soundsafe.ai

Department of Research & Development

SoundSafe.ai Detailed Task Sheet with Macro, Sub & Micro Tasks

Audio Watermark Model

1.1 Macro Task: Develop Perceptual Branch

- **Subtask 1.1.1: Implement Psychoacoustic Analyzer**

  - **Micro Task 1.1.1.1:**

    Implement the Moore-Glasberg model in code, optimizing for real-time performance.

    Utilize a library like `psychoacoustics` in Python for a pre-built implementation of the Moore-Glasberg model. Optimize for real-time performance by leveraging NumPy for efficient array operations and consider using Just-In-Time (JIT) compilation with libraries like Numba to speed up execution.

  - **Micro Task 1.1.1.2:**

    Develop a module to calculate masking thresholds for different frequency bands based on the implemented Moore-Glasberg model.

    After obtaining the global masking threshold from the Moore-Glasberg model, divide the audio spectrum into critical bands (e.g., using Bark or ERB scales with `librosa.filters.bark_frequencies` or similar functions). Calculate the masking threshold for each band by considering the spreading function of the masker within the critical band structure.

  - **Micro Task 1.1.1.3:**

    Integrate the analyzer with the watermark embedding process.

    Modify the watermark embedding function to accept masking thresholds as input. The function should scale the watermark signal's power in each frequency band, ensuring it remains below the corresponding masking threshold to maintain imperceptibility.

- **Subtask 1.1.2: Design Adaptive Watermark Embedding**

  - **Micro Task 1.1.2.1:**

    Develop an algorithm to adjust watermark strength based on masking thresholds and the desired Signal-to-Noise Ratio (SNR) of 38 dB.

    An iterative algorithm can be employed:

    - Initialize the watermark with low power across all frequencies.
    - Calculate the SNR of the watermarked audio.
    - If the SNR is below 38 dB, increase the watermark power in each band proportionally to the masking threshold until the target SNR is reached or the masking threshold is met.

  - **Micro Task 1.1.2.2:**

    Implement the algorithm in code, ensuring efficient embedding and extraction.

    Utilize the Short-Time Fourier Transform (STFT) (`librosa.stft`) for efficient frequency-domain manipulation during embedding. For extraction, implement a correlation-based detector that compares the potentially watermarked signal with the original watermark pattern in the frequency domain.

  - **Micro Task 1.1.2.3:**

    Test and refine the algorithm using various audio samples and attack scenarios.

    Create a diverse test set with various audio genres (speech, music, environmental sounds). Simulate real-world attacks using libraries like `audiomentations` for noise injection, compression (`pydub`), filtering (`scipy.signal`), and resampling. Evaluate the watermark detection accuracy after each attack and adjust the embedding algorithm accordingly.

- **Subtask 1.1.3: Incorporate Perceptual Loss Functions**

    - **Micro Task 1.1.3.1:**

      Implement the Mel-frequency cepstral coefficients (MFCCs)-based loss function in the training process.

      Extract MFCCs from both the original and watermarked audio using `librosa.feature.mfcc`. Calculate the Mean Squared Error (MSE) between the two sets of MFCCs to quantify the perceptual difference. Include this MSE as a component in the overall loss function during model training.
    - **Micro Task 1.1.3.2:**

      Experiment with different weights for the MFCC-based perceptual loss and the robustness loss to find the optimal balance between imperceptibility and robustness.

      Start with equal weights for both loss components. If the watermark is too perceptible, increase the weight of the perceptual loss. If the watermark is not robust enough, increase the weight of the robustness loss. Fine-tune these weights based on listening tests and robustness evaluations.

1.2 Macro Task: Develop Technical Branch

- **Subtask 1.2.1: Implement Error Correction Codes**

    - **Micro Task 1.2.1.1:**

      Implement Bose-Chaudhuri-Hocquenghem (BCH) codes in code.

      Utilize a library like `reedsolo` in Python for encoding and decoding BCH codes. This library provides functions for generating BCH code generators and performing encoding and decoding operations.
    - **Micro Task 1.2.1.2:**

      Determine optimal BCH code parameters (code rate, block length) based on robustness requirements.

      The code rate determines the amount of redundancy added for error correction, while the block length affects the computational complexity. Experiment with different code rates and block lengths, evaluating the trade-off between error correction capability and the overhead introduced by the code. Consider the types of attacks the watermark needs to withstand and adjust the parameters accordingly.
    - **Micro Task 1.2.1.3:**

      Integrate the BCH encoder and decoder into the watermark embedding and extraction processes.

      Encode the watermark bits using the BCH encoder before embedding them in the audio. During extraction, use the BCH decoder to correct any errors that may have occurred due to attacks or distortions.

- **Subtask 1.2.2: Design Adaptive Bit Allocation**

    - **Micro Task 1.2.2.1:**

      Develop a perceptual significance metric for different frequency bands based on the psychoacoustic model.

      Use the output of the psychoacoustic model (e.g., masking thresholds) to create a perceptual significance metric. Assign higher significance to frequency bands where the masking threshold is higher, as these bands can tolerate more watermark information without being perceived.
    - **Micro Task 1.2.2.2:**

      Design an algorithm to dynamically allocate watermark bits based on the significance metric.

      Allocate more watermark bits to frequency bands with higher perceptual significance. This ensures that the most important information is embedded in the most robust parts of the audio signal.
    - **Micro Task 1.2.2.3:**

      Implement the algorithm in code and integrate it with the watermark embedding process.

      Modify the watermark embedding function to incorporate the adaptive bit allocation algorithm. This function should distribute the watermark bits across different frequency bands based on their perceptual significance.

1.3 Macro Task: Construct and Augment Dataset

- **Subtask 1.3.1: Gather Diverse Audio Recordings**

  - **Micro Task 1.3.1.1:**
    Collect music samples from various genres and sources, including the Free Music Archive (FMA).
    Aim for a balanced representation of different instruments, tempos, and dynamic ranges.
  - **Micro Task 1.3.1.2:**
    Gather speech recordings from different speakers, languages, and accents.
    Include diverse speakers in terms of gender, age, and accent to ensure the model generalizes well.
  - **Micro Task 1.3.1.3:**
    Compile a library of sound effects.
    Include a variety of environmental sounds (rain, wind, traffic), animal sounds, and other common sound effects.
  - **Micro Task 1.3.1.4:**
    Record real-world audio in diverse environments.
    Include audio recordings in different acoustic environments, such as quiet rooms, busy streets, and concert halls. This is to ensure variable distances and angles which introduce different levels of reverberation and background noise.

- **Subtask 1.3.2: Implement Data Augmentation Techniques**

  - **Micro Task 1.3.2.1:**
    Add white, pink, and brown noise at various SNRs to the audio samples.
    Use libraries like `audiomentations` or `numpy` to generate and add different types of noise to the audio. Vary the Signal-to-Noise Ratio (SNR) to simulate different noise levels.
  - **Micro Task 1.3.2.2:**
    Apply low-pass, high-pass, band-pass, and notch filtering.
    Use `scipy.signal` to design and apply various filters. Experiment with different cutoff frequencies and filter orders to simulate different frequency response modifications.
  - **Micro Task 1.3.2.3:**
    Implement dynamic range compression with various compression ratios and attack/release times.
    Use libraries like `pydub` to apply dynamic range compression. Experiment with different compression ratios and attack/release times to simulate different compression effects.
  - **Micro Task 1.3.2.4:**
    Perform upsampling and downsampling to different sampling rates.
    Use `librosa.resample` to change the sampling rate of the audio. Experiment with different resampling methods (e.g., linear, cubic) and target sampling rates.
  - **Micro Task 1.3.2.5:** Apply MP3, AAC, and Opus codecs at various bitrates for lossy compression.
    Use libraries like `pydub` to encode and decode audio using different codecs. Experiment with different bitrates to simulate different levels of lossy compression.

1.4 Macro Task: Design and Execute Training Schedule

- **Subtask 1.4.1: Implement Curriculum Learning**
    - **Micro Task 1.4.1.1:**

      Refer to defined curriculum for gradually increasing the difficulty of training data, starting with clean audio and progressively introducing noise, distortion, and other artifacts.

      Start by training the model on clean audio samples. Gradually introduce increasingly challenging examples, such as audio with noise, compression, filtering, and other distortions. This can be done by adjusting the augmentation parameters over time or by introducing new, more challenging datasets.
    - **Micro Task 1.4.1.2:**

      Implement the curriculum in the training process.

      Structure the training process into different stages, with each stage using a different set of data and augmentation parameters. Gradually increase the difficulty of the training data as the model progresses through the stages.
- **Subtask 1.4.2: Set up Continuous Training Pipeline**
    - **Micro Task 1.4.2.1:**

      Develop a system for automatically collecting and labeling new data.

      Creating a system for users to submit audio samples, with incentives for contributing high-quality data.
    - **Micro Task 1.4.2.2:**

      Schedule regular retraining of the model on the updated dataset, considering retraining every 48 hours.

      Set up a pipeline that automatically collects new data, preprocesses it, and retrains the model on a 48 hour basis. Monitor the model's performance on a held-out validation set to track its progress and identify any potential issues.
- **Subtask 1.4.3: Establish Model Update Schedule**
    - **Micro Task 1.4.3.1:**

      Define a release schedule for model updates.

      Consider factors like the frequency of new data, the rate of model improvement, and user feedback when determining the release schedule. Aim for a balance between frequent updates to incorporate new improvements and stability to avoid disrupting users.
    - **Micro Task 1.4.3.2:**

      Establish a process for incorporating new features and improvements into the model.

      Implement a thorough testing process to ensure that new features and improvements do not introduce bugs or regressions.

1.5 Macro Task: Allocate and Manage Compute Resources

- **Subtask 1.5.1: Set up Cloud Infrastructure**
    - **Micro Task 1.5.1.1:**

      Utilize Microsoft Azure cloud resources (virtual machines, storage).
    - **Micro Task 1.5.1.2:**

      Set up a secure and scalable infrastructure for training and deployment.

- **Subtask 1.5.2: Optimize GPU Usage**

    - **Micro Task 1.5.2.1:**

      Select appropriate GPU instances for training and inference, considering NVIDIA A100 GPUs.
    - **Micro Task 1.5.2.2:**

      Optimize code for efficient GPU utilization.

      Use techniques like data parallelism, model parallelism, and mixed precision training to maximize GPU usage. Profile your code to identify bottlenecks and optimize them for improved performance.

- **Subtask 1.5.3: Ensure Scalability**

    - **Micro Task 1.5.3.1:**

      Implement distributed training techniques using PyTorch's distributed training capabilities.
    - **Micro Task 1.5.3.2:**

      Optimize the inference pipeline for low latency and high throughput.

      Optimize the model for inference by using techniques like model quantization and pruning. Implement efficient data loading and preprocessing pipelines. Use caching and load balancing to handle high request volumes and minimize latency.