

---

A Style-Based Generator Architecture for Generative Adversarial Networks

# StyleGAN

백승우

# 목차

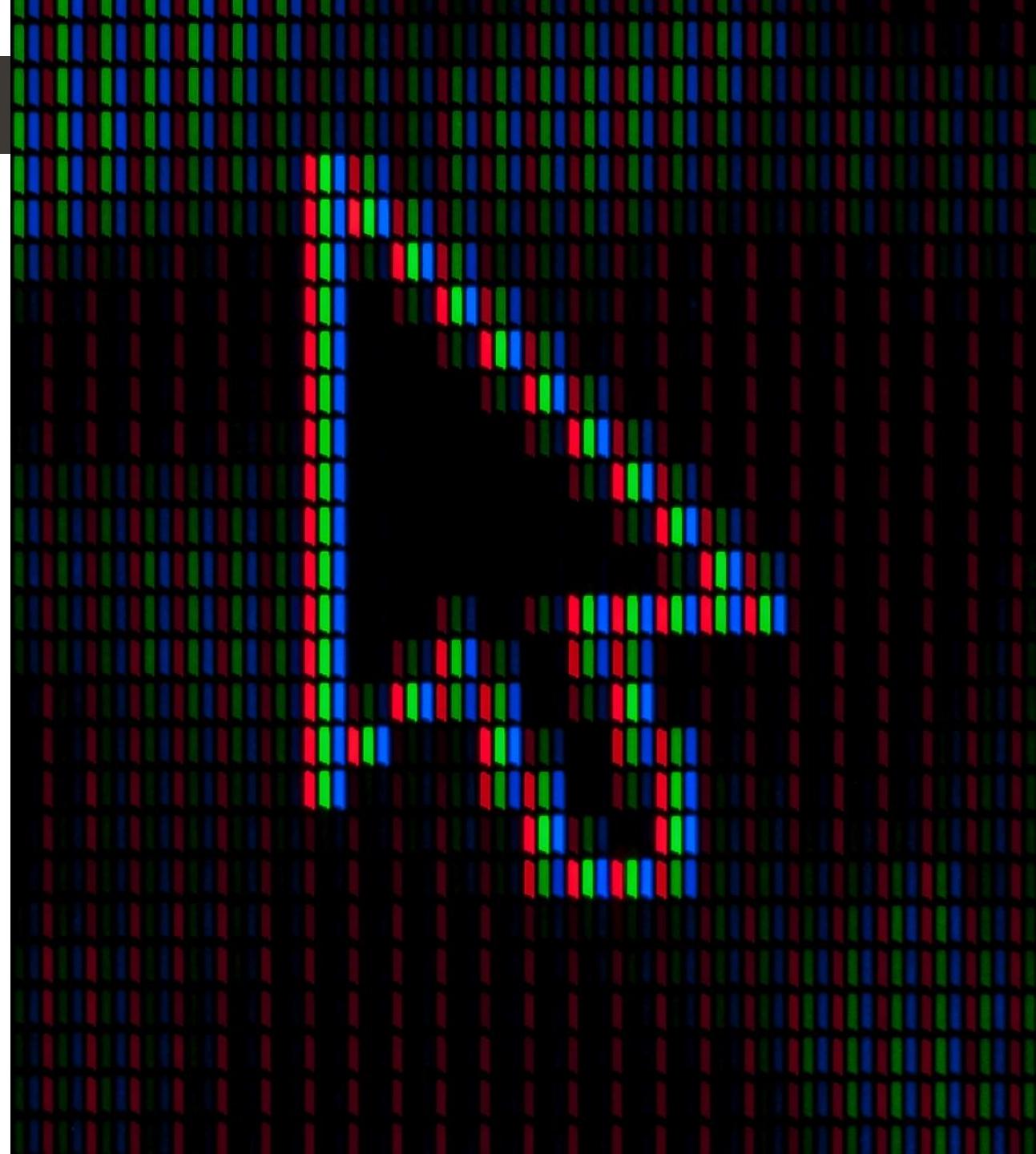
a table of contents

1 Abstract

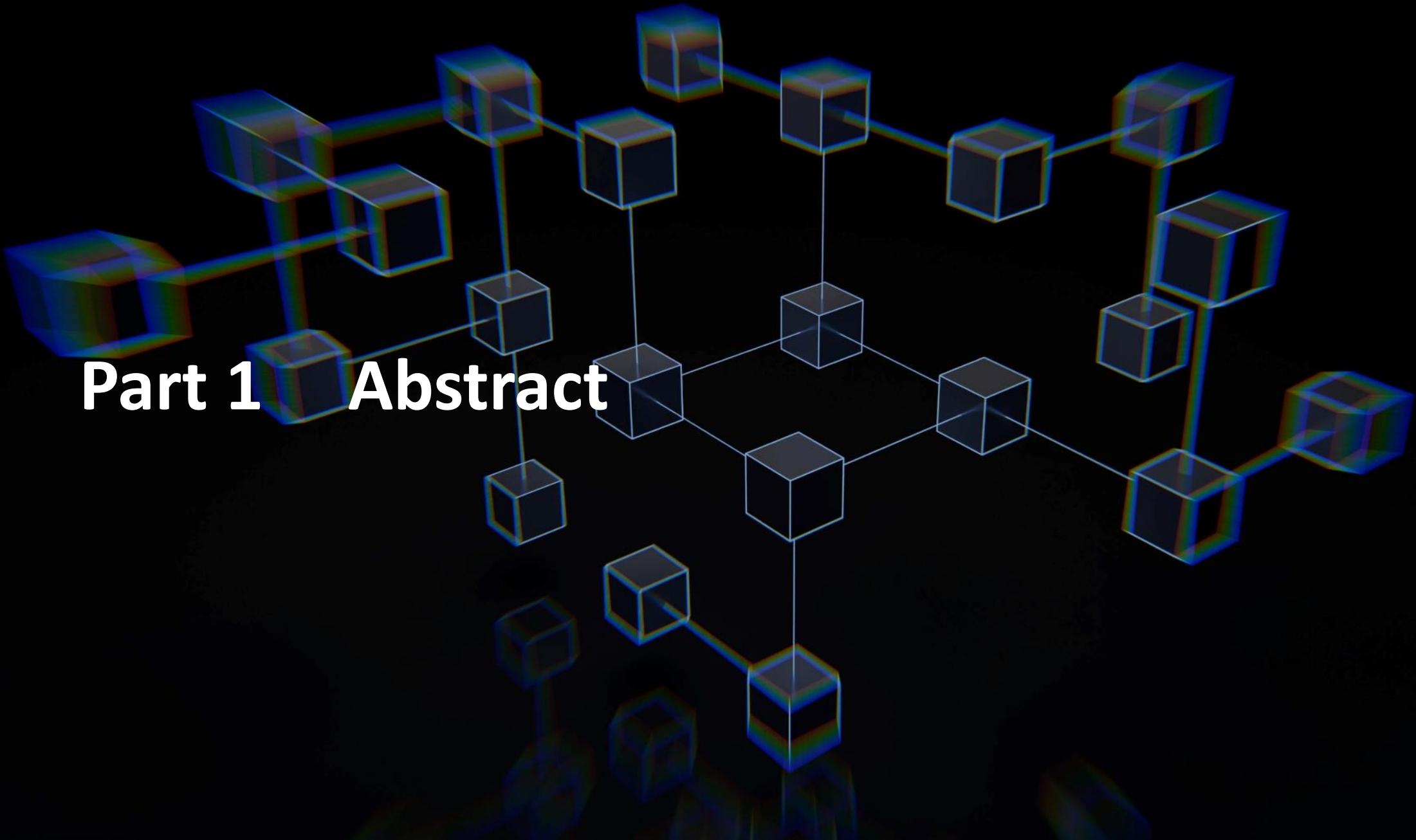
2 Prior Research

3 StyleGAN

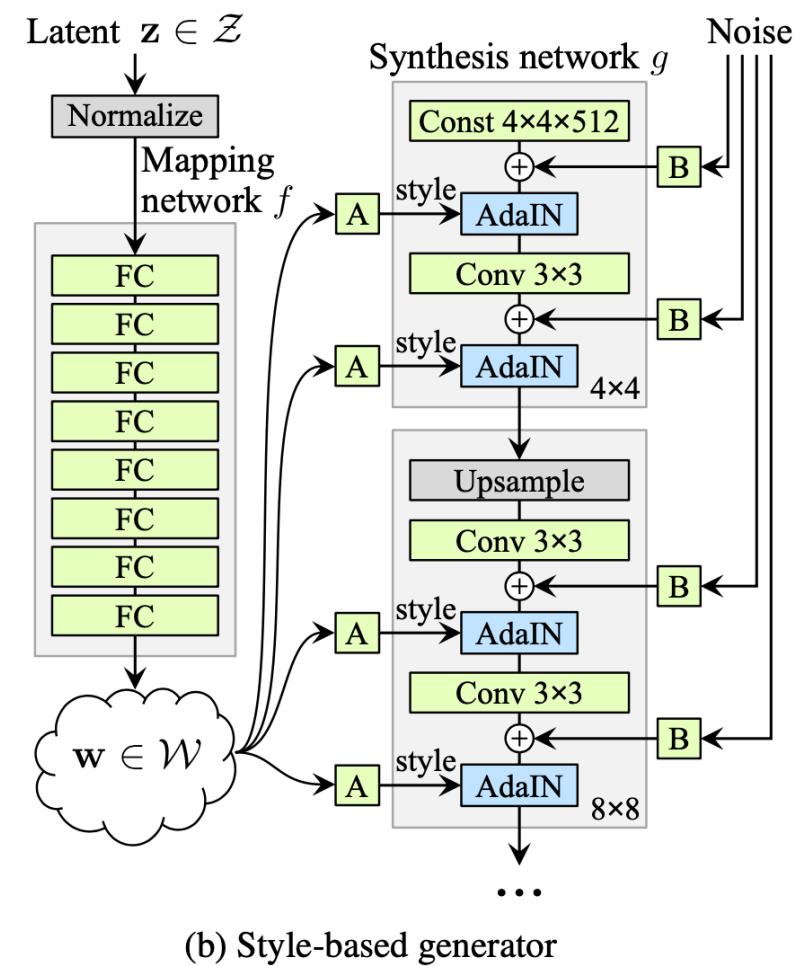
4 Conclusion



# Part 1 Abstract



We propose an alternative generator architecture for generative adversarial networks, borrowing from **style transfer literature**. The new architecture leads to an automatically learned, unsupervised separation of **high-level attributes** (e.g., pose and identity when trained on human faces) and **stochastic variation** in the generated images (e.g., freckles, hair), and it enables intuitive, scale-specific control of the synthesis. The new generator improves the state-of-the-art in terms of **traditional distribution quality metrics**, leads to demonstrably better interpolation properties, and also better disentangles the latent factors of variation. To quantify **interpolation quality** and **disentanglement**, we propose two new, automated methods that are applicable to any generator architecture. Finally, we introduce a new, highly varied and high-quality **dataset of human faces**.



(b) Style-based generator



## Part 2 Prior Research

# GAN

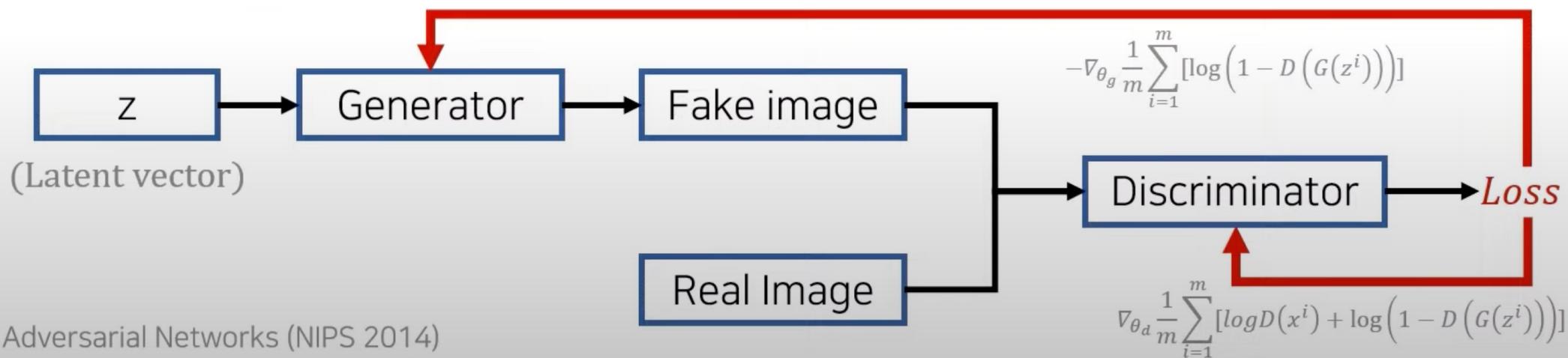
$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Generator

$G(z)$ : new data instance

Discriminator

$D(x)$  = Probability: a sample came from the real distribution (Real: 1 ~ Fake: 0)



# GAN

```
latent_dim = 100

# 생성자(Generator) 클래스 정의
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        # 하나의 블록(block) 정의
        def block(input_dim, output_dim, normalize=True):
            layers = [nn.Linear(input_dim, output_dim)]
            if normalize:
                # 배치 정규화(batch normalization) 수행(차원 동일)
                layers.append(nn.BatchNorm1d(output_dim, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        # 생성자 모델은 연속적인 여러 개의 블록을 가짐
        self.model = nn.Sequential(
            *block(latent_dim, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, 1 * 28 * 28),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 1, 28, 28)
        return img
```

# GAN

```
# 판별자(Discriminator) 클래스 정의
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(1 * 28 * 28, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    # 이미지에 대한 판별 결과를 반환
    def forward(self, img):
        flattened = img.view(img.size(0), -1)
        output = self.model(flattened)

        return output
```

# GAN

```

import time

n_epochs = 200 # 학습의 횟수(epoch) 설정
sample_interval = 2000 # 몇 번의 배치(batch)마다 결과를 출력할 것인지 설정
start_time = time.time()

for epoch in range(n_epochs):
    for i, (imgs, _) in enumerate(dataloader):

        # 진짜(real) 이미지와 가짜(fake) 이미지에 대한 정답 레이블 생성
        real = torch.cuda.FloatTensor(imgs.size(0), 1).fill_(1.0) # 진짜(real): 1
        fake = torch.cuda.FloatTensor(imgs.size(0), 1).fill_(0.0) # 가짜(fake): 0

        real_imgs = imgs.cuda()

        """ 생성자(generator)를 학습합니다. """
        optimizer_G.zero_grad()

        # 랜덤 노이즈(noise) 샘플링
        z = torch.normal(mean=0, std=1, size=(imgs.shape[0], latent_dim)).cuda()

        # 이미지 생성
        generated_imgs = generator(z)

        # 생성자(generator)의 손실(loss) 값 계산
        g_loss = adversarial_loss(discriminator(generated_imgs), real)

        # 생성자(generator) 업데이트
        g_loss.backward()
        optimizer_G.step()

        """ 판별자(discriminator)를 학습합니다. """
        optimizer_D.zero_grad()

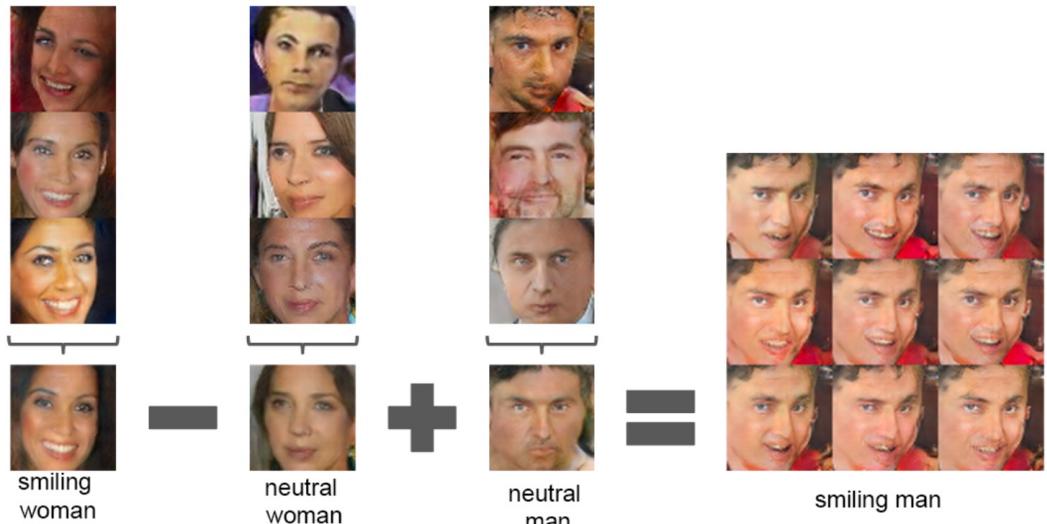
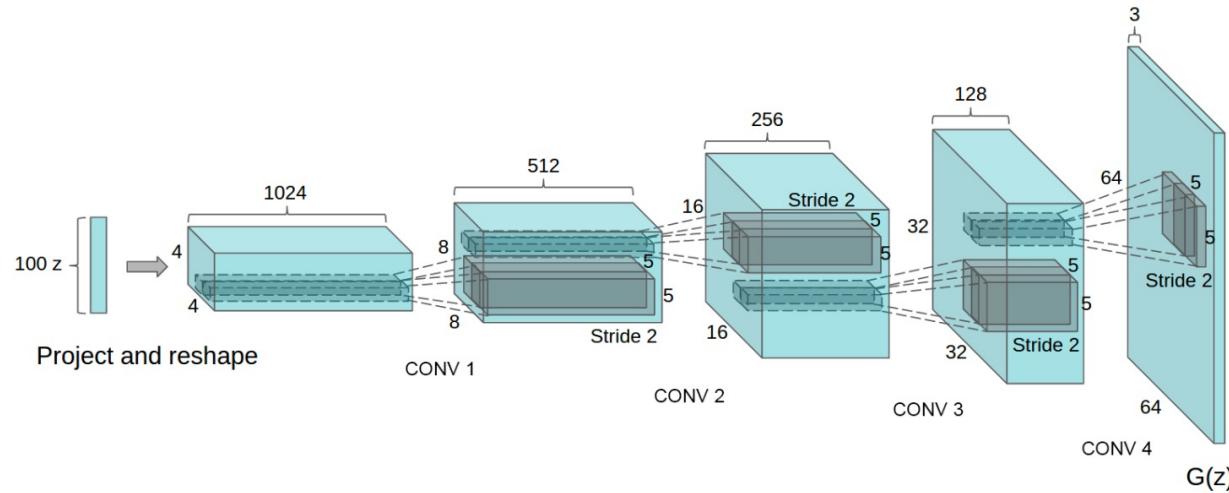
        # 판별자(discriminator)의 손실(loss) 값 계산
        real_loss = adversarial_loss(discriminator(real_imgs), real)
        fake_loss = adversarial_loss(discriminator(generated_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2

        # 판별자(discriminator) 업데이트
        d_loss.backward()
        optimizer_D.step()

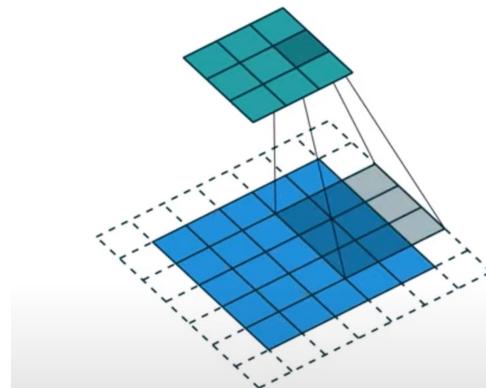
        done = epoch * len(dataloader) + i
        if done % sample_interval == 0:
            # 생성된 이미지 중에서 25개만 선택하여 5 x 5 격자 이미지에 출력
            save_image(generated_imgs.data[:25], f"{done}.png", nrow=5, normalize=True)

```

# DCGAN

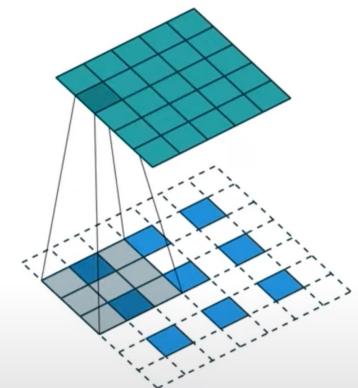


판별자(Discriminator)



Strided Convolution: 너비와 높이가 감소

생성자(Generator)



Transposed Convolution: 너비와 높이가 증가

# WGAN

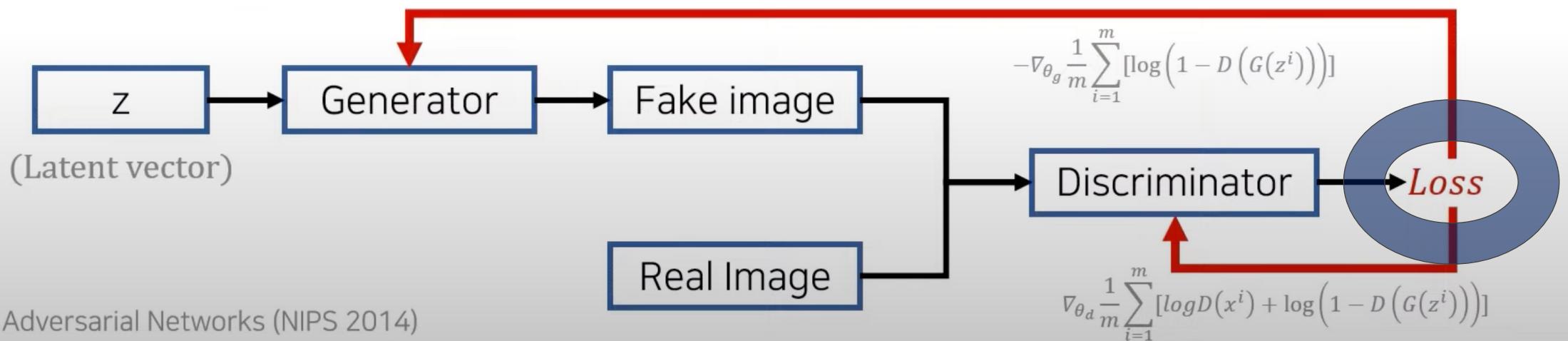
$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Generator

$G(z)$ : new data instance

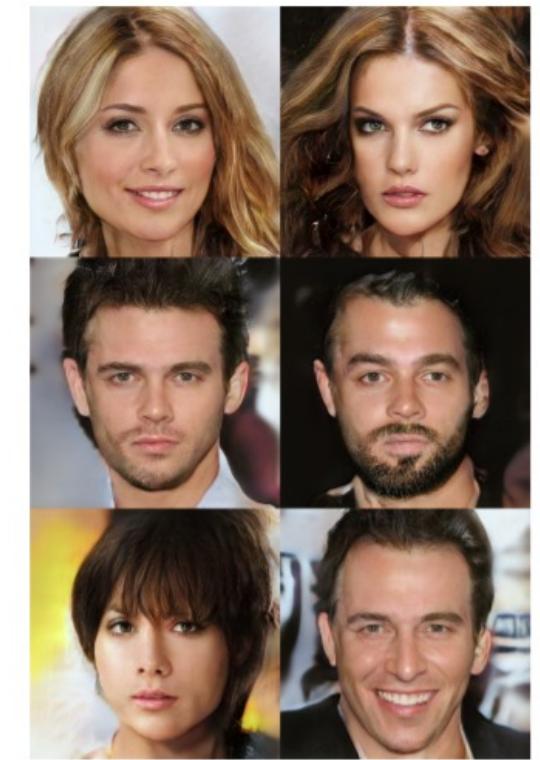
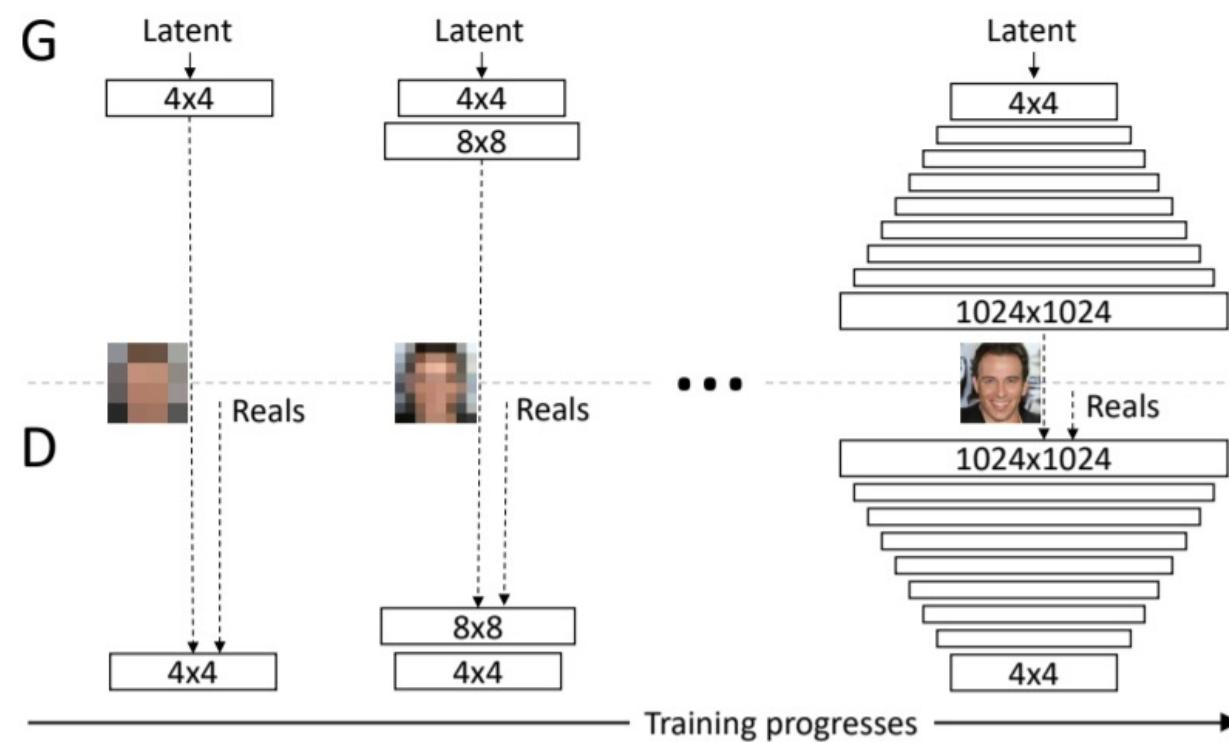
Discriminator

$D(x)$  = Probability: a sample came from the real distribution (Real: 1 ~ Fake: 0)



# PGGAN

<https://sensitivityit.tistory.com/508>



# Part 3 StyleGAN



# Mapping Network

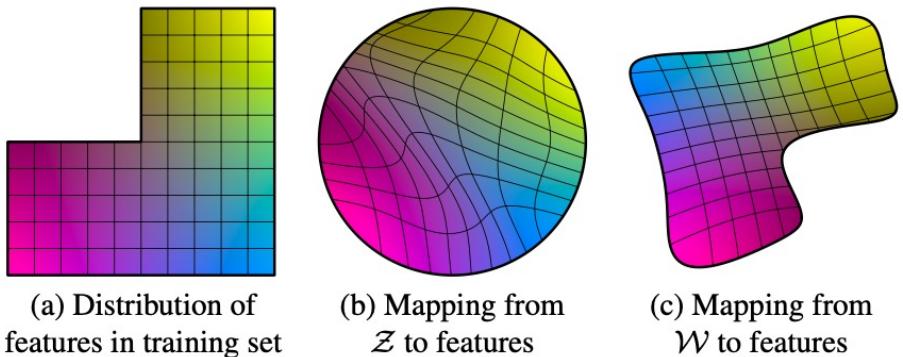
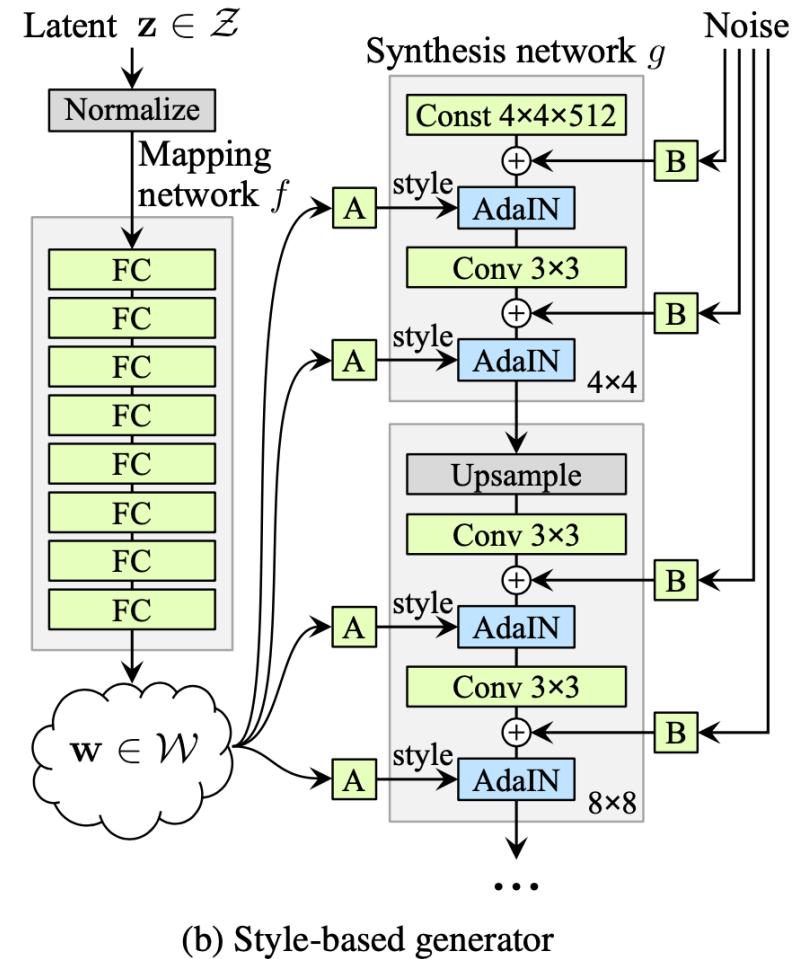
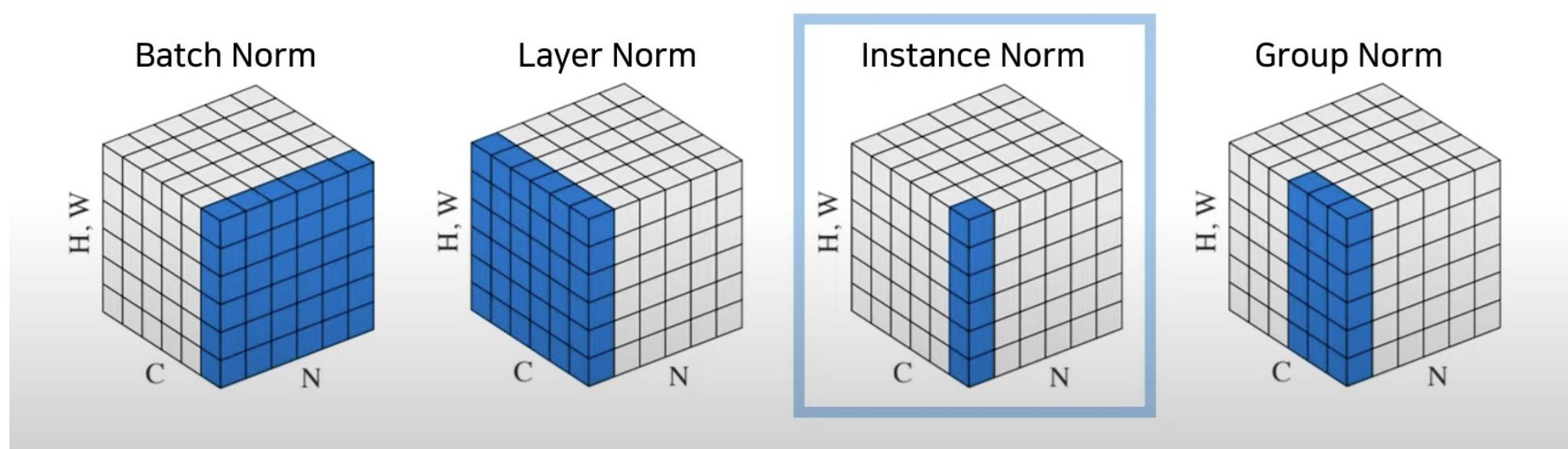


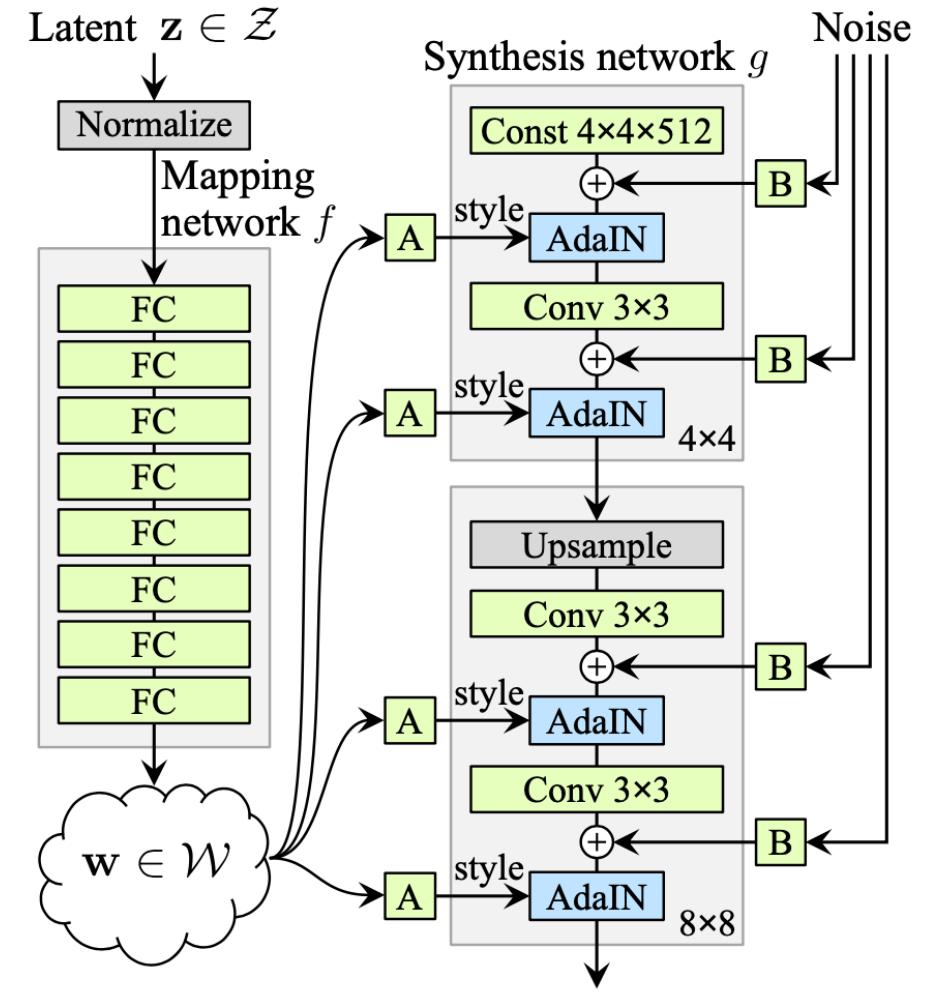
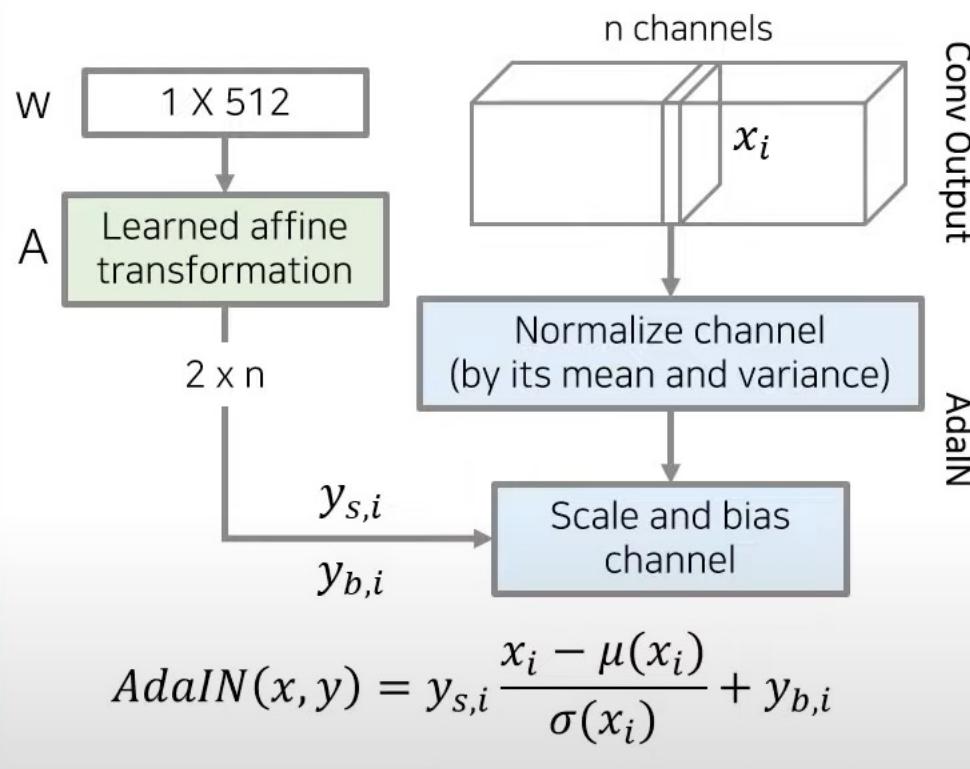
Figure 6. Illustrative example with two factors of variation (image features, e.g., masculinity and hair length). (a) An example training set where some combination (e.g., long haired males) is missing. (b) This forces the mapping from  $\mathcal{Z}$  to image features to become curved so that the forbidden combination disappears in  $\mathcal{Z}$  to prevent the sampling of invalid combinations. (c) The learned mapping from  $\mathcal{Z}$  to  $\mathcal{W}$  is able to “undo” much of the warping.



# AdaIN

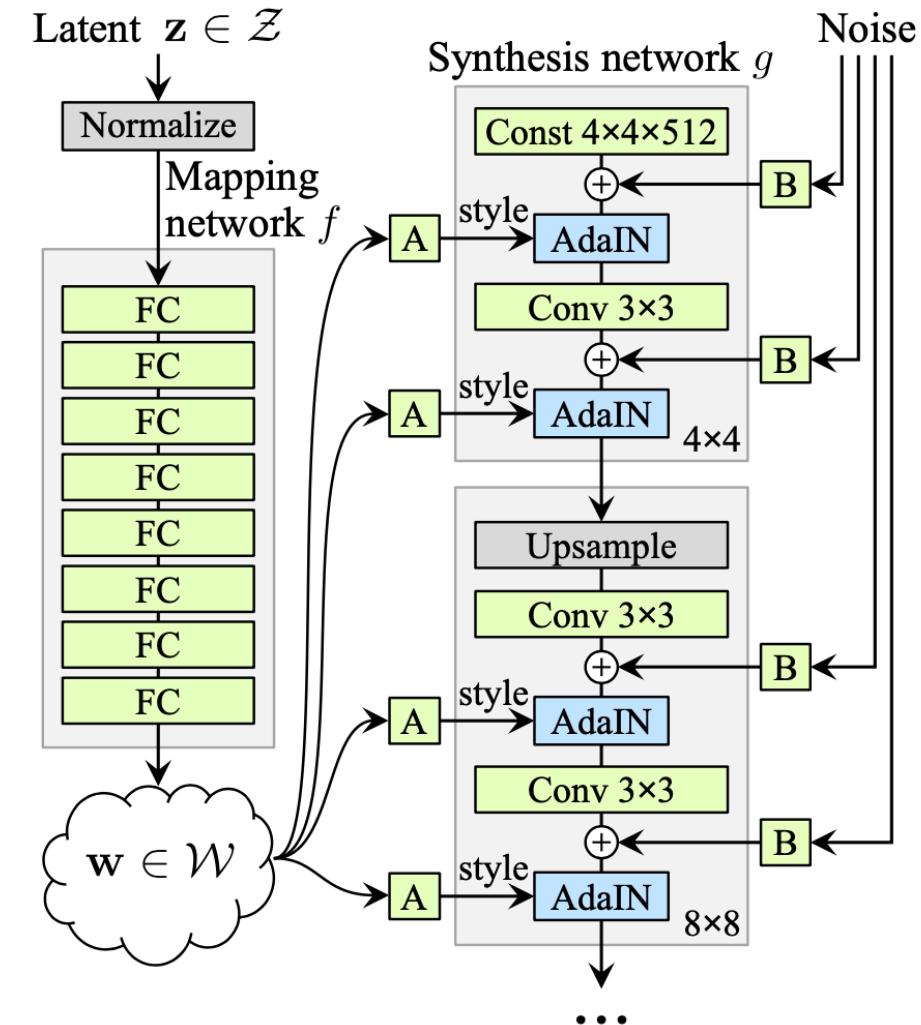


# AdaIN



(b) Style-based generator

# Removing Traditional input



(b) Style-based generator

# Stochastic Variation

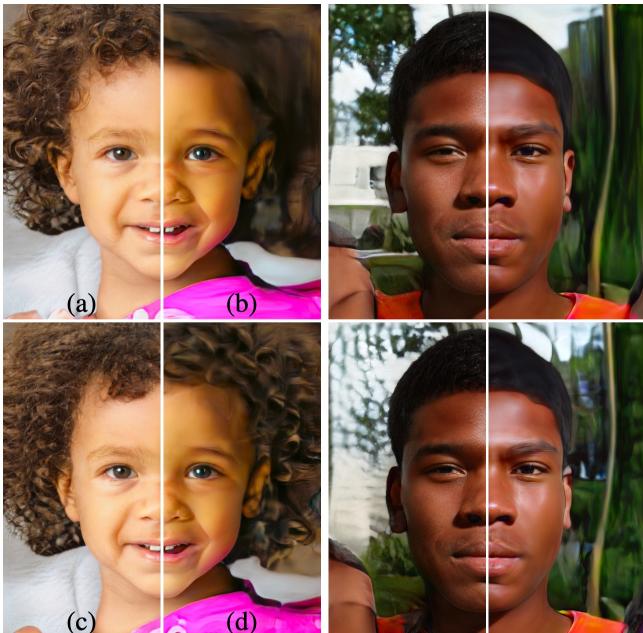
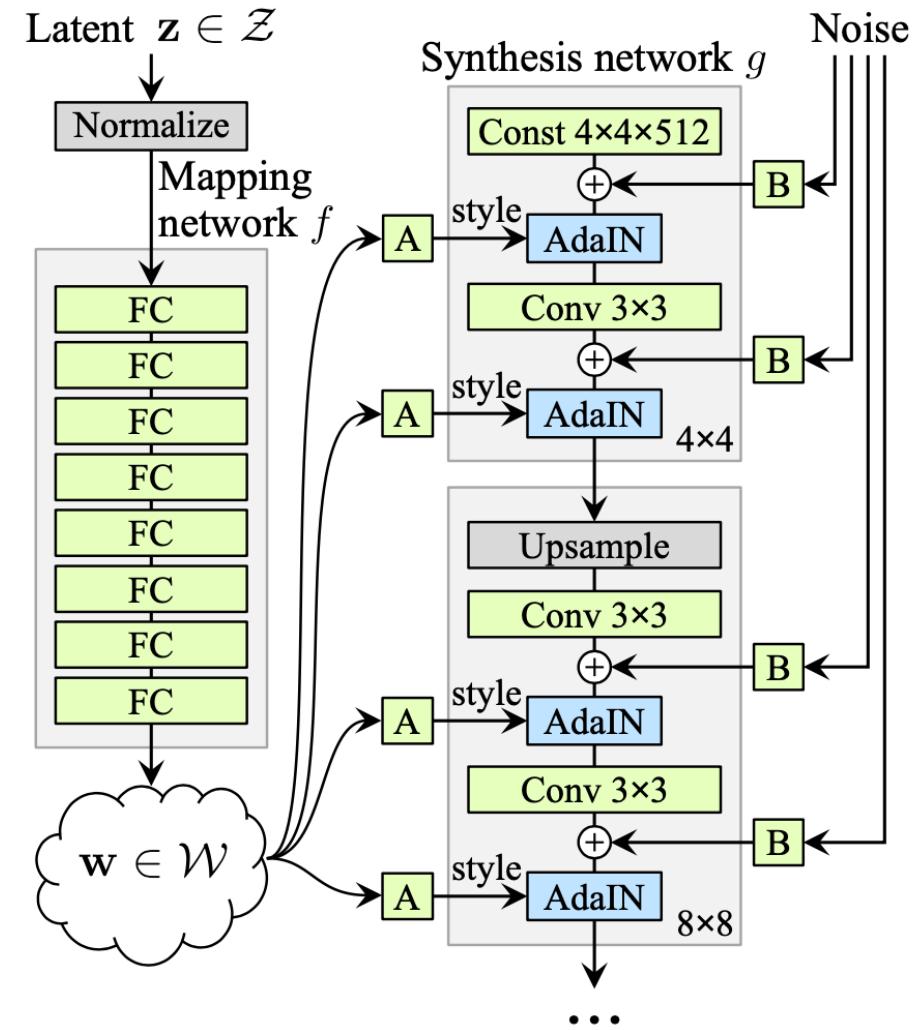
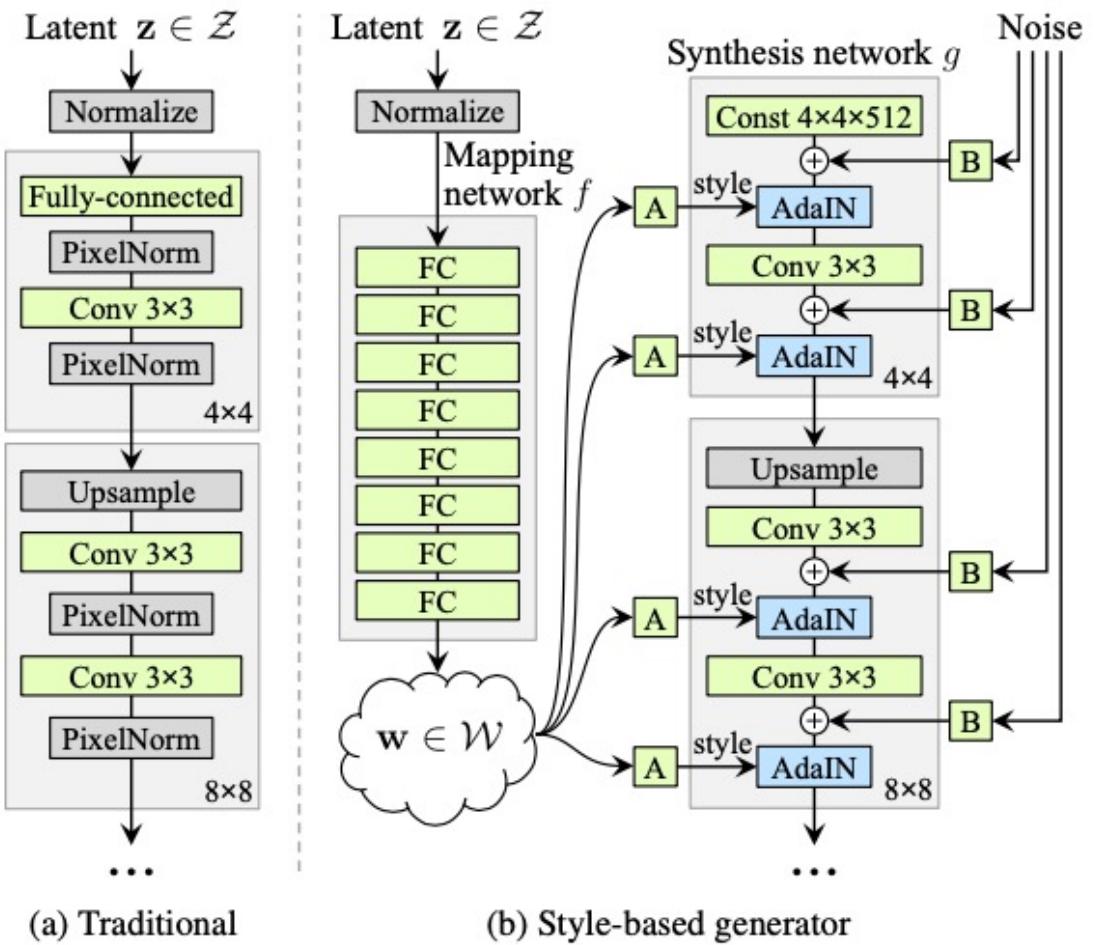


Figure 5. Effect of noise inputs at different layers of our generator. (a) Noise is applied to all layers. (b) No noise. (c) Noise in fine layers only ( $64^2 - 1024^2$ ). (d) Noise in coarse layers only ( $4^2 - 32^2$ ). We can see that the artificial omission of noise leads to featureless “painterly” look. Coarse noise causes large-scale curling of hair and appearance of larger background features, while the fine noise brings out the finer curls of hair, finer background detail, and skin pores.



(b) Style-based generator

# StyleGAN architecture: Disentanglement



(a) Traditional

(b) Style-based generator

# Latent vector Meanings of StyleGAN

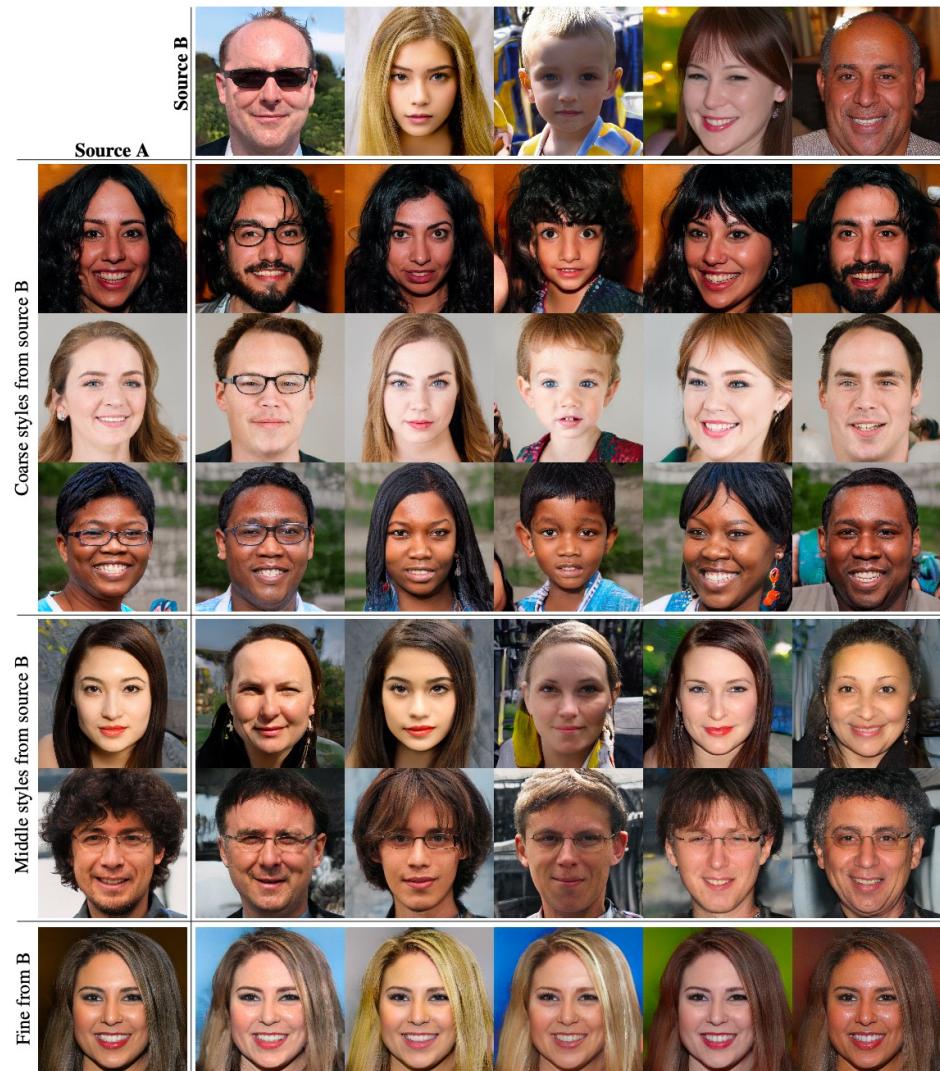
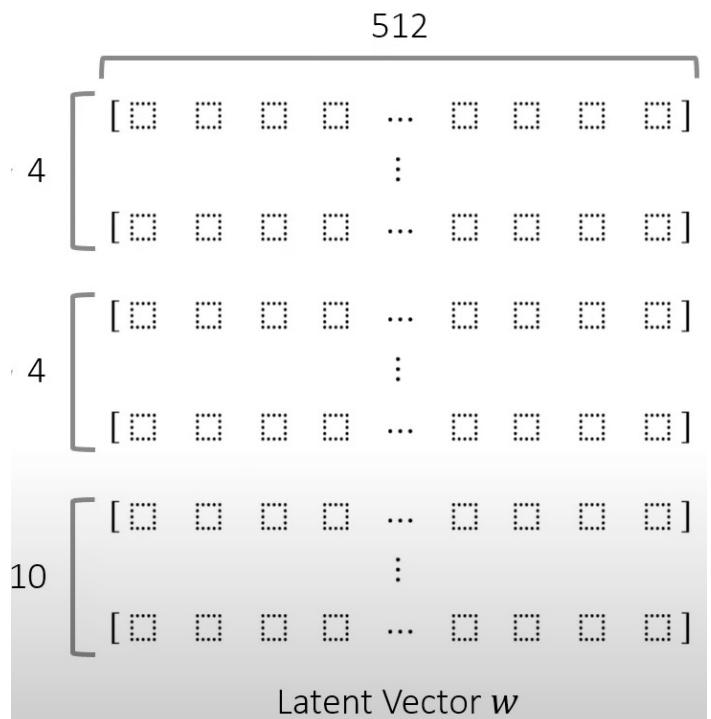
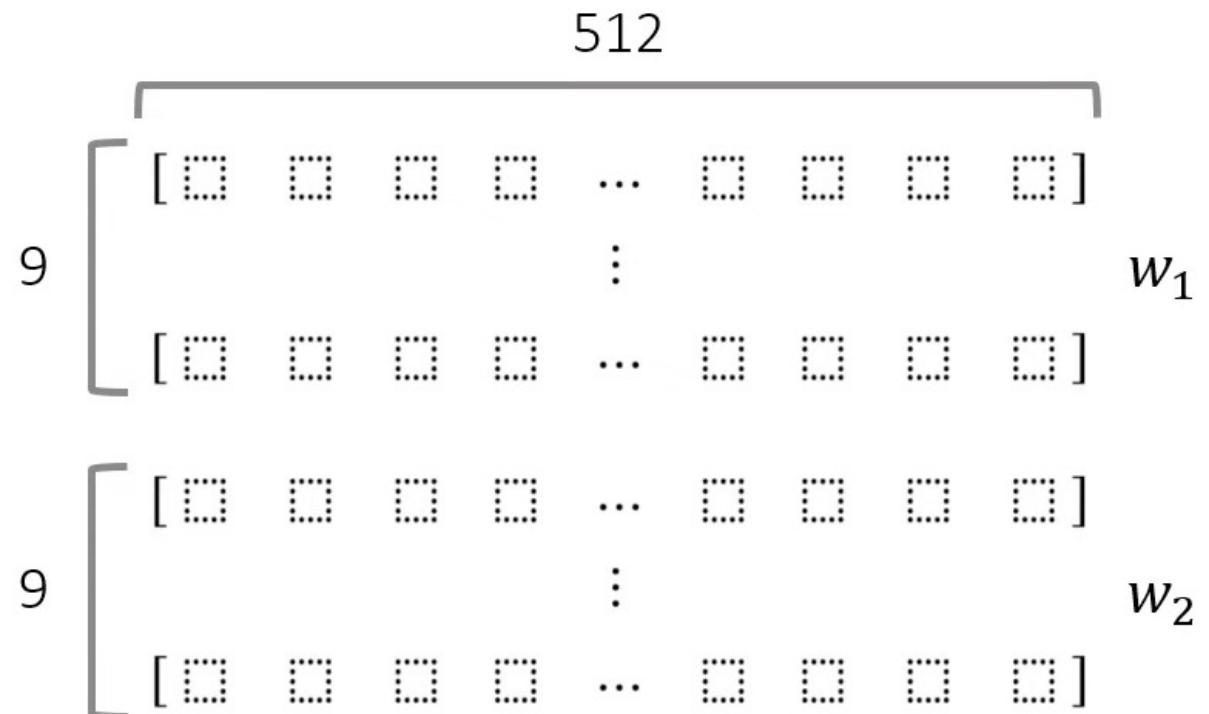


Figure 3. Two sets of images were generated from their respective latent codes (sources A and B); the rest of the images were generated by copying a specified subset of styles from source B and taking the rest from source A. Copying the styles corresponding to coarse spatial resolutions ( $4^2 - 8^2$ ) brings high-level aspects such as pose, general hair style, face shape, and eyeglasses from source B, while all colors (eyes, hair, lighting) and finer facial features resemble A. If we instead copy the styles of middle resolutions ( $16^2 - 32^2$ ) from B, we inherit smaller scale facial features, hair style, eyes open/closed from B, while the pose, general face shape, and eyeglasses from A are preserved. Finally, copying the fine styles ( $64^2 - 1024^2$ ) from B brings mainly the color scheme and microstructure.

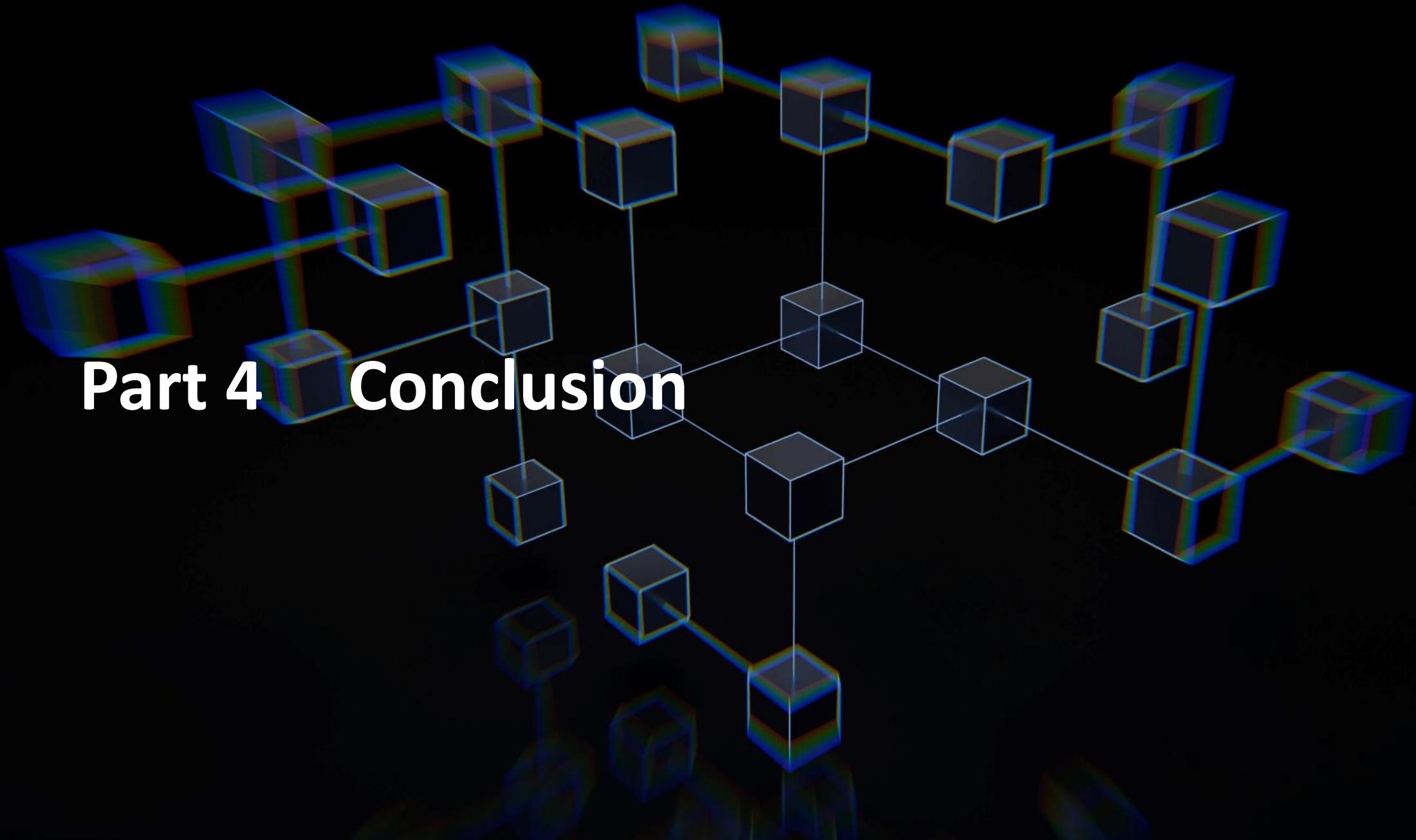
# Style Mixing

Mixing regularization	Number of latents during testing			
	1	2	3	4
E 0%	4.42	8.22	12.88	17.41
50%	4.41	6.10	8.71	11.61
F 90%	<b>4.40</b>	<b>5.11</b>	6.88	9.03
100%	4.83	5.17	<b>6.63</b>	<b>8.40</b>

Table 2. FIDs in FFHQ for networks trained by enabling the mixing regularization for different percentage of training examples. Here we stress test the trained networks by randomizing 1...4 latents and the crossover points between them. Mixing regularization improves the tolerance to these adverse operations significantly. Labels E and F refer to the configurations in Table 1.



## Part 4 Conclusion



Based on both our results and parallel work by Chen et al. [6], it is becoming clear that the traditional GAN generator architecture is in every way inferior to a style-based design. This is true in terms of established quality metrics, and we further believe that our investigations to the separation of high-level attributes and stochastic effects, as well as the linearity of the intermediate latent space will prove fruitful in improving the understanding and controllability of GAN synthesis.

We note that our average path length metric could easily be used as a regularizer during training, and perhaps some variant of the linear separability metric could act as one, too. In general, we expect that methods for directly shaping the intermediate latent space during training will provide interesting avenues for future work.