```
checkpoint_dir = 'drive/MyDrive/train/DCGAN_check'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                 discriminator_optimizer=discriminator_optimizer,
                 generator=generator,
                 discriminator=discriminator)
```

```
manager = tf.train.CheckpointManager(checkpoint, checkpoint_prefix,
max_to_keep=5)
checkpoint.restore(manager.latest_checkpoint)
```

We created a checkpoint to store the weights of the model we have in the training process, so we do not have to leave it running 24/7.

```
BATCH_SIZE = 256

all_images = tf.keras.preprocessing.image_dataset_from_directory(
    'drive/MyDrive/train/img_align_celeba/',
    batch_size=1,
    image_size=(218,178),
    shuffle=True,
    labels=None
)

def process(image):
  image = tf.reshape(image, [1, 218, 178, 3])
  image = tf.image.crop_and_resize(image, [[0.14, 0.205, 0.86, 0.795]],
[0], [128, 128])
  image = tf.cast((image-127.5) / 127.5 ,tf.float32)
  return image

all_images = all_images.map(process)
all_images = all_images.batch(BATCH_SIZE)
```

This section takes our images of size 218 x 178 and recrops them all to a 128 x 128 image.

```
noise_dim = 100

@tf.function
def train_step(real_images, current_batch_size):
  noise = tf.random.normal([current_batch_size, noise_dim])

  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    generated_images = generator(noise, training=True)

    real_output = discriminator(real_images, training=True)
    fake_output = discriminator(generated_images, training=True)
```

```
    gen_loss = generator_loss(fake_output)
    disc_loss = discriminator_loss(real_output, fake_output)

  gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
  generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
  gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
  discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))

  return gen_loss, disc_loss, real_images, generated_images, fake_output


def train(dataset, epochs, last_epoch):
  for epoch in range(epochs):
    start = time.time()
    recent_epoch = epoch + last_epoch + 1
    data = []
    real_images = []
    generated_images = []
    batch_num = 0
    print('Start training for epoch {}'.format(recent_epoch))

    for images in dataset:
      current_batch_size = images.shape[0]
      images = tf.reshape(images, [current_batch_size, images.shape[2],
images.shape[3], images.shape[4]])

      gen_loss, disc_loss, real_image, generated_image, fake_output =
train_step(images, current_batch_size)
      batch_num += 1
      if (batch_num % 10) == 0:
        r = np.random.randint(current_batch_size)
        data.append((gen_loss.numpy(), disc_loss.numpy(),
fake_output.numpy()[r]))
        real_images.append(real_image.numpy()[r])
        generated_images.append(generated_image.numpy()[r])
        print('Batch {} training finished'.format(batch_num))
```

```
    if (recent_epoch % 5) == 0:
      manager.save()

    save_result(data, real_images, generated_images, recent_epoch,
'drive/MyDrive/train/DCGAN_result')
    display.clear_output(wait=True)
    print ('Time for epoch {} is {} sec'.format(recent_epoch,
time.time()-start))
    print ('generator loss:', gen_loss.numpy())
    print ('discriminator loss:', disc_loss.numpy())


def save_result(data, real_images, generated_images, epoch_num, loc):
  wb = xlsxwriter.Workbook(f'{loc}/epoch{epoch_num:03}.xlsx')
  os.makedirs(f'{loc}/epoch{epoch_num:03}/real', exist_ok=True)
  os.makedirs(f'{loc}/epoch{epoch_num:03}/generated', exist_ok=True)
  ws = wb.add_worksheet()
  ws.write_row(0, 0, ('Batch Index', 'Generator Loss', 'Discriminator
Loss', 'Generated Image Prediction'))
  batch_num = 1
  for result, real_img, gene_img in zip(data, real_images,
generated_images):
    ws.write_row(batch_num, 0, (batch_num, result[0], result[1],
result[2]))
    save_img = (real_img * 127.5 + 127.5)
    save_img = PIL.Image.fromarray(np.uint8(save_img))
    save_img.save(f'{loc}/epoch{epoch_num:03}/real/{batch_num:03}.png')
    save_img = (gene_img * 127.5 + 127.5)
    save_img = PIL.Image.fromarray(np.uint8(save_img))

save_img.save(f'{loc}/epoch{epoch_num:03}/generated/{batch_num:03}.png')
    batch_num += 1
  wb.close()
```

This section of code sets up the training for our algorithm to recognize if it is a fake image or a real image. The generator takes 100 dimension Gaussian noise input, and creates a fake image. Then, the discriminator takes both the real and fake image to determine if those images are real or not.

Justin worked on the discriminator while Louis worked on the generator. Sheng-Hung worked on implementing both the Binary Cross Entropy and the Adam Optimizer. Sheng also worked on implementing a checkpoint to store the weight of the model. We then came back together and worked on the image resize function, train_step, and train functions. We don't have any outputs yet as we are still training the algorithm. We changed our data set to be one with 50,000 images from the previous 200,000 images. We need a few days to train our algorithm to get good results. In the next report, we should be able to give some results on what our algorithm produces. We didn't run into many major errors. We mostly had small ones that were quickly resolved.

2. **Any challenges you faced during this week? If so, how are you planning to resolve it? Any solutions or ideas?**

Most of our challenges throughout the week were just getting the code to work as we wanted it to. We managed to eventually resolve it by conversing among each other and seeing what we were trying to do.