Musical Style Transfer Using Latent Diffusion Models

Andrei Prioteasa (Matr. No.: 4740844)
Theo Stempel-Hauburger (Matr. No.: 4740729)

March 26, 2025

Lecture: Generative Neural Networks for the Sciences, WS 2024/25
Lecturer: Prof. Dr. Ullrich Köthe

GitHub Repository: `https://github.com/PrioteasaAndrei/music-style-transfer-ldm.git`

Contents

1 Introduction **3**
- 1.1 Background . . . 3
- 1.2 Problem Statement & Goals . . . 4

2 Methodology **5**
- 2.1 Architecture Overview . . . 5
  - 2.1.1 Spectrogram Encoder and decoder . . . 5
  - 2.1.2 Style Encoder . . . 6
  - 2.1.3 UNet . . . 7
  - 2.1.4 ForwardDiffusion . . . 8
  - 2.1.5 DDIM Sampling . . . 9
  - 2.1.6 Sinusoidal Position Embeddings . . . 10
  - 2.1.7 VGGishFeatureLoss . . . 11
- 2.2 Training Process . . . 11

3 Dataset Creation and Processing **12**
3.1 Data Selection 12
3.2 Data Acquisition 13
3.3 Audio Preprocessing and Spectrogram Generation 14
3.4 PyTorch Dataset Creation 18

4 Experiments and Results **22**
4.1 Training Environment 22
4.2 Spectrogram Resolution 22
4.3 Pre-training the Compression Model 23
4.4 Training the Style Transfer Model 23
4.5 Cross Attention Conditioning 24
4.6 Parameter Count Analysis 25
4.7 LDM Forward Pass 26
4.8 Style Generation Example with DDIM 27
4.9 Style Transfer Example with DDIM 29
4.10 Discussion and Analysis 30
4.10.1 Diffusion Restrictions 30
4.10.2 DDIM Issues 31

5 Conclusion **31**
5.1 Achievements 31
5.2 Future Work 32

Abstract

Andrei P.
In this project we implement a version of the approach proposed in the paper 'Music Style Transfer With Diffusion Model' [6]. We experiment with the use of latent diffusion models for musical style transfer, working on grayscale mel-spectrograms (which are a visual representation of audio). We design a Latent Diffusion Model that can transfer the style of one instrument to another, while preserving the original tonal characteristics of the instrument. We collect and preprocess our own dataset from youtube videos, which we make available to the public (see `https://github.com/PrioteasaAndrei/music-style-transfer-ldm.git`).

We analyze the model capability of transferring the style of one instrument to another, generating new spectrograms conditioned on a style image and we run experiments on the architecture choices.

We conclude that we are able to obtain blurry reconstruction, limited style transfer capabilities. We consider that the limited results of our model are due to the limited computational resources available to us and the complexity of the task, further training and refinement of our code is required to improve the results. We present a sample of our conditioned generation result at: `https://github.com/PrioteasaAndrei/music-style-transfer-ldm/blob/main/report/figures/content_aware_generated_audio_200ep.mp3`

1 Introduction

1.1 Background

Theo S.

Music style transfer involves altering the style of a musical piece, such as changing the instrument its played with, while preserving original characteristics like melody or rhythm. This could be done by relying on rule-based systems, by e.g. modifying the instruments in a MIDI file inside a music production software, or by leveraging machine learning techniques for symbolic music genre transfer, like Brunner et al. with MIDI-VAE or CycleGAN [4, 5]. However, these approaches are limited to leveraging MIDI files with the latter also requiring an exceptional amount of compute to train.

However, capturing the essence of a musical piece and transferring it to another one can involve much more than just changing what instrument is used to play the notes, if even applicable.

Recent advancements in machine learning have introduced approaches that work very well for the same corresponding task in the image domain. Notably, Latent Diffusion Models (LDMs) have demonstrated remarkable success in more efficiently generating high-quality images [8], and these models can be adapted to effectively transfer styles.

The principles of LDMs can be extended to music by representing audio data as spectrograms, visual representations of the frequencies in a sound signal over time. This approach potentially allows for application of image-based generation and style transfer techniques to audio data. The paper 'Music Style Transfer With Diffusion Model' [6] presents exactly this approach. Proposing a framework that utilizes diffusion models for music style transfer.

In this project, we want to implement a version of the approach proposed in the paper. However, since their code is not publicly available, we will develop our own implementation and hope to achieve similar results.

1.2 Problem Statement & Goals

Theo S.

The goal of this project is to implement a music style transfer architecture based on the principles of Latent Diffusion Models (LDMs). We should be able to take a music sample and change its instrument while preserving its other characteristics. This involves several challenges, including:

- **Dataset:** Creating a suitable dataset for training the model. The dataset should contain music samples with different instruments. Having enough samples of each instrument is also important to model has enough data to learn from. This is important to ensure that the model can learn to transfer styles effectively. Since we are very constrained in terms of computing resources, we restrict ourselves to only a few instruments and small samples.

- **Model Architecture:** Designing the model architecture. We will be implementing a Latent Diffusion Model (LDM) as proposed in the paper. Implementing an LDM as it would be traditionally done for images and also incorporating the ideas from the paper to adapt it to music.

- **Hardware Limitations:** Since we only have access to our laptops and a single 3060ti GPU, we are very limited in terms of computing resources. Because of that we have to be very careful to not introduce too many parameters to the model in order to train it for enough epochs.

- **Quality of Generated Samples:** We want to generate music samples that still sound okay and let us hear the characteristics behind the song and instruments. We plan to do most of the evacuation concerning the quality of the generated samples by just listening to them. However since we are very restricted in terms of computing resources, we are totally fine with not archiving the highest quality. If we can generate samples that sound okay and are able to transfer the style of the music, this is already a success and validate our approach. With more data, larger models and more training, someone with more resources could then probably achieve better results.

2 Methodology

Andrei P.
In this section, we will describe the methodology used to implement the proposed method. We will describe our approach to the architecture, the main components and the training process.

2.1 Architecture Overview

Andrei P.
We tried to stay as close to the original paper as possible in terms of architecture, but the paper did not provide a detailed description. The main components of our model are:

Component	**Description**
Spectrogram Encoder	Compresses the input spectrogram into a latent space
Style Encoder	Processes style spectrograms to extract multi-resolution style embeddings
Forward Diffusion	Implements the noise scheduler
UNet	Denoises the latent representation
Spectrogram Decoder	Reconstructs the final spectrogram from the latent space
DDIM	Reverse sampling process for generating new samples
Cross-Attention	Adds style information to the denoising process
VGGishFeatureLoss	Pretrained VGGish model to extract features from the spectrogram

Table 1: Main components of the model architecture

We will now briefly describe each of the components.

2.1.1 Spectrogram Encoder and decoder

Andrei P.
The encoder compresses the input spectrogram into a latent space using a series of convolutional layers, which allows for unrestricted input size:

```
class SpectrogramEncoder(nn.Module):
    def __init__(self, latent_dim=4):
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
```

```
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, latent_dim, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(latent_dim)
        )
```

The decoder mirrors the encoder architecture but uses transposed convolutions to upsample back to the original dimensions, normalizing the output to be between -1 and 1:

```
class SpectrogramDecoder(nn.Module):
    def __init__(self, latent_dim=4):
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 128, kernel_size=3, stride=2, padding=1,
                output_padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1,
                output_padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 1, kernel_size=3, stride=2, padding=1, output_padding
                =1),
            nn.Tanh()
        )
```

Both the encoder and decoder need to be pretrained on the spectrograms to be able to reconstruct the original audio. During the training process, we froze the encoder weights to prevent them from being updated, while leaving the decoder weights trainable. We describe this process in the experiments section.

2.1.2 Style Encoder

Andrei P.

The style encoder processes style spectrograms to extract multi-resolution embeddings. Activation maps from different convolutional layers are extracted and used as conditioning mechanisms in the UNet, through the Cross Attention mechanism.

```
class StyleEncoder(nn.Module):
    def __init__(self, in_channels=1, num_filters=64):
        self.enc1 = nn.Conv2d(in_channels, num_filters, kernel_size=3, stride=2, padding
            =1)
        self.enc2 = nn.Conv2d(num_filters, num_filters * 2, kernel_size=3, stride=2,
            padding=1)
        self.enc3 = nn.Conv2d(num_filters * 2, num_filters * 4, kernel_size=3, stride=2,
             padding=1)
        self.enc4 = nn.Conv2d(num_filters * 4, num_filters * 4, kernel_size=3, stride=2,
             padding=1)
        self.enc5 = nn.Conv2d(num_filters * 4, num_filters * 4, kernel_size=3, stride=2,
             padding=1)
        self.enc6 = nn.Conv2d(num_filters * 4, num_filters * 8, kernel_size=3, stride=2,
             padding=1)
```

2.1.3 UNet

Andrei P.
The UNet is a standard denoising diffusion model, which is used to denoise the latent representation of the spectrogram. The UNet is conditioned on the style embeddings extracted by the style encoder using the Cross Attention mechanism. Skip connections are used to improve the training process. We use a sinusoidal position embedding to encode the time step in the UNet.

```python
class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1, num_filters=64):
        super(UNet, self).__init__()

        # Define the channel dimensions used in your UNet
        time_emb_dim = 128  # This should match the channel dimension where you add the time embedding

        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),  # Match the channel dimension
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.GELU(),
            nn.Linear(time_emb_dim, time_emb_dim),
        )

        # Downsampling path with proper padding to maintain spatial dimensions
        self.enc1 = nn.Conv2d(in_channels, num_filters, kernel_size=3, stride=1, padding=1)
        self.enc2 = nn.Conv2d(num_filters, num_filters * 2, kernel_size=3, stride=2, padding=1)  # 128x128
        self.enc3 = nn.Conv2d(num_filters * 2, num_filters * 4, kernel_size=3, stride=2, padding=1)  # 64x64
        self.enc4 = nn.Conv2d(num_filters * 4, num_filters * 8, kernel_size=3, stride=2, padding=1)  # 32x32

        # Cross attention layers with correct embedding dimensions
        self.cross_attention1 = CrossAttention(embed_dim=512, num_heads=4)  # For 2x2 feature maps with 512 channels
        self.cross_attention2 = CrossAttention(embed_dim=256, num_heads=4)  # For 4x4 feature maps with 256 channels

        # Bottleneck
        self.bottleneck = nn.Conv2d(num_filters * 8, num_filters * 8, kernel_size=3, stride=1, padding=1)

        # Upsampling path with proper padding to maintain spatial dimensions
        self.dec4 = nn.ConvTranspose2d(num_filters * 8, num_filters * 4, kernel_size=3, stride=2, padding=1, output_padding=1)  # 64x64
        self.dec3 = nn.ConvTranspose2d(num_filters * 4, num_filters * 2, kernel_size=3, stride=2, padding=1, output_padding=1)  # 128x128
        self.dec2 = nn.ConvTranspose2d(num_filters * 2, num_filters, kernel_size=3, stride=2, padding=1, output_padding=1)  # 256x256
        self.dec1 = nn.Conv2d(num_filters, out_channels, kernel_size=3, stride=1, padding=1)
```

2.1.4 ForwardDiffusion

Andrei P.

The forward diffusion process gradually adds Gaussian noise to the input data over a fixed number of timesteps. At each timestep t, the process is defined by:

$$q(z_t|z_{t-1}) = \mathcal{N}(z_t; \sqrt{1-\beta_t} z_{t-1}, \beta_t \mathbf{I}) \tag{1}$$

where β_t is a variance schedule that controls how much noise is added at each step. The process can be written in a closed form for any timestep t as:

$$q(z_t|z_0) = \mathcal{N}(z_t; \sqrt{\bar{\alpha}_t} z_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{2}$$

where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. This allows us to sample z_t directly for any timestep using the reparameterization trick:

$$z_t = \sqrt{\bar{\alpha}_t} z_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \tag{3}$$

The reverse process then learns to gradually denoise the data by predicting the noise ϵ at each timestep. Given a noisy sample z_t and timestep t, we can predict the original input z_0 using:

$$z_0 = \frac{z_t - \sqrt{1 - \bar{\alpha}_t} \epsilon_\theta(z_t, t)}{\sqrt{\bar{\alpha}_t}} \tag{4}$$

where ϵ_θ is our UNet model that predicts the noise. This formulation allows for stable training and high-quality generation.

2.1.5 DDIM Sampling

Theo S.

This process was originally introduced by Song et al. [9]. Opposed to the forward diffusion process, which gradually adds Gaussian noise to the input data, in DDIM sampling, our goal is to reverse this process.

Given the noisy latent representation of our original image z_t, we use a trained UNet model $\epsilon_\theta(z_t, t)$ to predict the noise in z_t. From this prediction, we first estimate the original latent variable z_0 by rearranging the forward process equation:

$$z_0^{\text{pred}} = \frac{z_t - \sqrt{1 - \bar{\alpha}_t}\, \epsilon_\theta(z_t, t)}{\sqrt{\bar{\alpha}_t}}. \tag{5}$$

When we have a lot of noise, this is a pretty hard task for the model and therefore the z_0^{pred} will only be a rough estimate. Once we have this estimate, we add back a portion of the noise to reconstruct z_{t-1}, the latent at the previous timestep. In other words, we again "mix" the estimated z_0 with a portion of the noise to obtain a less noisy latent z_{t-1}. This update rule is given by:

$$z_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\, z_0^{\text{pred}} + \sqrt{1 - \bar{\alpha}_{t-1}}\, \epsilon_\theta(z_t, t) \tag{6}$$

where $\sqrt{\bar{\alpha}_{t-1}}\, z_0^{\text{pred}}$ represents the estimated latent at the previous timestep, and $\sqrt{1 - \bar{\alpha}_{t-1}}\, \epsilon_\theta(z_t, t)$ the corresponding noise at that timestep.

This process is repeated iteratively, moving from the final noisy latent z_T down to z_0.

In summary, while the forward process gradually introduces noise, the reverse DDIM update gradually removes it step by step. A good side effect of this approach is that each individual prediction by our model (if we only want to get new samples) does not need to be perfect. Because as the noise level decreases in later stages, the task becomes easier, one can still generate good samples even if the model is not perfect at every step.

2.1.6 Sinusoidal Position Embeddings

Andrei P.
The diffusion process requires knowledge of the timestep t to properly denoise the data. However, neural networks work best with continuous representations rather than discrete timestep indices. Therefore, we use sinusoidal position embeddings to encode the timestep information in a way that the network can effectively utilize.

The sinusoidal encoding transforms a scalar timestep t into a high-dimensional vector using sine and cosine functions at different frequencies:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \tag{7}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \tag{8}$$

where pos is the timestep and i is the dimension. This encoding has several desirable properties:

- It provides a unique encoding for each timestep
- The encoding varies smoothly with the timestep, allowing the model to interpolate between timesteps
- It captures both absolute and relative position information through the different frequency components
- The encoding is deterministic and requires no training

We employ the sinusoidal position embeddings to condition the model on the timestep. This is implemented as a multi-layer perceptron (MLP) that processes the timestep embedding:

```
self.time_mlp = nn.Sequential(
    SinusoidalPositionEmbeddings(time_emb_dim),
    nn.Linear(time_emb_dim, time_emb_dim),
    nn.GELU(),
    nn.Linear(time_emb_dim, time_emb_dim),
)
```

The MLP first converts the scalar timestep into a high-dimensional embedding using sinusoidal position embeddings, then processes it through two linear layers with a GELU activation. This processed timestep embedding is then injected into multiple layers of the UNet to condition its denoising behavior on the specific timestep. This approach, originally introduced in the Transformer architecture [10], has proven effective for encoding sequential position information in various deep learning applications, including diffusion models.

2.1.7 VGGishFeatureLoss

Andrei P.
The VGGishFeatureLoss is a loss function that uses the pretrained VGGish[1] model to extract features from the spectrogram and the reconstructed spectrogram, and then computes the mean squared error at different resolutions between the two.

2.2 Training Process

Andrei P.
Our training objective is a weighted sum of a reconstruction loss, a style transfer loss and a diffusion loss. More formally, the training objective is:

$$L = \lambda_{reconstruction} L_{reconstruction} + \lambda_{style} L_{style} + \lambda_{diffusion} L_{diffusion} \tag{9}$$

Specifically, the reconstruction loss is defined as:

$$\begin{aligned} L_{reconstruction}(x, \hat{x}, z) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{x}_i)^2 + \\ \lambda_{perceptual} \frac{1}{L} \sum_{l=1}^{L} MSE(\phi_l(x), \phi_l(\hat{x})) + \\ \lambda_{kl} \frac{1}{2} \mathbb{E}[z^2 - 1 - \log(z^2 + \epsilon)] \end{aligned} \tag{10}$$

[1] `https://github.com/harritaylor/torchvggish`

where ϕ_l represents the feature maps at layer l of the pretrained feature extractor network (VGGish or LPIPS). These feature maps capture increasingly abstract representations of the input spectrogram at different scales, from low-level features like edges in early layers to high-level semantic features in deeper layers.

For the diffusion loss, we use the standard denoising diffusion loss which measures how well the model predicts noise at each timestep in the latent space:

$$L_{diffusion}(\epsilon_\theta, \epsilon, t) = \frac{1}{n}\sum_{i=1}^{n}(\epsilon_{\theta,i}(z_t, t) - \epsilon_i)^2 \tag{11}$$

where ϵ_θ is the predicted noise and ϵ is the true noise.

For the style loss, we decide on measuring the MSE in the feature space of the pretrained feature extractor network (VGGish or LPIPS).

$$L_{style}(x, \hat{x}, z) = \frac{1}{L}\sum_{l=1}^{L} MSE(\phi_l(x), \phi_l(\hat{x})) \tag{12}$$

3 Dataset Creation and Processing

Theo S.
The dataset can be generated using the following two commands:

Listing 1: Generating the dataset

```
python data/yt_audio_downloader.py
python data/build_dataset.py
```

3.1 Data Selection

Theo S.
For this project, we focused on isolated instrument recordings, so recordings where only i.e. a piano or only a guitar are played. The idea behind this is to maintain simplicity and clarity in our audio style transfer tasks and to potentially decrease the complexity of the task for the model. This approach allows us to focus on learning the characteristics of individual instruments without the interference of other sounds or instruments. For now we selected

the following instruments for our dataset: Piano, Acoustic Guitar, Harp and Violin.

3.2 Data Acquisition

Theo S.
We developed an automated pipeline for downloading instrument recordings from `YouTube` using the `yt-dlp` library [11, 3]. The yt-dlp additionally acts as a wrapper for `FFmpeg` [1] for certain audio conversion and processing tasks, providing easy access to its functionality.

The data acquisition process followed these steps:

1. Manually search for instrument-specific videos that contained isolated recordings

2. Create a CSV file with columns for instrument labels, titles, and YouTube URLs

3. Process the data using our custom `AudioDownloader` class which:

   - Reads the CSV file to extract instrument categories, titles, and URLs
   - Creates a hierarchical folder structure with separate directories for each instrument type
   - Downloads high-quality audio streams using `yt-dlp` with the `bestaudio/best` format option
   - Converts downloads to MP3 format via FFmpeg
   - Names files consistently based on titles and saves them in the corresponding instrument subfolder

4. The resulting folder structure follows common conventions for organizing datasets on disk:

```
\label{code:music_directory_structure}
downloads/
|-- {instrument_name}/
|   |-- {title}.mp3
```

The core functionality of our `AudioDownloader` class relies on the `download_audio` method, which handles the actual download process:

Listing 2: AudioDownloader's download_audio method

```python
def download_audio(self, youtube_url: str, filename=None) -> str:
    """
    Downloads the audio stream from the provided YouTube URL using youtube-dlp.
    Additional documentation: https://github.com/yt-dlp/yt-dlp?tab=readme-ov-file#format-selection
    :param youtube_url: URL of the YouTube video.
    :param filename: Desired filename (with extension). If None, uses video's title.
    :return: Path to the downloaded audio file.
    """
    ydl_opts = {
        "format": "bestaudio/best",
        "outtmpl": (
            os.path.join(self.output_path, "%(title)s.%(ext)s")
            if filename is None
            else os.path.join(self.output_path, filename)
        ),
        "postprocessors": [
            {
                "key": "FFmpegExtractAudio",
                "preferredcodec": self.codec,
                "preferredquality": "192",
            }
        ],
    }

    with ytdlp.YoutubeDL(ydl_opts) as ydl:
        info = ydl.extract_info(youtube_url, download=True)
        if filename is None:
            filename = os.path.join(self.output_path, f"{info.get('title', 'audio')}.{self.codec}")
        return filename
```

Using the `AudioDownloader` is straightforward, as shown by this simple code snippet that processes youtube videos from a CSV file:

Listing 3: Example usage of AudioDownloader with CSV

```python
downloader = AudioDownloader(output_path="downloads", codec="mp3")
# Multiple URLs from CSV
downloaded_files = downloader.download_from_csv("data/youtube_urls.csv")
```

3.3 Audio Preprocessing and Spectrogram Generation

Theo S.

Raw audio signals need to be transformed into a representation suitable for our models. For this we convert the audio signals into spectrograms, which are visual representations of the frequency spectrum of audio signals over time. Since the architecture we are using is most known for image generation, we think this is a good approach to make sure our model can understand the data. To archive this we undergo multiple steps, which we describe in the following:

Audio Loading and Preprocessing The first step involves loading audio files and preparing them for further processing:

Listing 4: Audio loading and silence trimming

```
def load_audio(self, filepath):
    """
    Loads an audio file using librosa.
    :param filepath: Path to the audio file.
    :return: Tuple of (audio time series, sampling rate).
    """
    audio, sr = librosa.load(filepath, sr=self.target_sr, mono=True)
    return audio, sr

def trim_silence(self, audio, top_db=20):
    """
    Trims the silence from the beginning and end of an audio signal.
    :param audio: Audio time series.
    :param top_db: Threshold (in decibels) below reference to consider as silence.
    :return: The trimmed audio.
    """
    trimmed_audio, _ = librosa.effects.trim(audio, top_db=top_db)
    return trimmed_audio
```

By loading all audio at a standardized sampling rate of 22050 Hz, we ensure consistent processing regardless of the original recording quality. Trimming silence removes non-musical portions that could bring inconsistency to the dataset.

Mel Spectrogram Generation Next, we convert the preprocessed audio into spectrograms. More specifically, we use mel-spectrograms, which are a type of spectrogram that uses the mel scale instead of the linear frequency scale in the y-direction. The mel-scale is different from the linear frequency scale in that it is more aligned with human perception of sound. It compresses the frequency axis, grouping frequencies which are perceived as similar by the human ear into bins. With the number of bins being adjustable. Grouping the frequencies into bins allows us to drastically reduce the dimensionality and complexity of the data in y-direction. All of this while keeping characteristics of the original instrument and audio intact.

We perform this transformation using the `librosa` [2], which is a powerful python library for audio analysis. To extract the mel-spectrogram, we utilize our `get_mel_spectrogram` method:

Listing 5: Mel spectrogram extraction

```
def get_mel_spectrogram(self, audio, sr, n_mels=128):
    """
    Extracts a Mel spectrogram from the audio.
    :param audio: Audio time series.
    :param sr: Sampling rate.
    :return: Log-scaled Mel spectrogram.
    """
    mel_spec = librosa.feature.melspectrogram(y=audio, sr=sr, n_mels=n_mels)
    log_mel_spec = librosa.power_to_db(mel_spec, ref=np.max)
```

```
    return log_mel_spec
```

This transformation involves:

- Applying a Short-Time Fourier Transform (STFT) to convert time-domain signals to frequency domain
- Mapping the linear frequency spectrum to the mel scale.
- We mostly use 128 mel frequency bands. While testing different values as well as their reconstructions, we found that 128 bands capture sufficient information to reconstruct the original audio file with acceptable quality while at the same time keeping the dimensions low.
- Converting the spectrograms to decibel scale.

Spectrogram to Image Conversion To make spectrograms easily compatible with our architecture and processing pipeline, we convert them to standardized grayscale images:

Listing 6: Converting spectrograms to grayscale images

```python
def mel_spectrogram_to_grayscale_image(self, spectrogram, max_db=80):
    """
    Converts a log-scaled Mel spectrogram to an image.
    :param spectrogram: Log-scaled Mel spectrogram.
    :param max_db: Maximum decibel value for clipping.
    :return: Image of the Mel spectrogram.
    """
    # Shift to positive values
    spectrogram = spectrogram + max_db
    # Scale to 0-255 (grayscale)
    spectrogram = spectrogram * (255.0 / max_db)
    # Clip out of bounds
    spectrogram = np.clip(spectrogram, 0, 255)
    # Do rounding trick and convert to uint8
    spectrogram = (spectrogram + 0.5).astype(np.uint8)

    # Create an image
    image = Image.fromarray(spectrogram)
    return image
```

This conversion process includes:

- Since log-scaled mel-spectrogram values typically range from -80 dB to 0 dB, adding an 80 dB offset shifts the values to the range [0, 80]. This step converts negative values to positive values, which is necessary for image representation.
- The shifted values are then scaled from the [0, 80] range to the standard [0, 255] range required for 8-bit grayscale images.

- After scaling, any values outside the [0, 255] range are clipped to ensure that they stay within valid image intensity bounds.
- To ensure each value is rounded to the nearest integer, a rounding step (by adding 0.5) is applied before conversion.
- Finally, the processed array is converted into a PIL Image, making it more suitable for storage.

Complete Processing Pipeline With these individual components established, we created a comprehensive pipeline that processes our entire dataset:

Listing 7: Full audio-to-spectrogram processing pipeline

```python
def build_dataset_folder_structure(
    mp3_dir="downloads", output_root="processed_images", chunk_size_sec=3, max_duration=1800, n_mels=128
):
    """
    Process audio files in the mp3_dir, generate spectrogram images,
    and save them into folders (named after instrument labels) under output_root.

    :param mp3_dir: Directory containing the audio files.
    :param output_root: Root directory to save processed spectrogram images.
    :param chunk_size_sec: Duration of each audio chunk in seconds.
    :param max_duration: Maximum duration to process per file (in seconds).
    """
    ap = AudioPreprocessor()
    mp3_dir = Path(mp3_dir)
    mp3_files = list(mp3_dir.rglob("*.mp3"))

    for mp3_file in mp3_files:
        # Use the parent directory's name as the instrument label.
        instrument = mp3_file.parent.name
        instrument_dir = Path(output_root) / instrument
        instrument_dir.mkdir(parents=True, exist_ok=True)

        print(f"Processing file: {mp3_file}")
        # Load and preprocess audio.
        audio, sr = ap.load_audio(mp3_file)
        audio = ap.trim_silence(audio)

        # Calculate the number of samples per chunk.
        chunk_size = int(chunk_size_sec * sr)

        for chunk_idx, i in enumerate(range(0, len(audio), chunk_size)):
            if max_duration is not None and (i / sr) >= max_duration:
                break
            chunk = audio[i : i + chunk_size]
            if len(chunk) < chunk_size:
                chunk = np.pad(chunk, (0, chunk_size - len(chunk)), mode="constant")

            spectrogram = ap.get_mel_spectogram(chunk, sr, n_mels=n_mels)
            image_pil = ap.mel_spectogram_to_grayscale_image(spectrogram)

            filename = f"{mp3_file.stem}_chunk{chunk_idx}.png"
            image_path = instrument_dir / filename
            image_pil.save(image_path)
            print(f"Saved image: {image_path}")
        print(f"Finished processing file: {mp3_file}")
```

For this we first split each recording into 3-second chunks. This allows us to on the one hand create more samples in our dataset, but also keep the later training time reasonable. Since right now our dataset consists of very long recordings, one per instrument, we additionally can archive an equal split per instrument label by limiting the maximum duration of each recording by the length of the shortest recording. This is 30 minutes in our case. We hereby ensure that no instrument is over- or underrepresented in our dataset. As mentioned before, we decide to use 128 mel-frequency bands for our mel-spectrograms. The chunking process is done by iterating over the audio signal in steps of 3 seconds, and for each chunk, we generate a mel-spectrogram and convert it to an image. The final dataset structure mirrors our original audio organization, with each instrument having its own folder of 3 second spectrogram images.

3.4 PyTorch Dataset Creation

Theo S.
To later enable easy usage and compatibility in training throughout the pytorch training process, we created a custom dataset class that handles loading and processing of our images.

SpectrogramDataset First for just simple loading of the spectrogram images we created the following `SpectrogramDataset` class, which inherits from `torch.utils.data.Dataset`:

Listing 8: Custom Dataset Class

```
class SpectrogramDataset(Dataset):
    def __init__(self, config):
        super(SpectrogramDataset, self).__init__()
        self.image_dir_path = config["processed_spectograms_dataset_folderpath"]
        self.data = datasets.ImageFolder(root=self.image_dir_path, transform=self._get_transform())

    def __getitem__(self, idx):
        x, y = self.data[idx]
        return x, y

    def __len__(self):
        return len(self.data)

    def _get_transform(self):
        """
        Define the transformations to be applied to the images.
        :return: Transformations
        """
        return transforms.Compose(
            [
                # add crop from 130 to 128
                # ! If the chunk size is different, this needs to be changed
                transforms.Lambda(lambda x: x.crop((0, 0, 128, 128))),  # Crop to 128x128
                transforms.Grayscale(),  # Needed because ImageFolder by default converts to RGB ->
                    convert back
                transforms.ToTensor(),  # Automatically normalizes [0,255] to [0,1]
            ]
        )
```

Our custom dataset class utilizes PyTorch's `ImageFolder` to efficiently load spectrograms organized by instrument folders. This approach leverages the folder structure we established during the preprocessing phase. The transforms in `_get_transform()` ensure consistency across all images while preserving their essential characteristics.

Transform steps include cropping the images to 128x128 pixels. This is necessary because, while the original images have a y-dimension of 128 (matching the number of mel-frequency bands), the x-dimension is 130 pixels due to an interplay of multiple factors during the processing pipeline:

The x-dimension (time frames) is calculated using the formula:

$$T = \left\lceil \frac{D \times F_s}{H} \right\rceil$$

Where D is the duration of each audio chunk in seconds, F_s is the sampling rate, and H is the hop length of the STFT.

Substituting with the actual values,

$$T = \left\lceil \frac{3 \times 22050}{512} \right\rceil = \lceil 129.19 \rceil = 130$$

this calculation results in 130 time frames, which explains the original x-dimension of the spectrogram images before cropping. We then again convert the images to grayscale, normalize and convert them to tensors.

SpectrogramPairDataset For our style transfer experiments, we needed a more specialized dataset that could provide pairs of spectrograms from different instrument categories.

Our implementation allows us to draw predetermined instrument-to-instrument combinations for training, and also ensures an equal distribution of samples across all instrument categories. Moreover, this approach avoids storing a new dataset on disk but allows to load the images from the original data folder structure on the fly.

To accomplish this, we implemented the `SpectrogramPairDataset`:

Listing 9: Paired Dataset for Style Transfer

```
class SpectrogramPairDataset(Dataset):
```

```python
    def __init__(self, root_folder, pairing_file, transform=None):
        """
        Args:
            root_folder (str): Root directory with subfolders (each for one label).
            pairing_file (str): Path to the CSV file with predetermined pairings.
            transform: Transformations to apply to each image.
        """
        self.root_folder = root_folder
        self.pairing_file = pairing_file
        self.transform = transform if transform is not None else self._get_transform()

        # Load the precomputed pairs from the CSV file.
        # Each row in the CSV should contain: label1, idx1, label2, idx2
        self.pairs = []
        with open(self.pairing_file, "r") as f:
            reader = csv.reader(f)
            for row in reader:
                # Convert index strings to integers.
                self.pairs.append((row[0], int(row[1]), row[2], int(row[3])))

        # Build a dictionary of ImageFolder datasets keyed by label.
        self.datasets = {}
        # Sorting the subfolders ensures deterministic order.
        for folder in sorted(os.listdir(root_folder)):
            folder_path = os.path.join(root_folder, folder)
            if os.path.isdir(folder_path):
                # Assume the folder name is the label.
                self.datasets[folder] = ImageFolderNoSubdirs(root=folder_path, transform=self.
                    transform)

    def __len__(self):
        return len(self.pairs)

    def __getitem__(self, index):
        # Use the index to get the predetermined pairing.
        label1, idx1, label2, idx2 = self.pairs[index]
        img1, _ = self.datasets[label1][idx1]
        img2, _ = self.datasets[label2][idx2]
        return (img1, label1), (img2, label2)
```

The `SpectrogramPairDataset` works by loading multiple `ImageFolderNoSubdirs` datasets with each only containing spectrograms from one instrument label. It then pairs these datasets based on the predetermined pairings stored in a CSV file on the fly.

`ImageFolderNoSubdirs` is a custom class we implemented, that is a modified version of the standard `ImageFolder` class from PyTorch. Since the `ImageFolder` class expects a specific folder structure of the dataset, as described in 3.2, it does not allow for loading images of only one label from a single folder without subdirectories. We fix this issue by overwriting different methods of the PyTorch `ImageFolder` class.

Rather than randomly selecting pairs during training, we generate these pairings in advance and save them to a CSV file:

Listing 10: Generating predetermined pairs for consistency

```python
@classmethod
def generate_pairings(cls, root_folder, output_file_path="spectrogram_pair_dataset_pairings.csv",
    num_pairs=15000):
    """
```

```
    Generates a CSV file containing the predetermined pairings.

    Args:
        root_folder (str): Root directory with subfolders for each label.
        output_file (str): Path where the CSV file will be saved.
        num_pairs (int): Number of pairs to generate.
    """
    # List of labels (i.e. subfolder names) sorted deterministically.
    labels = sorted(
        [folder for folder in os.listdir(root_folder) if os.path.isdir(os.path.join(root_folder, folder))]
    )

    if len(labels) < 2:
        raise ValueError("Need at least two classes to form pairs.")

    # Create ImageFolder datasets for each label.
    datasets_dict = {}
    for label in labels:
        folder_path = os.path.join(root_folder, label)
        datasets_dict[label] = ImageFolderNoSubdirs(root=folder_path, transform=cls._get_transform())

    pairs = []
    # We precompute the pairs and save them as a file. Like this the future sampling is deterministic.
    rng = np.random.RandomState(42)
    for _ in range(num_pairs):
        # Randomly select two distinct labels.
        label1, label2 = rng.choice(labels, size=2, replace=False)
        ds1, ds2 = datasets_dict[label1], datasets_dict[label2]
        # Randomly select indices within each dataset.
        idx1 = rng.randint(0, len(ds1))
        idx2 = rng.randint(0, len(ds2))
        pairs.append((label1, idx1, label2, idx2))

    # Write the pairs to a CSV file.
    with open(output_file_path, "w", newline="") as f:
        writer = csv.writer(f)
        for pair in pairs:
            writer.writerow(pair)
    print(f"Pairings saved to {output_file_path}")
```

DataLoader Creation By following the PyTorch conventions, we can now easily leverage the default PyTorch `DataLoader` class with all its functionality, to create data loaders for our datasets.

Listing 11: Creating data loaders with train-test split

```
def prepare_dataset(config):
    dataset = SpectrogramDataset(config)

    # Split into training (80%) and testing (20%) sets
    train_size = int(0.8 * len(dataset))
    test_size = len(dataset) - train_size
    train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, test_size])

    train_loader = DataLoader(train_dataset, batch_size=config["batch_size"], shuffle=True, num_workers=0)
    test_loader = DataLoader(test_dataset, batch_size=config["batch_size"], shuffle=False, num_workers=0)

    return train_loader, test_loader
```

4 Experiments and Results

Andrei P.
In this section we will go in detail over the experiments we have conducted to evaluate the performance of our model, implementation choices, problems we encountered and how we solved them. Given the limited training capacity of the available computing resources, most architectural decisions were made with a trade-off between model complexity and training time, which may have resulted in suboptimal performance and the reason why the model is not able to transfer the style.

4.1 Training Environment

Andrei P.
The training was conducted on a single NVIDIA GeForce RTX 3060 Ti GPU with 8GB of VRAM. We were bound by this limitation in choosing the model architecture and the hyperparameters, which may have directly resulted in the suboptimal performance of the model and lack of convergence, even if we worked on such a reduced dataset of 128x128 grayscale spectrograms.

We developed two training scripts, one for the pre-training phase and one for the main training of the LDM. The trained weights of the autoencoder are then loaded into the LDM and frozen.

```
# Pre-training the autoencoder
python models/train.py --model autoencoder

# Training the latent diffusion model
python models/train.py --model ldm
```

The hyperparameter configuration is available in the `config.py` file.

4.2 Spectrogram Resolution

Andrei P.
We experimented with different spectrogram resolutions, by manually inspecting the reconstructed audio. We decided on using 128 frequency ranges for the spectrograms, as it provided a good balance between quality and training time and limiting the temporal resolution to 3 seconds.

4.3 Pre-training the Compression Model

Andrei P.

The compression model (encoder and decoder) is jointly pre-trained on the training set for 100 epochs using a learning rate of 0.5e-4 with an AdamW optimizer and ReduceLROnPlateau scheduler. The learning rate is reduced by a factor of 0.5 when the validation loss plateaus for 10 epochs, with a minimum learning rate of 1e-6. This adaptive learning rate strategy helps prevent overfitting and ensures stable convergence of the compression model training.

The pre-training phase focuses solely on reconstruction quality, optimizing the encoder-decoder architecture to effectively compress and reconstruct the input spectrograms while preserving essential musical features. This step is crucial as it establishes a strong foundation for the latent space that will later be used by the diffusion model for style transfer. We decide to regularize our latent space using a KL divergence loss, which helps to ensure that the latent space is normally distributed. Both the encoder and decoder are fully convolutional, allowing for dynamic input sizes.

We originally experimented with a latent space dimension of [32,8,8], which we found out to be too limiting in terms of the expressiveness of the latent space. We increased the spatial resolution of the latent space to [32,16,16], which allowed the model to learn a more complex latent space and improve the quality of the style transfer.

At this point we are able to reconstruct the input spectrograms with a high degree of accuracy, as shown in Figure 1. We now freeze the weights of the encoder and leave only the decoder trainable during the ldm training phase.

4.4 Training the Style Transfer Model

Andrei P.

The style transfer model is trained with similar hyperparameters (in term of learning rate and optimizer) as the pre-training phase and it is trained jointly with the diffusion model and the decoder. Since a simple MSE loss does not encourage the model to learn style characteristics, we decide on a pretrained feature extractor network and compute the loss as the MSE between the feature maps of the input and the target style spectrogram at different resolutions. Initially we opted for LPIPS loss, but since LPIPS is pretrained on ImageNet, we found out that it is not suitable for our task

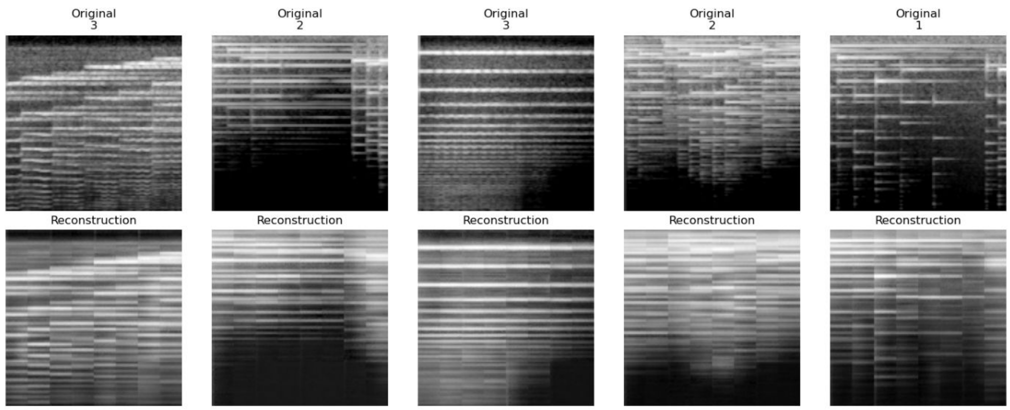

Figure 1: Reconstruction results from the pre-trained autoencoder. The first row shows the input spectrograms, while the second row shows the reconstructed spectrograms.

which deals with grayscale spectrograms. We then decided to use the VGGish feature extractor, which is pretrained on the AudioSet dataset and is more suitable for our task. Unfortunately, even after this modification, we were not able to diagnose while our style loss is not improving.
Moreover, at this point we observe that the compression loss which now accounts only for the decoder, is not improving, which may indicate over-training of the decoder or other issues.

4.5 Cross Attention Conditioning

Andrei P.
We apply the cross attention at 2x2 and 4x4 resolutions to inject style information into the latent space. For this, the default cross attention layer has to be rewritten in order to handle the shape of the style spectrogram.

```
def forward(self, unet_features, style_embedding):
    batch_size, c, h, w = unet_features.shape

    # Reshape feature maps for attention
    # [B, C, H, W] -> [H*W, B, C]
    unet_features = unet_features.permute(2, 3, 0, 1)  # [H, W, B, C]
    unet_features = unet_features.reshape(h * w, batch_size, c)  # [H*W, B, C]

    # Reshape style_embedding
    # [B, C, H, W] -> [H*W, B, C]
    style_embedding = style_embedding.permute(2, 3, 0, 1)  # [H, W, B, C]
    style_embedding = style_embedding.reshape(h * w, batch_size, c)  # [H*W, B, C]

    # Apply cross-attention
    attended_features, _ = self.multihead_attn(unet_features, style_embedding,
        style_embedding)
```

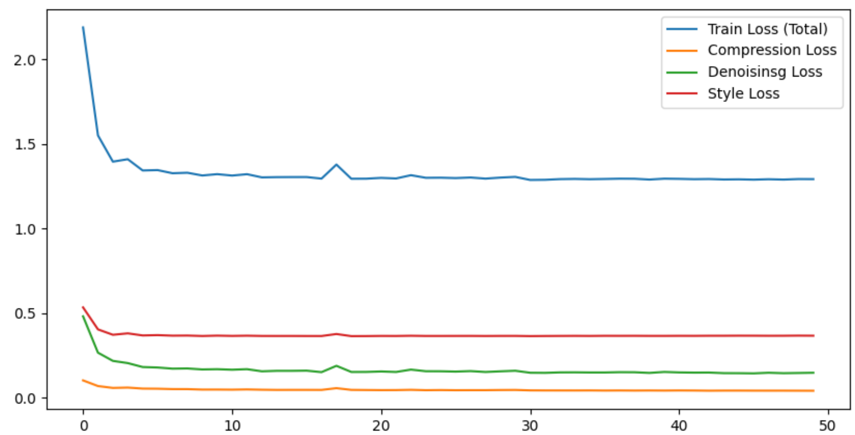

Figure 2: Loss curves during the training of the style transfer model. The style loss is not improving, which indicates that the model is not able to learn the style characteristics.

```

    # Reshape back to feature map
    # [H*W, B, C] -> [B, C, H, W]
    attended_features = attended_features.reshape(h, w, batch_size, c)
    attended_features = attended_features.permute(2, 3, 0, 1)

    return attended_features
```

4.6 Parameter Count Analysis

Theo S.

We analyze the parameter counts of different model components to understand the model complexity and computational requirements. Table 2 shows the breakdown of parameters for each component of our final model. The limiting number of parameters was intentionally chosen given the limited training resources.

It may be exactly because of this reason that the model is not able to learn the style characteristics and generate good results.

Table 2: Parameter counts for different model components

Component	Total Parameters	Trainable Parameters
SpectrogramEncoder	111,840	111,840
SpectrogramDecoder	198,209	198,209
StyleEncoder	2,729,984	2,729,984
CrossAttention	1,313,792	1,313,792
UNet	8,155,296	8,155,296
VGGishFeatureLoss	88M	0 (pre-trained)
LDM (full)	12,609,985	12,609,985

4.7 LDM Forward Pass

Theo S.
First we look at the output of the forward pass of the LDM model. Figure 3 visualizes the input spectrograms and the generated output spectrogram.

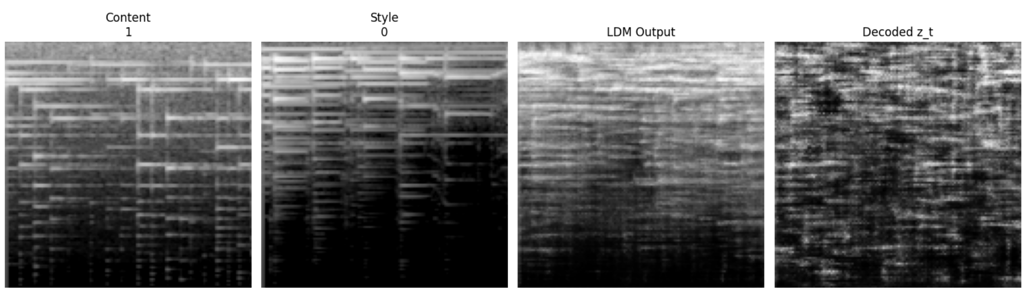

Figure 3: LDM forward pass results. The figure displays the spectrograms used as content and style input as well the generated output spectrogram. The right most column shows the initial content spectrogram with its applied noise (t=200), after we pass it through the decoder. The LDM model was trained for 200 epochs.

When comparing the decoded z_t, the LDM generated output, and the content spectrogram, we observe that while the decoded z_t looks noisy, the LDM seems to be able to capture some level of structure. This is an indication that the LDM did learn some style and reconstruction characteristics. Both insufficient training and a problematic loss convergence for the style encoder make a conclusive assessment of the style transfer capabilities of the LDM difficult. To get the audio output of the generated output spectrogram, we then reverse our preprocessing steps with librosa to get from the generated output spectrogram to an audio output.

Listening to the resulting audio, we find that it does not have meaningful content and sounds very unnatural and noise-like with, if any, very slight rhythmic structure. Because the output looks still very unrefined and noisy this is expected.

Next, we look at what happens when generating without a content spectrogram as input but instead only random noise. Figure 4 shows the results of this. The model is able to actually generate something with resemblance

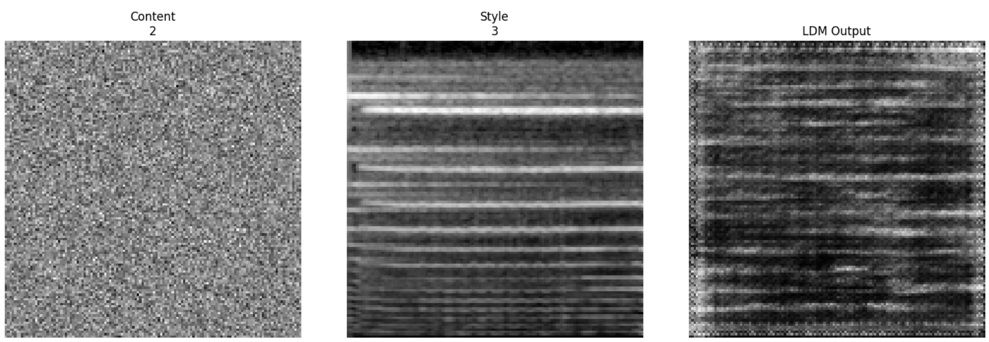

Figure 4: LDM forward pass results with random noise as input. The figure displays the spectrograms used as content and style input as well the generated output spectrogram. Here the content spectrogram is just random noise.

to a spectrogram, however the generated output is still very noisy. As before the corresponding audio output sounds very unnatural and noise-like.

4.8 Style Generation Example with DDIM

Theo S.

For the generation of the style-infused spectrograms, we utilize the ddim-sampling method as described in 2.1.5. By using ddim-sampling, we hope to archive better and less noisy results compared to the plain forward pass of the LDM.

For generation only, we again do not use an content-spectrogram but instead a random noise tensor and style-embeddings from another spectrogram as starting point for the sampling process. We then feed the generated spectrograms through the decoder to obtain the output spectrogram.

Results are shown in Figure 5. The resulting spectrograms show our models inability to actually generate and transfer the style characteristics in its current state. Increasing the number of sampling timesteps in the ddim-sampling further, did not yield any improvements. Also while testing multiple different input images and random seeds, the resulting spectrograms changed drastically, however the quality, and style-transfer capabilities, of them remained comparable to the ones in Figure 5.

Looking at how increasing the number of sampling timesteps in the ddim-sampling process affects the generated spectrograms, we observe that it does not seem to have a big impact on the generated output. The generated spectrograms do not seem to improve in quality or detail, but rather just change in their overall structure. This indicates problems with our ddim-sampling process or the model itself.

When listening to the resulting audio, it did have near to no meaningful content or structure and sounded very unnatural and noise-like. Representing near to no musical content. This however would be expected from the generated spectrograms in this state.

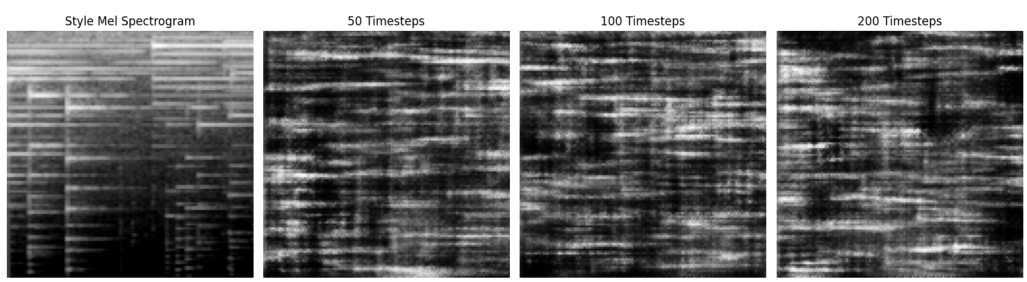

Figure 5: Style generation results. The figure displays the used spectrogram for style transfer and the spectrograms generated with different number of timesteps used in the ddim-sampling process. The LDM model was trained for 200 epochs.

4.9 Style Transfer Example with DDIM

Theo S.

Next, we also tested the style transfer capabilities of our model using the ddim-sampling method. Hoping that maybe some of the lacking generation capabilities come from the fact that we are not using a content spectrogram as input but only random noise.

Results are shown in Figure 6. For this we again use a content spectrogram as input and a style spectrogram as conditioning. Results are shown in Figure 6.

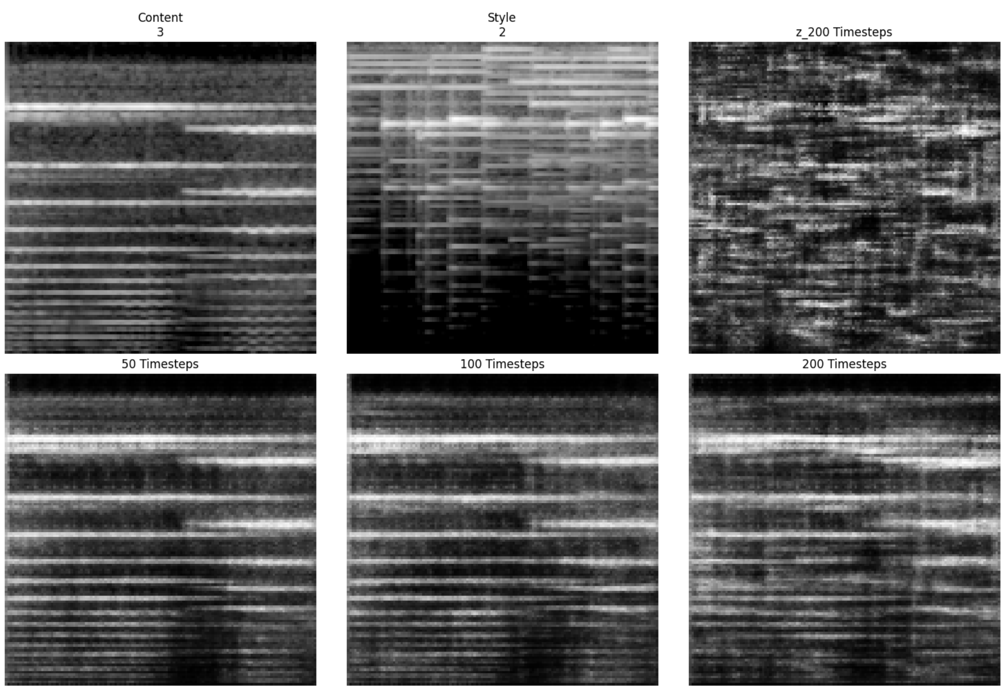

Figure 6: Style transfer results. The first row shows the content and style spectrograms used as input. For visualization we added the initial content spectrogram with its applied noise (t=200) from which we start the sampling process in the top right. The second row shows spectrograms with different number of timesteps used in the style transfer process with ddim-sampling. The LDM model was trained for 200 epochs.

While we are again not sure if the actual number of timesteps used in the sampling process even has an impact on the generated output, now the generated spectrograms have a strong resemblance to the content spectrogram. Comparing the initial z_t (top right) with the generated outputs (bottom row), we see a big change from the noisy input to the generated output.

This time, when listening to the resulting audio, we find that while it is still very noisy and has very bad quality, it does already have some meaningful content, structure and resemblance to music.

4.10 Discussion and Analysis

Andrei P.

Since our architecture was specifically designed to allow for quick training (using only a single RTX 3060 Ti GPU) and further research shows that significantly larger training time is needed (the original authors use 3 RTX 3090s), we cannot conclude if the poor results in our reconstruction or style transfer are due to the restricted architecture of a mistake in the training process. In this section we will propose our interpretation of the results and discuss possible causes.

4.10.1 Diffusion Restrictions

Theo S.

Our dataset consists of pairs of spectrograms, one for the content and one for the style. For each individual pair we generate a random noise threshold which will be passed to the forward diffusion model. Because of the limited training resources, we only sample a single noise threshold for each pair during training. Ideally, with enough data, all possible levels of noise (per spectrogram pair) should be encountered during training (both low levels as well as high levels) which would ensure that the model learns to denoise at any noise level. But because of the limited length of the dataset (only 15000 samples) there may be the case that only a small subset of possible noise thresholds (which may as well be too big) are used during the denoising process. This would be a possible cause for poor results during the DDIM generation. In order to address this issue, we augmented the dataset by generating multiple noise levels for each pair of spectrograms, but this increased the training time by a factor up to 100, which could not be tested.

4.10.2 DDIM Issues

Theo S.
This may be the reason why the resolution of the generated image is not better with increasing DDIM time steps. DDIM uses an iterative denoising process where each iteration improves the resolution of the previous iteration. If the model is not trained to denoise at all noise levels, this will result in the same low quality output for all generated images.

5 Conclusion

5.1 Achievements

Andrei P.
We remind the reader that we defined our goals for this project as follows:

- **To implement a method for musical style transfer and conditional generation using latent diffusion models.**: In this regard our results demonstrate the potential of LDMs for audio style transfer and provide a foundation for future research. While there are limitations, and that is mainly due to the complex training and computational requirements, we believe the existing unmodified code based would provide better results with significantly more training (we estimate that around 60 GPU hours of A100 would meet the requirement).

- **To show that the method works for a variety of musical styles and instruments.**: The style transfer capabilities of our model remain unclear (*that is we the style conditioned generated spectrograms look blurry (see Figures X and Y) and that can be attributed both to insufficient denoising and to poor interpolation of style, which may suggest the style network is not learning relevant style characteristics*). Since the produced results may be invalid because of the limited training, we cannot indicate for certain that the style transfer is not successful. We do believe that the style transfer may be problematic, and this is illustrated by the almost constant style loss, but further investigation is required.

- **To show that the model is able to understand tonal characteristics of different instruments.**: This objective was not achieved, we were not able to assess in any way the tonal characteristics of the generated spectrograms or see any noticeable difference between the generated spectrograms of different instruments.

We tackled a new problem with little available resources, which combines both audio and image processing. We were thus able to expand our knowledge in both fields which we consider a success of this project. While there are limitations and areas for improvement, the results demonstrate the potential of LDMs for audio style transfer and provide a foundation for future research in this direction.

5.2 Future Work

Andrei P.
Given more resources, we would like to explore the following directions:

- **Increase the number of parameters and training time**: The current model is relatively small, having only 12M parameters. We wish to try a bigger model, but that would require even more training time.

- **Augment the dataset**: At this point a random noise level is generated per each sample of the dataset, hoping that our model will learn to denoise at all noise levels. We considered augmenting the dataset, by training the model for each sample at multiple noise levels, but this increased the training effort substantially (by a factor of 100). This was not feasible given our previous resources.

- **Different style loss**: We would research for different style loss functions, and try to find pretrained models that work better on spectrograms.

- **Changes to noise scheduler and embedding**: We would experiment with a different noise scheduler and embedding mechanism to see if it improves results.

- **Embed labels into the loss**: Since we have information about the instrument being played, we could perhaps embed this into the loss calculation, in the same way they do in CLIPs[2]. This could increase the capability of the model of style transfer.

References

[1] Ffmpeg. Available at `https://ffmpeg.org`, last accessed: March 24, 2025.

[2] Librosa. Available at `https://github.com/librosa/librosa`, last accessed: March 24, 2025.

[3] yt-dlp. Available at `https://github.com/yt-dlp/yt-dlp`, last accessed: March 24, 2025.

[4] Gino Brunner, Andres Konrad, Yuyi Wang, and Roger Wattenhofer. Midi-vae: Modeling dynamics and instrumentation of music with applications to style transfer. *ArXiv*, abs/1809.07600, 2018.

[5] Gino Brunner, Yuyi Wang, Roger Wattenhofer, and Sumu Zhao. Symbolic music genre transfer with cyclegan. *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 786–793, 2018.

[6] Hong Huang, Yuyi Wang, Luyao Li, and Jun Lin. Music style transfer with diffusion model. *arXiv preprint arXiv:2404.14771*, 2024.

[7] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. *International Conference on Machine Learning*, pages 8748–8763, 2021.

[8] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[2]Learning Transferable Visual Models From Natural Language Supervision [7]

[9] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *International Conference on Learning Representations*, 2021.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[11] YouTube. Available at `https://www.youtube.com`, last accessed: March 24, 2025.