Taming Transformers for High-Resolution Image Synthesis (VQ-GAN)

김지환

Contents

1. Introduction

1. Introduction

Abstract

- Transformer는 sequential data에 대해 long-range interaction을 학습하도록 설계되었고

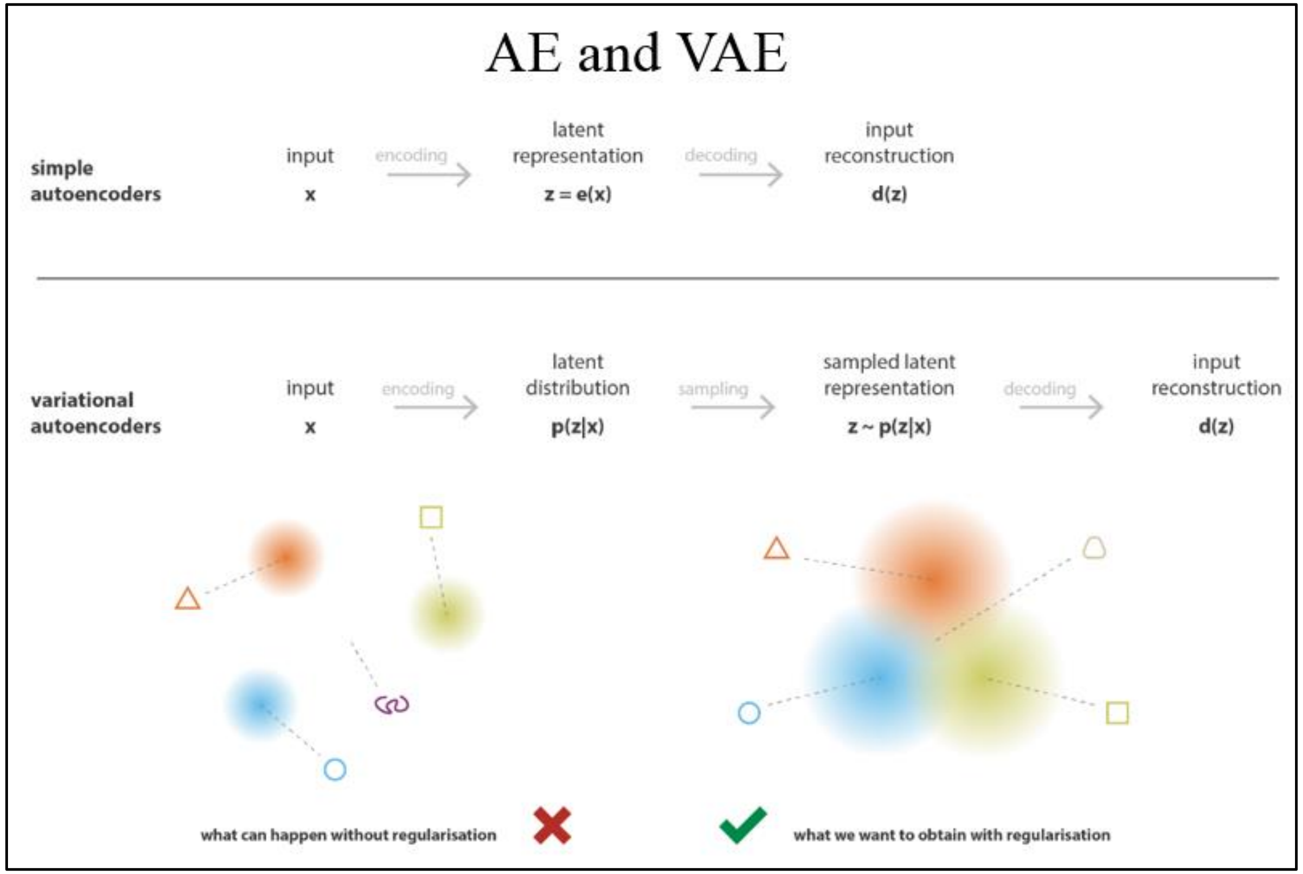
  CNN과 달리 local interaction을 우선시하는 inductive-bias가 없다.
- CNN의 inductive-bias와 transformer의 expressivity를 결합해 고해상도 이미지를 생성하는 것이 목적.
- CNN으로 이미지 구성요소의 contextrich vocabulary를 학습.
- Transformer로 고해상도 이미지 구성을 효율적으로 모델링.
- 클래스 정보나 segmentation 같은 공간 정보로 이미지를 제어하는 conditional synthesis task에 쉽게 적용.

1. Introduction

Background - VAE

Auto-Encoder: 입력을 latent variable로 보낸 후 latent variable로 원본 복원, Encoder 중심의 모델.

Variational-AE: 입력데이터의 분포를 근사하는 모델을 학습 후 새로운 이미지 생성, Decoder 중심의 모델.



http://egloos.zum.com/incredible/v/7473318

1. Introduction

Background - VAE

① 데이터의 분포 p(x)

② 데이터를 latent variable로 표현한 분포 p(z|x) : Encoder

③ latent variable 주었을때 데이터를 생성하는 p(x|z): Decoder

Posterior Likelihood Prior Evidence

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \Rightarrow p(x) = \frac{p(z,x)}{p(z|x)}$$

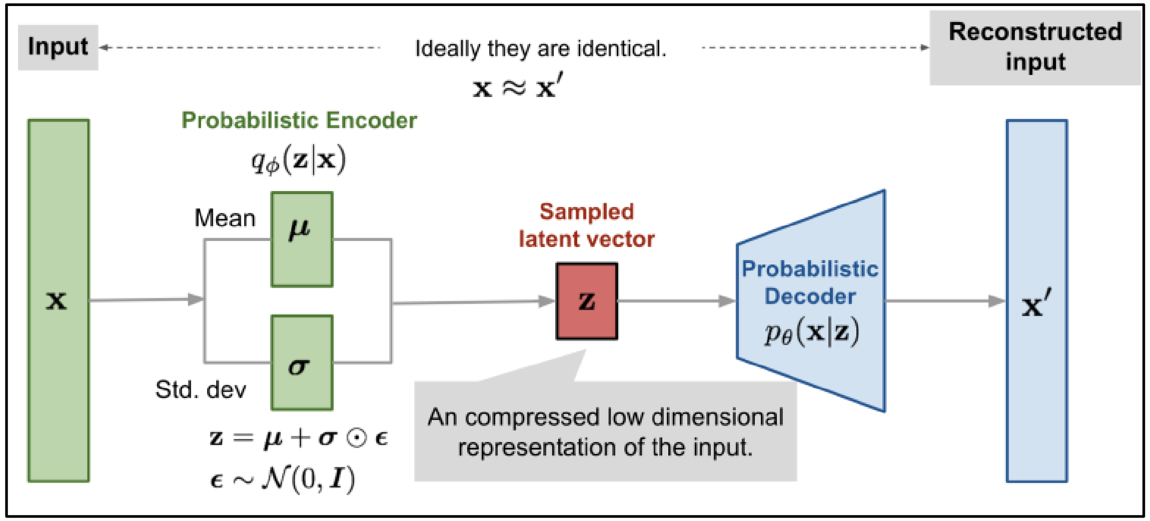
$$D_{KL}(P||Q) = \int P(x)\log\frac{P(x)}{Q(x)}dx$$

1. Introduction

Background - VAE

Maximum likelihood: $\arg\max_\theta p_\theta(x) = \int_z p_\theta(x,z) \;=\int_z p_\theta(x|z)p_\theta(z)$ 모든 z에 대해 계산할수 없기때문에 Intractable.

Variational Inference: 복잡한 p가 있을때 단순한 q 로 근사하겠다. $q_\phi(z|x) \approx p_\theta(z|x)$



$$
\begin{aligned}
\log(p_\theta(x)) &= \int_z q_\phi(z|x)\log(p_\theta(x)) \\
&= \int_z q_\phi(z|x)\log\frac{p_\theta(z,x)}{p_\theta(z|x)} \\
&= \int_z q_\phi(z|x)\log\left(\frac{p_\theta(z,x)}{q_\phi(z|x)}\frac{q_\phi(z|x)}{p_\theta(z|x)}\right) \\
&= \int_z q_\phi(z|x)\log\frac{p_\theta(z,x)}{q_\phi(z|x)} + \int_z q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(z|x)} \\
&= \mathcal{L}(\theta,\phi;x) + D_{KL}(q_\phi(z|x)||p_\theta(z|x)) \\
&\geq \underline{\mathcal{L}(\theta,\phi;x)} \qquad \because D_{KL} \geq 0
\end{aligned}
$$

$$
\begin{aligned}
\underline{\mathcal{L}(\theta,\phi;x)} &= \int_z q_\phi(z|x)\log\frac{p_\theta(z,x)}{q_\phi(z|x)} \\
&= -\int_z q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(z)} + \int_z q_\phi(z|x)\log p_\theta(x|z) \qquad \because p_\theta(z,x) = p(x|z)p(z) \\
&= -D_{KL}(q_\phi(z|x)||p(z)) + \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]
\end{aligned}
$$

https://blog.naver.com/PostView.naver?blogId=jk96491&logNo=222177759271

1. Introduction

Background - VAE

Prior z: 다루기 쉬운 표준 정규분포로 가정.

원 데이터에 대한 likelihood 선택

Variational inference를 위한 approximation class 중 선택

다루기 쉬운 확률 분포 중 선택

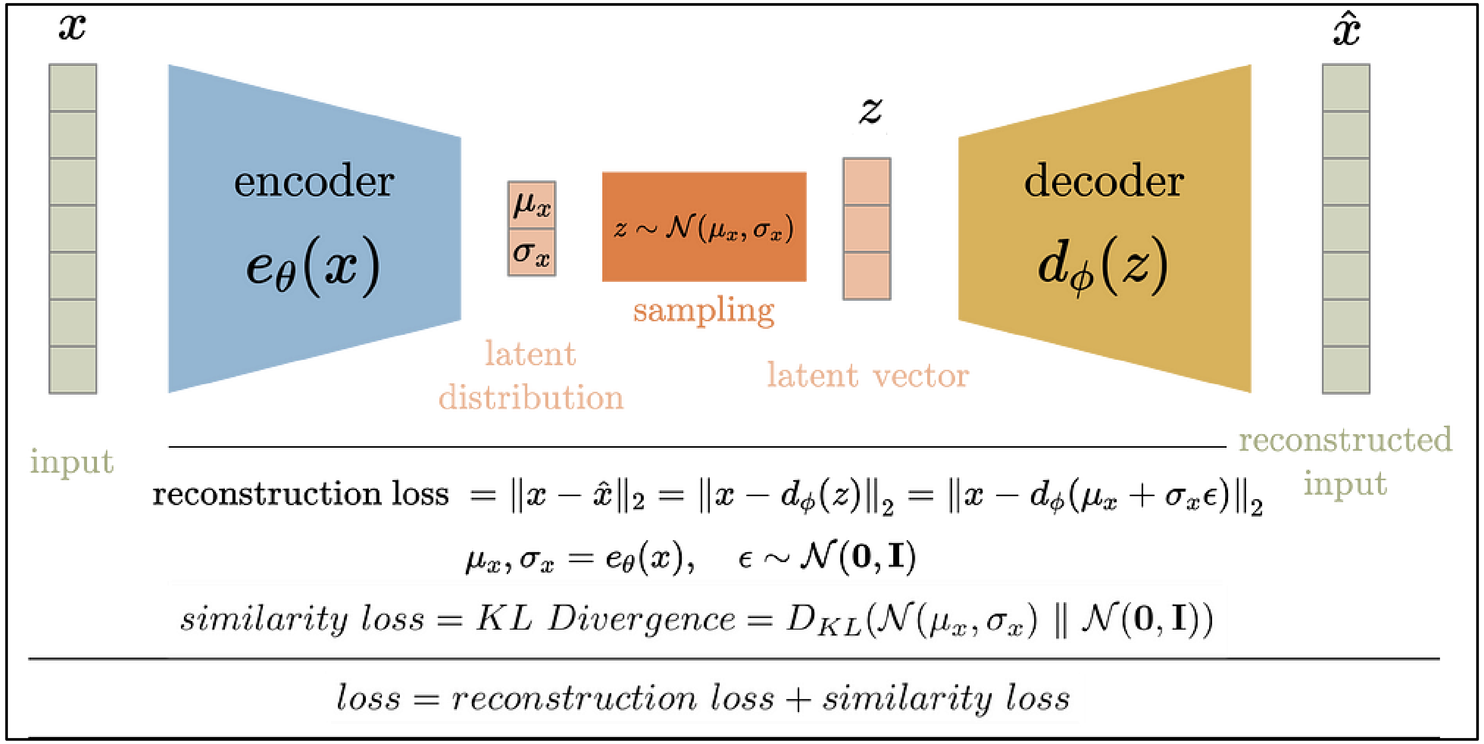
$$L_i(\phi,\theta,x_i) = \underbrace{-\mathbb{E}_{q_\phi(z|x_i)}[\log(p_\theta(x_i|z))]}_{\text{Reconstruction Error}} + \underbrace{KL(q_\phi(z|x_i)||p(z))}_{\text{Regularization}}$$

Reconstruction Error

- 현재 샘플된 z에 대한 negative log likelihood
- x_i에 대한 복원 오차 (AutoEncoder 관점)

Regularization

- 현재 샘플된 z에 대한 대한 추가 조건
- 샘플링되는 z들에 대한 통제성을 prior를 통해 부여, Variational distribution q(z|x)가 p(z)와 유사해야 한다는 조건을 부여



https://towardsdatascience.com/difference-between-autoencoder-ae-and-variational-autoencoder-vae-ed7be1c038f2

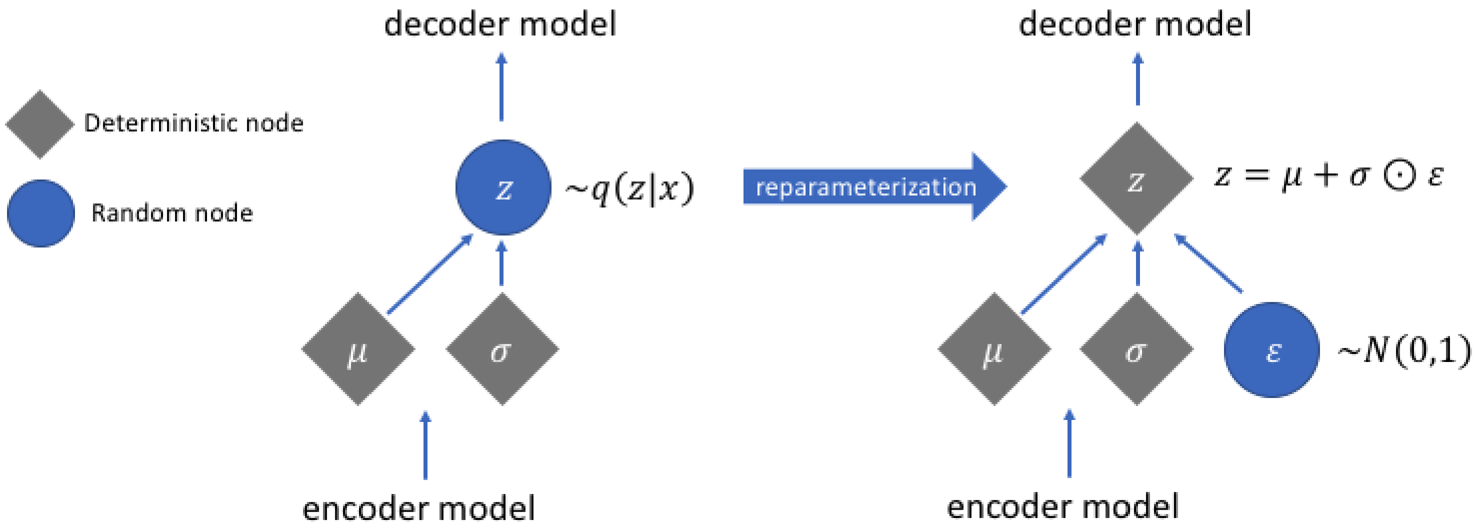
1. Introduction

Background - VAE

<u>Reparameterization trick</u>

mu, sigma를 통해 z를 샘플링을하게 되면 deterministic하지 않아 역전파가 불가능.

e를 확정적으로 샘플링 해놓고 z를 만들면 deterministic해져서 역전파가 가능해진다.



https://nrhan.tistory.com/entry/Variational-autoencoder

1. Introduction

Background - VAE

Code

```python
def encode(self, input: Tensor) -> List[Tensor]:
    result = self.encoder(input)
    result = torch.flatten(result, start_dim=1)
    mu = self.fc_mu(result)
    log_var = self.fc_var(result)
    return [mu, log_var]

def decode(self, z: Tensor) -> Tensor:
    result = self.decoder_input(z)
    result = result.view(-1, 512, 2, 2)
    result = self.decoder(result)
    result = self.final_layer(result)
    return result

def reparameterize(self, mu: Tensor, logvar: Tensor) -> Tensor:
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return eps * std + mu

def forward(self, input: Tensor, **kwargs) -> List[Tensor]:
    mu, log_var = self.encode(input)
    z = self.reparameterize(mu, log_var)
    return  [self.decode(z), input, mu, log_var]
```

```python
def loss_function(self,
                  *args,
                  **kwargs) -> dict:
    recons = args[0]
    input = args[1]
    mu = args[2]
    log_var = args[3]

    kld_weight = kwargs['M_N'] # Account for the minibatch samples from the dataset
    recons_loss =F.mse_loss(recons, input)
    kld_loss = torch.mean(-0.5 * torch.sum(1 + log_var - mu ** 2 - log_var.exp(), dim = 1)
                          , dim = 0)

    loss = recons_loss + kld_weight * kld_loss
    return {'loss': loss, 'Reconstruction_Loss':recons_loss.detach(), 'KLD':-kld_loss.detach()}
```

https://github.com/AntixK/PyTorch-VAE/blob/master/models/vanilla_vae.py

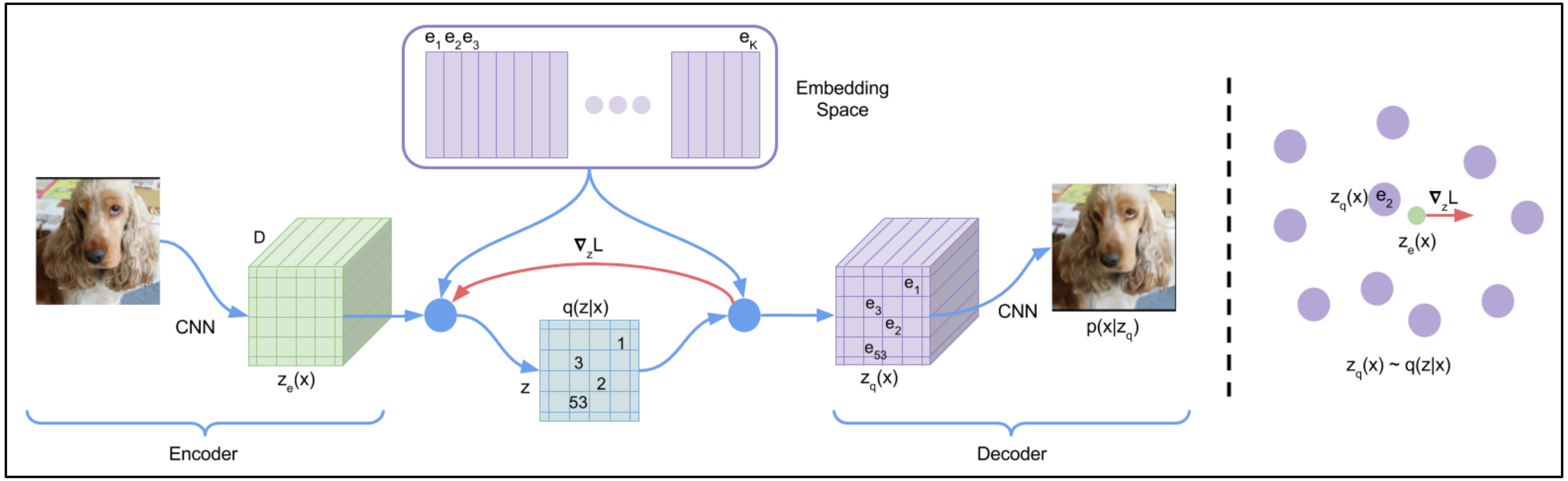
1. Introduction

Background - VQ VAE

Neural Discrete Representation Learning

Prior 분포를 연속적인 특정 분포로 가정하지 말고

이산적이고 훈련이 가능한 분포로 만들자!



$$e \in R^{K \times D}$$

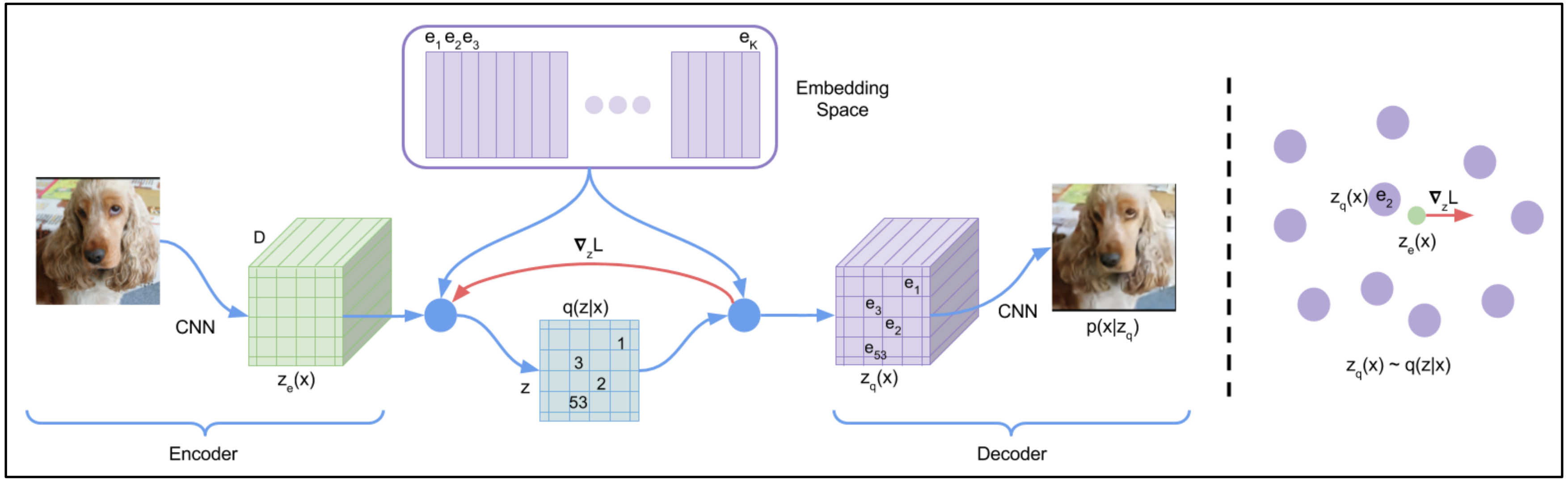
$$q(z=k|x) = \begin{cases} 1 & \text{for } k = \arg\min_j \|z_e(x) - e_j\|_2, \\ 0 & \text{otherwise} \end{cases}$$

$$z_q(x) = e_k, \quad \text{where} \quad k = \text{argmin}_j \|z_e(x) - e_j\|_2$$

1. Introduction

Background - VQ VAE

Discrete Latent variables



eg.

Input image **x**: 16x3x224x224

Encoded feature $\mathbf{Z_e(x)}$: 16x<u>64</u>x32x32 (B*D*H*W) -> 16,384 * <u>64</u>

Code book **e**: 512x<u>64</u> (num_embedding * D)

Encoding one hot **q(z|x)**: 16,384 * 512 (16x32x32x512)

Quantized latents $\mathbf{Z_q(x)}$: Encoding one hot * Code book: 16,384 * <u>64</u>

1. Introduction

Background - VQ VAE

Loss function

$$L = \log p(x|z_q(x)) + \|sg[z_e(x)] - e\|_2^2 + \|z_e(x) - sg[e]\|_2^2$$

Reconstruction Codebook Commitment

① Reconstruction: Decoder, Encoder 훈련 -> Encoder는 gradient를 복사해서 훈련
② Codebook: Codebook의 벡터가 Encoder의 output과 가깝게 되도록 훈련
③ Commitment: Encoder가 Codebook의 벡터와 가까운 벡터를 내도록 훈련

1. Introduction

Background - VQ VAE

1) Codebook 정의

```
self.embedding = nn.Embedding(self.K, self.D)
self.embedding.weight.data.uniform_(-1 / self.K, 1 / self.K)
```

2) 벡터 양자화

```
# Compute L2 distance between latents and embedding weights
dist = torch.sum(flat_latents ** 2, dim=1, keepdim=True) + \
       torch.sum(self.embedding.weight ** 2, dim=1) - \
       2 * torch.matmul(flat_latents, self.embedding.weight.t())  # [BHW x K]

# Get the encoding that has the min distance
encoding_inds = torch.argmin(dist, dim=1).unsqueeze(1)  # [BHW, 1]

# Convert to one-hot encodings
device = latents.device
encoding_one_hot = torch.zeros(encoding_inds.size(0), self.K, device=device)
encoding_one_hot.scatter_(1, encoding_inds, 1)  # [BHW x K]

# Quantize the latents
quantized_latents = torch.matmul(encoding_one_hot, self.embedding.weight)  # [BHW, D]
quantized_latents = quantized_latents.view(latents_shape)  # [B x H x W x D]
```

3) Loss 계산

```
# Compute the VQ Losses
commitment_loss = F.mse_loss(quantized_latents.detach(), latents)
embedding_loss = F.mse_loss(quantized_latents, latents.detach())

vq_loss = commitment_loss * self.beta + embedding_loss

# Add the residue back to the latents
quantized_latents = latents + (quantized_latents - latents).detach()

return quantized_latents.permute(0, 3, 1, 2).contiguous(), vq_loss  # [B x D x H x W]
```

$$L = \log p(x|z_q(x)) + \|sg[z_e(x)] - e\|_2^2 + \|z_e(x) - sg[e]\|_2^2$$

https://github.com/AntixK/PyTorch-VAE/blob/master/models/vq_vae.py

1. Introduction

Background - VQ VAE

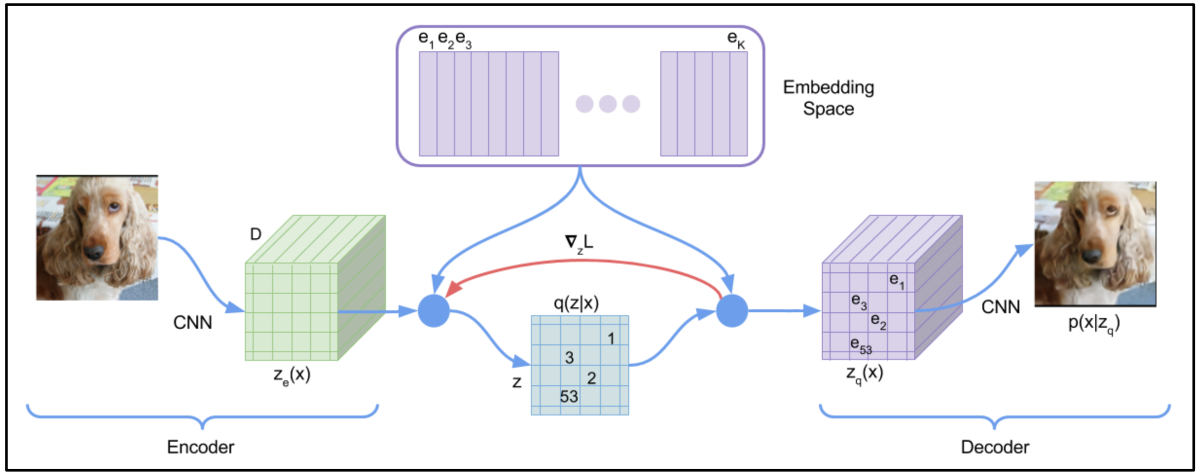
- VQ-VAE는 long-term dependency를 잘 모델링할 수 있다.
- 원본 source를 작은 latent로 수십 배 압축할 수 있다.
- 이러한 latent는 discrete하며, continuous latent와 비교하여 성능이 필적할 만하다.
- Image, audio, video 모두에 대해서 잘 모델링 및 압축한 후 중요한 내용을 잘 보존하면서 복원이 가능하다.

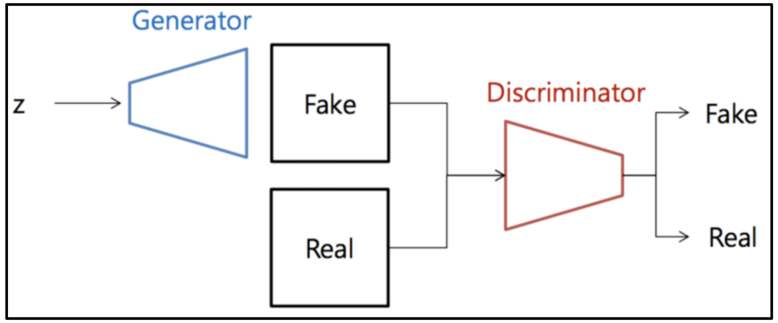
2. Methods

2. Methods

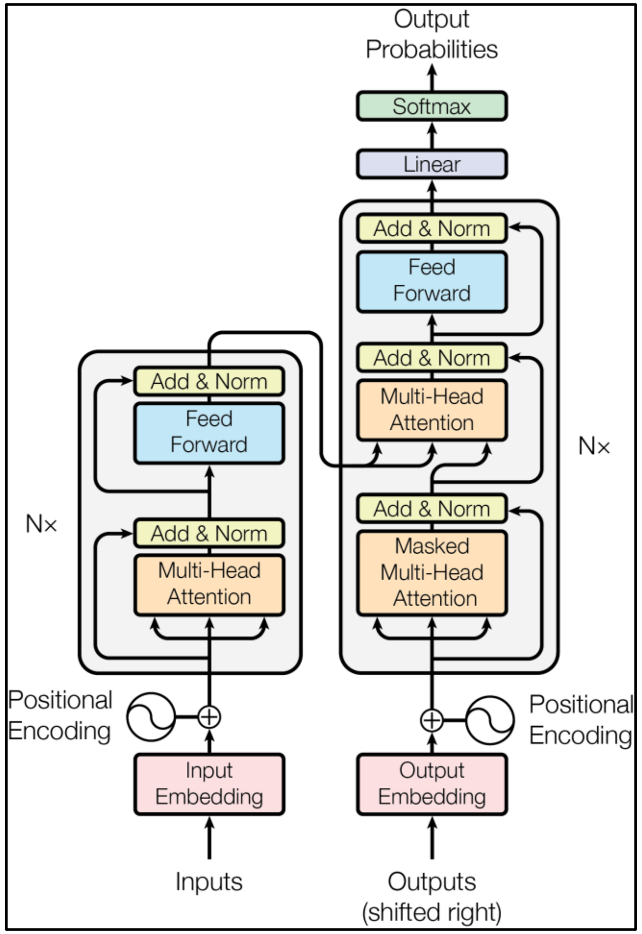
Training-procedure

First-stage





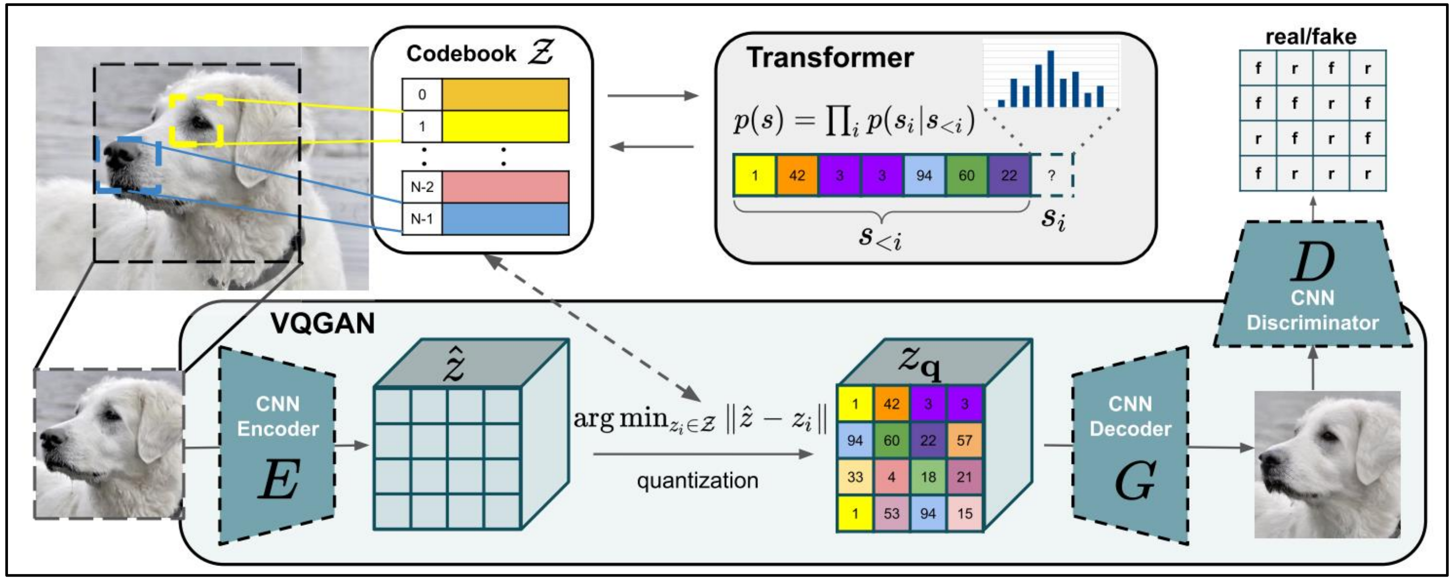
Second-stage



https://kjhov195.github.io/2020-03-09-generative_adversarial_network/

2. Methods

Architecture



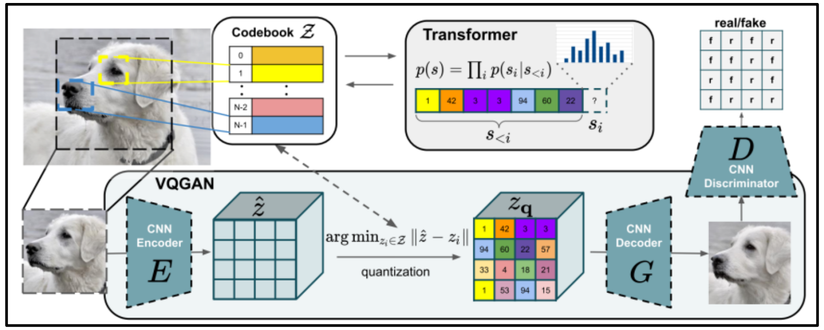
① CNN을 사용해 local feature가 풍부한 context를 Codebook에 encoding.

② Transformer을 사용해 global composition을 학습.

2. Methods

(1) Learning Codebook

- VQ-VAE와 유사하게 discrete한 latent embedding을 추출할 수 있도록 Codebook Z 학습.
- 이미지 생성후 Discriminator를 활용한 adversarial training 방식을 사용하는 것이 차이점.



$$\mathcal{L}_{\mathrm{VQ}}(E, G, \mathcal{Z}) = \boxed{\begin{aligned}\|x - \hat{x}\|^2 + \ &\|\mathrm{sg}[E(x)] - z_{\mathbf{q}}\|_2^2 \\ &+ \beta\|\mathrm{sg}[z_{\mathbf{q}}] - E(x)\|_2^2.\end{aligned}}$$

$$\mathcal{L}_{\mathrm{GAN}}(\{E, G, \mathcal{Z}\}, D) = [\log D(x) + \log(1 - D(\hat{x}))]$$

VQ-VAE Loss

$$L = \log p(x|z_q(x)) + \|sg[z_e(x)] - e\|_2^2 + \|z_e(x) - sg[e]\|_2^2$$

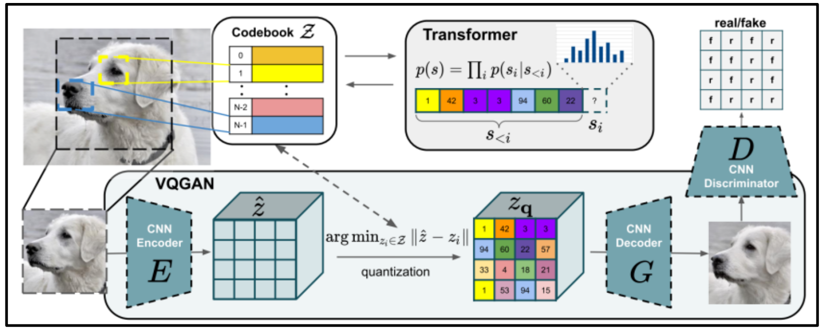
Reconstruction　　Codebook　　Commitment

① Reconstruction: Decoder, Encoder 훈련 -> Encoder는 gradient를 복사해서 훈련
② Codebook: Codebook의 벡터가 Encoder의 output과 가깝게 되도록 훈련
③ Commitment: Encoder가 Codebook의 벡터와 가까운 벡터를 내도록 훈련

2. Methods

(1) Learning Codebook

- VQ-VAE와 유사하게 discrete한 latent embedding을 추출할 수 있도록 Codebook Z 학습.
- 이미지 생성후 Discriminator를 활용한 adversarial training 방식을 사용하는 것이 차이점.



$$\mathcal{L}_{\mathrm{VQ}}(E,G,\mathcal{Z}) = \|x-\hat{x}\|^2 + \|\mathrm{sg}[E(x)] - z_{\mathbf{q}}\|_2^2 + \beta\|\mathrm{sg}[z_{\mathbf{q}}] - E(x)\|_2^2.$$

$$\mathcal{L}_{\mathrm{GAN}}(\{E,G,\mathcal{Z}\},D) = \boxed{[\log D(x) + \log(1-D(\hat{x}))]}$$

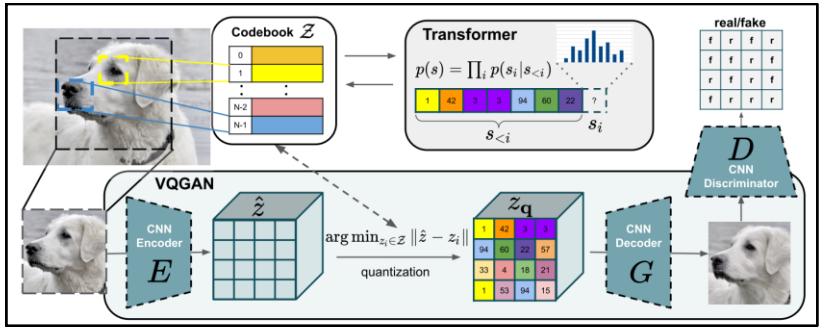
GAN Loss

Patchgan의 구조를 활용해 patch 단위로 real/fake를 구별하는 discriminator 사용.

2. Methods

(1) Learning Codebook

- VQ-VAE와 유사하게 discrete한 latent embedding을 추출할 수 있도록 Codebook Z 학습.
- 이미지 생성후 Discriminator를 활용한 adversarial training 방식을 사용하는 것이 차이점.



VQ-VAE Loss

GAN Loss

$$\mathcal{L}_{\mathrm{VQ}}(E, G, \mathcal{Z}) = \|x - \hat{x}\|^2 + \|\mathrm{sg}[E(x)] - z_{\mathbf{q}}\|_2^2 + \beta \|\mathrm{sg}[z_{\mathbf{q}}] - E(x)\|_2^2.$$

$$\mathcal{L}_{\mathrm{GAN}}(\{E, G, \mathcal{Z}\}, D) = [\log D(x) + \log(1 - D(\hat{x}))]$$

$$\mathcal{Q}^* = \arg\min_{E,G,\mathcal{Z}} \max_{D} \mathbb{E}_{x \sim p(x)} \Big[\mathcal{L}_{\mathrm{VQ}}(E, G, \mathcal{Z}) + \lambda \mathcal{L}_{\mathrm{GAN}}(\{E, G, \mathcal{Z}\}, D) \Big]$$

$$\lambda = \frac{\nabla_{G_L}[\mathcal{L}_{\mathrm{rec}}]}{\nabla_{G_L}[\mathcal{L}_{\mathrm{GAN}}] + \delta}$$

2. Methods

(2) Transformer

- Codebook Z를 학습 완료후 Transformer 를 학습
- GPT와 같은 Decoder-only 아키텍처 사용
- Transformer 훈련시에 index 일정 부분을 랜덤으로 만들고 그것을 올바르게 맞추는 방식으로 학습
- 샘플 생성 시 Computational Cost를 줄이기 위해 Sliding window 사용

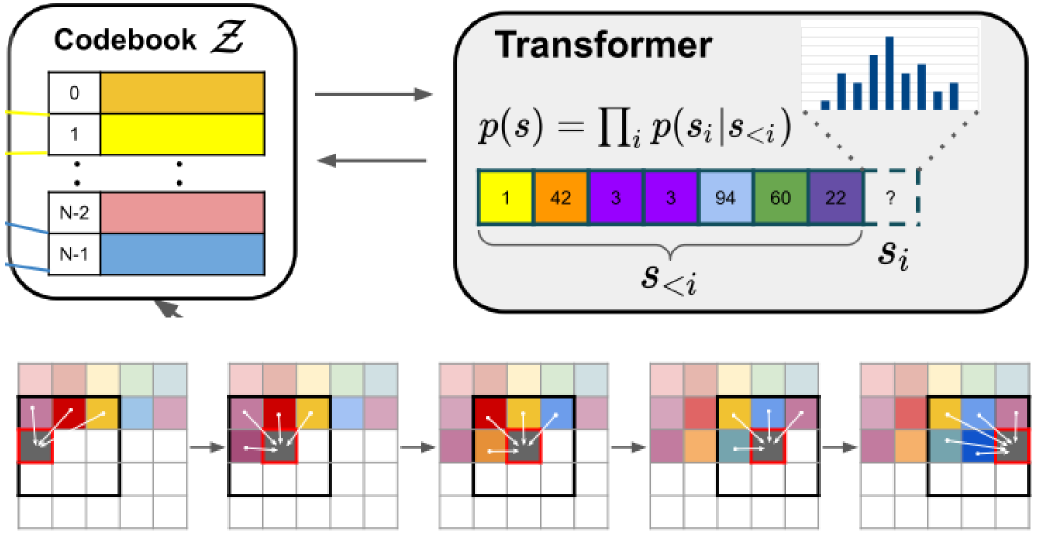
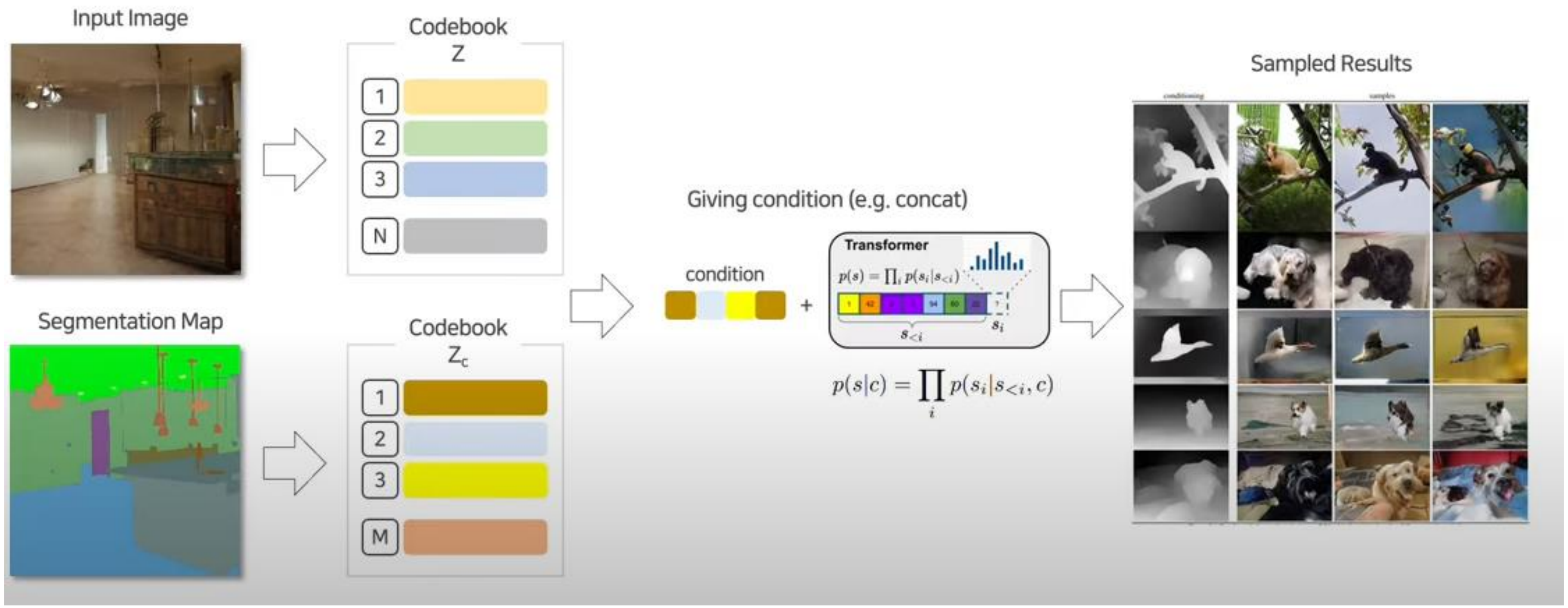


Figure 3. Sliding attention window.

2. Methods

(2) Transformer

Conditional generation



https://devocean.sk.com/blog/techBoardDetail.do?ID=164090

2. Methods

(2) Transformer

1) Transformer Training

```python
def forward(self, x, c):
    # one step to produce the logits
    _, z_indices = self.encode_to_z(x)
    _, c_indices = self.encode_to_c(c)

    if self.training and self.pkeep < 1.0:
        mask = torch.bernoulli(self.pkeep*torch.ones(z_indices.shape,
                                                     device=z_indices.device))
        mask = mask.round().to(dtype=torch.int64)
        r_indices = torch.randint_like(z_indices, self.transformer.config.vocab_size)
        a_indices = mask*z_indices+(1-mask)*r_indices
    else:
        a_indices = z_indices

    cz_indices = torch.cat((c_indices, a_indices), dim=1)

    # target includes all sequence elements (no need to handle first one
    # differently because we are conditioning)
    target = z_indices
    # make the prediction
    logits, _ = self.transformer(cz_indices[:, :-1])
    # cut off conditioning outputs - output i corresponds to p(z_i | z_{<i}, c)
    logits = logits[:, c_indices.shape[1]-1:]

    return logits, target
```

```python
def shared_step(self, batch, batch_idx):
    x, c = self.get_xc(batch)
    logits, target = self(x, c)
    loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), target.reshape(-1))
    return loss
```

https://github.com/CompVis/taming-transformers

2. Methods

(2) Transformer

2) Autoregressive Sampling

```
        for k in range(steps):
            callback(k)
            assert x.size(1) <= block_size # make sure model can see conditioning
            x_cond = x if x.size(1) <= block_size else x[:, -block_size:]  # crop context if needed
            logits, _ = self.transformer(x_cond)
            # pluck the logits at the final step and scale by temperature
            logits = logits[:, -1, :] / temperature
            # optionally crop probabilities to only the top k options
            if top_k is not None:
                logits = self.top_k_logits(logits, top_k)
            # apply softmax to convert to probabilities
            probs = F.softmax(logits, dim=-1)
            # sample from the distribution or take the most likely
            if sample:
                ix = torch.multinomial(probs, num_samples=1)
            else:
                _, ix = torch.topk(probs, k=1, dim=-1)
            # append to the sequence and continue
            x = torch.cat((x, ix), dim=1)
        # cut off conditioning
        x = x[:, c.shape[1]:]
    return x
```

https://github.com/CompVis/taming-transformers

3. Results

3. Results

vs SOTA

- Autoregressive generative model SOTA PixelSNAIL 보다 좋은 성능.
- Semantic image synthesis task와 VAE 모델간의 비교에서도 더 좋은 성능을 보여줌.

Data / # params	Transformer *P-SNAIL steps*	Transformer *P-SNAIL time*	PixelSNAIL *fixed time*
	Negative Log-Likelihood (NLL)		
RIN / 85M	**4.78**	4.84	4.96
LSUN-CT / 310M	**4.63**	4.69	4.89
IN / 310M	**4.78**	4.83	4.96
D-RIN / 180 M	**4.70**	4.78	4.88
S-FLCKR / 310 M	**4.49**	4.57	4.64

Table 1. Comparing Transformer and PixelSNAIL architectures across different datasets and model sizes. For all settings, transformers outperform the state-of-the-art model from the PixelCNN family, PixelSNAIL in terms of NLL. This holds both when comparing NLL at fixed times (PixelSNAIL trains roughly 2 times faster) and when trained for a fixed number of steps. See Sec. 4.1 for the abbreviations.

Dataset	**ours**	SPADE [53]	Pix2PixHD (+aug) [75]	CRN [9]
COCO-Stuff	22.4	22.6/23.9(*)	111.5 (54.2)	70.4
ADE20K	35.5	33.9/35.7(*)	81.8 (41.5)	73.3

Table 2. FID score comparison for semantic image synthesis (256×256 pixels). (*): Recalculated with our evaluation protocol based on [50] on the validation splits of each dataset.

Model	Codebook Size	dim $\mathcal{Z}$	FID/val	FID/train
VQVAE-2	64×64 & 32×32	512	n/a	~ 10
DALL-E [59]	32×32	8192	32.01	33.88
VQGAN	16×16	1024	7.94	10.54
VQGAN	16×16	16384	4.98	7.41
*VQGAN**	32×32	8192	1.49	3.24
VQGAN	64×64 & 32×32	512	1.45	2.78

Table 5. FID on ImageNet between reconstructed validation split and original validation (FID/val) and training (FID/train) splits. *trained with Gumbel-Softmax reparameterization as in [59, 29].

3. Results

vs SOTA

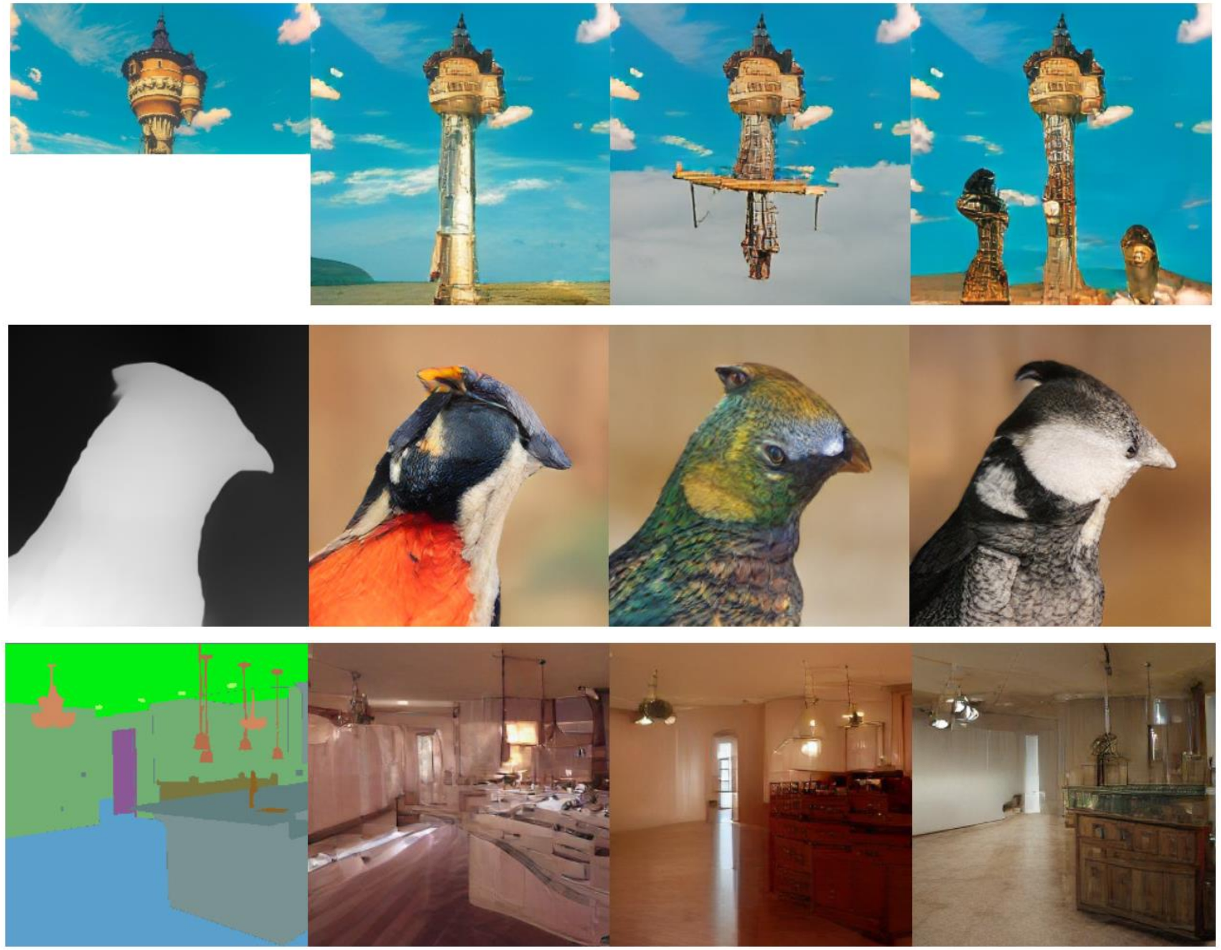
CelebA-HQ 256 × 256		FFHQ 256 × 256	
Method	FID ↓	Method	FID ↓
GLOW [37]	69.0	VDVAE ($t = 0.7$) [11]	38.8
NVAE [69]	40.3	VDVAE ($t = 1.0$)	33.5
PIONEER (B.) [23]	39.2 (25.3)	VDVAE ($t = 0.8$)	29.8
NCPVAE [1]	24.8	VDVAE ($t = 0.9$)	28.5
VAEBM [77]	20.4	*VQGAN*+P.SNAIL	21.9
Style ALAE [56]	19.2	BigGAN	12.4
DC-VAE [54]	15.8	**ours** (k=300)	9.6
ours (k=400)	10.2	U-Net GAN (+aug) [66]	10.9 (7.6)
PGGAN [31]	8.0	StyleGAN2 (+aug) [34]	3.8 (3.6)

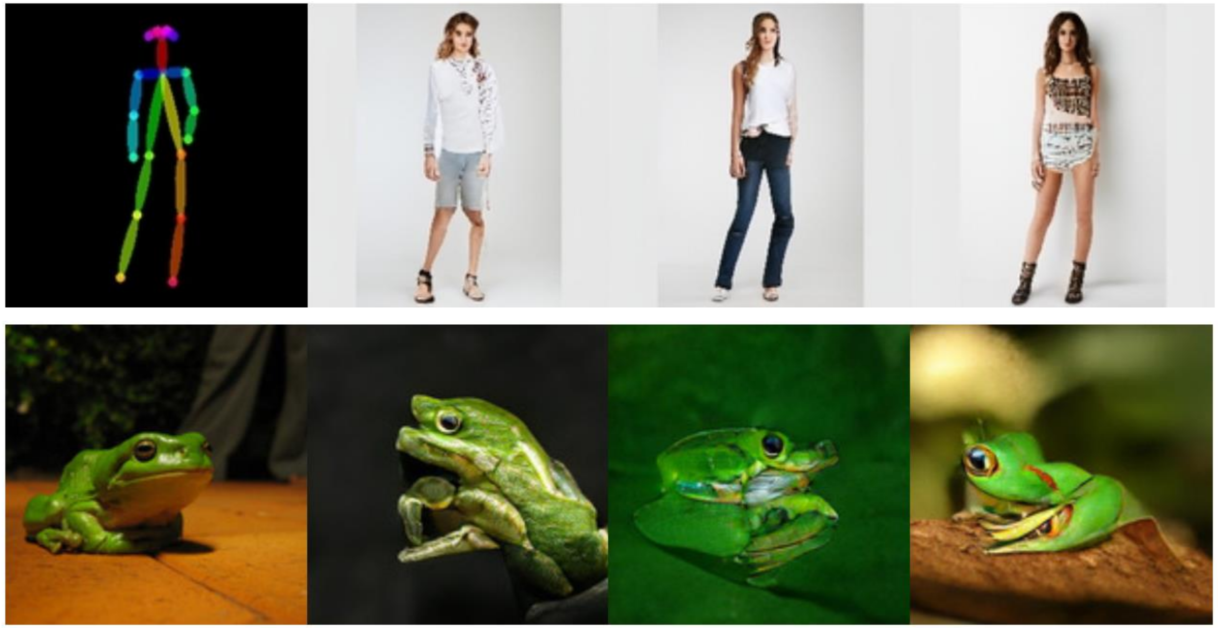
Table 3. FID score comparison for face image synthesis. CelebA-HQ results reproduced from [1, 54, 77, 24], FFHQ from [66, 32].

3. Results

Conditioning

- Completion, Depth-to-Image, Semantic-guide, Pose-guide, Class-condition





3. Results

High Resolution

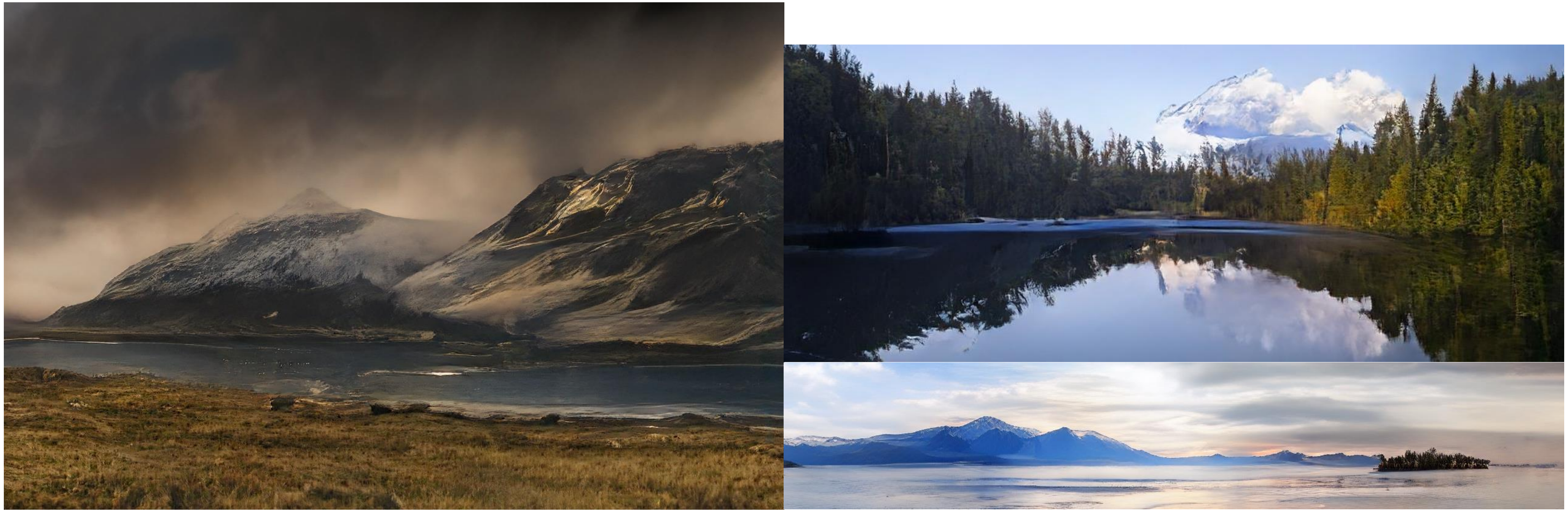


Figure 5. Samples generated from semantic layouts on S-FLCKR. Sizes from top-to-bottom: 1280×832, 1024×416 and 1280×240 pixels. Best viewed zoomed in. A larger visualization can be found in the appendix, see Fig 29.

3. Results

Conclusion

- 저해상도 이미지에 제한되어 있던 transformer의 문제를 해결.
- 처음으로 Transformer 기반 아키텍처로 고해상도 이미지를 생성.
- CNN과 Transformer의 구성요소를 상호 보완적으로 잘 활용함.

Thank you