Deep generative model을 활용한

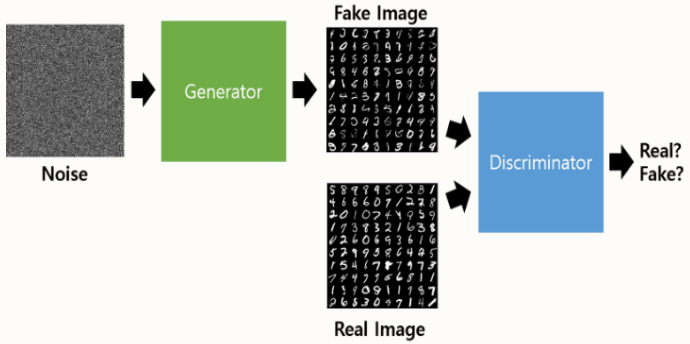
웹툰 스타일 이미지 생성

팀명: UphillGate
팀장: 박재성
지도 교수: 양희경 교수님
팀원: 박지훈, 백서희, 유상민

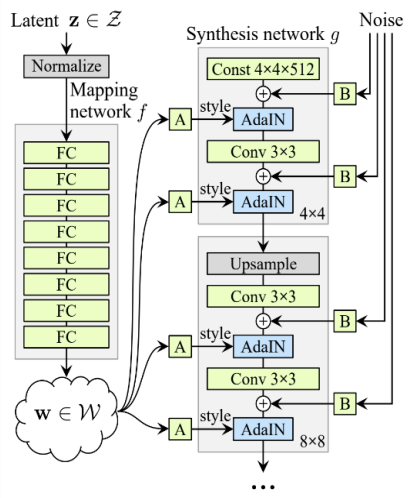
목차

기술 조사

기술 흐름도



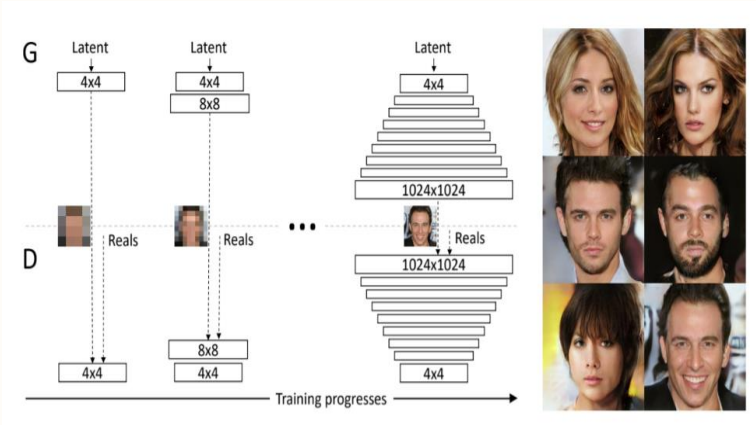
GAN(Generative Adversarial Network)



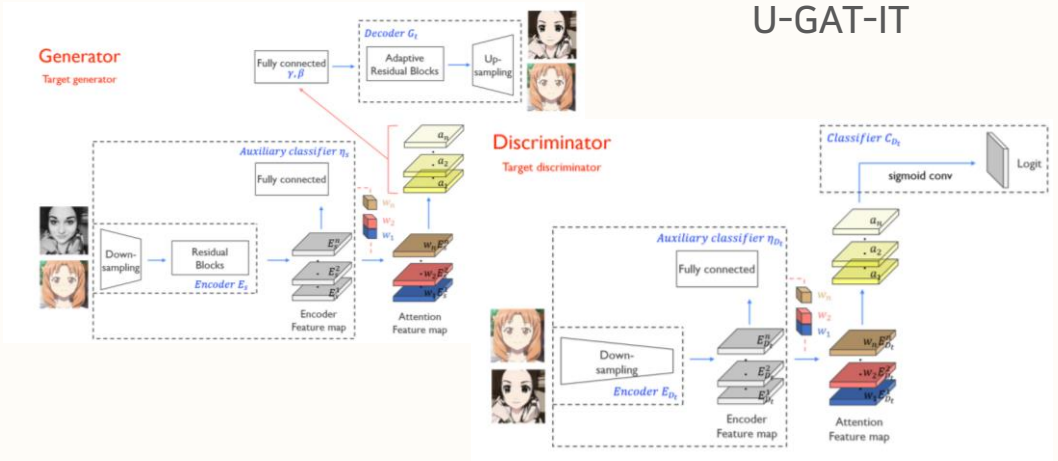
StyleGAN

1 2 3 4

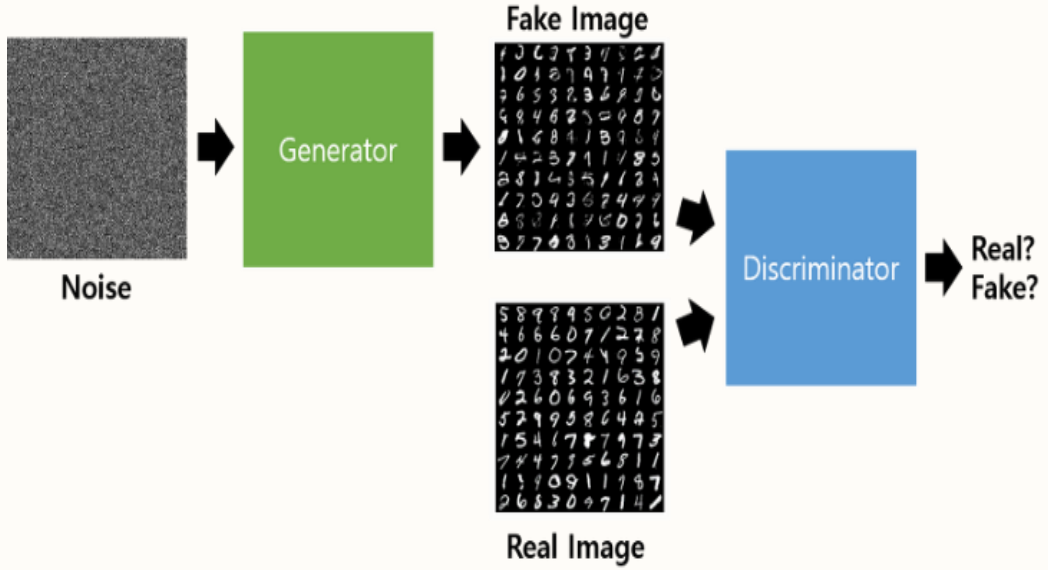
PGGAN



U-GAT-IT



GAN(Generative Adversarial Network)



출처 : https://hyeongminlee.github.io/post/gan001_gan/

적대적 생성 모델로, 데이터의 형태를 만들어내는 모델

1) Generator: Noise -> Fake image
2) Discriminator: Real VS Fake

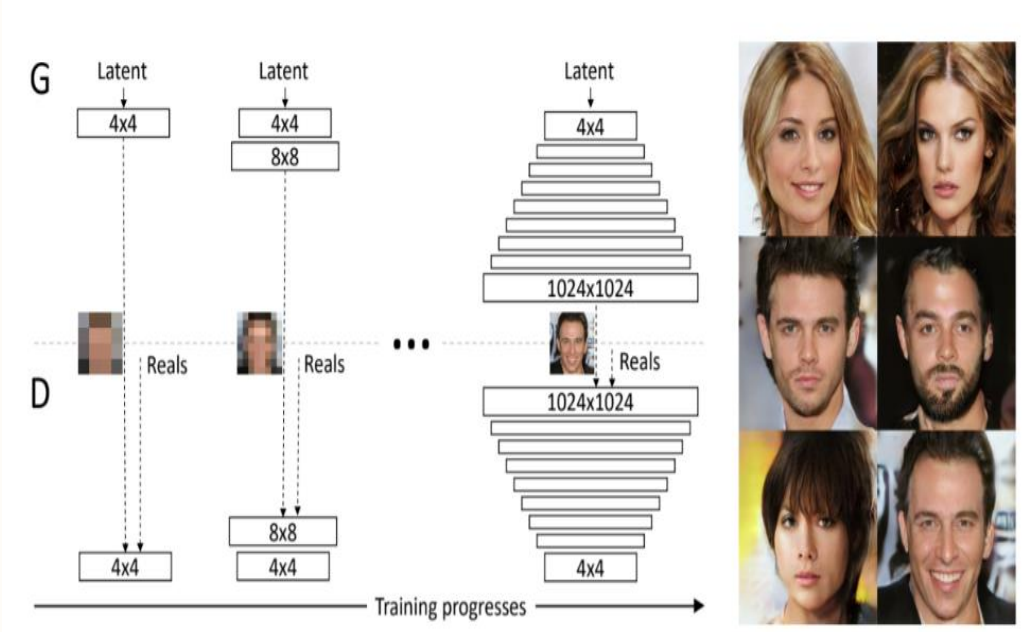
=> 1)과 2)의 과정을 반복하여,
Generator가 더욱 정교한 이미지를 만들도록 함

장점: Generator <-상호작용-> Discriminator
=> 좋은 출력 결과

단점: 학습의 불안정성, 낮은 해상도의 출력만 가능

PGGAN

낮은 해상도의 이미지를 먼저 생성한 후, 점진적으로 높은 해상도의 이미지를 만들도록 훈련 시키는 모델



Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

저해상도-이미지의 전체적인 형태에 대해 학습

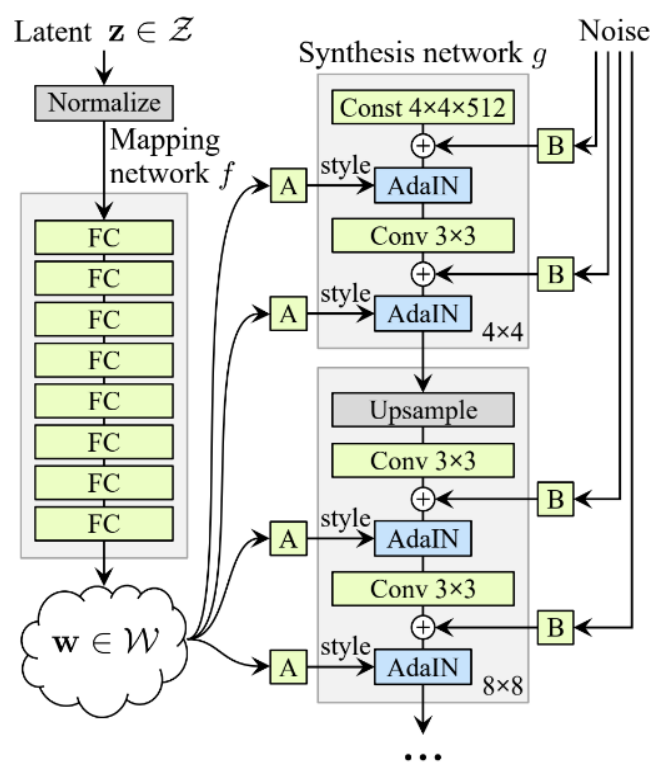
고해상도-더 디테일한 부분에 집중

Generator가 만든 저해상도 이미지가 Discriminator에게 같다고 인정받으면,

레이어를 하나씩 추가해 나가는 것으로 학습방향을 결정

StyleGAN

원하는 style을 수치화 시켜, GAN에 적용하기 위해 style을 scale에 넣어 학습시키는 방법



Karras, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4401-4410).

합성되는 이미지의 특성을 조절하기 위해,
style transfer에 기반한 새로운 Generator 구조를 가짐

여기서의 Style은 화풍이나 그림체 같은 스타일이 아닌,
성별 피부색 연령 헤어스타일 등을 의미함.

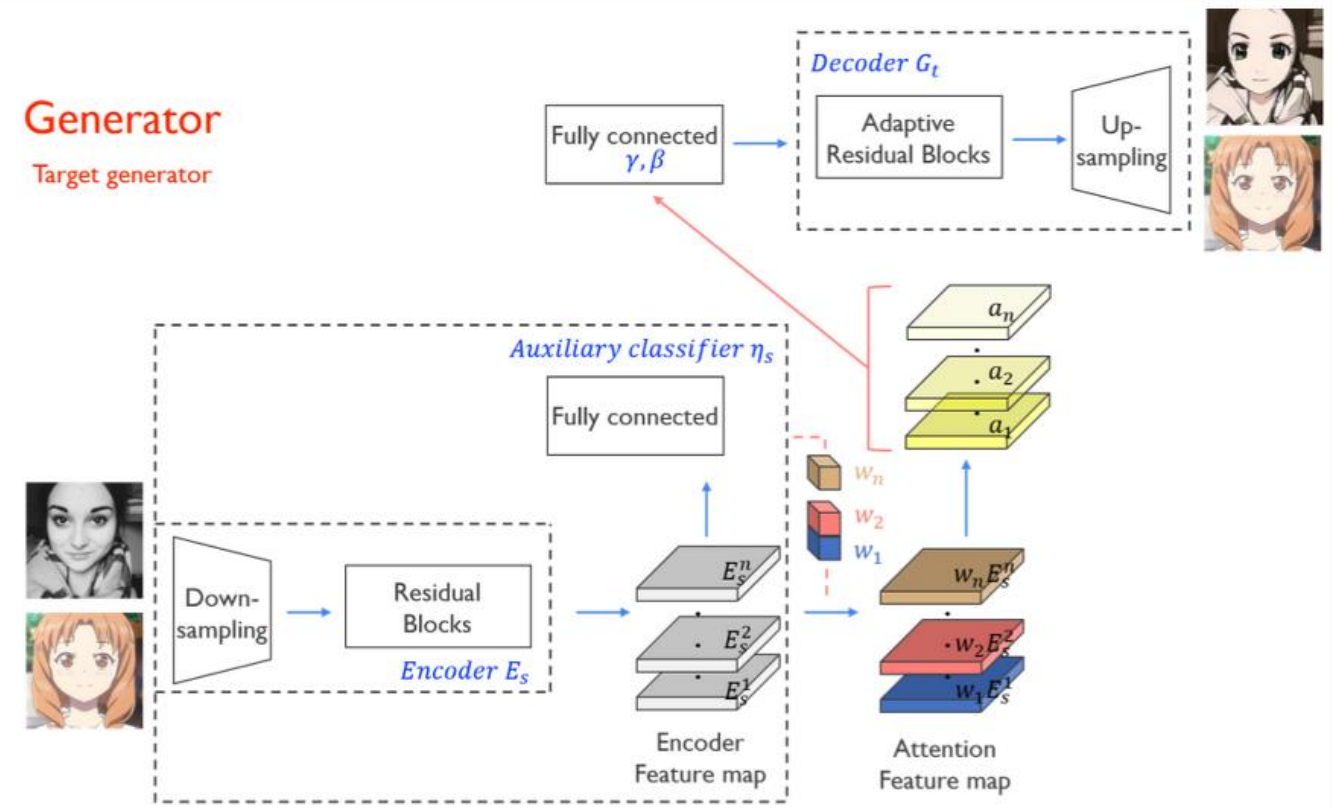
U-GAT-IT 이란?

Unsupervised image-to-image translation 의 새로운 방법

Attention module + AdaLIN(Adaptive Layer-Instance Normalization)
class activation map　　normalization

Generator와 Discriminator로 구성

Generator



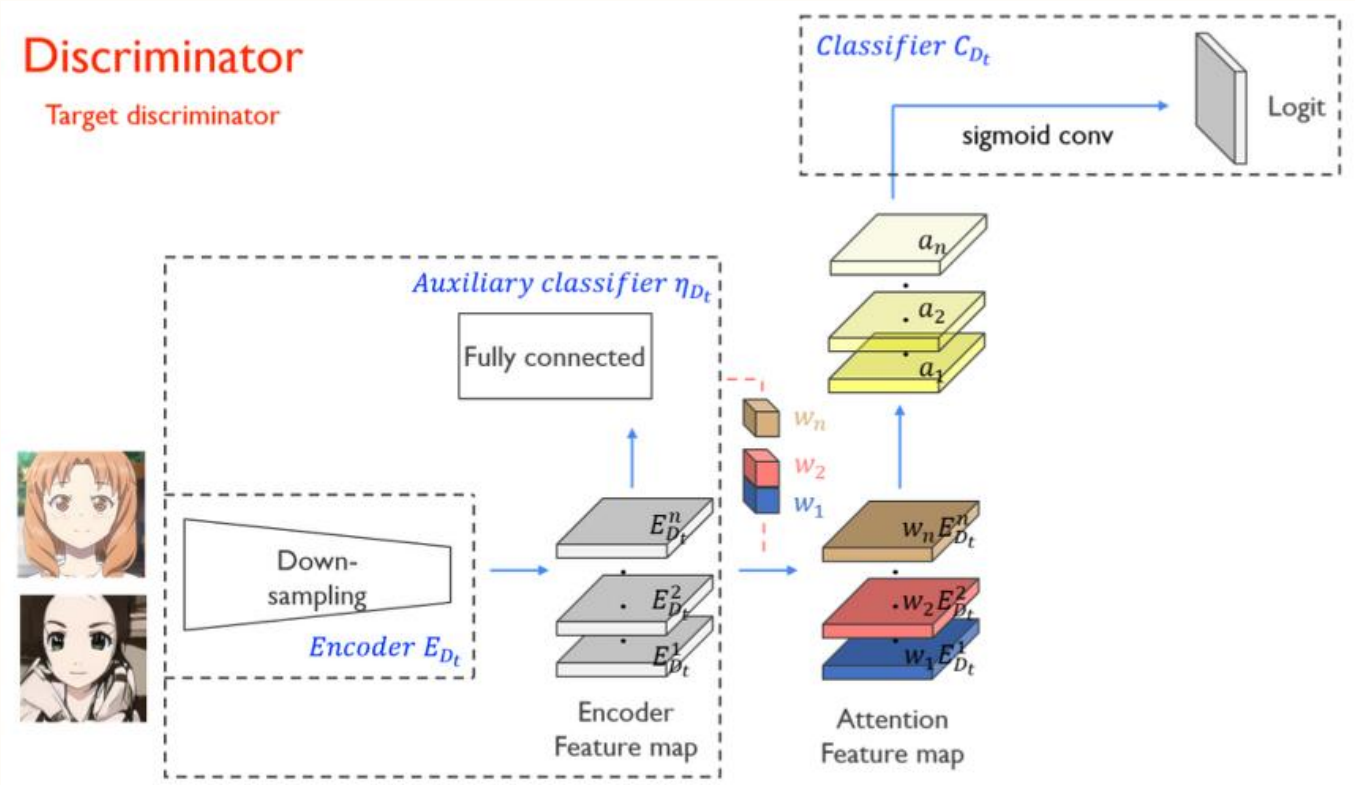
Kim, J., Kim, M., Kang, H., & Lee, K. (2019). U-GAT-IT: unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation. arXiv preprint arXiv:1907.10830.

Encoder와 Decoder로 구성

Encoder – Feature 추출

Decoder- 추출된 Feature로 이미지 생성

Discriminator



Kim, J., Kim, M., Kang, H., & Lee, K. (2019). U-GAT-IT: unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation. arXiv preprint arXiv:1907.10830.

Encoder와 Classifier로 구성

Encoder-Feature map 형성

Classifier-Attention feature map을 통해
어느 부분에 집중해야 하는지 판단

Attention module-class activation map(cam)

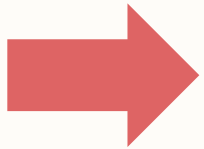
Class activation map이란?

훈련된 Classifier network를 통한 새로운 이미지를 분류 시,

어떤 점을 차이점으로 염두에 두고 결과값을 판단했는지를 나타내는 기술

Attention map이란?

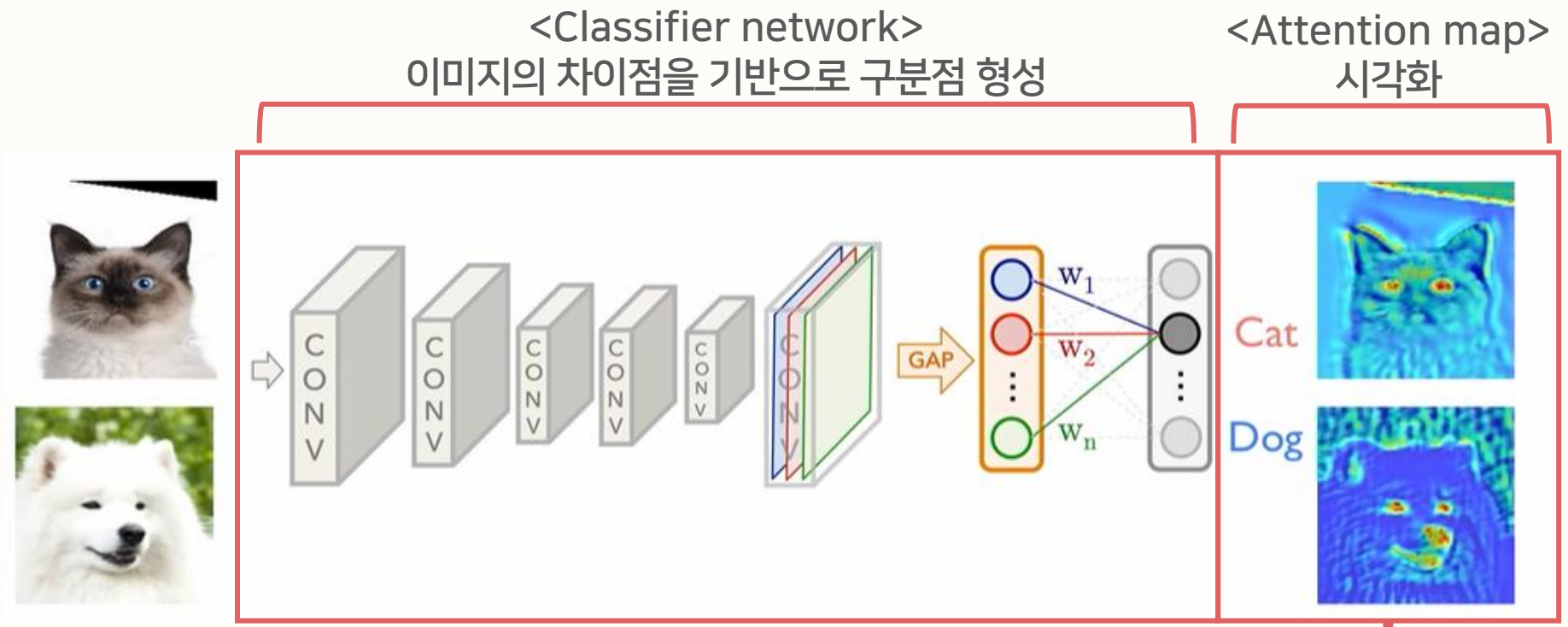
Class activation map의 과정 중 차이점을 기반으로 한 판단을 시각화 한 것



Source Domain과 Target Domain간의 변환 시,

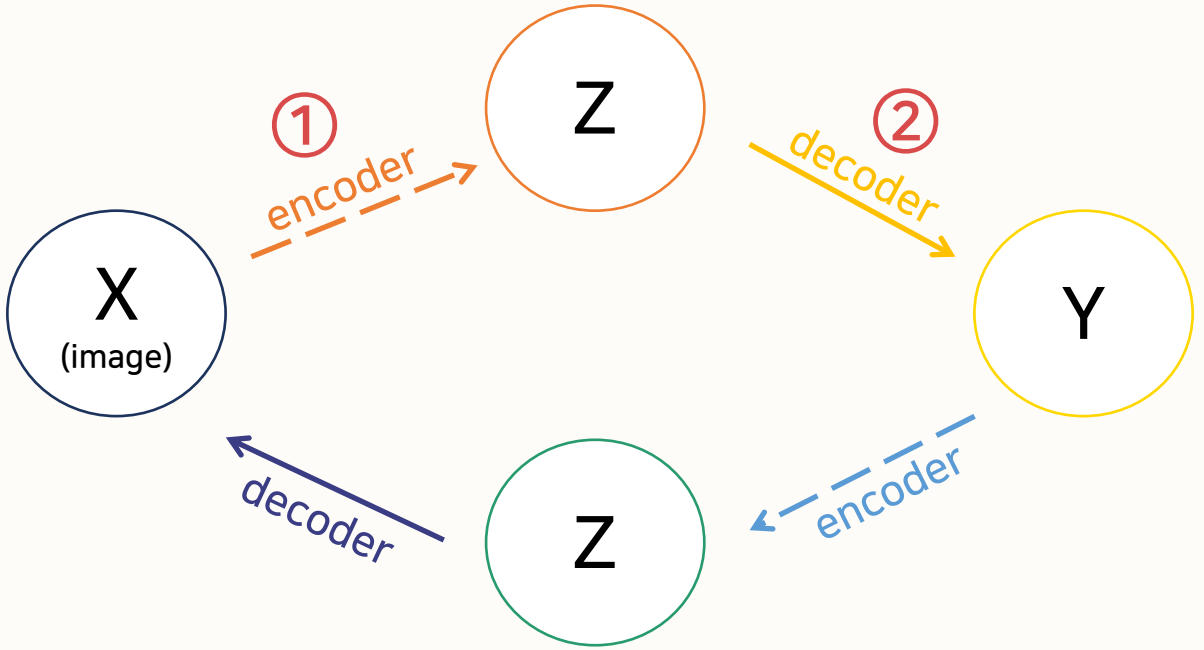
가장 차이가 나는 영역에 집중해 변환을 하도록 Attention module을 결합

Attention module-class activation map(cam)



Attention map을 바탕으로,
Attention Module에서 강조되는 구간이 설정됨

Attention module-class activation map(cam)



① Encoding image X

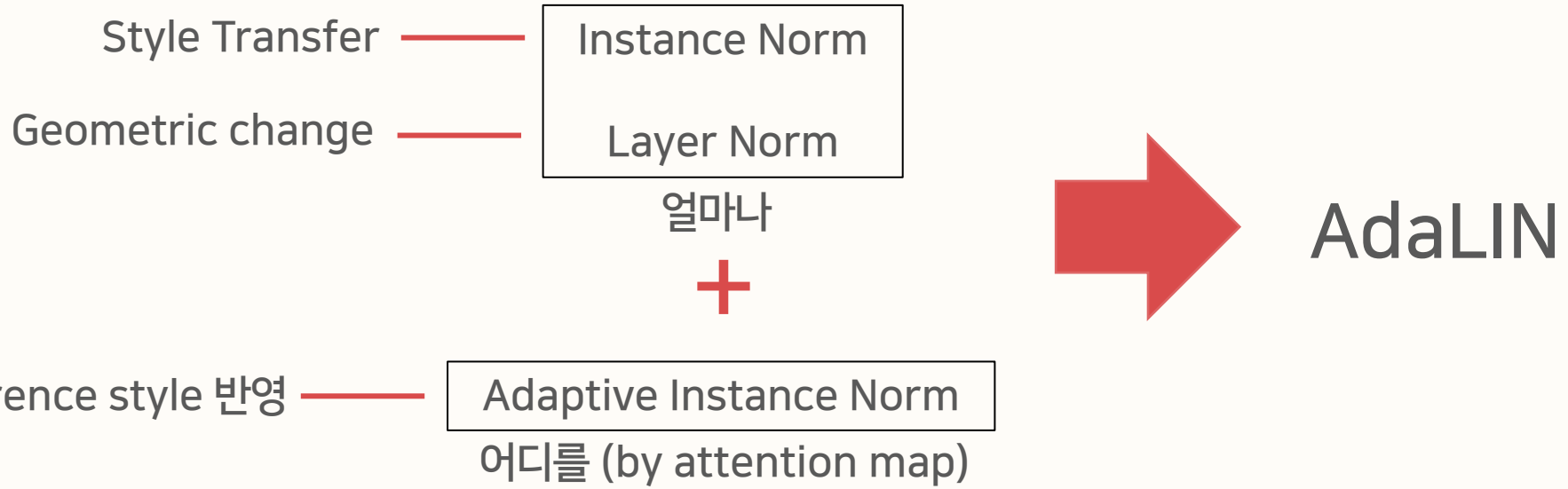
② Z(encoded X) -> Y decoding 시, attention map 삽입
=> network: 어느 부분을 기준으로 이미지 변환을 해야 할지 판단

AdaLIN (Adaptive Layer-Instance Normalization)

변환 할 때, 데이터셋에 따라 얼마만큼 변환할 지 네트워크가 스스로 학습하는 기법

Instance Norm + Layer Norm + Adaptive Instance Norm

Geometric change ↓ Geometric change ↑



관련성

AI 기술과의 연관성

U-GAT-IT 기술은 GAN의 변형이다.

학습의 반복

여러가지 모습으로 이미지를 변환

<학습 과정>

Generator
이미지 생성

Discriminator
진짜 VS 가짜

Generator
더 정교한 가짜 이미지
생성&학습

기업과의 연관성

1. NC Soft: 게임 icon 생성

실제 img -> 게임 스타일 img (image-to-image translation)

=> 시간적인 효율성 극대화 가능(사람 N시간/AI 0.0N시간)

2. Naver- starGAN V2 기술

다양한 도메인과 스타일을 적용하여 이미지 변환

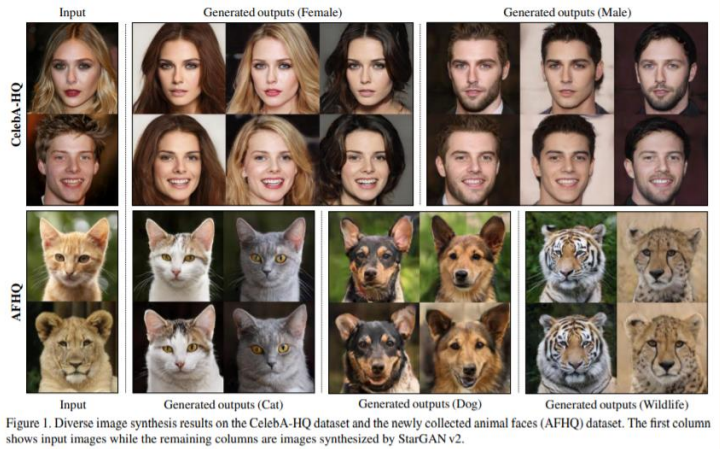


Figure 1. Diverse image synthesis results on the CelebA-HQ dataset and the newly collected animal faces (AFHQ) dataset. The first column shows input images while the remaining columns are images synthesized by StarGAN v2.

Choi, Y., Uh, Y., Yoo, J., & Ha, J. W. (2020). Stargan v2: Diverse image synthesis for multiple domains. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 8188-8197).

카메라 어플 B612에 적용한 모습

기술 개발

Train code-Generator: Encoder

Generator-source img 와 target img를 구분하는 가장 큰 region 탐색, S->T/ T->T 로 바뀐 이미지 출력

```
def generator(self, x_init, reuse=False, scope="generator"):
    channel = self.ch
    with tf.variable_scope(scope, reuse=reuse) :
        x = conv(x_init, channel, kernel=7, stride=1, pad=3, pad_type='reflect', scope='conv')
        x = instance_norm(x, scope='ins_norm')
        x = relu(x)

        # Down-Sampling -> encoder
        for i in range(2) :
            x = conv(x, channel*2, kernel=3, stride=2, pad=1, pad_type='reflect', scope='conv_'+str(i))
            x = instance_norm(x, scope='ins_norm_'+str(i))
            x = relu(x)

            channel = channel * 2

        # Down-Sampling Bottleneck -> residualblock을 통해 feature map 형성
        for i in range(self.n_res):
            x = resblock(x, channel, scope='resblock_' + str(i))
```

Encoder: feature map 생성

Down Sampling의 residual block을 통해 각 이미지의 feature map 생성

Train code-Generator: Auxiliary classifier

Auxiliary classifier-feature map의 중요한 부분만 학습하도록 함 이 과정에서 CAM img를 얻을 수 있음

Input: Encode 된 feature map

Output: 중요도에 따라 가중된 feature map

```
cam_x = global_avg_pooling(x)
cam_gap_logit, cam_x_weight = fully_connected_with_w(cam_x, scope='CAM_logit')
x_gap = tf.multiply(x, cam_x_weight)

cam_x = global_max_pooling(x)
cam_gmp_logit, cam_x_weight = fully_connected_with_w(cam_x, reuse=True, scope='CAM_logit')
x_gmp = tf.multiply(x, cam_x_weight)

cam_logit = tf.concat([cam_gap_logit, cam_gmp_logit], axis=-1)
x = tf.concat([x_gap, x_gmp], axis=-1)

x = conv(x, channel, kernel=1, stride=1, scope='conv_1x1')
x = relu(x)

heatmap = tf.squeeze(tf.reduce_sum(x, axis=-1))
```

global average pooling & global max pooling
=> source domain의 k번째 feature map의
weight를 학습하는 훈련 진행

Weight 값을 통해 domain에 특화된 attention feature map 계산 가능

Train code-Generator: Fully connected layer

Fully connected layer-Decoder에서 사용될 β, γ을 구함

```
def MLP(self, x, use_bias=True, reuse=False, scope='MLP'):
    channel = self.ch * self.n_res

    if self.light :
        x = global_avg_pooling(x)

    with tf.variable_scope(scope, reuse=reuse):
        for i in range(2) :
            x = fully_connected(x, channel, use_bias, scope='linear_' + str(i))
            x = relu(x)


        gamma = fully_connected(x, channel, use_bias, scope='gamma')
        beta = fully_connected(x, channel, use_bias, scope='beta')

        gamma = tf.reshape(gamma, shape=[self.batch_size, 1, 1, channel])
        beta = tf.reshape(beta, shape=[self.batch_size, 1, 1, channel])

        return gamma, beta
```

Input: Encoder에서 나온 weighted feature map

Output: β, γ

Train code-Generator: Decoder

Decoder-모든 입력 이미지(Weighted feature map)를 Target domain 이미지로 바꿈

```
# Gamma, Beta block
gamma, beta = self.MLP(x, reuse=reuse) #fullyconnected에서 구한 감마, 베타 값

# Up-Sampling Bottleneck #decoder 과정
for i in range(self.n_res):
    x = adaptive_ins_layer_resblock(x, channel, gamma, beta, smoothing=self.smoothing, scope='adaptive_resblock' + str(i)) #AdaLIN적용

# Up-Sampling #decoder 과정
for i in range(2) :
    x = up_sample(x, scale_factor=2)
    x = conv(x, channel//2, kernel=3, stride=1, pad=1, pad_type='reflect', scope='up_conv_'+str(i))
    x = layer_instance_norm(x, scope='layer_ins_norm_'+str(i))
    x = relu(x)

    channel = channel // 2


x = conv(x, channels=3, kernel=7, stride=1, pad=3, pad_type='reflect', scope='G_logit')
x = tanh(x)

return x, cam_logit, heatmap #모든 입력 이미지를 T 도메인 이미지로 바꿈(return)
```

Input: Weighted feature map
(Auxiliary network의 output),
β, γ(Fully connected layer)

Output: Target domain image

→ 이 과정에서 AdaLIN 적용됨

Train code-Discriminator

Discriminator- 생성 이미지와 target domain 이미지에서 가장 다른 부분을 찾아 이 부분을 Generator에 알려줌

```
def discriminator(self, x_init, reuse=False, scope="discriminator"):
    D_logit = []
    D_CAM_logit = []
    with tf.variable_scope(scope, reuse=reuse) :
        local_x, local_cam, local_heatmap = self.discriminator_local(x_init, reuse=reuse, scope='local')
        global_x, global_cam, global_heatmap = self.discriminator_global(x_init, reuse=reuse, scope='global')

        D_logit.extend([local_x, global_x]) #logit 생성
        D_CAM_logit.extend([local_cam, global_cam])

        return D_logit, D_CAM_logit, local_heatmap, global_heatmap
```

Input: 생성 이미지 OR
실제 Target domain 이미지

Output: True/False Logit

Train code-Discriminator

```python
def discriminator_global(self, x_init, reuse=False, scope='discriminator_global'):
    with tf.variable_scope(scope, reuse=reuse):
        channel = self.ch
        x = conv(x_init, channel, kernel=4, stride=2, pad=1, pad_type='reflect', sn=self.sn, scope='conv_0')
        x = lrelu(x, 0.2)

        for i in range(1, self.n_dis - 1):
            x = conv(x, channel * 2, kernel=4, stride=2, pad=1, pad_type='reflect', sn=self.sn, scope='conv_' + str(i))
            x = lrelu(x, 0.2)

            channel = channel * 2

        x = conv(x, channel * 2, kernel=4, stride=1, pad=1, pad_type='reflect', sn=self.sn, scope='conv_last')
        x = lrelu(x, 0.2)

        channel = channel * 2
        #Auxiliary network global average pooling 과 glomal max pooling를 이용해 학습됨
        cam_x = global_avg_pooling(x)
        cam_gap_logit, cam_x_weight = fully_connected_with_w(cam_x, sn=self.sn, scope='CAM_logit')
        x_gap = tf.multiply(x, cam_x_weight)

        cam_x = global_max_pooling(x)
        #auxiliary classifier -> fully connected
        cam_gmp_logit, cam_x_weight = fully_connected_with_w(cam_x, sn=self.sn, reuse=True, scope='CAM_logit')
        x_gmp = tf.multiply(x, cam_x_weight)

        cam_logit = tf.concat([cam_gap_logit, cam_gmp_logit], axis=-1)
        x = tf.concat([x_gap, x_gmp], axis=-1)

        x = conv(x, channel, kernel=1, stride=1, scope='conv_1x1')
        x = lrelu(x, 0.2)

        heatmap = tf.squeeze(tf.reduce_sum(x, axis=-1))

        x = conv(x, channels=1, kernel=4, stride=1, pad=1, pad_type='reflect', sn=self.sn, scope='D_logit')

        return x, cam_logit, heatmap
```

```python
def discriminator_local(self, x_init, reuse=False, scope='discriminator_local'):
    with tf.variable_scope(scope, reuse=reuse) :
        channel = self.ch
        x = conv(x_init, channel, kernel=4, stride=2, pad=1, pad_type='reflect', sn=self.sn, scope='conv_0')
        x = lrelu(x, 0.2)

        for i in range(1, self.n_dis - 2 - 1):
            x = conv(x, channel * 2, kernel=4, stride=2, pad=1, pad_type='reflect', sn=self.sn, scope='conv_' + str(i))
            x = lrelu(x, 0.2)

            channel = channel * 2

        x = conv(x, channel * 2, kernel=4, stride=1, pad=1, pad_type='reflect', sn=self.sn, scope='conv_last')
        x = lrelu(x, 0.2)

        channel = channel * 2

        cam_x = global_avg_pooling(x)
        cam_gap_logit, cam_x_weight = fully_connected_with_w(cam_x, sn=self.sn, scope='CAM_logit')
        x_gap = tf.multiply(x, cam_x_weight)

        cam_x = global_max_pooling(x)
        cam_gmp_logit, cam_x_weight = fully_connected_with_w(cam_x, sn=self.sn, reuse=True, scope='CAM_logit')
        x_gmp = tf.multiply(x, cam_x_weight)

        cam_logit = tf.concat([cam_gap_logit, cam_gmp_logit], axis=-1)
        x = tf.concat([x_gap, x_gmp], axis=-1)

        x = conv(x, channel, kernel=1, stride=1, scope='conv_1x1')
        x = lrelu(x, 0.2)

        heatmap = tf.squeeze(tf.reduce_sum(x, axis=-1))

        x = conv(x, channels=1, kernel=4, stride=1, pad=1, pad_type='reflect', sn=self.sn, scope='D_logit')

        return x, cam_logit, heatmap
```

다른 크기(local, global)를 가진 2개의 Discriminator 존재

Train Model

```
def generate_a2b(self, x_A, reuse=False):
    out, cam, _ = self.generator(x_A, reuse=reuse, scope="generator_B")

    return out, cam

def generate_b2a(self, x_B, reuse=False):
    out, cam, _ = self.generator(x_B, reuse=reuse, scope="generator_A")

    return out, cam

def discriminate_real(self, x_A, x_B):
    real_A_logit, real_A_cam_logit, _, _ = self.discriminator(x_A, scope="discriminator_A")
    real_B_logit, real_B_cam_logit, _, _ = self.discriminator(x_B, scope="discriminator_B")

    return real_A_logit, real_A_cam_logit, real_B_logit, real_B_cam_logit

def discriminate_fake(self, x_ba, x_ab):
    fake_A_logit, fake_A_cam_logit, _, _ = self.discriminator(x_ba, reuse=True, scope="discriminator_A")
    fake_B_logit, fake_B_cam_logit, _, _ = self.discriminator(x_ab, reuse=True, scope="discriminator_B")

    return fake_A_logit, fake_A_cam_logit, fake_B_logit, fake_B_cam_logit
```

Gs→t, Gt→s, Ds, Dt 총 4개의 모델을 학습 시킴

Train code-loss function

Loss function- Adversarial, Cycle, identity, CAM loss 사용 (총 4개)

```
            G_ad_loss_A = (generator_loss(self.gan_type, fake_A_logit) + generator_loss(self.gan_type, fake_A_cam_logit))
            G_ad_loss_B = (generator_loss(self.gan_type, fake_B_logit) + generator_loss(self.gan_type, fake_B_cam_logit))

            D_ad_loss_A = (discriminator_loss(self.gan_type, real_A_logit, fake_A_logit) + discriminator_loss(self.gan_type, real_A_cam_logit,
fake_A_cam_logit) + GP_A + GP_CAM_A)
            D_ad_loss_B = (discriminator_loss(self.gan_type, real_B_logit, fake_B_logit) + discriminator_loss(self.gan_type, real_B_cam_logit,
fake_B_cam_logit) + GP_B + GP_CAM_B)

            reconstruction_A = L1_loss(x_aba, self.domain_A) # reconstruction
            reconstruction_B = L1_loss(x_bab, self.domain_B) # reconstruction

            identity_A = L1_loss(x_aa, self.domain_A)
            identity_B = L1_loss(x_bb, self.domain_B)

            cam_A = cam_loss(source=cam_ba, non_source=cam_aa)
            cam_B = cam_loss(source=cam_ab, non_source=cam_bb)

            Generator_A_gan = self.adv_weight * G_ad_loss_A
            Generator_A_cycle = self.cycle_weight * reconstruction_B
            Generator_A_identity = self.identity_weight * identity_A
            Generator_A_cam = self.cam_weight * cam_A


            Generator_B_gan = self.adv_weight * G_ad_loss_B
            Generator_B_cycle = self.cycle_weight * reconstruction_A
            Generator_B_identity = self.identity_weight * identity_B
            Generator_B_cam = self.cam_weight * cam_B


            Generator_A_loss = Generator_A_gan + Generator_A_cycle + Generator_A_identity + Generator_A_cam
            Generator_B_loss = Generator_B_gan + Generator_B_cycle + Generator_B_identity + Generator_B_cam


            Discriminator_A_loss = self.adv_weight * D_ad_loss_A
            Discriminator_B_loss = self.adv_weight * D_ad_loss_B

            self.Generator_loss = Generator_A_loss + Generator_B_loss + regularization_loss('generator')
            self.Discriminator_loss = Discriminator_A_loss + Discriminator_B_loss + regularization_loss('discriminator')
```

Adversarial loss:
T 도메인 이미지와 S 도메인을 이용해 생성한 이미지 구분함

Cycle loss:
같은 이미지를 계속 만드는 문제(mode collapse 문제)를 완화

identity loss:
Input img 와 output img의 컬러분포를 비슷하게 만들기 위함

CAM loss
In G: 입력 이미지의 도메인을 알아내도록 함
In D: 생성 이미지인지 실제 이미지인지 알아내도록 함

기술 시연

Test code

```
import tensorflow as tf # tensorflow 1.13.1+
import cv2
import dlib
import numpy as np
import matplotlib.pyplot as plt
import os
from glob import glob

from UGATIT_noargs import UGATIT

tf.logging.set_verbosity(tf.logging.ERROR)
```

Test시 필요한 모듈 선언
Tensorflow는 2.0 미만의 버전 사용

직접 학습시키는 것에 어려움이 있어,
Pre-Trained된 모델을 사용

```
checkpoint_path = 'checkpoint/UGATIT_selfie2anime_lsgan_4resblock_6dis_1_1_10_10_1000_sn_smoothing/UGATIT.model-1000000' #dataset 파일 경로

sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True)) #Session 객체는 Operation 객체가 실행되고 Tensor 객체가 계산되는 환경을 캡슐화.
allow_soft_placement=True로 TensorFlow가 자동으로 존재하는 디바이스 중 선택.

gan = UGATIT() #UGATIT 객체 생성
gan.build_model() #모델 객체 생성

saver = tf.train.Saver() #훈련을 위한 클래스 객체 생성
saver.restore(sess, checkpoint_path) #세션에 체크 포인트 경로에 있는 dataset 훈련

print('Model loaded!')
```

UGATIT 과 Model 객체 생성

Test code

```
img_path = 'imgs/soyeon.jpg' #테스트용 이미지 경로

img = cv2.imread(img_path, flags=cv2.IMREAD_COLOR) #이미지 로드
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) #RGB 배열이 이미지 상에서는 BGR순서대로 들어가 있기 때문에 뒤집어 줌

brightness = 0 #밝기 조정
contrast = 30 #색채 대비 조정
img = np.int16(img)
img = img * (contrast / 127 + 1) - contrast + brightness
img = np.clip(img, 0, 255)
img = np.uint8(img)

fig, ax = plt.subplots(1, 1, figsize=(16, 8)) # 이미지 출력 용도
ax.imshow(img) # 이미지 출력
```

```
detector = dlib.get_frontal_face_detector() # 얼굴 인식용 클래스 생성 (기본 제공되는 얼굴 인식 모델 사용)
sp = dlib.shape_predictor('checkpoint/shape_predictor_5_face_landmarks.dat') # 인식된 얼굴에서 랜드마크 찾기위한 클래스 생성

dets = detector(img) # 데이터 업 샘플링
s = sp(img, dets[0]) # 인식된 좌표에서 랜드마크 추출
img = dlib.get_face_chip(img, s, size=256, padding=0.65) # 추출된 이미지 정보에 따라 size 및 padding값에 따른 이미지 저장

plt.figure(figsize=(6, 6)) # 이미지 출력 용도
plt.imshow(img) # 이미지 출력
```

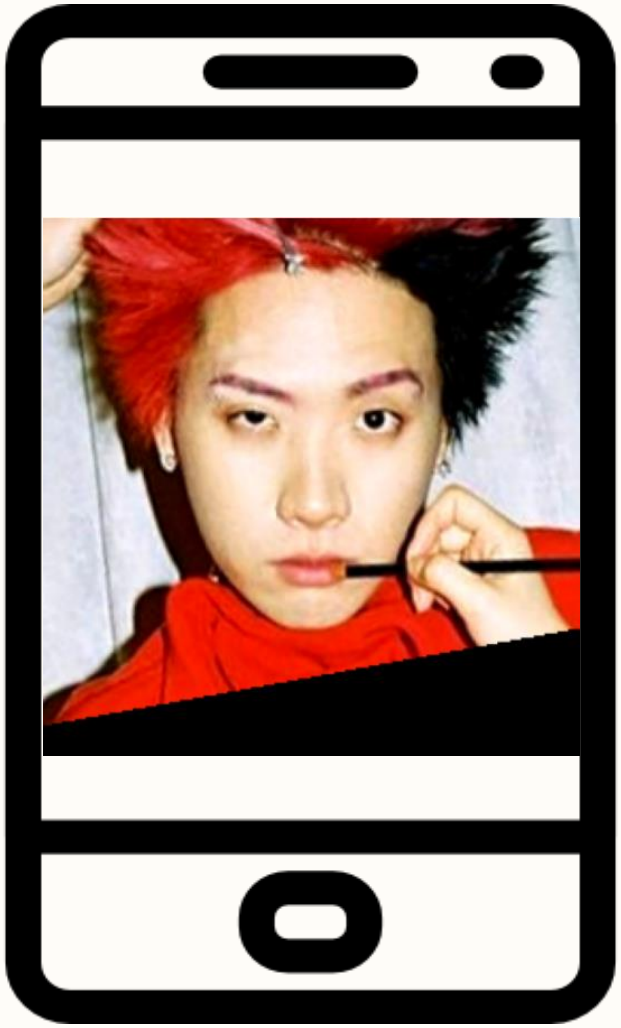
이미지 load후, test를 위해 data를 sampling 해주는 과정

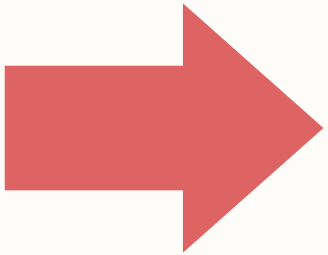
```
# preprocessing
img_input = cv2.resize(img, dsize=(256, 256), interpolation=cv2.INTER_NEAREST) # 이미지 크기 재조정
img_input = np.expand_dims(img_input, axis=0) # 배열에 차원 추가
img_input = img_input / 127.5 - 1 # sess.run 연산 처리를 위해 값 조정

# inference
img_output = sess.run(gan.test_fake_B, feed_dict={gan.test_domain_A: img_input}) # UGATIT 모델에서 지원하는 이미지 변환 연산그래프 실행. sess.run(연산
방식, input 맵핑 요소에 대한 값)

# postprocessing
img_output = (img_output + 1) * 127.5 # sess.run 연산 처리 후 원래 값으로 조정
img_output = img_output.astype(np.uint8).squeeze() # 배열에 차원 축소

#출력 용도
fig, ax = plt.subplots(1, 2, figsize=(16, 8))
ax[0].imshow(img)
ax[1].imshow(img_output)
```

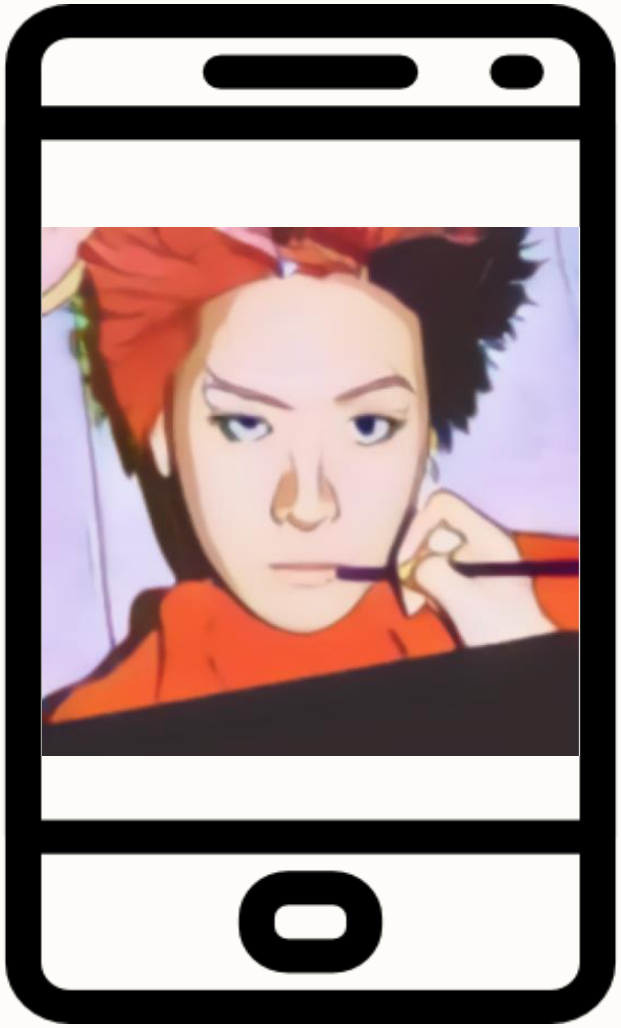
Test 결과

<Input> <Output>

예상 구현도







한계 및 보완점

한계 및 보완점

1. 의도하지 않은 결과물 산출 가능성 존재
   (ex. Shape, 색 변화, 특징 유실 등)

   => Reconstruction loss를 높여 색의 변화나 shape의 변화를 조절하여 보완 가능

2. 결과 이미지에 aliasing 현상 발생 가능

   => Super resolution 기술 中 Waifu2x기술을 사용하여 결과 이미지 퀄리티 향상

   * Waifu2x-애니메이션 스타일 이미지에 최적화된 super resolution 기술

3. 작가의 한정된 그림 수 -> 데이터 수집에 한계 존재

향후 발전 계획

향후 발전 계획

1. 애니메이션 스타일 외에도 다양한 이미지 스타일을 학습을 통해 얻을 수 있음
   활용: 카메라 필터 적용 가능
2. 간단한 스케치나 드로잉 그림을 대상으로 학습시킬 경우 실제 웹툰에서 작업량을 줄일 수 있음
3. 은퇴한 작가의 그림체로 만화 제작 가능
4. 그림 실력이 부족해도 손쉽게 그림을 얻을 수 있고 더 나아가 웹툰 제작 가능
5. 이미지 뿐만 아니라 영상에도 활용 가능

-사업화-

1) 사용자가 원하는 사진을 원하는 웹툰의 그림체로 변환 가능
   => 웹툰 제작자는 이득을 취하고, 소비자는 원하는 사진을 획득
2) 카메라 필터 어플로 적용하여 광고 및 구독제를 통한 수익 창출

감사합니다:)