(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

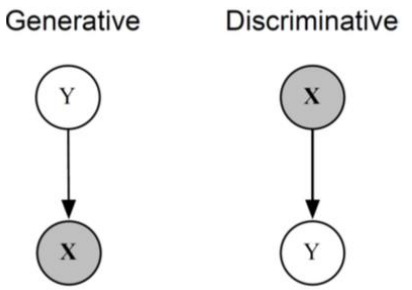
이 소논문은 Diffusion 모델의 음악 합성에 대한 이론적 배경과 관련 연구를 탐구한다. 특히 최근 고안되어 공간-시간적으로 연속적(spatiotemporally continuous)이고 정형적인 특성을 지닌 매체를 적은 컴퓨팅 자원만을 가지고 생성하는 데 효과적인 것으로 알려진 Latent Diffusion Model을 이용해 파형 그대로의 형식을 가진 비상징(非象徵, non-symbolic)적, 생파형(生波形)식 음악 합성에 중점을 둔다.

단계를 거쳐, 이 연구는 Variational Auto-Encoder 와 Diffusion 모델을 이용하여, 생파형 음악의 합성을 실현하는 과정을 상세히 설명한다. 논문은 Pytorch 를 사용한 Variational Auto-Encoder 와 U-Net 의 구현 과정을 포함하며, 이를 통해 Latent Diffusion Model 이 기존에 이미지 생성 분야에서만 적용되던 것을 넘어 음악 합성에도 효과적으로 활용될 수 있음을 보여준다. 또한, 본 연구는 Latent Diffusion Model 의 핵심 구성요소와 그들의 구현에 사용된 다양한 기술들을 논의하며, 이 모델이 어떻게 효과적인 비상징적, 생파형 음악 합성을 가능하게 하는지에 대한 깊은 이해를 제공한다. 분석 및 함의 부분에서는 구현된 모델의 성능 평가를, 결론 부분에서는 이 연구의 한계점과 미래의 연구 방향, 그리고 음악 합성에 대한 Latent Diffusion Model 의 활용 가능성에 대해 고찰한다.

Agenda

1 Introduction

통계적 분류의 관점에서 전통적으로 인공지능 네트워크들을 구분하는 접근 방식 중에 하나는 특정 모델이 generative 한지, discriminative 한지 나누는 이항 분류이다. Generative Model (생성 모델)들은 종속변수 y 가 주어졌을 때 독립변수 X 의 조건부 확률(conditional probability) 모델, 즉 $P(X \mid Y = y)$ 로 정의되며, Discriminative Model (판별 모델)들은 독립변수 x 가 주어졌을 때 종속변수 Y 의 조건부 확률 모델 – $P(Y \mid X = x)$ 로 정의된다. 그 중 Generative Model 들을 이용하는 Generative Artificial Intelligence, 즉 생성형 인공지능은 텍스트와 이미지 등의 미디어를 생성해낼 수 있는, 인공지능의 학문에서 Neural Network 의 기술적 패러다임이 Deep-Learning Era 를 넘어 Large-Scale Era 로 진입함과 함께 극명히 대두되는 모델의 종류이다.



https://towardsdatascience.com/generati
v 1

2020 년대 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 Multimodality 의 추세에 따라 영화 따위의 영상 생성이나 게임 제작과 같은 인지적으로 더욱 복잡한 예술 형태의 생성까지 차후 십여 년 이내에 가능해질 것이라고 예측도 가능해진다.

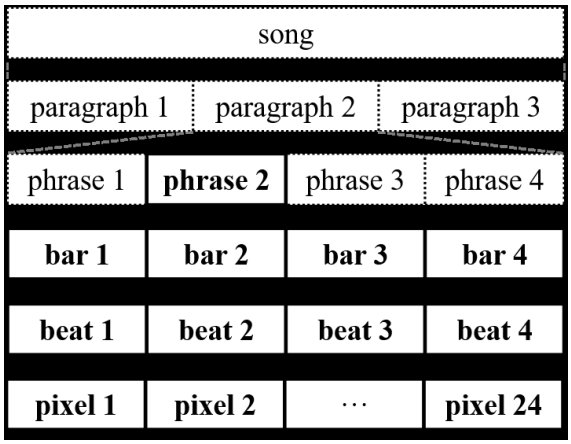
그리고 이 생성형 인공지능 학문이 이러한 발전을 이루도록 가장 큰 영향을 끼친 모델들 중 두 가지는 Transformer Architecture [1]와 Diffusion Models [2]이라고 할 수 있겠다. 이 두 모델들은 각각 Sequence-to-Sequence 의 시계열 데이터 처리 및 생성과 크기가 크며 인지적으로 복잡한 단일 샘플을 생성하는 데 사용되며, 각 분야에서 가장 범용적으로 사용되던 모델들을 대체해, 새로운 state-in-art 경지들을 성취해 내고 있다. 이 소논문에서는 두 모델들을 상대적으로 시도가 덜 이루어진 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 feasible 한 샘플들을 생성하는지, 고유한 Melodic 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 Modulation 과 같은 요소들은 음악의 생성을 다른 미디어의 생성보다 어렵게 만드는 데 한 몫을 하기 때문이다.

1.1 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.



https://www.researchgate.net/figure/Hier 1

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. [3] 오디오와 음악의 맥락에서 pixel은 일반적으로 이미지의 픽셀이 해당 도메인에서 가장 작은 정보 단위인 것과 마찬가지로 오디오 파일의 단일 샘플 포인트를 의미하며, $n\,kHz$의 샘플 레이트(sample rate)를 가지는 오디오 파일은 1초당 $n * 1000$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 2^n ($2 \leq n \leq 3$, n은 자연수)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.

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.

2 Background and Related Work

참조할 만한 선행하는 연구들은 Transformer Architecture을 사용한 것과 Diffusion Model을 사용한 것들로 나뉜다. 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) [4]를 꺾고 현존하는 방식들 중 가장 최적화된 성능을 보이기에 음악 생성에도 다양한 방식으로 사용되었다.

Cheng-Zhi Anna Huang et al.의 Music Transformer [5]는 Peter Shaw et al. [6]에 의해 제안된 상대적 attention 메커니즘을 확장하여, 음악 시퀀스의 장기 의존성을 처리할 수 있게 하였다. 이 모델은 기존의 Transformer architecture를 기반으로 하면서, 음악의 구조적 특성과 시간적 특성을 고려하여 효과적인 음악 생성

을 가능케 하였다. Music Transformer는 또한 이러한 상대적 attention 메커니즘을 통해 더 긴 시퀀스의 음악을 처리하고 생성할 수 있으며, 더 나아가 ABA 구조의 작곡과 유사하게 음악적 모티브, 문구 및 전체 섹션을 모델링하기 위해 자기 주의를 어떻게 사용할 수 있는지 보여준다.

Prateek Verma et al.의 A Generative Model for Raw Audio Using Transformer Architectures [7] 에서는, Transformer Architecture를 사용한 파형 수준의 오디오 합성을 위한 새로운 접근법이 제안되었다. 모델은 WaveNet 모델과 유사한 방식으로 파형을 생성하기 위한 심층 신경망을 채용한다. 그러나 생성된 각 샘플은 이전에 관찰된 샘플에만 의존하는 auto-regressive process와 causal generation process를 활용한다. 아키텍처 내의 주의 메커니즘은 어떤 오디오 샘플이 미래의 샘플을 예측하는데 중추적인지 학습하는 것을 용이하게 한다. 이 방법론은 인과적 트랜스포머 생성 모델을 통해 원시 파형 합성을 가능하게 한다. 이 논문은 이 새로운 접근법이 가능성이 있지만, 잠재 코드나 메타데이터의 도움 없이 의미 있는 음악의 생성을 아직 달성하지 못했다고 하였는데, 모델의 성능을 더 넓은 맥락에서 샘플을 조건화함으로써 향상시킬 수 있으며, 이를 통해 더 정교한 오디오 합성 모델을 위한 길을 열 수 있다고 언급한다.

Diffusion Model은 확률론적 score-based 생성 모델의 일종으로서, 기존에 이미지 생성에서 가장 pervasive하게 사용되던 Generative Adversarial Network (GAN, 적대적 생성 신경망) [8]을 훈련의 안전성, 생성된 sample의 질적 우월성 [9], 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 [10]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 잠재 자의 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.

3 Methodology

3.1 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.

```python
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 곡 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.

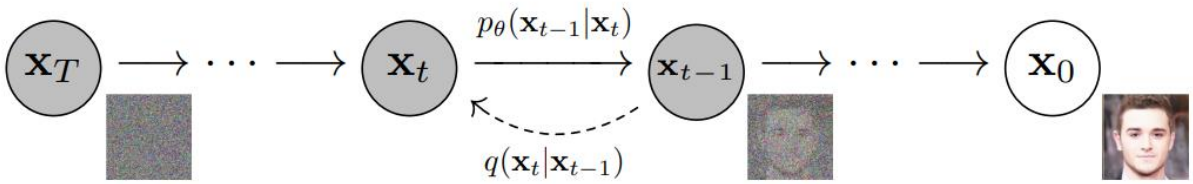
따라서 전처리에서 유일하게 신경 쓸 요소는 tensor의 크기를 좌우할 오디오의 sampling rate와 bit depth뿐이다. 가정용 GPU에서도 사용 가능하게 설계된, 이 프로젝트에서 사용하는 Latent Diffusion Model을 사용할 영감을 준 Stable Diffusion의 경우, 하나의 이미지 tensor는 일반적으로 512x512x3의 크기 (가로, 세로의 길이가 각각 512, color channel의 수가 R, G, B로 3), 786,432개의 scalar 값으로 이루어져 있으며, 각 scalar 값은 32-bit 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)
```

3.2 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 $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$ 에서는 sample $\mathbf{x}_0$가 주어졌을 때 같은 크기를 가진 posterior sample $\mathbf{x}_1, \ldots, \mathbf{x}_T$를 고정된 Gaussian Distribution를 이용해 계산적으로 구한다. 이때 timestep T는 sample x_t의 분포가 거의 완전한 isotropic Gaussian (대칭적 정규분포)를 이루도록 설정된다. 이 뉴럴 네트워크 모델이 사용되는 목적은 backward step – sample x_t가 주어졌을 때 x_{t-1}를 추정하는 $p_\theta(x_{t-1}|x_t)$를 구현하기 위해서이다.

Diffusion Model은 이러한 점차적인 sample 형성, sample 자체를 추정하는 것이 아닌 이전 forward step에서 첨가된 noise를 추정한다는 점 등 덕분에 패러다임이 끝자락을 향하던 GAN을 밀어내고 이미지 생성 분야에서 가장 뛰어난 성능을 보이는 모델이 되었다. [11] 게다가 Diffusion Model은 GAN의 문제점으로 자주 거론되던 Mode Collapse로 인한 훈련의 어려움과 생성되는 sample의 다양성 상실을 해결한 데 더해 점차적 형성으로 인한 자연어 embedding 등의 conditioning의 용이성 또한 가져 다양한 매체들 (음악, 동영상)을 생성하려는 시도들이 Diffusion Model을 사용하며 이루어지고 있다.

3.2.1. Understanding the Diffusion Samplers

위에서 상술한 Forward Step과 Backward Step을 구현하기 위해서는 Diffusion Sampler가 사용된다. Diffusion Sampler들은 Diffusion Model 내에서 모델의 state space을 횡단함으로써 샘플들을 생성하는데 사용된다. 이러한 샘플러들은 연속적인 (continuous) 또는 이산적 (discrete)인 방식, 다른 말로 확률적인 (stochastic) 방법 또는 결정적인 (deterministic) 방법으로 동작할 수 있다. Stochastic한 관점에서, 샘플들은 Stochastic Gradient

Equation (SDE, 확률적 미분 방정식)을 적분하여 생성되는 반면, discrete한 접근은 이산 단계들을 통해 반복하는 것을 택한다. 역사적으로 여러 가지 Diffusion Sampler들이 고안되었고, 이들 중 역사적 가치가 가장 높다고 생각한 Denoising Diffusion Probabilistic Models (DDPM)과 Denoising Diffusion Implicit Models (DDIM)을 구현해 보았다.

3.2.1.1 Denoising Diffusion Probabilistic Models

DDPM (Denoising Diffusion Probabilistic Models) [2]은 2020년 Jonathan Ho et al.에 의해 고안된 현실적인 샘플을 생성하기 위해 사용되는 Diffusion Model의 instance로, 특히 이전의 방법과 비교하여 안정성과 일반화 가능성에 대해 이미지 생성 [12]의 맥락에서 이전부터 이론상으로 존재해 왔던 확산 모델을 practical하게 만든 첫 모델이다. DDPM은 DDPM의 기반이 되는 연구들과 마찬가지로 Nonequilibrium Thermodynamics의 Langevin Dynamics의 원리에 뿌리를 두며, DDPM의 특징은 데이터를 점진적으로 noise화하도록 훈련된 파라미터화된 Markov Chain이라는 점이다.

DDPM은 Markov Chain으로 매개변수화되는데, 이는 Markov chain의 parameter가 모델의 잠재 변수들이며 잠재 변수 x1…xt가 이전(또는 이후) 시간 단계에만 의존한다는 것을 의미한다. DDPM의 저자들은 1000여 step에 이르는 Forward process를 Reparameterization trick을 써서 하나의 식으로 나타내는 것을 손보였다.

$$q(x_t|x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, \sqrt{1-\bar{\alpha}_t}I) = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$$
$$\epsilon \sim \mathcal{N}(0,1)$$

이는 Noise schedule로부터 계산되는 $\bar{a}_t$이 주어졌을 때 step 0에서 t까지를 한 번에 계산해 내는 cumulative product 연산이며, $\bar{a}_t = \prod_{s=1}^{t} a_s$로 계산된다. β_t는 우리의 noise scheduler라는 이름으로 정의된다. DDPM 논문의 저자들은 β_t로 10^-4와 0.02 사이의 linear scheduler를 사용하는데, 이때 시간 t=0에서 β_t의 값은 10^-4가 될 것이다. 이러한 값은 시간 t-1에 상대적으로 시간 t에서 추가된 소음의 양에 대한 백분율과 같은 역할을 한다.

3.2.1.2 Denoising Diffusion Implicit Models

Figure 1: Graphical models for diffusion (left) and non-Markovian (right) inference models.

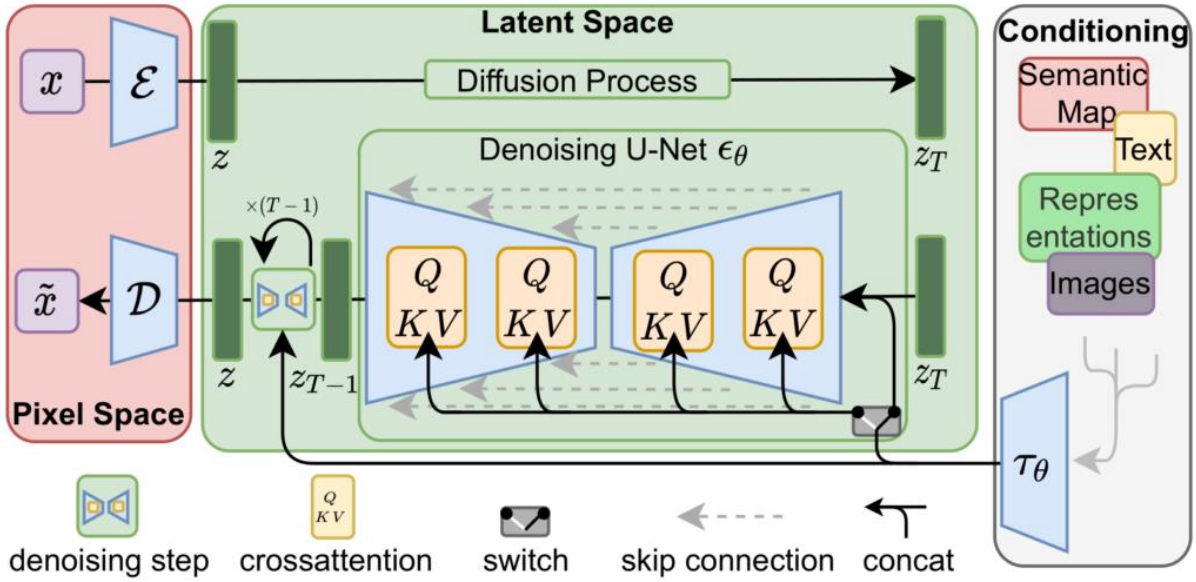
왼쪽 그림은 원래의 DDPM 논문으로, 시간 t에서 다음 노이즈 이미지를 얻기 위해 시간 T에서 시간 t-1까지의 모든 과거 노이즈 제거 단계를 요구한다. DDPM은 마르코프 체인으로 모델링되며, 이는 시간 t에서의 sample는 t 이전의 모든 sample이 생성되기 전까지 전체 체인이 생성될 수 없음을 의미한다.

DDIM (Denoising Diffusion Implicit Models) [13]은 광범위한 순방향 및 역방향 확산 단계로 인해 DDPM에서 느린 샘플링 프로세스를 가속화하기 위해 2020년 Jiaming Song et al.에 의해 고안되었다. DDIM의 프로세스를 non-Markovian (오른쪽 그림)으로 만드는 방법을 제안하여 노이즈 제거 프로세스의 단계를 건너뛸 수 있게, 현재 상태 이전에 모든 과거 상태를 방문할 필요가 없게 만들었다. DDIM의 장점은 모델을 교육한 후에 적용할 수 있으므로 새로운 모델을 재교육하지 않고도 DDPM 모델을 쉽게 DDIM으로 변환할 수 있다는 것이다.

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1-\bar{\alpha}_{t-1}} \cdot \epsilon_\theta^{(t)}$$

이것을 위해서 DDIM의 저자들은 reverse diffusion process를 위와 같이 새롭게 정의한다. 새로운 정의에서는 데이터에 noise가 추가되지 않는데, 이것이 DDIM의 장점이다. σ = 0일 때 reverse diffusion process은 완전히 결정론적이 되며, reverse diffusion 과정에서 새로운 noise이 추가되지 않으므로 존재하는 noise는 오직 x_0에 첨가된 단 한 stepd의 noise이다.. 역과정에서 noise가 없기 때문에 과정은 결정론적이고, 이는 sample의 생성에 필요한 step size를 20 정도의 작은 숫자로 획기적으로 줄이는 계기가 되었다.

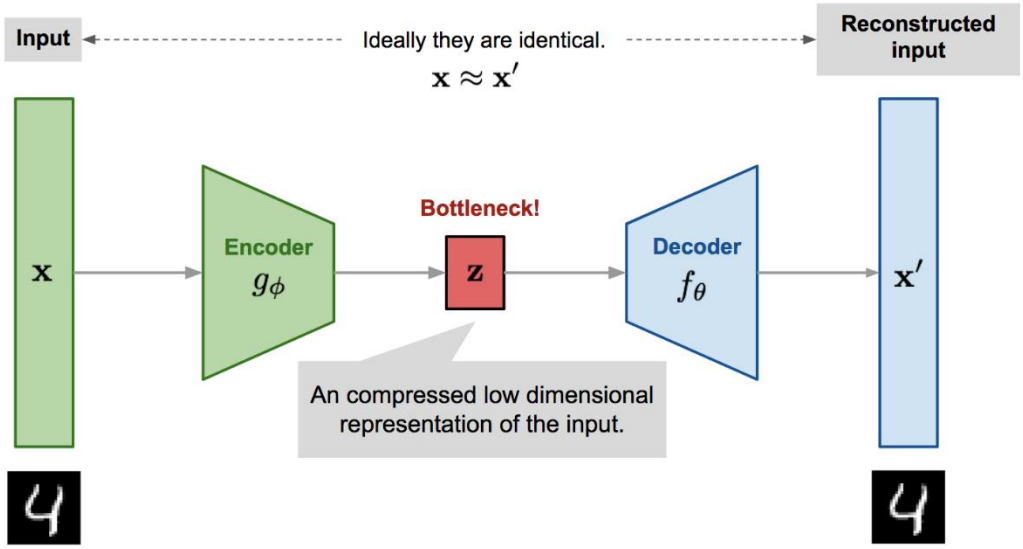
3.2.2 Implementation the Latent Diffusion Model



Latent Diffusion Model은 단순히 VAE 등의 네트워크로 압축된 잠재 표상 위에서 작동하는 Diffusion Model을 말한다. 2021년 Robin Rombach et al.의 High-Resolution Image Synthesis with Latent Diffusion Models [14]에서 고안되었으며, computational cost가 높은 Diffusion Model을 VAE로 압축시킨 잠재 벡터를 sample로 삼아 훈련시킴으로써 더 경제적인 접근을 가능하게 한다. 위의 논문 속 Latent Diffusion Model은 text conditioning과 inpainting 기능 등 다양한 기능들을 구현해 놓았지만, 이 프로젝트에는 단순성을 위해서 일반적 샘플 생성만을 다룬다.

3.2.2.1 Overview of the Model & Synthetization Process

3.2.2.1.1 Implementation of the Variational Auto-Encoder



Autoencoder (AE, 오토인코더)는 1985년 David H et al.의 A Learning Algorithm for Boltzmann Machines [14]에서 그 개념에 기초가 잡힌, 샘플 분포에 대한 효율적인 표상을 학습하고 이를 위해 Encoder와 Decoder가 존재하는 Bottleneck 구조를 가진 뉴럴 네트워크 구조들의 통칭이다. Bottleneck 구조를 가지고 Encoder와 Decoder 구조가 존재하지만 skip connection을 가지기 때문에 분리 가능한 잠재 표상을 가지지 않는 U-Net과는 다르다. 적은 차원의 잠재 공간을 이용해 input sample을 정확하게 재구축하는 것이 목적이기 때문에, input sample이 곧 output label이며, 비지도 학습 (Unsupervised Learning)을 사용하는 모델로 분류된다. 특징 학습 (feature learning), 변칙 탐지 (anomaly detection), 차원 축소 (dimensionality reduction) 등의 다양한 용도로 사용되며, Latent Diffusion Model에서의 쓰임은 Diffusion process를 거치는 저수준의 정보를 포함하고 복잡한 원형 tensor를 더 계산적으로 효율적으로 만드는 차원 축소에 해당한다.

역사적으로 완전 연결 계층 (fully connected layer)만으로 이루어진 Vanilla Autoencoder, 이에 Convolutional Layer을 추가해 이미지 재구축에 최적화한 Convolutional Autoencoder, 효율성을 위해 은닉층에 최소한의 노드들만 활성화되게 하는 Sparse Autoencoder 등의 여러 하위 모델들이 고안되었지만, 현재 가장 많이 사용되는 Autoencoder의 하위 모델 종류는 Variational Autoencoder (VAE, 변이형 오토인코더)이다.

VAE는 2013년 Kingma et al.의 Auto-Encoding Variational Bayes [15]에서 처음 고안된, 고차원의 Gaussian Distribution (정규 분포) 속의 한 포인트에 원형 sample을 맵핑하는, data space와 latent space 사이의 통계적 관계를 학습하는 Autoencoder의 분류이다. VAE의 Encoder의 output은 Gaussian Distribution들의 파라미터 – 평균 m과 표준편차 σ로 나타내어진다. 이에 따라 VAE는 일반 Autoencoder보다 data space에 대한 연속적이고 범용적인 표상을 학습하는 데 유리하며, latent space로부터 sampling한 point로부터 새로운 샘플을 생성하는 데에도 사용될 수 있다. 현재에는 주로 Vector Quantization (벡터 양자화)의 개념을 도입한 VQ-VAE로의 수학적 변형에 더해 ResNet, Attention Layer 등의 모델 구조 상 변형 등을 가해 구축되기도 한다.

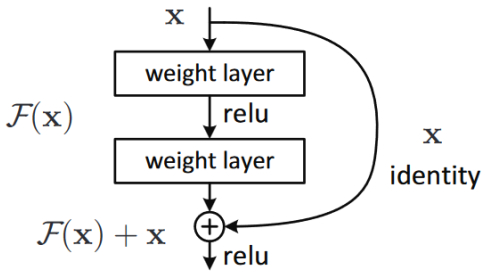
3.2.2.1.1.1 Convolutional Layers & Upsample & Downsample Layers

Upsample Layer와 Downsample Layer는 각각 VAE의 Decoder와 Encoder structure에 사용되는 합성곱 레이어들이다. 신호 처리 영역에서 기본적인 연산이므로 여러 신경망 구조, 특히 이미지 및 오디오 처리와 관련된 작업 전반에서 널리 사용된다. Upsampling은 주어진 신호 또는 이미지의 공간 해상도를 증가시키는 프로세스이다. Autoencoder의 관점에서, Upsampling은 네트워크를 통해 진행할 때 feature map의 dimension를 증가시키기 위해 Decoder structure에서 사용된다. 뉴럴 네트워크를 사용하지 않는 연산은 Nearest-Neighbor Interpolation

혹은 Bilinear Interpolation이 있지만, 인공신경망의 구현에서 Upsampling layer는 훈련되는 계층이다. 다음은 1D Convolutional Layer 하나와 Linear Interpolation로 이루어진 Upsample과 Downsample layer이다.

```
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
```

3.2.2.1.1.2 The Resnet Block and the Linear Attention



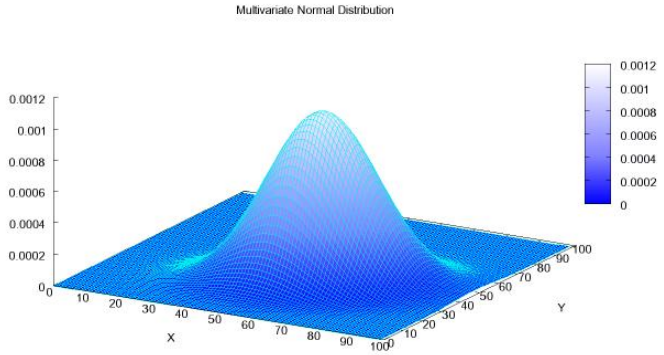
ResNet Block은 컴퓨팅 수준의 향상에 따라 딥러닝 모델들 또한 복잡해지던 무렵, 레이어의 수가 몇 십 겹을 넘어가면 Graident Vanishing problem에 의해 훈련이 잘 되지 않고, 오히려 복잡도가 더 낮은 모델들보다도 과소적합되는 문제를 해결하기 위해 2015년 Kaiming He et al.의 Deep Residual Learning for Image Recognition [16]을 통해 처음 고안되었다. ResNet Block의 구조를 이용하여 해당 연구팀은 Computer Vision 영역의 모델 성능 척도와도 같이 작용하는 대회인 ILSVRC에서 2015년 기존의 state-in-art 모델 VGG nets보다 무려 8배 깊은 152개의 레이어를 가진 모델을 설계해 ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation의 부문들에서 우승했는데, 이는 복잡한 모델의 설계에서 ResNet Block의 사용의 이점을 명확히 드러낸다.

ResNet의 핵심이 되는 아이디어는 'shortcut connection' 또는 하나 이상의 레이어를 우회하는 'skip connection'의 도입이다. ResNet은 각 레이어는 전통적인 순차적 방식으로 레이어를 쌓는 대신 F(x) + x로 나타내어지는, 입력에 자신의 출력을 추가하는 두 개의 연결된 레이어로 구성되는 ResNet Block으로 구성된다. 여기서 F(x)는 레이어의 출력이고 x는 레이어의 입력이다. 각 블록은 Normalization 레이어 및 활성화 함수에 이어지는 여러 Convolutional Layer로 구성된다. Skip connection은 입력을 출력에 직접 추가하기 위해 이 레이어들을 건너뛰어 모델을 'residual'하게 만든다. 이 블록들의 적층은 역전파 동안 그래디언트를 위한 일종의 고속도로를 형성하며, 이

는 소실되는 그래디언트 문제를 완화하고 매우 깊은 네트워크의 훈련을 가능하게 한다. 아래는 LayerNormalization, 1D Convolutional Layer, SiLU를 사용한 ResNetBlock을 구현한 Torch 코드이다.

```
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
```

3.2.2.1.1.3 The Diagonal Gaussian Distribution



VAE(Variational Autoencoder)에서 Diagonal Gaussian Distribution은 학습에서 가장 중요한 부분인 variational posterior를 나타내기 위해 사용된다. 단순성과 계산 효율성 때문에 선택된 이 분포는, latent space의 다변수 분포의 covariance matrix (공분산 행렬)의 off-diagonal 원소들이 모두 0 – 즉, 분포의 모든 차원들이 서로와 독립적이라는 이상적 가정을 내포하고 있기 때문에 Diagonal Gaussian Distribution이라고 불린다.

이러한 독립성 가정은 특히 full-covariance Gaussian Distribution과 비교할 때 계산을 의미 있게 단순화시킨다. VAE에서 목표는 관측된 데이터가 주어진 잠재변수들의 참된 사후분포에 대한 좋은 근사치를 학습하는 것이기에, Diagonal Gaussian Distribution는 이를 수행하기 위한 추적 가능하고 계산적으로 효율적인 방법을 제공한다. 이 분포의 파라미터들, 즉 각각의 차원에 따른 means (x) 및 variance (Var)들은 뉴럴 네트워크에 의해 훈련되며, 이는 VAE가 단순하고 효율적인 posterior approximation를 유지하면서 원형 데이터와 잠재 변수들 사이의 복잡한 관계들을 학습할 수 있게 한다. 아래는 VAE의 latent distribution을 나타낼 DiagonalGaussianDistribution의 Pytorch 코드이다.

이 코드 내에서 분산은 모델의 학습을 돕기 위해 log scale 내에서 측정이 되며, 차후 이 잠재 분포를 이용한 encoding 과정을 쉽게 하기 위해 분포로부터 posterior을 생성하는 sample() 함수와 평균을 돌려주는 mode() 함수를 노출시키고, 모델의 훈련을 위해 사용될 KL-Divergence와 Negative-Log Likelihood를 정의한다.

```
# 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
```

3.2.2.1.1.4 Structure of the Encoder & Decoder

Upsample&Downsample 레이어, ResNet block과 Diagonal Gaussian Distribution을 정의한 다음은 Encoder와 Decoder를 정의할 차례이다. 다시 언급하자면, Recognition Network라고도 알려진 인코더는 입력 공간으로부터 데이터를 가져와서 확률적 잠재 공간에 매핑한다. 이것은 잠재 공간에서 Gausian Distribution – sample posterior를 계산함으로써 수행된다. Encoder는 기본적으로 원형 데이터가 주어진 latent variables의 를 근사화하는 함수를 학습한다. Encoder의 핵심 부분은 원형 데이터를 받아들여 잠재 공간에서 관련된 확률 분포의 파라미터, 즉 mean과 variance을 출력하는 신경망인데, 여기서의 목적은 확률적인 방식으로 입력 데이터에 내재된 고차원 정보를 저차원 잠재 공간에서 포착하는 것이다.

아래는 Encoder network의 구현이다. 후에 이 Encoder을 encapsulate할 VAE로부터 channel multiplier, 레리어 군집 당 ResNet Block의 개수 등의 파라미터를 입력받으며, Downsample 레이어를 사용해 본격적인 차원 축소를 개시하는 Downsampling layer와 ResNet Block으로 이루어진 Middle layer로 구성된다. 출력값은 이후에 VAE에서 하나의 추가 레이어를 거쳐 DiagonalGaussianDistribution으로 원형 샘플을 확률적 잠재 공간 속에 맵핑한 sample posterior, 뒤따라오는 Decoder의 입력값이 된다.

```
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
```

Generative Network라고도 알려진 Decoder는 잠재 공간으로부터 sample posterior을 가져와서 다시 data space로 맵핑 - 재구축한다. Decoder는 본질적으로 잠재 변수가 주어진 관측 데이터의 가능성을 근사화하기 위한 함수를 학습하며, 인코더와 마찬가지로 뉴럴 네트워크로 이를 학습한다. 디코더의 목표는 학습된 잠재 공간으로부터 샘플링한 posterior을 원본과 비슷하게 재구축하는 것이며, 아래는 Decoder을 구현한 Pytorch 코드이다. 마찬가지로 VAE로부터 모델 파라미터를 입력받으며, Encoder와는 달리 텐서의 노드 개수를 증가시키는 Upsampling Layer가 Middle Layer을 뒤따른다.

```
class Decoder(nn.Module):
```

```python
    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
```

따로 정의된 Encoder와 Decoder structure는 중앙에 DiagonalGaussianDistribution을 둔 채로 VAE를 구성한다. VAE 모델은 간편한 정의를 위해 Pytorch Lightning을 사용하여 제작되었다. Lightning은 Pytorch로 뉴럴 네트워크를 정의할 때 따라와야 하는 training loop의 작성을 배제할 수 있게 해 주는 PyTorch의 확장 라이브러리이다. 이를 사용하면 딥러닝 모델을 개발할 때 VAE의 Encoding과 Decoding과 같이 하나의 작업이 하나 이상의 상황에서 사용될 때, 딥러닝 모듈이 pl.lightningModule을 override하게 만듦으로써 코드를 단순화시켜 사용자가 모델의 forward pass, loss 계산, optimizer step 등의 핵심 로직에만 집중할 수 있게 해 준다. Pytorch스러운 코드 스타일을 추구한다는 점, Callback function 등의 customization이 자유롭다는 등의 이유로 최근 인공지능 개발자들 사이에서 사용되는 경우가 많아지는 추세이다.

아래의 VAE 코드에서는 앞서 정의한 Encoder, Decoder의 resolution (=4800000)과 z_channels (=512) 등의 파라미터를 정의하고, Encoder와 DiagonalGaussianDistribution, DiagonalGaussianDistribution과 Decoder 사이를 연결하는 quant_conv 레이어와 post_quant_conv 레이어를 정의한다. 샘플 X가 주어졌을 때 Diagonal Gaussian Distribution Z를 반환하는 encode(x) 함수와 Diagonal Gaussian Distribution이 주어졌을 때 재구축한 샘플 $\bar{X}$를 반환하는 decode(z) 함수를 정의하고, encode(x)와 decode(z) 과정을 모두 거치는 forward(x, sample_posterior) 함수를 정의하여 VAE의 기능을 구현하였다. training_step(batch, batch_idx)와 validation_step(batch, batch_idx)는 자신의 forward(x) 함수를 접근하여 차후의 training loop 코드를 omit할 수 있게 해주며, 이렇게 VAE의 코드는 마무리가 된다.

```
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
```

VAE의 훈련은 Encoder과 Decoder 둘 다의 네트워크의 매개 변수를 최적화하여 모델 아래에서 관측된 데이터의 log-likelihood에 대한 대리 역할을 하는 Evidence Lower Bound(ELBO)을 최대화하는 것을 포함한다. 이 프로세스를 통해 VAE는 잠재 공간에서 데이터의 압축적이고 확률적인 표현을 학습하며, 이렇게 훈련된 VAE는 이후 작업에 사용될 수 있다.

3.2.2.1.1.5 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, $\boldsymbol{L1} = \frac{1}{N}\sum_{i=1}^{N} |y_i - y_i|$)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 코드이다.

```python
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 T4 GPU로 이루어졌다. 잠재 벡터 Z는 (1, 512)로, 입출력 텐서인 X와 $\bar{X}$는 (1, 4800000)으로 설정하였을 때, 모델은 대략 512M parameter 수를 가지게 되었다. GPU를 이용한 훈련 중 모델은 VRAM 사용량은 모델의 파라미터, 입력 데이터의 크기, 배치 크기, 중간 활성화, 최적화 상태, 그래디언트 등을 저장하기 위해 필요한 추가 공간 등 다양한 요소에 의해 영향을 받는다.

각 파라미터는 32비트 부동 소수점 숫자(4바이트)이다. 따라서 512M 개의 파라미터에 대해서는

$$512 \times 10^6\ parameters \times 4\frac{bytes}{parameter} = 2048\ MB$$

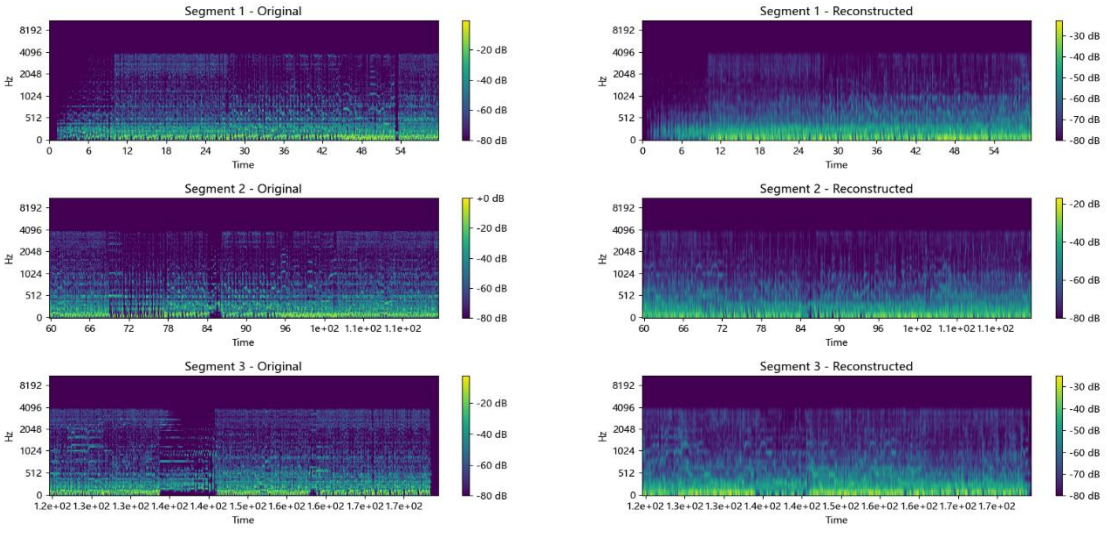
의 VRAM 점유율이 필요하다. 뉴럴 네트워크 모델이 훈련될 때 사용되는 gradient는 각 파라미터와 같은 크기를 가진다. 이를 고려하면 필요한 VRAM의 용량은

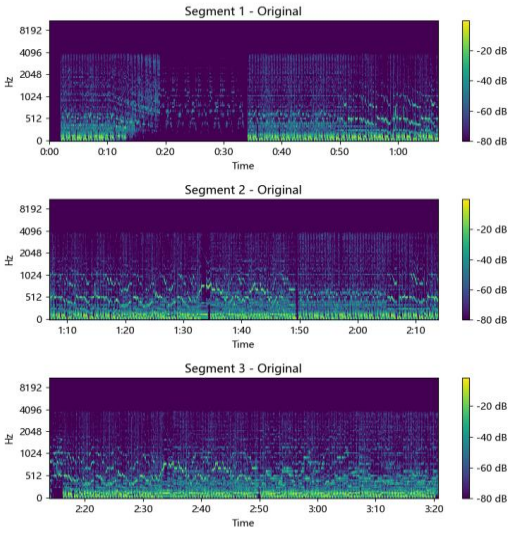
$$2048\ MB \times 2 = 4096\ MB$$

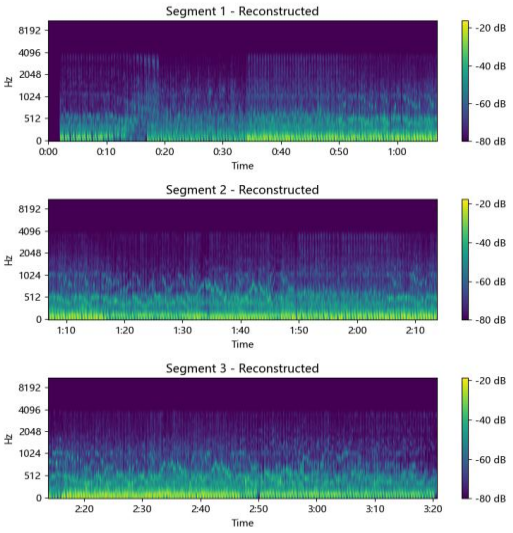
이다. 모델의 훈련에 사용되는 Adam optimizer 역시 gradient의 평균을 계산하는 등의 수행을 위해 일정 용량의 VRAM을 점유한다. Optimizer가 점유하는 VRAM의 용량은 정확하게 예측할 수 없지만, 주로 파라미터의 2~3배의 VRAM 용량을 점유한다. 이에 필요 VRAM의 용량은

$$2048\ MB \times (2 + 3) = 10240\ MB$$

까지 늘어난다. Batch size를 늘리면 input batch 자체의 크기와 활성화 함수, gradient의 계산에 필요한 VRAM이 증가 비율에 따라 선형적으로 증가한다. 따라서 메모리 단편화와 같은 변수들을 고려했을 때 16GB의 용량을 가진 T4 GPU에서 훈련하기 위해서 train batch size는 2로, validation batch size는 4로 설정하여, 3일 동안 150 epoch을 훈련하였다. 훈련된 모델은 앞서 정의한 CombinedAudioLoss로 0.04246655464172360의 training loss를, 0.088774223327636의 validation loss를 기록함으로써 음악의 구조적 유사성과 인지적 형태를 포착할 수 있게 되었다. 다음은 예시들은 훈련한 모델로 재구축한 오디오 파일들을, 원본은 왼쪽에, 재구축한 버전은 오른쪽에 Mel Spectogram으로 도식화시킨 것이다.




Segment 1 - Original
Hz
Time
dB


Segment 1 - Reconstructed
Hz
Time
dB

Segment 2 - Original
Hz
Time
dB

Segment 2 - Reconstructed
Hz
Time
dB

Segment 3 - Original
Hz
Time
dB

Segment 3 - Reconstructed
Hz
Time
dB

Segment 1 - Original
Hz
Time
dB

Segment 1 - Reconstructed
Hz
Time
dB

Segment 2 - Original
Hz
Time
dB

Segment 2 - Reconstructed
Hz
Time
dB

Segment 3 - Original
Hz
Time
dB

Segment 3 - Reconstructed
Hz
Time
dB

Segment 1 - Original
Hz
Time
dB

Segment 1 - Reconstructed
Hz
Time
dB

Segment 2 - Original
Hz
Time
dB

Segment 2 - Reconstructed
Hz
Time
dB

Segment 3 - Original
Hz
Time
dB

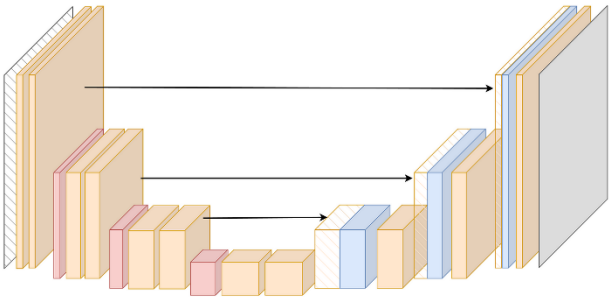
Segment 3 - Reconstructed
Hz
Time
dB

재구성된 음악은 Multivariate Gaussian Distribution을 잠재 공간으로 사용하는 VAE의 특성상 이미지에서 사용되는 VAE와 같이 재구축의 음향적 선명도가 크게 저하되는 결과가 나타났다. 이는 사용된 손실 함수 혹은 훈련하는 데 사용된 너무나도 작은 batch size의 문제라고 생각할 수 있겠다.

3.2.2.1.2 Implementation of the Diffusion Models

다음은 VAE에서 압축된 음악 데이터 위에서 작동하는 Diffusion Model을 구현한 것이다. Diffusion Model은 Diffusion Process (DDPM, DDIM)의 forward step과 backward step의 정의, 그리고 backward step에서 실질적인 역할을 하는 U-Net의 구현이 필요하다.

3.2.2.1.2.1 Implementation of the Sampler U-Net



왼쪽에 도식으로 나타내어진 U-Net은 Input과 Output tensor의 모양이 같고 중앙에 bottleneck structure와 skip connection을 가지고 있는 Convolutional Neural Network 딥러닝 모델의 구조 중 하나로, 영상에서 객체의 경계를 정확하게 탐지하고 분할하는 데 뛰어난 Image Segmentation, Depth Estimation 등의 여러 Computer Vision 과제에서 사용되는 모델이다.

U-Net의 구조는 이름으로 유추할 수 있듯이 U자 형태를 가지고 있는데, 이는 영상의 낮은 수준의 정보를 추출하는 인코더 부분과 이 정보를 바탕으로 고수준의 잠재 정보를 복원해내는 디코더 부분으로 구성되어 있다. U-Net은 인코더와 디코더 사이에 직접적인 skip connection들을 가지고 있어, 낮은 수준의 정보가 손실되지 않고 디코더로 전달될 수 있게 한다. 이러한 구조 덕분에 U-Net은 영상의 세밀한 특징까지 정확하게 파악하고 복원할 수 있는 능력을 갖추게 된다.

Diffusion Model에서는 U-Net이 backward step - sample x_t가 주어졌을 때 anterior sample x_{t-1}를 추정하는 과정에서 사용되는데, 이는 sample x_t가 모두 동일한 tensor shape를 가지고 있기 때문에 자연스럽게 U-Net이 noise를 추정하는 과정에 알맞기 때문이다. U-Net 모델은 따라서 noise가 첨가된 sample x_t를 입력으로 받아 anterior sample x_{t-1} 상태로 도달하기 위해 추가된 noise을 예측한다. 이 예측은 Diffusion Process을 반전시켜 목표 분포로부터 표본을 생성하기 위해 데이터를 점진적으로 noise을 제거하는 데 사용된다. Skip Connection을 갖는 인코더-디코더 구조 덕분에, U-Net은 다수의 스케일에서 특징들을 캡처하고 사용할 수 있다. 이는 상이한 특징들이 상이한 잡음 레벨들에서 다소 두드러질 수 있기 때문에, 추가된 잡음을 정확하게 예측하고 데이터를 잡음화하는 데 있어서 매우 중요하게 작용하는 특징이다.

다음은 앞서 VAE를 구축하며 사용하였던 뉴럴 레이어들을 재활용하여 구성한 U-Net이다. Channel multiplier, 각 레이어 군집당 ResNet block의 개수와 input resolution, z channel의 수를 노출시키며, 레이어 군집의 개수만큼의 Upsample Layer, Downsample Layer이 각각 Decoder와 Encoder에 ResNet block과 skip connection을 통해 연결된다. 이 모델이 구축에는 Pytorch의 ModuleList()의 활용이 중요하게 작용되었는데, 뉴럴 네트워크의 함수적 정의와 순차적 정의가 적절하게 공존하며 사용되는 Pytorch로 작성된 뉴럴 네트워크 코드의 Python스러움을 잘 보여준다고 할 수 있겠다.

```
import torch.nn as nn
import torch
import pytorch_lightning as pl
```

```python
from ..blocks import *


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
```

3.2.2.1.2.2 Implementation of DDPM

DDPM은 diffusion process을 reverse하도록 훈련되며, 이 과정은 정해진 수의 step을 거쳐 점진적으로 데이터에 노이즈를 추가한다. 모델의 목표는 이 과정의 각 단계에서 데이터에 추가된 noise를 예측하는 방법을 배우는 것이다.

이 훈련은 원본 DDPM 논문으로부터 발췌한 다음과 같은 순서로 이루어진다.

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Data point x_0을 data distribution으로부터 추출하고, [1, T]로부터 균일하게 timestep t를 추출한다. 정의된 Forward Diffusion Processs $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \; where \; \epsilon \sim N(0, I)$를 이용해 x_t를 계산한다. 모델의 예측값 $\epsilon: \hat{\epsilon} = f_\theta(x_t, t)$를 연산한다. ϵ와 $\hat{\epsilon}$ 사이의 오차 L2 함수를 계산하고, 오차를 역전파시킨다.

다음은 이를 구현한 Pytorch Lightning의 코드다. 앞서 정의한 U-Net을 파라미터로 받아, training_step() 중에 노이즈를 첨가하는 forward diffusion process와 이를 역진행하는 self.forward() 연산을 수행한다.

```
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)
```

3.2.2.1.2.3 Implementation of DDIM

DDM(Denoising Diffusion Implicit Models)은 DDPM의 변형으로 결정론적 샘플링을 허용하고 생성 프로세스에 대한 더 나은 제어를 제공하며, DDIM의 코드는 DDPM의 lightning module의 미세한 조정밖에 필요로 하지 않는다. 다음은 DDIM을 구현한 Pytorch 코드다.

```
def ddim_step(x_t, t, eps_theta, alpha_bar, alpha, sigma):
    """
    Perform a DDIM step.

    :param x_t: Noisy sample at time t
    :param t: Time step
    :param eps_theta: Noise prediction from the model
    :param alpha_bar: Cumulative product of alphas
    :param alpha: Alpha values for each time step
    :param sigma: Sigma values for each time step
    :return: Predicted sample for next time step
    """
    sqrt_alpha_bar_next = torch.sqrt(alpha_bar[t - 1])
    sqrt_one_minus_alpha = torch.sqrt(1.0 - alpha[t])

    # Calculate the predicted sample for the next time step
    x_t_next = (x_t - sqrt_one_minus_alpha * eps_theta) / sqrt_alpha_bar_next
    return x_t_next

class DDIM(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 = torch.nn.MSELoss()

        self.alpha = torch.linspace(0.9999, 0.01, T)
        self.alpha_bar = torch.cumprod(self.alpha, dim=0)
        self.sigma = torch.zeros(T)

    def forward(self, x, t):
        return self.unet_model(x, t)

    def training_step(self, batch, batch_idx):
        x_0 = batch
        B, C, L = x_0.shape

        # Sample a random timestep for each example in the batch
        t = torch.randint(1, self.T + 1, (B,), device=self.device).long()
        t_index = t - 1  # Convert to 0-indexing

        # Perform the forward diffusion process
        noise = torch.randn_like(x_0)
```

```
        x_t = torch.sqrt(self.alpha_bar[t_index][:, None, None]) * x_0 + torch.sqrt(1.0 -
self.alpha_bar[t_index][:, None, None]) * noise

        eps_theta = self(x_t, t)

        # Perform the DDIM step
        x_t_next_pred = ddim_step(x_t, t, eps_theta, self.alpha_bar, self.alpha, self.sigma)

        loss = self.loss_fn(x_t_next_pred, x_0)

        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)
```

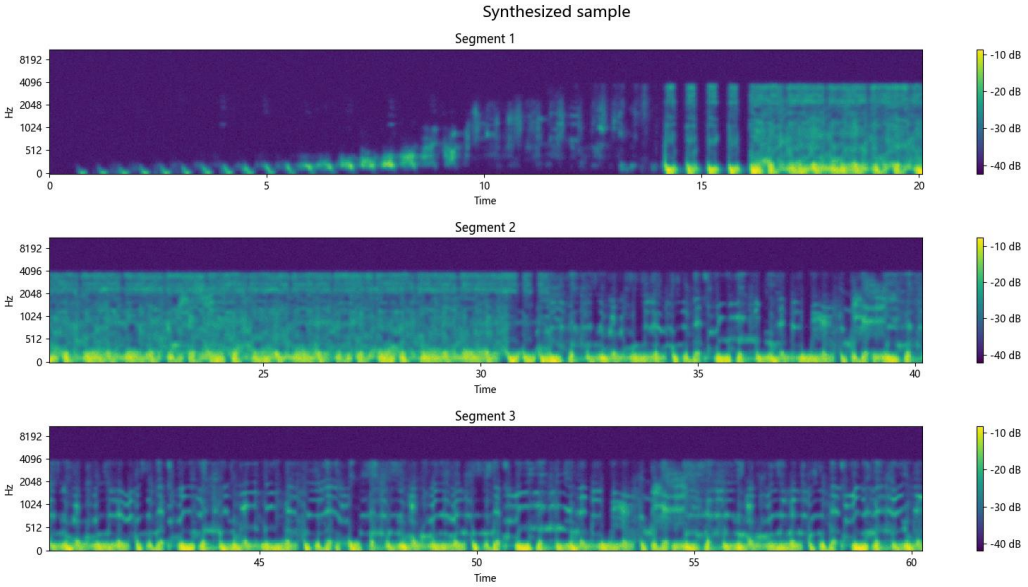
위의 과정에서 α는 diffusion process의 각 단계에서 추가되는 noise의 양을 조절하는 variance multipliers (분산 승수열)이다. 일반적인 선택은 α가 시간에 따라 등비수열을 따라 감소하도록 하는 것이다. α_bar는 timestep T까지의 α의 cumulative product (누적곱)으로 계산되며, σ는 DDIM 샘플링 과정에서 각 diffusion step에서의 noise 수준을 제어한다. σ의 선택은 샘플링의 결정성 (determinism)에 영향을 미친다. 결정성 샘플링을 하는 DDIM의 경우, σ는 0으로 설정될 수 있다.

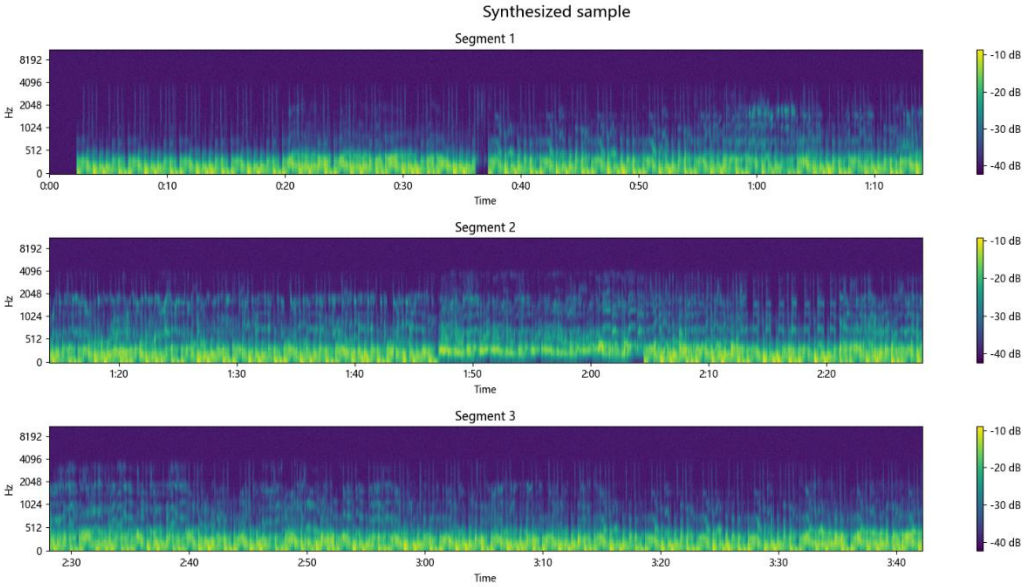
DDIM 훈련 단계에서, 주요 목표는 noise가 있는 데이터와 time step 정보가 주어진 특정 time step에서 입력 데이터에 추가된 noise를 예측하도록 모델을 훈련하는 것이다. 이에 따라 위의 코드에서는, 먼저 dataset과 random timestep t에서 랜덤 배치를 선택한다. 그리고 나서 α, α_bar 값들에 의해 결정된 noise schedule에 따라 input batch에 noise를 추가한다. 이 noise 데이터는 timestep t와 함께 DDIM 모델에 대한 입력의 역할을 한다. 이러한 맥락에서 통상적으로 U-Net 아키텍처인 DDIM 모델은 noise가 있는 데이터와 timestep을 받아들여 원본 데이터에 추가된 noise를 예측하며, 예측된 결과는 ground-truth와 대조되어 모델의 훈련에 사용된다.

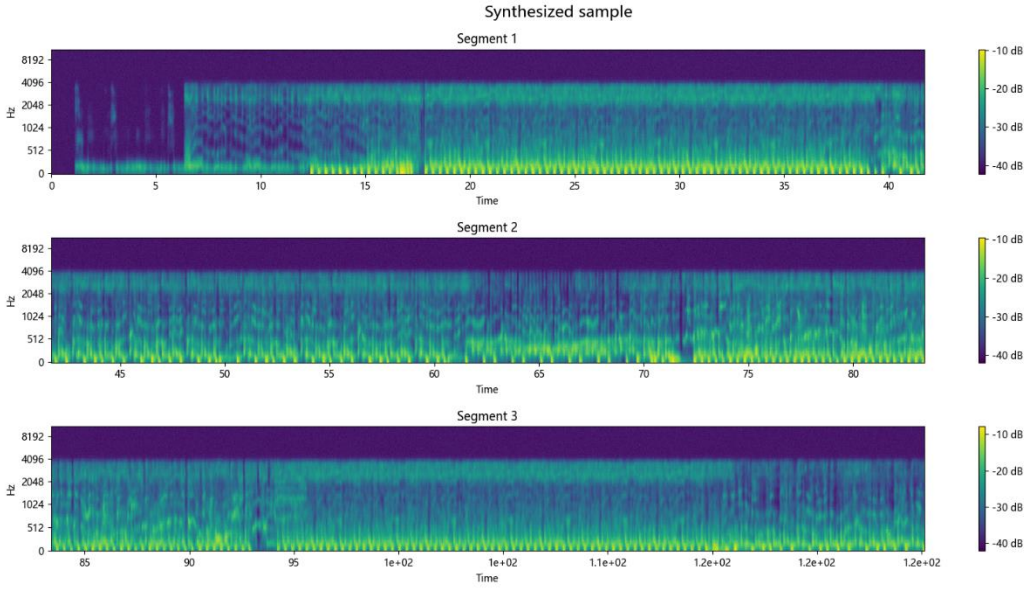
이 전체 과정은 다양한 시간 단계에서 추가된 잡음을 예측하는 모델의 성능을 점진적으로 향상시키기 위해 여러 epoch에 걸쳐 반복된다. 이러한 방식으로 U-Net에게 backward step을훈련시킴으로써 latent space으로부터 sample space으로 샘플을 생성할 수 있게 한다.

4 Analysis and Implications

DDIM 의 훈련은 T4 GPU 에서 진행되었다. 150 epoch 을 훈련하였는데, epoch 73 기준으로 validation loss 가 13.38 로 최소값을 도달하고 더 이상 수렴하지 않았다. DDIM 은 오디오의 전체적인 구조는 제대로 이해하는 경향을 보였으나, VAE 의 한계로 정상적인 음악을 담은 오디오로 보기에는 힘든 결과를 내놓았다. 이는 부족한 batch size 로 인한 gradient estimate 과정에서의 noise 와 훈련에서의 파라미터 진동 현상이 원인일 것으로 생각되어지지만, 더 좋은 컴퓨팅 파워를 동원하여 훈련한다면 Latent Diffusion Model 을 사용한 음악 생성이 상용화될 것이라는 확신을 주기에 충분하였다.


Synthesized sample
Segment 1
Segment 2
Segment 3
Hz
Time
8192
4096
2048
1024
512
0
-10 dB
-20 dB
-30 dB
-40 dB


Synthesized sample
Segment 1
Segment 2
Segment 3
Hz
Time
8192
4096
2048
1024
512
0
-10 dB
-20 dB
-30 dB
-40 dB



5 Conclusion – Limitations, Future Works, and Applications

인공지능의 분야는 정말 빠르게 발전한다. OpenAI 가 매년 새롭게 인류가 단기간 안에는 성취 못할 것이라고 생각하던 과제들 – GPT 3, 3.5, 4 로 선보인 인간의 '논리적 사고를 필요로 하던' 자연어 분석 / 이해 / 생성이나 Dall-E 2, 3 으로 선보인 '창의력을 필요로 하던' 정교한 이미지의 생성 등이 그것이다. 올해에는 GPT-4 의 Computer Vision 능력을 새롭게 공개하며 마치 이전에는 뉴럴 네트워크의 근본적 패러다임 전환 없이는 까마득해 보였던 인공 일반 지능 (Artificial General Intelligence, AGI) 또한 근방에 가능할 것이라는 인상을 주었다.

작년 GAN 을 뛰어넘는 성능을 보이며 여러 기술 기업들의 집중과 투자를 받은 Diffusion Model, 더 정확히는 이를 개인용 컴퓨터에서 작동시킬 수 있도록 설계한 Latent Diffusion Model 을 이로써 음악 생성의 과제에 맞게 구현하였다. 사용할 수 있는 GPU 의 VRAM 한계로 비롯한 구현할 수 있던 모델의 복잡성과 batch size 의 제한이 있었고, 빠른 시일 이내에 개인용 컴퓨터에 더 향상된 VRAM 용량을 가진 GPU 나 Neuromorphic Chip 이 부착되어 더 큰 인공지능 모델을 작동시키고 훈련할 수 있는 날이 왔으면 한다.

차후에는 Diffusion Model 과 함께 인공지능 분야에서 가장 인기 있는 모델인 Transformer Model 을 연구해 보고 싶다. Token limitation 으로 인해 지속적인 사용에 불편함이 있는 Transformer Model 을 개선하기 위해, 인간의 Long-Term Potentiation (장기 기억 생성)에서 영감을 받은 모델의 고안이 재밌을 듯 하다. 아니면 인공지능을 구현하는 다른 방안을 탐색하기 위해 오차 역전파를 사용하는 전통적인 뉴럴 네트워크가 아닌, 함께 활성화되는 뉴런들이 함께 연결된다는 Hebbian Rule 로부터 영감을 받았다는 Spiking Neural Networks 에 대해 탐구해 보고 싶다.

인공지능을 통한 음악 생성은 다른 미디어 생성 인공지능 모델들과 마찬가지로 인간의 창작과 저작권에 대한 근본적인 재고찰과 함께 윤리적 문제들을 야기할 것이다. 현재에는 정형화된 tensor shape 를 가진 이미지나 오디오 따위에 한정되어 있더라도, Moore Law 를 기반으로 한 컴퓨팅 파워의 향상과 인공지능 기술의 계속되는 발전으로 인해, 빠른 시일 이내에 영화와 같은 복잡한 영상, 게임과 같은 복합적 매체 등도 생성이 가능해질 지도 모르는 일이기 때문이다. 이런 간편한 매체 생성은 사용자를 위한 매체 customization – 선호하는 음악 장르에 따른 개인을 위한 음악 생성과 같은 – 으로 인해 편리함을 야기할 것이지만, 이는 창작을 계속하는 인간들이 차별화를 꾀해야 할 직접적인 계기가 된다. 아마 당시의 인공지능이 따라할 수 없는 비정형화된 예술 형태의 주류화나 창작의 복잡성의 개선으로 이러한 차별화가 일어나지 않을까 추측하며 이 프로젝트를 마친다.

6 References

[1] 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.

[2] J. Ho, "Denoising Diffusion Probabilistic Models," *Neural Information Processing Systems,* 2020.

[3] 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.

[4] S. Hochreiter, "Long Short-Term Memory," *Neural Computation,* 1997.

[5] 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.

[6] Peter Shaw, Jakob Uszkoreit, Ashish Vaswani, "Self-Attention with Relative Position Representations," *arXiv preprint arXiv:1803.02155v2,* 2018.

[7] Prateek Verma, "A Generative Model for Raw Audio Using Transformer Architectures," *arXiv preprint arXiv:2106.16036v3,* 2021.

[8] I. J. Goodfellow, "Generative Adversarial Nets," *Neural Information Processing Systems,* 2014.

[9] Prafulla Dhariwal, Alex Nichol, "Diffusion Models Beat GANs on Image Synthesis," *arXiv preprint arXiv:2105.05233v4,* 2021.

[10] J. E. C. H. I. S. Gautam Mittal, "Symbolic Music Generation with Diffusion Models," *arXiv preprint arXiv:2103.16091v2,* 2021.

[11] Prafulla Dhariwal, Alex Nichol, "Diffusion Models Beat GANs on Image Synthesis," *arxiv preprint arXiv:2105.05233v4,* 2021.

[12] Esteban Hernandez Capel, Jonathan Dumas, "Denoising diffusion probabilistic models for probabilistic energy forecasting," *arxiv preprint arXiv:2212.02977v5,* 2022.

[13] C. M. S. E. Jiaming Song, "Denoising Diffusion Implicit Models," *arxiv preprint arXiv:2010.0250*

2v4, 2020.

[14] David H. Ackley, Geoffrey E. Hinton, Terrence J. Sejnowski, "A learning algorithm for boltzma nn machines," *Cognitive Science,* vol. 9, no. 1, pp. 147-169, 1985.

[15] Diederik P Kingma, Max Welling, "Auto-Encoding Variational Bayes," *arxiv preprint arXiv:1312. 6114v11,* 2013.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Rec ognition," *arxiv preprint arXiv:1512.03385,* 2015.

[17] Flavio Schneider, Zhijing Jin, Bernhard Schölkopf, "Moûsai: Text-to-Music Generation with Lo ng-Context Latent Diffusion," *arXiv preprint arXiv:2301.11757v2,* 2023.

[18] 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, Zhi feng Chen, Wei Han, "Noise2Music: Text-conditioned Music Generation with Diffusion Model s," *arXiv preprint arXiv:2302.03917v2,* 2023.

[19] K. Cho, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *Association for Computational Linguistics,* 2014.

[20] J. Hopfield, "Neural networks and physical systems with emergent collective computational a bilities.," *Proceedings of the National Academy of Sciences of the United States of America,* 1 982.