**Task: Train a Non-Hindi Language Model using the UCLA Data Corpus with VITS**

# Training Non-Hindi Language Models with VITS

## Table of Contents

1. [Introduction](#introduction)
2. [Theoretical Foundations](#theoretical-foundations)
3. [System Requirements and Setup](#system-requirements-and-setup)
4. [Data Processing and Preparation](#data-processing-and-preparation)
5. [Model Architecture and Configuration](#model-architecture-and-configuration)
6. [Training Process](#training-process)
7. [Model Deployment](#model-deployment)
8. [Troubleshooting and Optimization](#troubleshooting-and-optimization)
9. [Contributing Guidelines](#contributing-guidelines)
10. [References and Citations](#references-and-citations)

## 1. Introduction

### 1.1 Project Overview

This project implements a Text-to-Speech (TTS) synthesis system using the VITS (Conditional Variational Autoencoder with Adversarial Learning) architecture, specifically designed for training on non-Hindi language datasets from the ULCA corpus.

### 1.2 Key Features

* End-to-end TTS model training
* Support for custom dataset integration
* Real-time audio conversion capabilities
* WebSocket support for streaming applications
* FLAC audio output format

## 2. Theoretical Foundations

### 2.1 VITS Architecture Overview

#### 2.1.1 Conditional Variational Autoencoder (CVAE)

VITS builds upon a CVAE framework with three main components:

1. **Prior Encoder**
   * Purpose: Maps text input to prior distribution parameters (μ, σ)
   * Implementation:
     + Normalizing flows for expressive prior
     + Transformer-based sequence modeling
   * Mathematical foundation:
   * p(z|x) = N(μ(x), σ(x)²)
2. **Posterior Encoder**
   * Function: Encodes audio into posterior distribution
   * Benefits:
     + Robust latent representation learning
     + Text-speech consistency maintenance
   * Implementation details:
   * posterior\_dist = encoder(audio\_input)  
     z = posterior\_dist.rsample()
3. **Flow-based Decoder**
   * Purpose: Converts latent variables to acoustic features
   * Technical aspects:
     + Invertible mapping through normalizing flows
     + Exact likelihood computation
     + Deterministic transformation

### 2.2 Adversarial Learning Components

* Multiple discriminators for different time scales
* Feature matching loss implementation
* Mel-spectrogram discrimination techniques

## 3. System Requirements and Setup

### 3.1 Hardware Requirements

* CUDA-compatible GPU (recommended)
* Minimum 16GB RAM
* 50GB available storage space

### 3.2 Software Dependencies

# Install required packages  
pip install torch torchaudio librosa numpy tqdm unidecode phonemizer tensorboard  
pip install matplotlib pyyaml  
pip install monotonic-align  
pip install aiohttp websockets

### 3.3 Initial Setup

1. Clone the repository:

git clone https://github.com/jaywalnut310/vits.git  
cd vits\_model

1. Configure Monotonic Align module:

def fix\_monotonic\_align():  
 vits\_path = os.path.abspath('vits\_model')  
 monotonic\_path = os.path.join(vits\_path, 'monotonic\_align')  
   
 # Create directory structure  
 monotonic\_align\_dir = os.path.join(monotonic\_path, 'monotonic\_align')  
 os.makedirs(monotonic\_align\_dir, exist\_ok=True)  
   
 # Setup files and build  
 source\_files = ['core.c', 'core.pyx']  
 for file in source\_files:  
 src = os.path.join(monotonic\_path, file)  
 dst = os.path.join(monotonic\_align\_dir, file)  
 if os.path.exists(src):  
 shutil.copy2(src, dst)  
   
 # Create initialization files  
 with open(os.path.join(monotonic\_path, '\_\_init\_\_.py'), 'w') as f:  
 f.write('from .monotonic\_align.core import maximum\_path\_c\n')

## 4. Data Processing and Preparation

### 4.1 Audio Signal Processing Theory

#### 4.1.1 Spectral Analysis

* Mel-spectrogram generation principles
* Frequency domain conversion
* Human perception modeling

#### 4.1.2 Implementation

def process\_audio(data\_path):  
 for file in os.listdir(data\_path):  
 if file.endswith(".wav"):  
 # Load audio with theoretical sampling rate  
 audio, sr = librosa.load(os.path.join(data\_path, file), sr=22050)  
   
 # Convert to tensor with proper dimensionality  
 audio\_tensor = torch.tensor(audio).unsqueeze(0)  
   
 # Save processed audio  
 torchaudio.save(os.path.join(data\_path, file), audio\_tensor, sr)

### 4.2 Dataset Organization

#### 4.2.1 Split Theory

* Training set (80%): Pattern learning and feature extraction
* Validation set (10%): Generalization monitoring
* Test set (10%): Unbiased evaluation

#### 4.2.2 Implementation

def split\_dataset(data\_path):  
 split\_paths = {  
 'train': os.path.join(data\_path, 'train'),  
 'val': os.path.join(data\_path, 'val'),  
 'test': os.path.join(data\_path, 'test')  
 }  
   
 # Create directories  
 for split in split\_paths.values():  
 os.makedirs(split, exist\_ok=True)  
   
 # Split files  
 files = [f for f in os.listdir(data\_path) if f.endswith('.wav')]  
 random.shuffle(files)  
   
 train\_split = int(0.8 \* len(files))  
 val\_split = int(0.9 \* len(files))  
   
 # Move files to respective directories  
 for i, file in enumerate(files):  
 source = os.path.join(data\_path, file)  
 if i < train\_split:  
 destination = split\_paths['train']  
 elif i < val\_split:  
 destination = split\_paths['val']  
 else:  
 destination = split\_paths['test']  
 shutil.move(source, os.path.join(destination, file))

## 5. Model Architecture and Configuration

### 5.1 Model Components

#### 5.1.1 Transformer Encoder

model = SynthesizerTrn(  
 num\_vocab=len(voc),  
 spec\_channels=80,  
 segment\_size=512,  
 inter\_channels=192,  
 hidden\_channels=192,  
 filter\_channels=768,  
 n\_heads=2,  
 n\_layers=6,  
 kernel\_size=5,  
 p\_dropout=0.1,  
 resblock='1',  
 resblock\_kernel\_sizes=[3,7,11],  
 resblock\_dilation\_sizes=[[1,3,5],[1,3,5],[1,3,5]],  
 upsample\_rates=[8,8,2,2],  
 upsample\_initial\_channel=512,  
 upsample\_kernel\_sizes=[16,16,4,4],  
 gin\_channels=0,  
 n\_speakers=1  
)

#### 5.1.2 Parameter Explanation

* spec\_channels: Matches mel-spectrogram dimensions
* segment\_size: Balances context and memory
* n\_heads: Enables parallel attention computation
* n\_layers: Determines model capacity
* filter\_channels: Controls information bottleneck
* kernel\_size: Affects receptive field
* p\_dropout: Prevents overfitting

### 5.2 Configuration Setup

def setup\_config():  
 config\_path = "/content/vits\_model/configs/ljs\_base.json"  
 with open(config\_path, "r") as f:  
 config = json.load(f)  
   
 # Modify configuration  
 config['data']['training\_files'] = 'dataset/train'  
 config['data']['validation\_files'] = 'dataset/val'  
 config['model']['n\_speakers'] = 1  
 config['train']['batch\_size'] = 16  
   
 # Save updated configuration  
 with open(config\_path, "w") as f:  
 json.dump(config, f)

## 6. Training Process

### 6.1 Loss Functions

#### 6.1.1 Multi-Resolution STFT Loss

class MultiResolutionSTFTLoss(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.scales = [(2048, 512), (1024, 256), (512, 128)]  
   
 def forward(self, y\_pred, y\_true):  
 magnitude\_loss = 0  
 complex\_loss = 0  
   
 for scale in self.scales:  
 mag\_l, comp\_l = self.stft\_loss(y\_pred, y\_true, scale[0], scale[1])  
 magnitude\_loss += mag\_l  
 complex\_loss += comp\_l  
   
 return magnitude\_loss, complex\_loss

#### 6.1.2 Adversarial Loss Components

* Discriminator loss implementation
* Generator adversarial loss
* Feature matching mechanisms

### 6.2 Training Implementation

def train\_model():  
 # Initialize data loaders  
 train\_loader = data.DataLoader(  
 train\_dataset,  
 batch\_size=32,  
 shuffle=True,  
 num\_workers=4  
 )  
   
 val\_loader = data.DataLoader(  
 val\_dataset,  
 batch\_size=32,  
 shuffle=False,  
 num\_workers=4  
 )  
   
 # Training command  
 command = [  
 "python",  
 "train.py",  
 "-c",  
 "configs/ljs\_base.json",  
 "-m",  
 "my\_vits\_model"  
 ]  
 subprocess.run(command, cwd="/content/vits\_model")

## 7. Model Deployment

### 7.1 API Implementation

async def convert\_to\_flac(request):  
 data = await request.post()  
 wav\_file = data['wav\_file']  
 flac\_file = convert\_wav\_to\_flac(wav\_file)  
 return aiohttp.web.Response(  
 body=flac\_file,  
 content\_type='audio/flac'  
 )

### 7.2 WebSocket Integration

async def stream\_audio(websocket, path):  
 async for message in websocket:  
 flac\_audio = await process\_audio\_stream(message)  
 await websocket.send(flac\_audio)  
  
async def main():  
 app = aiohttp.web.Application()  
 app.add\_routes([  
 aiohttp.web.post('/convert', convert\_to\_flac)  
 ])  
 runner = aiohttp.web.AppRunner(app)  
 await runner.setup()  
 site = aiohttp.web.TCPSite(  
 runner,  
 'localhost',  
 8080  
 )  
 await site.start()

## 8. Troubleshooting and Optimization

### 8.1 Common Issues

1. **Monotonic Align Import Error**
   * Solution: Run fix\_monotonic\_align()
   * Verify Python path
2. **CUDA Out of Memory**
   * Reduce batch size
   * Implement gradient accumulation
   * Enable mixed precision training
3. **Audio Processing Errors**
   * Verify sample rate consistency
   * Check for corrupt files
   * Validate format conversions

### 8.2 Performance Optimization

1. **Memory Management**
   * Gradient accumulation implementation
   * Mixed precision training setup
   * Batch size optimization
2. **Computational Efficiency**
   * CUDA optimization techniques
   * Data loading optimization
   * Resource monitoring

## 9. Contributing Guidelines

### 9.1 Development Process

1. Fork the repository
2. Create feature branch
3. Follow code style guidelines
4. Add unit tests
5. Submit pull request

### 9.2 Code Style

* Follow PEP 8
* Use descriptive variable names
* Add comprehensive documentation
* Include docstrings

## 10. References

### 10.1 Additional Resources

* VITS GitHub Repository
* ULCA Dataset Documentation
* PyTorch Documentation
* Librosa Documentation