2020 딥러닝 프로젝트 최종보고서

학과	전자통신공학과
학번	2016707072
이름	신민정

프로젝트 명

Emotional Voice Conversion

문제정의

아이폰의 시리, 안드로이드의 빅스비 같은 음성 인공지능 서비스는 스마트폰의 서비스를 음성으로 알려주는 역할을 한다. 뿐만 아니라 음성 인식, 번역과 같은 다양한 작업을 수행한다. 이렇게 인공지능으로 만들어낸 목소리는 평탄한 기계음으로 출력된다. 내용에 상관없이 평탄한 톤의 음성을 출력하기 때문에 이질감이 느껴질 때가 있다. 이러한 문제는 이제 vocoder로 쉽게 해결할 수 있다. 가장 sota model인 WaveNet vocoder로 만들어진 음성은 실제 사람의 음성과 매우 흡사한 톤을 가진다.

하지만, 이는 neutral tone에 국한되어있어 감정을 갖는 음성을 구현하기는 힘들다는 문제가 있다. 이러한 문제를 해결하기 위해 음성에 감정을 추가하여 새로운 음성을 생성하는 프로젝트를 진행하였다.

과제목표

기존 음성에 다양한 **감정을 추가**하고 **정도를 조절**하여 새로운 감정을 음성을 **생성**한다.

Happy한 음성 -> 상대적으로 50% sad음성

제안방법

- RelGAN (Multi-Domain Image-to-Image Translation via

Relative Attributes)

기존 음성에서 스타일을 변화하여 새로운 음성을 생성하는 task이므로 생성모델인 GAN을 사용하였다. 이번 프로젝트에서 가장 중요한 점은 1. 하나의 모델로 입력 음성에 추가할 감정을 선택할 수 있으며, 2.그 농도를 조절할 수 있는 것 이다.

이러한 이유로, 하나의 모델로 attribute를 선택할 수 있으며 그 정도를 조절할 수 있는 **RelGAN**을 사용하였다. (프로젝트에서 attribute는 음성의 감정이 된다.)

RelGAN은 RelGAN (Multi-Domain Image-to-Image Translation via Relative Attributes)논문에서 제안되었다.

#main idea

1) 속성에 보간법을 사용하여 translation을 연속적인 상태로 표현하자

2) Relative Attributes를 사용하여 바꾸고 싶은 속성만 선택하여 쉽게 조작□ 수 있도록 만들자

Relative Attributes

input image $\boldsymbol{x}$와 $\boldsymbol{x}$의 속성 $\boldsymbol{a}$, 목표 속성 $\hat{\boldsymbol{a}}$이 주어질 때, relative attributes v를 구한 후 목표 이미지 $\boldsymbol{y}$를 출력한다.

$$\boldsymbol{v} \triangleq \hat{\boldsymbol{a}} - \boldsymbol{a}$$

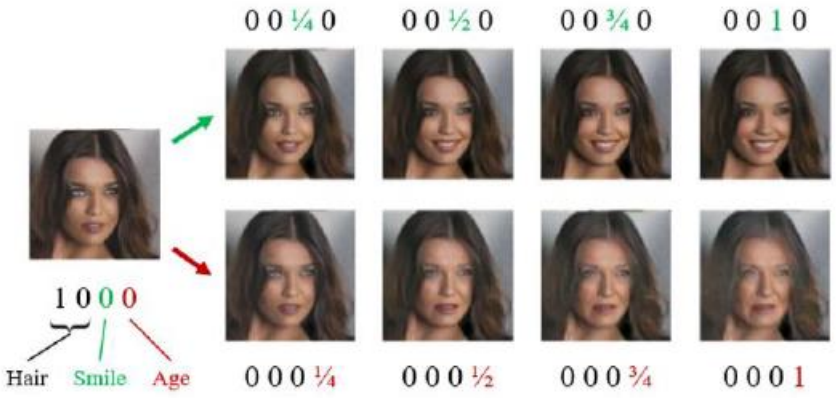
여기서 $\boldsymbol{v}$는 $\boldsymbol{x}$를 $\boldsymbol{y}$로 바꾸기 위해서 기존 $\boldsymbol{a}$ 에서 얼만큼 변화해야 한다는 가에 대한 지표이다.

#Attribute interpolation

속성에 강도 scaling을 하여 translation을 연속적인 형태로 표현한 것이다.

타겟 $\boldsymbol{y}$를 $\boldsymbol{G}(\boldsymbol{x},\boldsymbol{v})$ 라고 할 때, **α**를 relative attributes **v**에 곱해준다. **(α->0~1사이의 값)**

G(x, αv)



#Loss

1)Adversarial loss

기존 GAN의 것과 동일하다. G를 통해 생성된 이미지가 진짜처럼 보이게 만들면 된다. D_{Real}는 실제 이미지 x와 생성된 이미지 중에서 진짜를 구별한다.

$$\min_G \max_{D_{\text{Real}}} \mathcal{L}_{\text{Real}} = \mathbb{E}_{\mathbf{x}} \left[\log D_{\text{Real}}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{x},\mathbf{v}} \left[\log(1 - D_{\text{Real}}(G(\mathbf{x},\mathbf{v})))\right]$$

2)Conditioal Adversarial loss

G로부터 생성된 이미지는 진짜 같을 뿐만 아니라 relative attributes v만큼 충실히 히 변화를 주어야한다.D_{Match}는 x와 v를 입력 받아서 실제 y와 G(x,v) 둘 중 어느 것이 y인지 맞춘다.

$$\min_G \max_{D_{\text{Match}}} \mathcal{L}_{\text{Match}} = \mathbb{E}_{\mathbf{x},\mathbf{v},\mathbf{x}'} \left[\log D_{\text{Match}}(\mathbf{x},\mathbf{v},\mathbf{x}')\right] + \mathbb{E}_{\mathbf{x},\mathbf{v}} \left[\log(1 - D_{\text{Match}}(\mathbf{x},\mathbf{v},G(\mathbf{x},\mathbf{v})))\right].$$

3)Reconstruction loss

G가 v에 따라서 속성에 변□를 줄 때, 각각의 속성이 다른 특징까지 건드려서는 안된다.(e.g. 배경, 얼굴의 다른 특징들)

속성의 변화가 우리가 기대하는 부분만 정확히 영향을 끼칠 수 있도록 2가지 loss를 사용한다.

1. Cycle-reconstruction loss

CycleGAN의 loss와 동일하다. 아이디어는 x에 v만큼 변화를 준 후, 정반대로 -v로 변화를 준다면 x로 돌아와야 한다는 것.

x → G(v) → G(-v) → x

$$\min_G \mathcal{L}_{\text{Cycle}} = \mathbb{E}_{\mathbf{x},\mathbf{v}} \left[\| G(G(\mathbf{x}, \mathbf{v}), -\mathbf{v}) - \mathbf{x} \|_1 \right].$$

2. Self-reconsruction loss

Attributes v가 0벡터라면 어떻게될까? 우리의 목적에 따르면 0벡터 v를 입력하면 아무런 변화가 없어야 한다.

$$\min_G \mathcal{L}_{\text{Self}} = \mathbb{E}_{\mathbf{x}} \left[\| G(\mathbf{x}, \mathbf{0}) - x \|_1 \right],$$

4)Interpolation loss

v에 interpolation coefficient인 **α**를 곱할 때 우리가 원하는 만큼 scaling되길 바란다. 즉, **G(x, αv)**를 진짜처럼 잘 만들면 된다.

$$\min_{D_{\text{Interp}}} \mathcal{L}^D_{\text{Interp}} = \mathbb{E}_{\mathbf{x},\mathbf{v},\alpha} [\| D_{\text{Interp}}(G(\mathbf{x}, \alpha\mathbf{v})) - \hat{\alpha} \|^2 + \| D_{\text{Interp}}(G(\mathbf{x}, \mathbf{0})) \|^2 + \| D_{\text{Interp}}(G(\mathbf{x}, \mathbf{v})) \|^2],$$

$$\min_G \mathcal{L}^G_{\text{Interp}} = \mathbb{E}_{\mathbf{x},\mathbf{v},\alpha} \left[\| D_{\text{Interp}}(G(\mathbf{x}, \alpha\mathbf{v})) \|^2 \right],$$

WORLD Vocoder

학습에 사용될 feature을 뽑는데 WORLD Vocoder를 사용하였다.

WOLRD Vocoder로 뽑을 수 있는 feature중 f_0, MCEPs 두개를 사용하였다.

f_0: 사람이 소리를 들었을 때 인지하는 음의 뿌리가 되는 주파수

MCEPs : Mel Filter가 적용되지 않은 Spectral Envelop

WORLD Vocoder로 뽑은 음성의 feature로 GAN이 경쟁학습하며 새로운 음성을 만들어 낸다.

구현

제안논문에 paper with code로 첨부된 keras로 구현되어있는 official code를 참고하였다. Image domain의 code를 음성 domain의 code로 변환하였고, dataloader, preprocessing 코드를 추가하여 구현하였다.

-Train.py-

1. Load pickle

MCEPs와 MCEPs의 평균, 표준편차, log f_0값 평균과 표준편차를 각 도메인 데이터셋 별 음성 feature data 적재

(WORLD vocoder 코드 생략

WORLD Vocoder를 이용한 preprocessing= 다음 github참조 https://github.com/alpharol/Voice_Conversion_CycleGAN2)

[[화남1 MCEPs, 화남2 MCEPs, · · · ·]

[기쁨1 MCEPs, 기쁨2 MCEPs, · · · ·]]

```
print('Loading cached data...')
coded_sps_norms = []
coded_sps_means = []
coded_sps_stds = []
log_f0s_means = []
log_f0s_stds = []
for f in glob.glob(os.path.join(argv.dataset_dir, '*/')): # All attributes
    coded_sps_norm, coded_sps_mean, coded_sps_std, log_f0s_mean, log_f0s_std = load_pickle(
        os.path.join(f, f'cache{hp.num_mceps}.p'))
    coded_sps_norms.append(coded_sps_norm)
    coded_sps_means.append(coded_sps_mean)
    coded_sps_stds.append(coded_sps_std)
    log_f0s_means.append(log_f0s_mean)
    log_f0s_stds.append(log_f0s_std)
```

2. Compile model

Domain의 개수를 구한다 (화남, 기쁨, 슬픔 = 3).

RelGAN 모델 객체를 선언한다.

schedules클래스를 사용하여 Discriminator와 Generator에 쓰일 optimizer를 선언한다.

```
# model and optimizers.
# num_domains = len(coded_sps_norms)
domains = glob.glob(os.path.join(argv.dataset_dir, '*/'))
num_domains = len(domains)
print('num_domains = ', num_domains)
model = RelGAN(num_domains, hp.batch_size)

gen_lr_fn = tf.optimizers.schedules.PolynomialDecay(hp.generator_lr, hp.num_iterations, 1e-05)
dis_lr_fn = tf.optimizers.schedules.PolynomialDecay(hp.discriminator_lr, hp.num_iterations, 2e-05)
generator_optimizer = tf.optimizers.Adam(learning_rate=gen_lr_fn, beta_1=0.5)
discriminator_optimizer = tf.optimizers.Adam(learning_rate=dis_lr_fn, beta_1=0.5)
```

3. Sample train data

전체 학습 셋에서 batch size만큼 데이터셋을 리턴한다.

각 배치마다 랜덤한 도메인 3개를 선택한다 (x, y, z)

예시)

x = [batch_size, num_mcep(36), n_frames(128)]

x_atr = [1, batch_size] # [1, 3, 2, 2, 1, 2, 3, 1]

x_label = [0, 1, 0, 0] # if domain = 1

```
# batch sample
# x = [batch_size, num_mcep, n_frames]
# x_atr = [1, batch_size]
x, x2, x_atr, y, y_atr, z, z_atr = sample_train_data(dataset_A=coded_sps_norms, nBatch=hp.batch_size)

x_labels = np.zeros([hp.batch_size, num_domains])
y_labels = np.zeros([hp.batch_size, num_domains])
z_labels = np.zeros([hp.batch_size, num_domains])
for b in range(hp.batch_size):  # one-hot encoding
    x_labels[b] = np.identity(num_domains)[x_atr[b]]
    y_labels[b] = np.identity(num_domains)[y_atr[b]]
    z_labels[b] = np.identity(num_domains)[z_atr[b]]

rnd = np.random.randint(2)
alpha = np.random.uniform(0, 0.5, size=hp.batch_size) if rnd == 0 else np.random.uniform(0.5, 1.0,
                                                                                         size=hp.batch_size)

inputs = [x, x2, y, z, x_labels, y_labels, z_labels, alpha]
# training.
train_step(inputs)
```

4. Train step

모든 batch input이 준비되면 train_step함수에 넣는다.

실제로 학습이 진행되는 함수이다. 지정한 iteration 후 weight를 저장한다.

```
if iteration % argv.ckpt_interval == 0:
    weight_dir = os.path.join(argv.output_dir, 'weights')
    os.makedirs(weight_dir, exist_ok=True)
    model.save_weights(os.path.join(weight_dir, 'weights_{:}'.format(iteration)))
```

5. Back-prop

각 모델 별 Loss 취합 후 gradient 전파, TF에 간단한 역전파 모듈로 역전파를 진행한다.

Full Loss

$$\min_D \mathcal{L}^D = -\mathcal{L}_{\text{Real}} + \lambda_1 \mathcal{L}^D_{\text{Match}} + \lambda_2 \mathcal{L}^D_{\text{Interp}}$$

$$\min_G \mathcal{L}^G = \mathcal{L}_{\text{Real}} + \lambda_1 \mathcal{L}^G_{\text{Match}} + \lambda_2 \mathcal{L}^G_{\text{Interp}} + \lambda_3 \mathcal{L}_{\text{Cycle}} + \lambda_4 \mathcal{L}_{\text{Self}} + \lambda_5 \mathcal{L}_{\text{Ortho}},$$

```
# merge the losses.
generator_loss = generator_loss_A2B + hp.lambda_backward * backward_loss + mode_seeking_loss + hp.lambda_cycle
                * cycle_loss + hp.lambda_identity * identity_loss + hp.lambda_triangle * triangle_loss + \
                hp.lambda_conditional * generator_loss_cond_sf + hp.lambda_interp * generator_loss_interp_alp
discriminator_loss = discriminator_loss_B + hp.lambda_interp * discriminator_loss_interp + \
                     hp.lambda_conditional * discriminator_loss_cond

# Optimizers
generator_vars = model.generator.trainable_variables
discriminator_vars = model.discriminator.trainable_variables + model.adversarial.trainable_variables + \
                     model.interpolate.trainable_variables + model.matching.trainable_variables
# compute grad
grad_gen = gen_tape.gradient(target=generator_loss, sources=generator_vars)
grad_dis = dis_tape.gradient(target=discriminator_loss, sources=discriminator_vars)
generator_optimizer.apply_gradients(zip(grad_gen, generator_vars))
discriminator_optimizer.apply_gradients(zip(grad_dis, discriminator_vars))
```

역전파 시켜줌

역전파 시켜줌

-Model.py-

Class RelGAN(tf.keras.Model):

Call : keras model이 호출되면 자동 실행

#Generator : train batch set을 받아 generator가 작동하고 output에 추가

#Discriminator : 논문에서 제안된 3가지 loss를 위한 값들

- Adversarial loss

$$\min_G \max_{D_{\text{Real}}} \mathcal{L}_{\text{Real}} = \mathbb{E}_{\mathbf{x}} \left[\log D_{\text{Real}}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{x},\mathbf{v}} \left[\log(1 - D_{\text{Real}}(G(\mathbf{x}, \mathbf{v})))\right],$$

- Conditional loss

Algorithm 1 Conditional adversarial loss

function MATCH_LOSS($\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$)
  $\mathbf{v}_{12}, \mathbf{v}_{32}, \mathbf{v}_{13} \leftarrow \mathbf{a}_2 - \mathbf{a}_1, \mathbf{a}_2 - \mathbf{a}_3, \mathbf{a}_3 - \mathbf{a}_1$
  $s_r \leftarrow D_{\text{Match}}(\mathbf{x}_1, \mathbf{v}_{12}, \mathbf{x}_2)$ {real triplet}
  $s_f \leftarrow D_{\text{Match}}(\mathbf{x}_1, \mathbf{v}_{12}, G(\mathbf{x}_1, \mathbf{v}_{12}))$ {fake triplet}
  $s_{w_1} \leftarrow D_{\text{Match}}(\mathbf{x}_3, \mathbf{v}_{12}, \mathbf{x}_2)$ {wrong triplet}
  $s_{w_2} \leftarrow D_{\text{Match}}(\mathbf{x}_1, \mathbf{v}_{32}, \mathbf{x}_2)$ {wrong triplet}
  $s_{w_3} \leftarrow D_{\text{Match}}(\mathbf{x}_1, \mathbf{v}_{13}, \mathbf{x}_2)$ {wrong triplet}
  $s_{w_4} \leftarrow D_{\text{Match}}(\mathbf{x}_1, \mathbf{v}_{12}, \mathbf{x}_3)$ {wrong triplet}
  $\mathcal{L}^D_{\text{Match}} \leftarrow (s_r - 1)^2 + s_f^2 + \sum_{i=1}^{4} s_{w_i}^2$
  $\mathcal{L}^G_{\text{Match}} \leftarrow (s_f - 1)^2$
  return $\mathcal{L}^D_{\text{Match}}, \mathcal{L}^G_{\text{Match}}$

- Interpolation loss

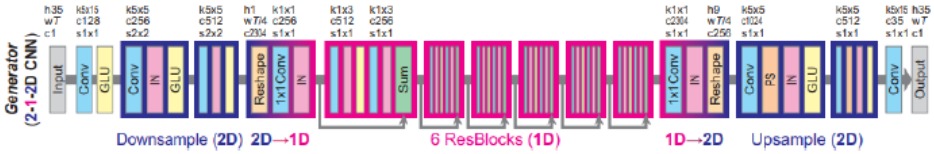
$$\min_{D_{\text{Interp}}} \mathcal{L}^{D}_{\text{Interp}} = \mathbb{E}_{\mathbf{x},\mathbf{v},\alpha}[\, \|D_{\text{Interp}}(G(\mathbf{x}, \alpha\mathbf{v})) - \hat{\alpha}\|^2 + \|D_{\text{Interp}}(G(\mathbf{x}, \mathbf{0}))\|^2 + \|D_{\text{Interp}}(G(\mathbf{x}, \mathbf{v}))\|^2],$$

```python
def call(self, inputs, training=None, mask=None):
    input_A_real = inputs[0]
    input_A2_real = inputs[1]
    input_B_real = inputs[2]
    input_C_real = inputs[3]
    input_A_label = inputs[4]
    input_B_label = inputs[5]
    input_C_label = inputs[6]
    alpha = inputs[7]
    alpha_1 = tf.reshape(alpha, [-1, 1])

    vector_A2B = input_B_label - input_A_label
    vector_C2B = input_B_label - input_C_label
    vector_A2C = input_C_label - input_A_label

    outputs = []

    # Generator.
    generation_B = self.generator([input_A_real, vector_A2B])  # G(A, v)
    generation_B2 = self.generator([input_A2_real, vector_A2B])  # G(A2, v)
    generation_C = self.generator([input_B_real, -vector_C2B])  # G(B, v_bc)
    generation_A = self.generator([input_C_real, -vector_A2C])  # G(C, v_ca)
    generation_cycle_A = self.generator([generation_B, -vector_A2B])  # G(G(A, v), -v)
    generation_A_identity = self.generator([input_A_real, vector_A2B - vector_A2B])  # G(A, v-v) = G(A, 0)
    generation_alpha = self.generator([input_A_real, vector_A2B * alpha_1])  # G(A,αv)
    generation_cycle_A_back = self.generator([generation_alpha, -vector_A2B * alpha_1])  # G(G(A,αv), -αv)

    outputs += [generation_B, generation_B2, generation_C, generation_A, generation_cycle_A, generation_A_identity,
                generation_cycle_A_back]
```

```python
    # Adversarial
    discrimination_B_real = self.discriminator(input_B_real)
    discrimination_B_real = self.adversarial(discrimination_B_real)  # D(B)
    discrimination_B_fake = self.discriminator(generation_B)
    discrimination_B_fake = self.adversarial(discrimination_B_fake)  # D(G(A, v))

    outputs += [discrimination_B_fake, discrimination_B_real]

    # Conditional adversarial
    sr = [self.discriminator(input_A_real), self.discriminator(input_B_real), vector_A2B]
    sr = self.matching(sr)  # D_match(A, B, v)
    sf = [self.discriminator(input_A_real), self.discriminator(generation_B), vector_A2B]
    sf = self.matching(sf)  # D_match(A, G(A, v), v)
    w1 = [self.discriminator(input_C_real), self.discriminator(input_B_real), vector_A2B]
    w1 = self.matching(w1)  # D_match(C, B, v)
    w2 = [self.discriminator(input_A_real), self.discriminator(input_B_real), vector_C2B]
    w2 = self.matching(w2)  # D_match(A, B, v_cb)
    w3 = [self.discriminator(input_A_real), self.discriminator(input_B_real), vector_A2C]
    w3 = self.matching(w3)  # D_match(A, B, v_ac)
    w4 = [self.discriminator(input_A_real), self.discriminator(input_C_real), vector_A2B]
    w4 = self.matching(w4)  # D_match(A, C, v)

    outputs += [sr, sf, w1, w2, w3, w4]

    # Interpolation.
    interpolate_alpha = self.discriminator(generation_alpha)
    interpolate_alpha = self.interpolate(interpolate_alpha)  # D_interp(G(A, αv)) = interpolated
    interpolate_identity = self.discriminator(generation_A_identity)
    interpolate_identity = self.interpolate(interpolate_identity)  # D_interp(G(A, 0)) = non-interpolated
    interpolate_B = self.discriminator(generation_B)
    interpolate_B = self.interpolate(interpolate_B)  # D_interp(G(A, v)) = non-interpolated

    outputs += [interpolate_identity, interpolate_B, interpolate_alpha]
    return outputs
```

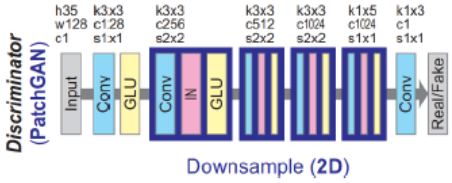
Generator

1. Gated Linear Units(GLU)
2. Downsample (2D)
3. 6 ResBlock (1D)
4. Upsample (2D)



Discriminator

1. Gated Linear Units(GLU)
2. Downsample (2D)



#Generator

```python
# 2-1-2D CNN from CycleGAN-VC2 model
class Generator(tf.keras.Model):
    def __init__(self, num_domains, batch_size):
        super(Generator, self).__init__()
        self.num_domains = num_domains
        self.batch_size = batch_size

        # Gated Linear Units
        # h(X) = (X*W+b)⊗σ(X*V + c)
        self.h1 = tf.keras.layers.Conv2D(128, kernel_size=(5, 15), padding='same', name='h1_conv', dtype='float32')
        self.h1_gates = tf.keras.layers.Conv2D(128, kernel_size=(5, 15), padding='same', name='h1_conv_gates', dtype='float32')
        self.h1_glu = tf.keras.layers.Multiply(name='h1_glu', dtype='float32')

        # Downsamp (2D)
        self.d1 = Downsample2DBlock(filters=256, kernel_size=(5, 5), strides=2, name_prefix='downsample2d_block1_')
        self.d2 = Downsample2DBlock(filters=256, kernel_size=(5, 5), strides=2, name_prefix='downsample2d_block2_')

        # 2D -> 1D
        self.resh1 = tf.keras.layers.Conv1D(256, kernel_size=1, strides=1, padding='same', name='resh1_conv', dtype='float32')
        self.resh1_norm = tfa.layers.InstanceNormalization(epsilon=1e-6, name='resh1_norm')

        # 6 ResBlocks(1D)
        self.res1 = Residual1DBlock(512, kernel_size=3, name_prefix='resid_block1_')
        self.res2 = Residual1DBlock(512, kernel_size=3, name_prefix='resid_block2_')
        self.res3 = Residual1DBlock(512, kernel_size=3, name_prefix='resid_block3_')
        self.res4 = Residual1DBlock(512, kernel_size=3, name_prefix='resid_block4_')
        self.res5 = Residual1DBlock(512, kernel_size=3, name_prefix='resid_block5_')
        self.res6 = Residual1DBlock(512, kernel_size=3, name_prefix='resid_block6_')

        # 1D -> 2D
        self.resh2 = tf.keras.layers.Conv1D(2304, kernel_size=1, padding='same', name='resh2_conv')
        self.resh2_norm = tfa.layers.InstanceNormalization(epsilon=1e-6, name='resh2_norm')

        # Upsample (2D)
        self.u1 = Upsample2DBlock(filters=1024, kernel_size=5, name_prefix='upsampling2d_block1_')
        self.u2 = Upsample2DBlock(filters=512, kernel_size=5, name_prefix='upsampling2d_block2_')

        self.conv_out = tf.keras.layers.Conv2D(1, kernel_size=(5, 15), padding='same', name='conv_out')
```

#Discriminator

```python
# Patch GAN from CycleGAN-VC2
class PatchGanDiscriminator(tf.keras.Model):
    def __init__(self):
        super(PatchGanDiscriminator, self).__init__()
        # Gated Linear Units & Downsample (2D)
        self.h1 = tf.keras.layers.Conv2D(128, kernel_size=(3, 3), padding='same', kernel_regularizer=spectral_norm,
                                         name='h1_conv', dtype='float32')
        self.h1_gates = tf.keras.layers.Conv2D(128, kernel_size=(3, 3), padding='same',
                                               kernel_regularizer=spectral_norm, name='h1_conv_gates', dtype='float32')
        self.h1_glu = tf.keras.layers.Multiply(name='h1_glu', dtype='float32')

        self.d1 = Downsample2DBlock(256, kernel_size=(3, 3), strides=2, sn=True, name_prefix='downsample2d_block1_')
        self.d2 = Downsample2DBlock(512, kernel_size=(3, 3), strides=2, sn=True, name_prefix='downsample2d_block2_')
        self.d3 = Downsample2DBlock(1024, kernel_size=(3, 3), strides=2, sn=True, name_prefix='downsample2d_block3_')
        self.d4 = Downsample2DBlock(1024, kernel_size=(1, 5), strides=1, sn=True, name_prefix='downsample2d_block4_')

        self.out = tf.keras.layers.Conv2D(1, kernel_size=(1, 3), strides=1, padding='same',
                                          kernel_regularizer=spectral_norm, name='out_conv', dtype='float32')
```

실험 결과 및 분석

```
Iteration: 100  Generator loss: 17.064491271972656      Discriminator loss: 4.667359352111816
Iteration: 200  Generator loss: 15.275678634643555      Discriminator loss: 4.561468124389648
Iteration: 300  Generator loss: 14.703805923461914      Discriminator loss: 4.507226943969727
Iteration: 400  Generator loss: 14.197807312011719      Discriminator loss: 4.472631454467773
Iteration: 500  Generator loss: 13.663023948669434      Discriminator loss: 4.431839466094971
Iteration: 600  Generator loss: 13.227020263671875      Discriminator loss: 4.442312240600586
Iteration: 700  Generator loss: 13.091564178466797      Discriminator loss: 4.427208423614502
Iteration: 800  Generator loss: 12.962154388427734      Discriminator loss: 4.411061763763428
Iteration: 900  Generator loss: 12.819465637207031      Discriminator loss: 4.423252105712891
Iteration: 1000         Generator loss: 12.724388122558594      Discriminator loss: 4.416530609130859
```

```
Iteration: 1100         Generator loss: 12.644235610961914      Discriminator loss: 4.416619777679443
Iteration: 1200         Generator loss: 12.605001449584961      Discriminator loss: 4.394216537475586
Iteration: 1300         Generator loss: 12.563719749450684      Discriminator loss: 4.384000778198242
Iteration: 1400         Generator loss: 12.465240478515625      Discriminator loss: 4.370443344116211
Iteration: 1500         Generator loss: 12.376626014709473      Discriminator loss: 4.3832688331604
Iteration: 1600         Generator loss: 12.38540267944336       Discriminator loss: 4.372719764709473
Iteration: 1700         Generator loss: 12.350442886352539      Discriminator loss: 4.359674453735352
Iteration: 1800         Generator loss: 12.3056058883667        Discriminator loss: 4.356139659881592
Iteration: 1900         Generator loss: 12.195042610168457      Discriminator loss: 4.347680568695068
Iteration: 2000         Generator loss: 12.191181182861328      Discriminator loss: 4.331737518310547
Iteration: 2100         Generator loss: 12.12770652709961       Discriminator loss: 4.320141315460205
Iteration: 2200         Generator loss: 12.125083923339844      Discriminator loss: 4.30532693862915
Iteration: 2300         Generator loss: 12.02168083190918       Discriminator loss: 4.278015613555908
Iteration: 2400         Generator loss: 11.968084335327148      Discriminator loss: 4.26969051361084
Iteration: 2500         Generator loss: 11.720359802246094      Discriminator loss: 4.214087963104248
Iteration: 2600         Generator loss: 11.574833869934082      Discriminator loss: 4.205646514892578
Iteration: 2700         Generator loss: 11.480443000793457      Discriminator loss: 4.183983325958252
Iteration: 2800         Generator loss: 11.46356201171875       Discriminator loss: 4.1382670402526855
Iteration: 2900         Generator loss: 11.462458610534668      Discriminator loss: 4.129915237426758
Iteration: 3000         Generator loss: 11.377847671508789      Discriminator loss: 4.093634128570557
Iteration: 3100         Generator loss: 11.310443878173828      Discriminator loss: 4.079222679138184
Iteration: 3200         Generator loss: 11.149134635925293      Discriminator loss: 4.060810565948486
Iteration: 3300         Generator loss: 11.300651550292969      Discriminator loss: 4.067541599273682
Iteration: 3400         Generator loss: 11.193005561828613      Discriminator loss: 4.029587745666504
Iteration: 3500         Generator loss: 11.112325668334961      Discriminator loss: 4.0289411544799805
Iteration: 3600         Generator loss: 11.144966125488281      Discriminator loss: 4.005340576171875

Iteration: 3700         Generator loss: 11.10163688659668       Discriminator loss: 3.9938018321990967
Iteration: 3800         Generator loss: 10.917231559753418      Discriminator loss: 4.0004119873046875
Iteration: 3900         Generator loss: 10.89622688293457       Discriminator loss: 3.9904894828796387
Iteration: 4000         Generator loss: 10.936223030090332      Discriminator loss: 3.9805638790130615
Iteration: 4100         Generator loss: 10.893993377685547      Discriminator loss: 3.9874396324157715
Iteration: 4200         Generator loss: 10.74929428100586       Discriminator loss: 3.980334758758545
Iteration: 4300         Generator loss: 10.69869613647461       Discriminator loss: 3.951045513153076
Iteration: 4400         Generator loss: 10.786127090454102      Discriminator loss: 3.9157607555389404
Iteration: 4500         Generator loss: 10.69076919555664       Discriminator loss: 3.908278465270996
Iteration: 4600         Generator loss: 10.541130065917969      Discriminator loss: 3.8963589668273926
Iteration: 4700         Generator loss: 10.547018051147461      Discriminator loss: 3.876527786254883
Iteration: 4800         Generator loss: 10.502135276794434      Discriminator loss: 3.848594903945923
Iteration: 4900         Generator loss: 10.524696350097656      Discriminator loss: 3.8331520557403564
Iteration: 5000         Generator loss: 10.62326431274414       Discriminator loss: 3.805957078933716
Iteration: 5100         Generator loss: 10.43476676940918       Discriminator loss: 3.7837066650390625
Iteration: 5200         Generator loss: 10.34143352508545       Discriminator loss: 3.7666373252868652
Iteration: 5300         Generator loss: 10.277427673339844      Discriminator loss: 3.7421305179595947
Iteration: 5400         Generator loss: 10.30938720703125       Discriminator loss: 3.750797986984253
Iteration: 5500         Generator loss: 10.227890014648438      Discriminator loss: 3.7132129669189453
Iteration: 5600         Generator loss: 10.240747451782227      Discriminator loss: 3.709106206893921
Iteration: 5700         Generator loss: 10.184128761291504      Discriminator loss: 3.6771328449249268
Iteration: 5800         Generator loss: 10.121233940124512      Discriminator loss: 3.709017753601074
Iteration: 5900         Generator loss: 9.9215726852417         Discriminator loss: 3.672884225845337


Iteration: 6100         Generator loss: 10.026266098022461      Discriminator loss: 3.624149799346924
Iteration: 6200         Generator loss: 9.925288200378418       Discriminator loss: 3.645158290863037
Iteration: 6300         Generator loss: 9.901174545288086       Discriminator loss: 3.6366164684295654
Iteration: 6400         Generator loss: 9.804664611816406       Discriminator loss: 3.562387466430664
Iteration: 6500         Generator loss: 9.707025527954102       Discriminator loss: 3.515397310256958
Iteration: 6600         Generator loss: 9.730149269104004       Discriminator loss: 3.5563323497772217
Iteration: 6700         Generator loss: 9.690670013427734       Discriminator loss: 3.532818555831909
Iteration: 6800         Generator loss: 9.700032234191895       Discriminator loss: 3.50605845451355
Iteration: 6900         Generator loss: 9.752659797668457       Discriminator loss: 3.5920071601867676
Iteration: 7000         Generator loss: 9.536027908325195       Discriminator loss: 3.474421977996826
```

```
Iteration: 7100        Generator loss: 9.633672714233398       Discriminator loss: 3.493881940841675
Iteration: 7200        Generator loss: 9.518526077270508       Discriminator loss: 3.5141546726226807
Iteration: 7300        Generator loss: 9.544452667236328       Discriminator loss: 3.5276143550872803
Iteration: 7400        Generator loss: 9.445075988769531       Discriminator loss: 3.5255985260009766
Iteration: 7500        Generator loss: 9.408249855041504       Discriminator loss: 3.458280563354492
Iteration: 7600        Generator loss: 9.326560020446777       Discriminator loss: 3.4423460960388184
Iteration: 7700        Generator loss: 9.341133117675781       Discriminator loss: 3.5118956565856934
Iteration: 7800        Generator loss: 9.248261451721191       Discriminator loss: 3.4636166095733643
Iteration: 7900        Generator loss: 9.32721996307373        Discriminator loss: 3.4379138946533203
Iteration: 8000        Generator loss: 9.385726928710938       Discriminator loss: 3.482572317123413
```

```
Iteration: 8100        Generator loss: 9.316174507141113       Discriminator loss: 3.4522833824157715
Iteration: 8200        Generator loss: 9.221351623535156       Discriminator loss: 3.4688859939575195
Iteration: 8300        Generator loss: 9.24046516418457        Discriminator loss: 3.4305918216705322
Iteration: 8400        Generator loss: 9.085041046142578       Discriminator loss: 3.4042530059814453
Iteration: 8500        Generator loss: 9.218391418457031       Discriminator loss: 3.458298921585083
Iteration: 8600        Generator loss: 9.190052032470703       Discriminator loss: 3.4438202381134033
Iteration: 8700        Generator loss: 9.21442699432373        Discriminator loss: 3.4285361766815186
Iteration: 8800        Generator loss: 9.138998031616211       Discriminator loss: 3.407801866531372
Iteration: 8900        Generator loss: 9.131251335144043       Discriminator loss: 3.4101779460906982
Iteration: 9000        Generator loss: 9.13964557647705        Discriminator loss: 3.352229118347168
Iteration: 9100        Generator loss: 9.165791511535645       Discriminator loss: 3.36631441116333
Iteration: 9200        Generator loss: 9.20913314819336        Discriminator loss: 3.3840200901031494
Iteration: 9300        Generator loss: 9.26045036315918        Discriminator loss: 3.2645480632781982
Iteration: 9400        Generator loss: 9.276666641235352       Discriminator loss: 3.2935004234313965
Iteration: 9500        Generator loss: 9.28481674194336        Discriminator loss: 3.2904622554779053
Iteration: 9600        Generator loss: 9.230844497680664       Discriminator loss: 3.2818896770477295
Iteration: 9700        Generator loss: 9.321380615234375       Discriminator loss: 3.2438957691192627
Iteration: 9800        Generator loss: 9.273369789123535       Discriminator loss: 3.2684786319732666
Iteration: 9900        Generator loss: 9.334859848022461       Discriminator loss: 3.271437644958496
```

최종 loss

```
Iteration: 9900        Generator loss: 9.334859848022461       Discriminator loss: 3.271437644958496
```

Epoch이 진행될수록 generator와 disciriminatoy의 loss가 줄어들며 잘 학습되고 있다. (시간관계상 10000epoch에서 학습을 종료하였다. Image를 학습시키는 제안 논문에서는 100,000epoch을 학습시켰고, 약 30,000epoch에서 loss가 수렴하였다.)

- Inference

Input : 음성(.wav), attribute (source = 입력 음성의 감정 index, target index = 원하는 감정의 index, a = 원하는 target감정의 정도)

Output : 생성된 음성(.wav)

(음성은 첨부가 불가능하므로 발표영상을 참고해주시기 바랍니다.)

결론

(음성data를 평가할때에는 MOS와 같은 주관적인 평가지표밖에 없어 개인 한사람이 하는 평가는 무의미하다고 판단하여 평가지표를 첨부하지 않았습니다.)

입력음성의 source감정에 상대적인 감정차이를 갖는 음성이 정도에 맞게 잘 생성되는것을 알 수 있었다. 지정한 domain 중 (happy, angry, fear, sad) 고주파가 많은 fear가 가장 좋은 결과를 내었다.

시간/컴퓨팅파워 문제로 인해 약 10,000epoch만 학습하였다. 10,000epoch학습시에도 확연한 감정차이를 느낄 수 있었다. 다만 학습량이 부족하여 감정을 세부적으로 조절하는것에는 많은 차이가 없었다. 더 많은 dataset과 많은 epoch을 학습한다면 인간의 음성과 거의 동일하며, 감정의 구분도 더 확실하고 정도를 아주 세부적으로 조절할 수 있을것이라고 생각한다.

이번 프로젝트에서 음성도메인에 적용한 RelGAN을 이용하여 음성에 원하는 감정을 입힐 수 있었고 만족스러운 결과를 얻었다.