|  |
| --- |
| **( 정보 (인공지능) ) 분야**  예)교육, 미디어, 예술, 생명과학 등 |

**2023 사고(思考)뭉치 프로젝트 탐구보고서**

**Synthetization of Raw Waveform Music via Latent Diffusion Models**

**( Latent Diffusion Model을 이용한 생파형(生波形)식 음악 생성 )**

**일산대진고등학교**

**2학년 13반 20번 이재현**

**Abstract**

이 소논문에서는 최근 고안되어 공간-시간적으로 연속적(spatiotemporally continuous)이고 정형적인 특성을 지닌 매체를 적은 컴퓨팅 자원만을 가지고 생성하는 데 효과적인 것으로 알려진 Latent Diffusion Model에 대해 알아보고, 이를 이용해 파형 그대로의 형식을 가진 비상징(非象徵, non-symbolic)적, 생파형(生波形)식 음악을 생성하는 것을 보인다.

이에는 생파형 음악 데이터의 수집의 방식부터 Latent Diffusion Model에 사용되는 Variational Autoencoder와 U-Net의 Pytorch 구현까지 코드와 설명이 병행되며 서술될 것이며, 궁극적으로는 이미지 생성의 분야에서만 적용되던 Latent Diffusion Model로 생파형 음악을 생성하는 모델을 구축한다.

**Agenda**

1. **Introduction**
   1. **Objective and Importance of Music Synthetization**
2. **Background and Related Work**
   1. **Transformer and their applications in Music Synthesis**
   2. **Diffusion Models and their applications in Music Synthesis**
3. **Methodology**
   1. **Creation of the Dataset**
      1. **Obtaining Musical Data**
      2. **Preprocessing Options**
   2. **Music Synthetization via Latent Diffusion Models**
      1. **Understanding the Diffusion Samplers**
         1. **Denoising Diffusion Probabilistic Models**
         2. **Denoising Diffusion Implicit Models**
      2. **Delving into the Latent Space**
      3. **Implementation the Latent Diffusion Model**
         1. **Overview of the Model & Synthetization Process**
            1. **Implementation of the Variational Auto-Encoder**

Convolutional Layers & Upsample & Downsample Layers

The Resnet Block and the Linear Attention

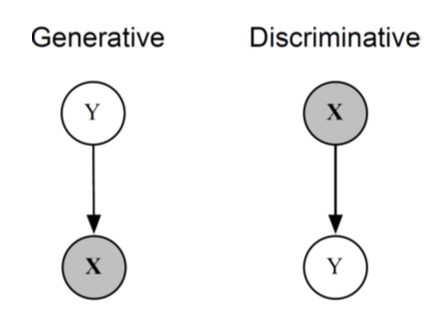
The Diagonal Gaussian Distribution

Structure of the Encoder & Decoder

**Metrics and Training**

* + - * 1. **Implementation of the Sampler U-Net**
      1. **Evaluation Metrics and Training**

1. **Comparative Analysis and Implications**
2. **Conclusion - Limitations, Future Works, and Applications**
3. **References**
4. Introduction

통계적 분류의 관점에서 전통적으로 인공지능 네트워크들을 구분하는 접근 방식 중에 하나는 특정 모델이 generative한지, discriminative 한지 나누는 이항 분류이다. Generative Model (생성 모델)들은 종속변수 y가 주어졌을 때 독립변수 X의 조건부 확률(conditional probability) 모델, 즉 로 정의되며, Discriminative Model (판별 모델)들은 독립변수 x가 주어졌을 때 종속변수 Y의 조건부 확률 모델 – 로 정의된다. 그 중Generative Model들을 이용하는 Generative Artificial Intelligence, 즉 생성형 인공지능은 텍스트와 이미지 등의 미디어를 생성해낼 수 있는, 인공지능의 학문에서 Neural Network의 기술적 패러다임이Deep-Learning Era를 넘어 Large-Scale Era로 진입함과 함께 극명히 대두되는 모델의 종류이다.

https://towardsdatascience.com/generativ

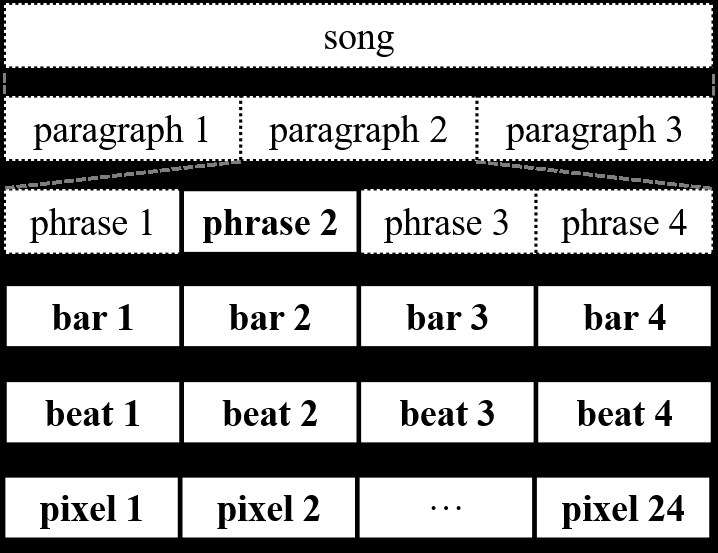
2020년대 초반 이전 생성 모델들의 고질적인 문제들을 해결한 딥 뉴럴 네트워크 구조들이 고안되고 발전되면서 사회에 모습을 드러낸 생성형 인공지능 서비스들 – 대형 언어 모델 (Large Language Model – abbreviated as LLM)들 ChatGPT, Bing Chat, Bard, LLaMA, Claude 등과 text-to-image 이미지 생성 프로그램들 Dall-E, Midjourney, Stable Diffusion, Adobe Firefly 등은 사회와 산업 전반에 이들의 유용성을 부각시켰다, 하나의 입력/출력 형식만 수용하고 생성할 수 있는 Unimodal 모델들과 두 개 이상의 형식을 생성할 수 있는 Multimodal 모델들은, 모두 산업 전반 – 일반 사무 직종은 물론이고 작화, 작곡 같은 창의력이 필수적이라고 생각되어지던 종목부터, 단백질 구조 예측이나 프로그래밍과 같은 기술 발전에 도움이 되는 작업까지 무궁무진한 활용 가능성을 보이고 있으며, 매 해 지속적으로 증가하는 새로운 모델들의 확장된 성능과 Multimodality의 추세에 따라 영화 따위의 영상 생성이나 게임 제작과 같은 인지적으로 더욱 복잡한 예술 형태의 생성까지 차후 십여 년 이내에 가능해질 것이라고 예측도 가능해진다.

그리고 이 생성형 인공지능 학문이 이러한 발전을 이루도록 가장 큰 영향을 끼친 모델들 중 두 가지는 Transformer Architecture [1]와 Diffusion Models [2]이라고 할 수 있겠다. 이 두 모델들은 각각Sequence-to-Sequence의 시계열 데이터 처리 및 생성과 크기가 크며 인지적으로 복잡한 단일 샘플을 생성하는 데 사용되며, 각 분야에서 가장 범용적으로 사용되던 모델들을 대체해, 새로운state-in-art 경지들을 성취해 내고 있다. 이 소논문에서는 두 모델들을 상대적으로 시도가 덜 이루어진 음악 생성의 미디어 생성 분야에 사용할 수 있도록 변형된 구현을 시도한다.

주목해야 할 점은, 동일한 데이터셋에 기반을 둔 각각의 모델들이, 각자의 특성을 살려 얼마나 효과적으로 feasible한 샘플들을 생성하는지, 고유한Melodic Structures (선율적 구조)를 생성하는지에 것이다. 생파형의 음악 생성은 근본적으로 복잡한 프로세스로써, 여러 음악적 요소와 이들 간의 복잡한 상호 작용을 포함한다. Harmonization (화음과 멜로디를 함께 사용하는 것), 리듬 생성, 그리고 Timbre Modulation과 같은 요소들은 음악의 생성을 다른 미디어의 생성보다 어렵게 만드는 데 한 몫을 하기 때문이다.

* 1. Objective and Importance of Music Synthetization

모든 샘플 형식들의 생성은 각 저마다의 특성과 어려움을 지니지만, 음악 생성은 몇 가지 측면에서 이미지와 같은 다른 샘플 형식들보다 생성이 어렵다고 할 수 있다. 첫째는 음악의 Temporal Dependency (시간적 의존성)에 있다. 음악은 본질적으로 sequential (순차적)하고 temporal (시간적)이다. 한 시점의 음표 (note) 또는 화음 (chord)의 정보는 그 이전과 이후에 오는 것에 깊이 의존하기에, 이는 음악의 모델링이 다른 매체의 생성보다 어려운 이유 중 하나가 된다. 또한 같은 측면을 다른 맥락에서 바라보자면, 음악 샘플이 가지는 시간적 의존성이 큰 간격을 두고 반복되는 주제와 패턴과 연관되는 것 역시 문제가 된다. Transformer Architecture가 같은 시계열 데이터인 자연어의 처리엔 뛰어난 성능을 보이는 것으로 증명되지만, 이 시간적 장기 의존성은 Token Context에 제한이 있는 Transformer Network가 처리하기에 어려울 수 있기 때문이다.

둘째로, 음악의 Hierarchical Structure (계층적 구조)의 측면이다. 음악은 다양한 주기로 반복되는 계층적 구조를 지니는데, 이는 pixel – beat – bar – phrase – paragraph의 구조로 정리될 수 있다. [1] 오디오와 음악의 맥락에서 pixel은 일반적으로 이미지의 픽셀이 해당 도메인에서 가장 작은 정보 단위인 것과 마찬가지로 오디오 파일의 단일 샘플 포인트를 의미하며, 의 샘플 레이트 (sample rate)를 가지는 오디오 파일은 1초당 개의 픽셀을 가지기에, 음악에서 단지 beat 하나를 인식함에도 몇 백 ~ 몇 천 개의 pixel들을 처리해야 함을 알 수 있다. 반복의 수가 적은 것도 문제가 되는데, bar들이 모여 만들어진 phrase는 고작 번 정도 반복되어 하나의 paragraph를 이루며, 하나의 음악은 Intro, Verse, Chorus, Bridge, Outro 정도의 적은 양의 paragraph으로만 이루어져 있기에 이 적은 양의 반복으로 음악 전체를 이해하는 것도 시계열 처리 모델로 처리하기 힘든 요소이다.

https://www.researchgate.net/figure/Hier

셋째 문제는 파형의 형식을 띄는 오디오 형식 자체의 처리에 대한 난해함이다. 음악은 Polynomial Characteristics (다항 특성)을 지니는데, 음악은 하나의 악기만으로 구성되지 않고 여러 악기의 사용으로 만들어진다. 이때 두 개 이상의 악기가 사용되면 Fourier’s Transform과 같은 알고리즘적 방법을 사용하더라도 주파수의 패턴이 완전하게 분리되지 않을 뿐더러 주파수를 복잡하게 변형시킨다. 게다가 파형이라는 특성 상 오디오 데이터는 압축이 굉장히 난해하다. 높은 sample rate로 저장된 오디오 파일을 낮은 sample rate로 전처리하면 높은 주파수의 파형들이 모두 손실되어 뭉개지는 특성이 있기 때문이다.

이러한 어려움 때문에 음악을 처리하고 생성하는 과제에서는 낮은 수준의 정보들 (low-level features)로부터 고차원적인 잠재 정보 (high level features)를 파악하는 모델 구조의 능력이 많이 요구된다. 이렇게 계속해서 고수준의 이해가 가능하도록 만드는 것이 결국 인공지능 학문이 지속적으로 추구하는 궁극적인 장기 목표이기도 한데, 이는 모델의 크기와 데이터의 양을 증폭시키는 것보다도 더 세분화되고 최적화된 모델 구조들이 – 마치 뇌의 특정 기능에 최적화된 중추들처럼, 더 좋은 결과를 내놓을 수 있기 때문이며 – 이러한 구조들을 고안하는 것이 인공지능 분야가 추구하는 것이기도 하기 때문이다.

1. Background and Related Work

Transformer Model은 본래 Natural Language Processing (NLP, 자연어 처리)를 위해 개발된 모델이지만, 순차 데이터를 처리하는 데에 기존에 널리 사용되던 RNN의 변종인 Long-Short Term Memory (LSTM, 장단기 메모리) [3]를 꺾고 현존하는 방식들 중 가장 최적화된 성능을 보이기에 음악 생성에도 다양한 방식으로 사용되었다.

Cheng-Zhi Anna Huang et al.의 Music Transformer [4]는 Peter Shaw et al. [5]에 의해 제안된 상대적 attention 메커니즘을 확장

Prateek Verma et al.의 A Generative Model for Raw Audio Using Transformer Architectures [6]

Diffusion Model은 확률론적 score-based 생성 모델의 일종으로서, 기존에 이미지 생성에서 가장 pervasive하게 사용되던 Generative Adversarial Network (GAN, 적대적 생성 신경망) [7]을 훈련의 안전성, 생성된 sample의 질적 우월성 [8], 자연어 conditioning의 가능함 등의 이유로 빠른 속도로 대체하고 있는 모델이다. Diffusion 모델은 음악 합성 분야에서의 응용을 찾아왔지만, 연속 데이터와 이산 데이터 간의 본질적인 차이로 인해 도전이 계속되고 있다.

Mittal et al.의 Symbolic Music Generation with Diffusion Models [9]은 Variational Autoencoder (VAE, 변이형 오토인코더)로 압축시킨 연속적 잠재 표상 (continuous latents) 위에서 Diffusion Model 가족의 가장 오래된 개체 중 하나인 Denoising Diffusion Probabilistic Model (DDPM)을 훈련시켜 이산 symbolic music을 생성한다. 주요 결과로는 DDPM을 사용한 이산 symbolic music의 고품질 무조건(無條件, unconditional) 샘플링, 연속 잠재자의 계층적 모델링에서 강력한 자기 회귀 기준선의 우수한 성능, 창의적인 응용을 위한 post-havoc 조건부 주입 등이 있다. 이는 Diffusion Model을 VAE와 접목시킨 Latent Diffusion Model이 음악 생성에서 어떻게 사용될 수 있는가를 잘 보여주지만, 가장 오래된 Diffusion Sampler 중 하나인 DDPM을 사용하였으며 symbolic music을 생성하는 것을 목표로 하였는 점에서 개선의 여지가 있다.

1. Methodology
   1. Creation of the Dataset

Symbolic Music data는 MIDI (Musical Instrument Digital Interface) 또는 MusicXML과 같은 형식을 사용하여 음악의 기호적인 부분들(음표, 쉼표, 다이내믹 등)을 인코딩하는 방법을 의미한다. 이는 실제 오디오 파일이 아닌, 음악의 구조와 특징을 나타내는 정보를 저장하는 방식이다. 이러한 Symbolic Music data의 예로는 Lakh MIDI Dataset, Classical Archives, Mutopia Project 등이 있는데, 이들은 모두 대규모로 제작되어 일반 대중에게 공개된 음악 데이터베이스들이다.

하지만 이 프로젝트에서는 non-symbolic, 파형 형식의 오디오를 필요로 했기 때문에, 특정한 아티스트의 명칭이 주어졌을 때, 그 아티스트의 모든 공개된 음악 작품들을 YouTube에서 다운로드하여 전처리하는 과정을 거치는 pipeline을 구축하였다. 이 pipeline의 구축은 생파형 오디오 데이터를 효과적으로 수집하고 관리하기 위해 필수적인 단계로, 음악의 다양한 특성과 형식을 포괄할 수 있는 강력한 데이터셋을 생성하는 데 도움을 주었다.

3.1.1 Obtaining Musical Data

음악 데이터를 획득하기 위해서는 먼저 아티스트의 이름을 입력으로 받아, 대규모 음악 metadata 데이터베이스인 MusicBrainz API를 통해 해당 아티스트의 상세한 정보와 그들의 모든 음악 작품들을 조회한다. 이렇게 얻어진 정보를 바탕으로, YouTube에서 해당 아티스트의 곡들을 순차적으로 다운로드하는 과정을 거쳐 데이터를 수집한다. 이러한 방식은 저작권과 관련된 여러 문제를 야기할 수 있는 다소 민감한 방법일 수 있기에, 본 프로젝트에서는 연구 및 교육 목적으로만 사용되었다.

음악 데이터의 획득과 관련된 프로그래밍 작업은 Python 언어를 사용하여 진행되었다. 이 과정에서 musicbrainzngs, youtubesearchpython, pytube와 같은 다양한 외부 라이브러리들을 활용하여, 데이터 수집 및 전처리 과정을 효율적이고 체계적으로 수행할 수 있도록 하였다. 밑은 음악 데이터 수집을 위해 작성한 프로그램의 전문이다.

|  |
| --- |
| *import* os  *import* re  *import* pytube  *import* musicbrainzngs  *from* youtubesearchpython *import* VideosSearch  musicbrainzngs.set\_useragent("MusicBrainzAPI", "0.1", *contact*="Somniaquia@gmail.com")  *def* download\_video(*video\_url*, *save\_location*, *audio\_only*=True):      yt = pytube.YouTube(video\_url)  *# video\_title = re.sub(r'[^\x00-\x7F]', '\_', yt.title)*      video\_title = re.sub(*r*'[\x00-\x1F\x7F-\x9F\/:?\*<>|]', '\_', yt.title)      video\_title = re.sub(*r*'\_+', '\_', video\_title)      path = *f*"data/raw/{save\_location}"      if not os.path.exists(path):          os.makedirs(path)      audio\_stream = yt.streams.filter(*only\_audio*=audio\_only, *file\_extension*='mp4').first()      audio\_stream.download(*filename*=*f*"{path}/{video\_title}.mp4")      return video\_title  *def* search\_artist\_songs(*artist\_name*):      try:          artist\_search = musicbrainzngs.search\_artists(artist\_name)          if "artist-list" not in artist\_search or not artist\_search["artist-list"]:  *print*("Artist not found.")              return          artist\_name = artist\_search["artist-list"][0]["name"]          artist\_id = artist\_search["artist-list"][0]["id"]          append\_link(artist\_name)          releases = musicbrainzngs.browse\_releases(*artist*=artist\_id)          for release in releases["release-list"]:              release\_id = release["id"]              release\_info = musicbrainzngs.get\_release\_by\_id(  release\_id, *includes*=["recordings"])['release']              if "medium-count" not in release\_info or release\_info["medium-count"] == 0:  *print*(release\_info)  *print*("No medium found for this release.\n")                  continue              medium\_list = release\_info["medium-list"][0]  *# Assuming one medium per release*              if "track-list" not in medium\_list:  *print*("No tracks found for this release.")                  continue  *# Iterate through the tracks on the medium*              for track in medium\_list["track-list"]:                  track\_title = track["recording"]["title"]                  youtube\_search = VideosSearch(*f*"{artist\_name} {track\_title} original")                  results = youtube\_search.result()['result']                  if results:                      youtube\_link = results[0]["link"]  *print*(*f*"Track found: {track\_title} - {artist\_name}, Link: {youtube\_link}")                      try:                          if not is\_link\_already\_saved(youtube\_link):                              download\_video(youtube\_link, artist\_name)  *print*("Download succeed! \n")                              append\_link(youtube\_link)                          else:  *print*("Skipping song as it is already downloaded \n")                      except Exception as e:  *print*(e)                  else:  *print*(*f*"Track: {track\_title} by {artist\_name}")  *print*("YouTube Link not found.")      except musicbrainzngs.WebServiceError as exc:  *print*(*f*"MusicBrainz API error: {exc}")  *def* is\_link\_already\_saved(*link*):      if os.path.isfile("data/links.txt"):          with *open*("data/links.txt", "r") as file:              return link in file.read()      return False  *def* append\_link(*link*):      with *open*("data/links.txt", "a") as file:          file.write(link + "\n")  if \_\_name\_\_ == "\_\_main\_\_":      while True:          artist\_names = *input*("Enter the artists' names to download albums of (seperated with commas): ")          for name in artist\_names.split(","):              search\_artist\_songs(name) |

작성된 프로그램을 이용하여 크롤링할 음악 아티스트들은 GPT-4를 이용하여 국적에 의한 분류에 따른 미국의 아티스트 93단위, 한국과 일본의 아티스트 각각 74단위와 55단위씩, 음악 장르에 따라 일렉트로닉 장르 아티스트 190단위와 역사적 피아노 좆 작곡가 183단위가 선정되었다. 수집의 결과로 총 10078 단위의 생파형 음악을 수집하였으며, MP4 형식의 AAC encoding을 사용한 결과 240GB 분량의 데이터셋을 형성할 수 있었다.

Preprocessing Options

파형의 형태의 음악을 저장하려면 오디오 인코딩 방식을 사용해야 한다. 오디오 인코딩 방식에는 몇 가지 선택지들이 있는데, 이 선택지들은 인공지능 모델로 불러올 때의 계산 효율성, 저장공간 효율성 등의 척도에서 차이를 낸다.

오디오 인코딩은 디지털 음향학(音響學)의 원리를 바탕으로 오디오를 저장하는 방식을 채택한다. 이 방식은 시간 domain에서의 연속적인 음파를 샘플링하여 이산적인 디지털 데이터로 변환시키는 과정을 통해 오디오를 재현한다. 샘플링 과정에서는 표준화된 시간 간격으로 음파의 진폭을 측정하고, 이를 16비트나 24비트와 같은 디지털 값으로 표현한다. 이렇게 변환된 디지털 데이터는 인코딩된 파일 형식에 저장되며, 재생 시에는 다시 연속적인 음파로 변환되어 음악으로 들을 수 있다. 이 과정에서 Wav 파일은 원본 음파의 특성을 최대한 보존하려 노력한다. 그러나 샘플링 속도와 비트 깊이에 따라 데이터의 크기와 음질이 결정되어, 높은 음질을 원할 경우 상당한 저장 공간을 요구하게 된다 – 즉, 파일의 크기와 음질 사이의 타협점을 찾아야 한다.

YouTube에서 수집한 음악 파일들은 MP4 컨테이너 속 AAC (Advanced Audio Coding) 인코딩으로 저장되어 있다. AAC 인코딩은 MP3 (MPEG-1 Audio Layer 3)의 개선형으로 고안되었는데, 인코딩 과정에서 사람의 청각 시스템의 특성을 고려하여, 인간이 잘 듣지 못하는 음역대의 정보를 제거하는 modified discrete cosine transform (MDCT)라는 기법을 사용한다는 특징이 있다. 이는 MP3와 같은 sampling rate에서 더 좋은 음질을 달성하게 해주며, 같은 음악을 저장할 때에 Wav 인코딩보다 10배 정도 크기가 작아 오디오를 저장하는 데 이상적인 인코딩이 된다.

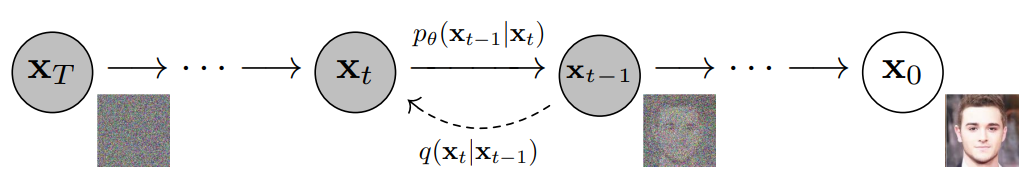
그럼에도 뉴럴 네트워크의 input으로 사용될 때에는 압축된 형식을 그대로 tensor로 바꿔 사용할 수는 없다 – 음악적 형식과 관계없는 복잡도가 늘어나면서 모델이 음악의 형식을 배우는 데에 방해를 하기 때문이다. 이에 high-level 기계학습 라이브러리인 Pytorch의 TorchAudio는 MP4를 비롯해 MP3, WAV, AAC, OGG, FLAC, AVR, CDDA, CVS/VMS, AIFF, AU, AMR, MP2, AC3, AVI, WMV, MPEG, IRCAM format들을 사용할 때 이 format들을 무손실 오디오의 형식으로 복원해준다.

따라서 전처리에서 유일하게 신경 쓸 요소는 tensor의 크기를 좌우할 오디오의 sampling rate와 bit depth뿐이다. 가정용 GPU에서도 사용 가능하게 설계된, 이 프로젝트에서 사용하는 Latent Diffusion Model을 사용할 영감을 준 Stable Diffusion의 경우, 하나의 이미지 tensor는 일반적으로 512x512x3의 크기 (가로, 세로의 길이가 각각 512, color channel의 수가 R, G, B로 3), 786,432개의 scalar 값으로 이루어져 있으며, 각 scalar 값은 32-bit float 값을 가진다. 즉, 하나의 이미지 tensor는 3,145,728byte의 크기를 가진다. 이는 44.1kHz의 sampling rate와 16-bit의 bit depth를 가진 5분 길이의 오디오 tensor의 크기 - 26,460,000byte보다 대략 10배 정도 작은 값이다.

이에 pydub을 이용해 모든 audio를 16kHz의 sampling rate와 8-bit의 bit depth – 앞서 언급된 이미지 텐서와 비슷한 크기인 4,800,000byte를 가지게 전처리하는 스크립트가 작성되었으며, 이의 전문은 아래와 같다.

|  |
| --- |
| *import* os  *from* pydub *import* AudioSegment  *from* pathlib *import* Path  *def* convert\_and\_resample(*root\_folder*, *target\_sample\_rate*=8000, *bit\_depth*=16):      for subdir, dirs, files in os.walk(root\_folder):          for file in files:              src\_filepath = os.path.join(subdir, file)              if src\_filepath.endswith(".mp4"):                  dst\_filepath = src\_filepath.replace('raw', str(target\_sample\_rate).replace('000', 'kHz') + '\_'  + str(bit\_depth) + 'bit')                  Path(os.path.dirname(dst\_filepath)).mkdir(*parents*=True, *exist\_ok*=True)                  if os.path.isfile(dst\_filepath):  *print*("File already exists: " + dst\_filepath)                      continue                  try:                      audio = AudioSegment.from\_file(src\_filepath, *format*="mp4")                      audio = audio.set\_frame\_rate(target\_sample\_rate)                      audio = audio.set\_sample\_width(bit\_depth // 8)                      audio.export(dst\_filepath.replace('mp4', 'wav'), *format*="wav")  *print*("Saved to " + dst\_filepath)                  except Exception as e:  *print*("Failed to process {}: {}".format(src\_filepath, str(e)))  convert\_and\_resample('data\\raw', 8000, 8) |

* 1. Music Synthetization via Latent Diffusion Models



인공지능에서 Diffusion Model은 여러 step에 걸쳐 sample에 Gaussian noise를 첨가해 붕괴시키는 forward step과, noise가 존재하는 sample과 해당 sample의 step의 번호가 주어졌을 때 이전 step의 sample에서 추가된 noise를 추측하는 backward step으로 이루어진 생성 모델이다. 그러니까, 고정된 Markov Chain을 사용하여 sample이 아닌 첨가된 noise를 잠재공간에 매핑하는 잠재(潛在)변수(變數) 모형이다.

Forward step 에서는 sample 가 주어졌을 때 같은 크기를 가진 posterior sample 를 고정된 Gaussian Distribution를 이용해 계산적으로 구한다. 이때 timestep *T*는 sample 의 분포가 거의 완전한 isotropic Gaussian (대칭적 정규분포)를 이루도록 설정된다. 이 뉴럴 네트워크 모델이 사용되는 목적은 backward step - sample 가 주어졌을 때 를 추정하는 를 구현하기 위해서이다.

3.2.1. Understanding the Diffusion Samplers

위에서 상술한 Forward Step과 Backward Step을 구현

* + - 1. Denoising Diffusion Probabilistic Models
      2. Denoising Diffusion Implicit Models
    1. Delving into the Latent Space
    2. Implementation the Latent Diffusion Model
    3. Overview of the Model & Synthetization Process
       1. Implementation of the Variational Auto-Encoder

Compvis/StableDiffusion

3.2.2.1 Convolutional Layers & Upsample & Downsample Layers

|  |
| --- |
| *class* Upsample(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *in\_channels*, *with\_conv*):          super().\_\_init\_\_()          self.with\_conv = with\_conv          if self.with\_conv:              self.conv = nn.Conv1d(in\_channels, in\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)  *def* forward(*self*, *x*):          x = F.interpolate(x, *scale\_factor*=2.0, *mode*='linear')  *# Changed to 'linear' for smoother results*          if self.with\_conv:              x = self.conv(x)          return x    *class* Downsample(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *in\_channels*, *with\_conv*):          super().\_\_init\_\_()          self.with\_conv = with\_conv          if self.with\_conv:              self.conv = nn.Conv1d(in\_channels, in\_channels, *kernel\_size*=4, *stride*=2, *padding*=1)  *def* forward(*self*, *x*):          if self.with\_conv:              x = self.conv(x)          else:              x = F.avg\_pool1d(x, *kernel\_size*=2, *stride*=2)          return x |

The Resnet Block and the Linear Attention

ResNet의 설명

Linear Attention

|  |
| --- |
| *class* ResnetBlock(*nn*.*Module*):  *def* \_\_init\_\_(*self*, \*, *in\_channels*, *out\_channels*=None, *conv\_shortcut*=False, *dropout*, *t\_emb\_channels*=512):          super().\_\_init\_\_()          self.in\_channels = in\_channels          self.out\_channels = in\_channels if out\_channels is None else out\_channels          out\_channels = self.out\_channels          self.use\_conv\_shortcut = conv\_shortcut          self.norm1 = nn.LayerNorm(in\_channels)          self.conv1 = nn.Conv1d(in\_channels, out\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)          if t\_emb\_channels > 0:              self.t\_emb\_proj = nn.Linear(t\_emb\_channels, out\_channels)            self.norm2 = nn.LayerNorm(out\_channels)          self.dropout = nn.Dropout(dropout)          self.conv2 = nn.Conv1d(out\_channels, out\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)            if self.in\_channels != self.out\_channels:              if self.use\_conv\_shortcut:                  self.conv\_shortcut = nn.Conv1d(in\_channels, out\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)              else:  *# The NiN (Network-in-Network) shortcut - https://arxiv.org/abs/1312.4400*                  self.nin\_shortcut = nn.Conv1d(in\_channels, out\_channels, *kernel\_size*=1, *stride*=1, *padding*=0)  *def* forward(*self*, *x*, *t\_emb*):          h = x          h = self.norm1(h.transpose(1, 2)).transpose(1, 2)          h = nn.SiLU()(h)          h = self.conv1(h)  *# Project t\_emb to the same dimensionality as the output of the first convolutional layer, add to the result*          if t\_emb is not None:  *# The original network processed 4D image tensors (b, h, w, c) but in case of 2D audio tensors (b, s) dimensionality augmentation is unnecessary*  *# h = h + self.t\_emb\_proj(nn.SiLU(t\_emb))[:,:,None,None]*              h = h + self.t\_emb\_proj(nn.SiLU()(t\_emb))[:, :, None]          h = self.norm2(h.transpose(1, 2)).transpose(1, 2)          h = nn.SiLU()(h)          h = self.dropout(h)          h = self.conv2(h)          if self.in\_channels != self.out\_channels:              if self.use\_conv\_shortcut:                  x = self.conv\_shortcut(x)              else:                  x = self.nin\_shortcut(x)  *# Implement the defining skip connection of a ResnetBlock*          return x+h |

|  |
| --- |
| *class* LinearAttention(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *dim*, *heads*=4, *dim\_head*=32):          super().\_\_init\_\_()          self.heads = heads          hidden\_dim = dim\_head \* heads          self.to\_qkv = nn.Conv1d(dim, hidden\_dim \* 3, 1, *bias* = False)          self.to\_out = nn.Conv1d(hidden\_dim, dim, 1)  *def* forward(*self*, *x*):          b, c, l = x.shape          qkv = self.to\_qkv(x)          q, k, v = rearrange(qkv, 'b (qkv heads c) l -> qkv b heads c l', *heads*=self.heads, *qkv*=3)          k = k.softmax(*dim*=-1)          context = torch.einsum('bhdn,bhen->bhde', k, v)          out = torch.einsum('bhde,bhdn->bhen', context, q)          out = rearrange(out, 'b heads c l -> b (heads c) l', *heads*=self.heads, *l*=l)          return self.to\_out(out) |

The Diagonal Gaussian Distribution

VAE의 특징

잠재 벡터로서의 Diagonal Gaussian Distribution의 역할

|  |
| --- |
| *# Gaussian Distribution with a diagonal covariance matrix*  *class* DiagonalGaussianDistribution(object):  *def* \_\_init\_\_(*self*, *parameters*, *deterministic*=False):  *# A tensor whose first half along the first dimension represents means, and the second half represents log variances of a Gaussian distribution.*          self.parameters = parameters  *# Means and Log Variances of the Gaussian distribution*          self.mean, self.logvar = torch.chunk(parameters, *chunks*=2, *dim*=1)  *# Clamp logvars in order to avoid numerical instability*          self.logvar = torch.clamp(self.logvar, -30.0, 20.0)  *# A boolean flag to determine whether the distribution should ignore the stochasticity and just use the mean during sampling.*          self.deterministic = deterministic  *# Compute standard deviation*          self.std = torch.exp(0.5 \* self.logvar)  *# Compute variance*          self.var = torch.exp(self.logvar)  *# If deterministic == True, variance and std are set to zero tensors*  *# This results in subsequent sampling yielding the means of the distributions*          if self.deterministic:              self.var = self.std = torch.zeros\_like(self.mean).to(*device*=self.parameters.device)  *# Sample from the random distribution - or return the mean if deterministic == True*  *def* sample(*self*):          x = self.mean + self.std \* torch.randn(self.mean.shape).to(*device*=self.parameters.device)          return x    *# Calculates KL Divergence from an another Gaussian distribution*  *# If the other distribution is not provided, this calculates KL Divergence with a normal distribution N(1, 0)*  *def* kl(*self*, *other*=None):          if self.deterministic:              return torch.tensor([0.], *device*=self.parameters.device)          else:              if other is None:                  return 0.5 \* torch.sum(torch.pow(self.mean, 2) + self.var - 1.0 - self.logvar, *dim*=[1])              else:                  return 0.5 \* torch.sum(torch.pow(self.mean - other.mean, 2) / other.var + self.var / other.var - 1.0 - self.logvar + other.logvar, *dim*=[1])    *# Calculate the negative log-likelihood of a 'sample' under the distribution.*  *# Common for loss functions of generative models*  *def* nll(*self*, *sample*, *dims*=[1]):          if self.deterministic:              return torch.Tensor([0.])          logtwopi = np.log(2.0 \* np.pi)          return 0.5 \* torch.sum(              logtwopi + self.logvar + torch.pow(sample - self.mean, 2) / self.var, *dim*=dims)    *# Returns the mode (most probable value) of the distribution, which is the mean in this case*  *def* mode(*self*):          return self.mean |

Structure of the Encoder & Decoder

Encoder Decoder

|  |
| --- |
| *class* Encoder(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *ch*=64, *ch\_mult*=(1, 2, 4, 8), *num\_res\_blocks*=1, *attn\_resolutions*=[], *dropout*=0.0, *resamp\_with\_conv*=True, *in\_channels*=1, *resolution*=4800000, *z\_channels*=512):          super().\_\_init\_\_()          self.ch = ch          self.num\_resolutions = *len*(ch\_mult)          self.num\_res\_blocks = num\_res\_blocks          self.attn\_resolutions = attn\_resolutions          self.resolution = resolution          self.in\_channels = in\_channels          self.z\_channels = z\_channels          self.t\_emb\_ch = 0  *# Downsampling layers*          self.conv\_in = nn.Conv1d(in\_channels, self.ch, *kernel\_size*=3, *stride*=1, *padding*=1)          curr\_res = self.resolution          in\_ch\_mult = (1,) + tuple(ch\_mult)          self.in\_ch\_mult = in\_ch\_mult          self.down = nn.ModuleList()          for i\_level in *range*(self.num\_resolutions):              block = nn.ModuleList()              attn = nn.ModuleList()              block\_in = ch \* in\_ch\_mult[i\_level]              block\_out = ch \* ch\_mult[i\_level]              for i\_block in *range*(self.num\_res\_blocks):                  block.append(ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_out, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout))                  block\_in = block\_out                  if curr\_res in attn\_resolutions:                      attn.append(LinearAttention(*dim*=block\_in))              down = nn.Module()              down.block = block              down.attn = attn              if i\_level != self.num\_resolutions-1:                  down.downsample = Downsample(block\_in, resamp\_with\_conv)                  curr\_res = curr\_res // 2              self.down.append(down)  *# Middle Layers*          self.mid = nn.Module()          self.mid.block\_1 = ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_in, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout)  *def* forward(*self*, *x*, *t\_emb*=None):  *# Initial convolution*          h = self.conv\_in(x)  *# Downsampling layers*          for i\_level in *range*(self.num\_resolutions):              for i\_block in *range*(self.num\_res\_blocks):                  h = self.down[i\_level].block[i\_block](h, t\_emb)                  if *len*(self.down[i\_level].attn) > 0:                      h = self.down[i\_level].attn[i\_block](h)              if i\_level != self.num\_resolutions - 1:                  h = self.down[i\_level].downsample(h)  *# Middle layers*          h = self.mid.block\_1(h, t\_emb)          return h |

|  |
| --- |
| *class* Decoder(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *ch*=64, *out\_ch*=1, *ch\_mult*=(8, 4, 2, 1), *num\_res\_blocks*=1, *attn\_resolutions*=[], *dropout*=0.0, *resamp\_with\_conv*=True, *in\_channels*=1, *resolution*=4800000, *z\_channels*=512):          super().\_\_init\_\_()          self.ch = ch          self.num\_resolutions = *len*(ch\_mult)          self.num\_res\_blocks = num\_res\_blocks          self.resolution = resolution          self.in\_channels = in\_channels          self.z\_channels = z\_channels          self.out\_ch = out\_ch          self.t\_emb\_ch = 0  *# Compute in\_ch\_mult, block\_in, and curr\_res at lowest res*          in\_ch\_mult = (1,) + tuple(ch\_mult)          block\_in = ch \* ch\_mult[self.num\_resolutions - 1]          curr\_res = resolution // 2\*\*(self.num\_resolutions - 1)          self.z\_shape = (1, z\_channels)  *print*(*f*"Working with z of shape {self.z\_shape} dimensions.")          self.conv\_in = nn.Conv1d(z\_channels, block\_in, *kernel\_size*=3, *stride*=1, *padding*=1)  *# middle*          self.mid = nn.Module()          self.mid.block\_1 = ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_in, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout)          self.mid.attn\_1 = LinearAttention(*dim*=block\_in)          self.mid.block\_2 = ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_in, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout)  *# upsampling*          self.up = nn.ModuleList()          for i\_level in *reversed*(*range*(self.num\_resolutions)):              block = nn.ModuleList()              attn = nn.ModuleList()              block\_out = ch \* ch\_mult[i\_level]              for i\_block in *range*(self.num\_res\_blocks):                  block.append(ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_out, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout))                  block\_in = block\_out                  if curr\_res in attn\_resolutions:                      attn.append(LinearAttention(*dim*=block\_in))              up = nn.Module()              up.block = block              up.attn = attn              if i\_level != 0:                  up.upsample = Upsample(block\_in, resamp\_with\_conv)                  curr\_res = curr\_res \* 2              self.up.insert(0, up)          self.norm\_out = nn.LayerNorm(block\_in)          self.conv\_out = nn.Conv1d(block\_in, out\_ch, *kernel\_size*=3, *stride*=1, *padding*=1)    *def* forward(*self*, *z*):          self.last\_z\_shape = z.shape          t\_emb = None          h = self.conv\_in(z)          h = self.mid.block\_1(h, t\_emb)          h = self.mid.attn\_1(h)          h = self.mid.block\_2(h, t\_emb)          for i\_level in *reversed*(*range*(self.num\_resolutions)):              for i\_block in *range*(self.num\_res\_blocks):                  h = self.up[i\_level].block[i\_block](h, t\_emb)                  if *len*(self.up[i\_level].attn) > 0:                      h = self.up[i\_level].attn[i\_block](h)              if i\_level != 0:                  h = self.up[i\_level].upsample(h)            h = self.norm\_out(h.transpose(1, 2)).transpose(1, 2)          h = nn.SiLU()(h)          h = self.conv\_out(h)          return h |

|  |
| --- |
| *class* VAE(*pl*.*LightningModule*):  *def* \_\_init\_\_(*self*, *embed\_dim*=64, *ckpt\_path*=None, *ignore\_keys*=[], *monitor*=None, *learning\_rate*=1e-3):          super().\_\_init\_\_()          self.encoder = Encoder(*ch*=64, *ch\_mult*=(1, 2, 4, 8), *resolution*=4800000, *z\_channels*=512)          self.decoder = Decoder(*ch*=64, *ch\_mult*=(8, 4, 2, 1), *resolution*=4800000, *z\_channels*=512)          self.loss = CombinedAudioLoss(*alpha*=0.5)          self.quant\_conv = nn.Conv1d(512, embed\_dim\*2, 1)  *# Matched to z\_channels*          self.post\_quant\_conv = nn.Conv1d(embed\_dim, 512, 1)  *# Matched to z\_channels*          self.embed\_dim = embed\_dim          self.learning\_rate = learning\_rate          if monitor is not None:              self.monitor = monitor          if ckpt\_path is not None:              self.init\_from\_ckpt(ckpt\_path, *ignore\_keys*=ignore\_keys)  *def* init\_from\_ckpt(*self*, *path*, *ignore\_keys*=list()):          sd = torch.load(path, *map\_location*='cpu')['state\_dict']          keys = list(sd.keys())          for k in keys:              for ik in ignore\_keys:                  if k.startswith(ik):  *print*('Deleting key {} from state\_dict'.format(k))                      del sd[k]          self.load\_state\_dict(sd, *strict*=False)  *print*(*f*'Reconstructed from {path}')  *# Return the 'latent vector (posterior)' of a VAE, which is a multivariate Gaussian Distribution*  *def* encode(*self*, *x*):          h = self.encoder(x)          moments = self.quant\_conv(h)          posterior = DiagonalGaussianDistribution(moments)          return posterior  *# Decode, try to reconstruct a sample from the latent vector*  *def* decode(*self*, *z*):          z = self.post\_quant\_conv(z)          dec = self.decoder(z)          return dec  *def* forward(*self*, *x*, *sample\_posterior*=True):          posterior = self.encode(x)          if sample\_posterior:              z = posterior.sample()          else:              z = posterior.mode()          dec = self.decode(z)  *# Return the reconstruction and the posterior*          return dec, posterior  *# A mandatory function to define what optimizers to use in pytorch lightning.*  *# Returns the optimizers and optionally rate schedulers used in training*  *def* configure\_optimizers(*self*):          lr = self.learning\_rate          opt\_ae = torch.optim.Adam(self.parameters(), *lr*=lr, *betas*=(0.5, 0.9))          return opt\_ae  *def* training\_step(*self*, *batch*, *batch\_idx*):          reconstructions, posterior = self(batch)  *# Calculate the combined audio loss*          loss = self.loss(reconstructions, batch)          self.log('train\_loss', loss, *prog\_bar*=True, *logger*=True, *on\_step*=True, *on\_epoch*=True)          return loss    *def* validation\_step(*self*, *batch*, *batch\_idx*):          reconstructions, \_ = self(batch)  *# Calculate the combined audio loss*          val\_loss = self.loss(reconstructions, batch)          self.log('val\_loss', val\_loss, *prog\_bar*=True, *logger*=True, *on\_step*=True, *on\_epoch*=True)          return val\_loss    *def* get\_last\_layer(*self*):          return self.decoder.conv\_out.weight |

Metrics and Training

Loss Function (손실 함수)는 모델의 예측 성능을 평가하는 데 사용되는 수학적 metric이다. 이 함수는 모델의 예측 값과 실제 값 사이의 차이를 측정하여, 이 차이가 얼마나 큰지를 나타내며, 뉴럴 네트워크의 가중치를 수정하는 데 사용되는 유일한 값이기에 훈련 과정에서 인공지능의 성능을 크게 좌우하는 지표이기도 하다. 이전에는 Mean-Squared Error (MSE, 평균 제곱 오차), Cross-Entropy (교차 엔트로피), Log-Loss (로그 손실)과 같은 sample의 scalar 값을 단위로 하는 low-level (저수준) 오차 함수들만이 사용되었지만, 사용되는 샘플의 크기와 복잡도가 크게 증가한 최근 인공지능 연구의 동향으로는 이에 더해 인간이 실제로 느끼는 샘플 사이의 인식적 차이인 Perceptual Loss (지각 손실) 함수도 병행되는 추세이다.

Perceptual Loss는 주로 Classification를 위해 훈련된 Convolutional Neural Network (CNN, 합성곱 뉴럴 네트워크)나 Variational Autoencoder의 Encoder의 최후반 레이어를 제거한 모델을 사용하여 측정되나, 이를 위해서는 생파형의 음악을 위해 고안된 그러한 CNN 모델이나 VAE 모델이 필요하고, 그것이 존재하지 않는 것이 이 프로젝트의 존재 이유였으므로 Perceptual Loss를 측정하는 것은 계란과 닭의 관계가 된다.

따라서 모델의 metric은 저수준 Mean Absolute Error (MAE, )에 더해, Perceptual Loss 대신 이와 비슷한 기능을 할 것으로 생각되게 고안한 함수 - Short-Time Fourier Transform으로 처리된 오디오의 절댓값 사이의 MAE loss의 합으로 구현되었다.

다음은 이를 구현한 Torch 코드의 전문이다.

|  |
| --- |
| *class* TimeDomainLoss(*nn*.*Module*):  *def* \_\_init\_\_(*self*):          super(TimeDomainLoss, self).\_\_init\_\_()          self.l1\_loss = nn.L1Loss()  *def* forward(*self*, *y\_pred*, *y\_true*):          return self.l1\_loss(y\_pred, y\_true)  *class* FrequencyDomainLoss(*nn*.*Module*):  *def* \_\_init\_\_(*self*):          super(FrequencyDomainLoss, self).\_\_init\_\_()          self.l1\_loss = nn.L1Loss()  *def* forward(*self*, *y\_pred*, *y\_true*):  *# Compute the STFT of y\_pred and y\_true*          stft\_pred = torch.stft(y\_pred.squeeze(), *n\_fft*=1024, *hop\_length*=256, *win\_length*=1024, *return\_complex*=True)          stft\_true = torch.stft(y\_true.squeeze(), *n\_fft*=1024, *hop\_length*=256, *win\_length*=1024, *return\_complex*=True)  *# Compute the magnitude of the STFT*          mag\_pred = stft\_pred.abs()          mag\_true = stft\_true.abs()  *# Compute the L1 loss between the magnitudes*          loss = self.l1\_loss(mag\_pred, mag\_true)          return loss    *class* CombinedAudioLoss(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *alpha*=0.5):          super(CombinedAudioLoss, self).\_\_init\_\_()          self.time\_domain\_loss = TimeDomainLoss()          self.frequency\_domain\_loss = FrequencyDomainLoss()          self.alpha = alpha  *def* forward(*self*, *y\_pred*, *y\_true*):          time\_loss = self.time\_domain\_loss(y\_pred, y\_true)          freq\_loss = self.frequency\_domain\_loss(y\_pred, y\_true)          loss = (1 - self.alpha) \* time\_loss + self.alpha \* freq\_loss          return loss |

훈련은 Google Colab에서 제공하는 Nvidia A100 GPU로 이루어졌다.

Implementation of the Sampler U-Net

U-Net 구현

|  |
| --- |
| *import* torch.nn *as* nn  *import* torch  *import* pytorch\_lightning *as* pl  *from* ..blocks *import* \*  *class* ResnetBlock(*nn*.*Module*):  *def* \_\_init\_\_(*self*, \*, *in\_channels*, *out\_channels*=None, *conv\_shortcut*=False, *dropout*, *t\_emb\_channels*=512):          super().\_\_init\_\_()          self.in\_channels = in\_channels          self.out\_channels = in\_channels if out\_channels is None else out\_channels          out\_channels = self.out\_channels          self.use\_conv\_shortcut = conv\_shortcut          self.norm1 = nn.LayerNorm(in\_channels)          self.conv1 = nn.Conv1d(in\_channels, out\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)          if t\_emb\_channels > 0:              self.t\_emb\_proj = nn.Linear(t\_emb\_channels, out\_channels)            self.norm2 = nn.LayerNorm(out\_channels)          self.dropout = nn.Dropout(dropout)          self.conv2 = nn.Conv1d(out\_channels, out\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)            if self.in\_channels != self.out\_channels:              if self.use\_conv\_shortcut:                  self.conv\_shortcut = nn.Conv1d(in\_channels, out\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)              else:  *# The NiN (Network-in-Network) shortcut - https://arxiv.org/abs/1312.4400*                  self.nin\_shortcut = nn.Conv1d(in\_channels, out\_channels, *kernel\_size*=1, *stride*=1, *padding*=0)  *def* forward(*self*, *x*, *t\_emb*):          h = x          h = self.norm1(h.transpose(1, 2)).transpose(1, 2)          h = nn.SiLU()(h)          h = self.conv1(h)  *# Project t\_emb to the same dimensionality as the output of the first convolutional layer, add to the result*          if t\_emb is not None:  *# The original network processed 4D image tensors (b, h, w, c) but in case of 2D audio tensors (b, s) dimensionality augmentation is unnecessary*  *# h = h + self.t\_emb\_proj(nn.SiLU(t\_emb))[:,:,None,None]*              h = h + self.t\_emb\_proj(nn.SiLU()(t\_emb))[:, :, None]          h = self.norm2(h.transpose(1, 2)).transpose(1, 2)          h = nn.SiLU()(h)          h = self.dropout(h)          h = self.conv2(h)          if self.in\_channels != self.out\_channels:              if self.use\_conv\_shortcut:                  x = self.conv\_shortcut(x)              else:                  x = self.nin\_shortcut(x)  *# Implement the defining skip connection of a ResnetBlock*          return x+h  *class* UNetDDPM(*nn*.*Module*):  *def* \_\_init\_\_(*self*, *ch*=64, *ch\_mult*=(1, 2, 4, 8), *num\_res\_blocks*=1, *attn\_resolutions*=[], *z\_channels*=512, *resolution*=4800000, *dropout*=0.0, *resamp\_with\_conv*=True):          super().\_\_init\_\_()          self.ch = ch          self.num\_resolutions = *len*(ch\_mult)          self.num\_res\_blocks = num\_res\_blocks          self.attn\_resolutions = attn\_resolutions          self.z\_channels = z\_channels          self.resolution = resolution          self.t\_emb\_ch = 0          self.resamp\_with\_conv = resamp\_with\_conv  *# Compute in\_ch\_mult, block\_in, and curr\_res at lowest res*          in\_ch\_mult = (1,) + tuple(ch\_mult)          block\_in = ch \* ch\_mult[self.num\_resolutions - 1]          curr\_res = resolution // 2\*\*(self.num\_resolutions - 1)  *# Downsampling layers*          self.down = nn.ModuleList()          for i\_level in *reversed*(*range*(self.num\_resolutions)):              block = nn.ModuleList()              attn = nn.ModuleList()              block\_out = ch \* ch\_mult[i\_level]              for i\_block in *range*(self.num\_res\_blocks):                  block.append(ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_out, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout))                  block\_in = block\_out                  if curr\_res in attn\_resolutions:                      attn.append(LinearAttention(*dim*=block\_in))              down = nn.Module()              down.block = block              down.attn = attn              if i\_level != 0:                  down.downsample = Downsample(block\_in, resamp\_with\_conv)                  curr\_res = curr\_res // 2              self.down.append(down)  *# Middle layers*          self.mid = nn.Module()          self.mid.block\_1 = ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_in, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout)          self.mid.attn\_1 = LinearAttention(*dim*=block\_in)          self.mid.block\_2 = ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_in, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout)  *# Upsampling layers*          self.up = nn.ModuleList()          for i\_level in *range*(self.num\_resolutions):              block = nn.ModuleList()              attn = nn.ModuleList()              block\_out = ch \* ch\_mult[i\_level]              for i\_block in *range*(self.num\_res\_blocks):                  block.append(ResnetBlock(*in\_channels*=block\_in, *out\_channels*=block\_out, *t\_emb\_channels*=self.t\_emb\_ch, *dropout*=dropout))                  block\_in = block\_out                  if curr\_res in attn\_resolutions:                      attn.append(LinearAttention(*dim*=block\_in))              up = nn.Module()              up.block = block              up.attn = attn              if i\_level != self.num\_resolutions - 1:                  up.upsample = Upsample(block\_in, resamp\_with\_conv)                  curr\_res = curr\_res \* 2              self.up.append(up)          self.norm\_out = nn.LayerNorm(block\_in)          self.conv\_out = nn.Conv1d(block\_in, z\_channels, *kernel\_size*=3, *stride*=1, *padding*=1)  *def* forward(*self*, *z*, *t\_emb*=None):  *# List to store the outputs of each downsampling block for skip connections*          skip\_connections = []  *# Downsampling*          for i\_level in *reversed*(*range*(self.num\_resolutions)):              for i\_block in *range*(self.num\_res\_blocks):                  z = self.down[i\_level].block[i\_block](z, t\_emb)                  if *len*(self.down[i\_level].attn) > 0:                      z = self.down[i\_level].attn[i\_block](z)              skip\_connections.append(z)              if i\_level != 0:                  z = self.down[i\_level].downsample(z)  *# Middle layers*          z = self.mid.block\_1(z, t\_emb)          z = self.mid.attn\_1(z)          z = self.mid.block\_2(z, t\_emb)  *# Upsampling*          for i\_level in *range*(self.num\_resolutions):              z = z + skip\_connections.pop()  *# Skip connection*              for i\_block in *range*(self.num\_res\_blocks):                  z = self.up[i\_level].block[i\_block](z, t\_emb)                  if *len*(self.up[i\_level].attn) > 0:                      z = self.up[i\_level].attn[i\_block](z)              if i\_level != self.num\_resolutions - 1:                  z = self.up[i\_level].upsample(z)          z = self.norm\_out(z.transpose(1, 2)).transpose(1, 2)          z = nn.SiLU()(z)          z = self.conv\_out(z)          return z |

다음은 U-Net을 train하는 pytorch lightning module

|  |
| --- |
| *import* torch  *import* torch.nn *as* nn  *import* pytorch\_lightning *as* pl  *class* DDPM(*pl*.*LightningModule*):  *def* \_\_init\_\_(*self*, *unet\_model*, *T*=1000, *learning\_rate*=1e-4):          super().\_\_init\_\_()          self.unet\_model = unet\_model          self.T = T          self.learning\_rate = learning\_rate          self.loss\_fn = nn.MSELoss()  *def* forward(*self*, *x*, *t*):          return self.unet\_model(x, t)  *def* training\_step(*self*, *batch*, *batch\_idx*):          x\_0 = batch  *# Original data*          B, C, L = x\_0.shape  *# Sample a random timestep for each example in the batch*          t = torch.randint(0, self.T, (B,), *device*=self.device).long()  *# Perform the forward diffusion process*          alpha\_bar\_t =          noise = torch.randn\_like(x\_0)          x\_t = torch.sqrt(alpha\_bar\_t[:, None, None]) \* x\_0 + torch.sqrt(1 - alpha\_bar\_t[:, None, None]) \* noise  *# Get the model's prediction for the noise*          eps\_hat = self(x\_t, t)  *# Compute the loss*          loss = self.loss\_fn(eps\_hat, noise)    *# Log loss*          self.log('train\_loss', loss, *on\_epoch*=True, *prog\_bar*=True)          return loss  *def* configure\_optimizers(*self*):          return torch.optim.Adam(self.parameters(), *lr*=self.learning\_rate)  if \_\_name\_\_ == "\_\_main\_\_":  *from* ..variational\_autoencoder.music\_dataset *import* MusicDataset, collate\_fn  *from* ..variational\_autoencoder.vae *import* VAE  *from* torch.utils.data *import* DataLoader      vae\_model = VAE()      unet\_model = UNetDDPM(vae\_model.encoder.z\_channels, *T*=1000)      ddpm\_model = DDPM(unet\_model)      train\_set = MusicDataset(*root\_dir*=*input*("Data direrctory: "))      train\_loader = DataLoader(train\_set, *batch\_size*=1, *shuffle*=True, *num\_workers*=1, *collate\_fn*=collate\_fn)      val\_loader = DataLoader(train\_set, *batch\_size*=1, *num\_workers*=1, *collate\_fn*=collate\_fn)      for batch in train\_loader:  *print*("Batch shape:", batch.shape)          break      trainer = pl.Trainer(*max\_epochs*=100, *precision*='16-mixed')      trainer.fit(ddpm\_model, train\_loader, val\_loader) |

* + - 1. Evaluation Metrics and Training

훈련은 Google Colab에서 제공하는 Nvidia A100 GPU로 이루어졌다.

1. Analysis and Implications

Batch size가 너무 작다

1. Conclusion - Limitations, Future Works, and Applications

Transformer Model을 사용한 다른 시도도 해보고 싶다

# 6 References

|  |  |
| --- | --- |
| [1] | Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, Yi-Hsuan Yang, "MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment," *arXiv preprint arXiv:1709.06298,* 2017. |
| [2] | Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, "Attention is All You Need," *In Advances in neural information processing systems,* pp. 5998-6008, 2017. |
| [3] | S. Hochreiter, "Long Short-Term Memory," *Neural Computation,* 1997. |
| [4] | Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, Douglas Eck, "Music Transformer," *arXiv preprint arXiv:1809.04281v3,* 2018. |
| [5] | Peter Shaw, Jakob Uszkoreit, Ashish Vaswani, "Self-Attention with Relative Position Representations," *arXiv preprint arXiv:1803.02155v2,* 2018. |
| [6] | Prateek Verma, "A Generative Model for Raw Audio Using Transformer Architectures," *arXiv preprint arXiv:2106.16036v3,* 2021. |
| [7] | I. J. Goodfellow, "Generative Adversarial Nets," *Neural Information Processing Systems,* 2014. |
| [8] | Prafulla Dhariwal, Alex Nichol, "Diffusion Models Beat GANs on Image Synthesis," *arXiv preprint arXiv:2105.05233v4,* 2021. |
| [9] | J. E. C. H. I. S. Gautam Mittal, "Symbolic Music Generation with Diffusion Models," *arXiv preprint arXiv:2103.16091v2,* 2021. |
| [10] | Flavio Schneider, Zhijing Jin, Bernhard Schölkopf, "Moûsai: Text-to-Music Generation with Long-Context Latent Diffusion," *arXiv preprint arXiv:2301.11757v2,* 2023. |
| [11] | Qingqing Huang, Daniel S. Park, Tao Wang, Timo I. Denk, Andy Ly, Nanxin Chen, Zhengdong Zhang, Zhishuai Zhang, Jiahui Yu, Christian Frank, Jesse Engel, Quoc V. Le, William Chan, Zhifeng Chen, Wei Han, "Noise2Music: Text-conditioned Music Generation with Diffusion Models," *arXiv preprint arXiv:2302.03917v2,* 2023. |
| [12] | K. Cho, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *Association for Computational Linguistics,* 2014. |
| [13] | J. Ho, "Denoising Diffusion Probabilistic Models," *Neural Information Processing Systems,* 2020. |
| [14] | J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities.," *Proceedings of the National Academy of Sciences of the United States of America,* 1982. |