Fine-tune two text-to-speech (TTS) models

Overview of Text-to-Speech (TTS)

Text-to-Speech (TTS) is a technology that converts written text into spoken voice output. It is a branch of artificial intelligence and computational linguistics, leveraging deep learning models, signal processing techniques, and linguistic rules to generate synthetic speech from text inputs. TTS systems are used in various applications where voice-based interaction or accessibility is required, and their sophistication has improved drastically in recent years, creating more human-like and natural-sounding voices.

Components of TTS:

1. **Text Processing (Front-End):**
    - The system processes input text, normalizes it, and converts it into phonetic or linguistic representations, such as phonemes. This stage also involves language-specific rules, pronunciation handling, prosody analysis (intonation, rhythm, and stress), and sometimes semantic analysis.
2. **Speech Synthesis (Back-End):**
    - The back-end involves converting phonetic representations into a continuous speech waveform. It can be based on traditional methods (concatenative synthesis, parametric synthesis) or modern deep learning methods (neural vocoders, sequence-to-sequence models).
3. **Vocoder:**
    - A vocoder is responsible for generating the actual waveform (sound signal) from the speech features generated in the TTS pipeline. Neural vocoders, such as **WaveNet**, **Tacotron**, or **HiFi-GAN**, have become standard in generating high-quality speech.

Applications of TTS

Text-to-Speech technology has wide-ranging applications across many industries, improving accessibility, enhancing user experience, and enabling voice-driven technologies. Below are some prominent use cases:

1. **Assistive Technology:**
    - **Accessibility for the Visually Impaired:** TTS is widely used in screen readers to help visually impaired users access digital content such as websites, eBooks, and apps by converting text to speech.
    - **Speech Disabilities:** People with speech disabilities use TTS devices for communication, allowing them to express themselves verbally using a synthesized voice.
2. **Voice Assistants:**

    - TTS is a core technology behind voice-activated assistants like **Amazon Alexa**, **Google Assistant**, and **Apple Siri**. These systems convert responses to user queries into speech, providing an interactive and natural conversational experience.
3. **Audiobooks and Podcast Creation:**
    - TTS can be used to generate audiobooks, making written content accessible to those who prefer auditory learning or entertainment. Some content creators use TTS to create podcasts from text-based articles or blog posts.
4. **Customer Support and IVR Systems:**
    - **Interactive Voice Response (IVR)** systems use TTS for automated customer service in call centers, providing users with a self-service interface that offers spoken responses.
    - **Chatbots** with integrated TTS capabilities can provide a more human-like interaction, helping customers resolve queries faster.
5. **Language Learning:**
    - TTS can be utilized in educational applications to help learners with pronunciation, intonation, and language practice, providing instant auditory feedback for written text.
6. **Gaming and Virtual Characters:**
    - TTS is used in gaming for creating dynamic voice responses for NPCs (Non-Player Characters) or virtual avatars, enriching the user experience by providing contextual voice output based on in-game events.
7. **Navigation Systems:**
    - GPS systems use TTS to give spoken directions to users, reducing distractions and making driving safer by allowing users to focus on the road.

The Importance of Fine-Tuning in TTS

Fine-tuning refers to the process of adapting a pre-trained model to a specific task, domain, or language to improve its performance. While pre-trained TTS models offer general capabilities, fine-tuning can significantly enhance the quality, naturalness, and context-appropriateness of the generated speech, especially for specialised or high-precision applications.

Reasons for Fine-Tuning in TTS:

1. **Domain-Specific Adaptation:**
    - TTS models trained on general text corpora may not perform well in specialised domains, such as medicine, law, or technical fields. Fine-tuning enables the model to learn domain-specific terminology, jargon, and linguistic patterns, resulting in more accurate and natural speech synthesis for niche applications.
2. **Accent, Dialect, and Language Variation:**
    - Different languages and dialects have unique pronunciations, intonations, and prosodic patterns. Fine-tuning the TTS model for specific accents (e.g., British vs. American English), dialects (e.g., regional variations), or entirely different

languages allows the system to generate speech that closely matches the linguistic norms of the target user base.

3. **Personalization:**
    - In certain cases, TTS systems need to generate personalised voices that match specific speakers' voice characteristics. Fine-tuning helps adjust the model to synthesise speech that mirrors an individual speaker's voice tone, pitch, and style, making it more personalised for uses such as cloned voices or character voices in gaming.
4. **Improved Prosody and Naturalness:**
    - Generic TTS models might generate speech that lacks emotional depth or appropriate intonation (prosody). Fine-tuning allows the model to better capture natural variations in speech, leading to more expressive and emotionally attuned synthetic voices. This can be essential for applications like audiobooks or virtual agents that require an engaging, human-like voice.
5. **Noise Robustness and Quality:**
    - Fine-tuning can enhance the TTS system's ability to generate high-quality speech even when the input text is noisy or contains errors. For real-world applications like voice assistants, the ability to handle imperfect input data is crucial for maintaining good user experiences.

Steps in Fine-Tuning a TTS Model

1. **Dataset Collection:**
    - The first step is to gather a high-quality, domain-specific dataset. This could include specialised vocabulary, recordings of specific accents or dialects, or even individual speaker recordings if creating a personalised voice. Text-audio pairs are used for training and evaluation.
2. **Preprocessing:**
    - The dataset must be preprocessed, including cleaning text (removing non-speech symbols), normalising phonetic representations, and aligning audio to text using techniques like forced alignment.
3. **Model Adaptation:**
    - The pre-trained model is then fine-tuned on the new dataset using transfer learning. This involves updating the model’s weights based on the new data while leveraging the knowledge the model has gained from its initial large-scale training.
4. **Fine-Tuning Prosody and Emotion:**
    - Fine-tuning is not just about phonetic accuracy. Techniques like prosody prediction and emotion modelling are incorporated to generate more expressive and natural-sounding speech.
5. **Evaluation and Iteration:**
    - After training, the fine-tuned model must be evaluated both subjectively (through listener ratings like MOS) and objectively (using metrics such as PESQ or WER). Iterative fine-tuning may be required to reach the desired quality levels.

Technical English TTS =>

```
from transformers import SpeechT5Processor, SpeechT5ForTextToSpeech,
SpeechT5HifiGan
from datasets import load_dataset
import torch
import soundfile as sf
from datasets import load_dataset

processor = SpeechT5Processor.from_pretrained("microsoft/speecht5_tts")
model =
SpeechT5ForTextToSpeech.from_pretrained("microsoft/speecht5_tts")
vocoder = SpeechT5HifiGan.from_pretrained("microsoft/speecht5_hifigan")

inputs = processor(text="A village is a small settlement typically
found in rural areas, smaller than a town but larger than a hamlet.",
return_tensors="pt")

# load xvector containing speaker's voice characteristics from a
dataset
embeddings_dataset = load_dataset("Matthijs/cmu-arctic-xvectors",
split="validation")
speaker_embeddings =
torch.tensor(embeddings_dataset[7306]["xvector"]).unsqueeze(0)

speech = model.generate_speech(inputs["input_ids"], speaker_embeddings,
vocoder=vocoder)

sf.write("speech.wav", speech.numpy(), samplerate=16000)
```

Regional Language TTS=>

```
import os
os.environ["SUNO_USE_SMALL_MODELS"] = "True"
from bark import SAMPLE_RATE, generate_audio, preload_models
from scipy.io.wavfile import write as write_wav
from IPython.display import Audio

# download and load all models
preload_models()

# generate audio from text
```

```
text_prompt = """
     गाँवों में लोग सामुदायिक जीवन जीते हैं, जहाँ पड़ोसी एक-दूसरे के सुख-दुख में भागीदार होते हैं। यहाँ परंपराएँ और रीति-रिवाज पीढ़ियों से चली आ रही हैं। लोग एक साथ त्योहार मनाते
"""
audio_array = generate_audio(text_prompt)

# save audio to disk
write_wav("bark_generation.wav", SAMPLE_RATE, audio_array)

# play text in notebook
Audio(audio_array, rate=SAMPLE_RATE)
```

Explain any challenges faced during the process=>

1. **Dataset Challenges**
   - **A. Data Quality and Alignment Issues**
   - **B. Dataset Size and Diversity**
   - **Noisy Data**
2. **Model Convergence Challenges**
   - A. **Training Instability**
   - Overfitting

Bonus Task:

```
from transformers import SpeechT5Processor, SpeechT5ForTextToSpeech, SpeechT5HifiGan
from datasets import load_dataset
import torch
import soundfile as sf
import time
import torch.nn.utils.prune as prune
from torch.quantization import quantize_dynamic

# Step 1: Load Model and Processor
processor = SpeechT5Processor.from_pretrained("microsoft/speecht5_tts")
model = SpeechT5ForTextToSpeech.from_pretrained("microsoft/speecht5_tts")
vocoder = SpeechT5HifiGan.from_pretrained("microsoft/speecht5_hifigan")

# Step 2: Prepare Input
inputs = processor(text="OOP is centred around four main concepts: encapsulation, inheritance, polymorphism, and abstraction.", return_tensors="pt")
```

```
# Load speaker embeddings
embeddings_dataset = load_dataset("Matthijs/cmu-arctic-xvectors",
split="validation")
speaker_embeddings =
torch.tensor(embeddings_dataset[7306]["xvector"]).unsqueeze(0)

# Step 3: Generate Speech Before Optimization
speech_before = model.generate_speech(inputs["input_ids"],
speaker_embeddings, vocoder=vocoder)
sf.write("myenv/speech_before.wav", speech_before.numpy(),
samplerate=16000)

# Step 4: Model Quantization
quantized_model = quantize_dynamic(model, {torch.nn.Linear},
dtype=torch.qint8)

# Step 5: Pruning the Model
# Example: Prune all linear layers in the model
for name, module in model.named_modules():
    if isinstance(module, torch.nn.Linear):
        prune.l1_unstructured(module, name='weight', amount=0.3)  #
Prune 30% of weights

# Step 6: Generate Speech After Optimization
speech_after = quantized_model.generate_speech(inputs["input_ids"],
speaker_embeddings, vocoder=vocoder)
sf.write("myenv/speech_after.wav", speech_after.numpy(),
samplerate=16000)

# Step 7: Measure Inference Time
def measure_inference_time(model, inputs, speaker_embeddings):
    start_time = time.time()
    _ = model.generate_speech(inputs["input_ids"], speaker_embeddings,
vocoder=vocoder)
    end_time = time.time()
    return end_time - start_time

# Timing inference
inference_time_before = measure_inference_time(model, inputs,
speaker_embeddings)
inference_time_after = measure_inference_time(quantized_model, inputs,
speaker_embeddings)
```

```
print(f'Inference time before optimization: {inference_time_before:.4f}
seconds')
print(f'Inference time after optimization: {inference_time_after:.4f}
seconds')

# Step 8: Save the Optimised Model
torch.save(quantized_model.state_dict(),
"optimized_speecht5_model.pth")
print("Optimised model saved.")
```

Conclusion=>

Findings and Key Takeaways

- **Data Quality and Alignment**: High-quality, diverse datasets are essential for natural speech synthesis. Misaligned or noisy data negatively affects performance.
- **Optimization Techniques**: Model optimization through **quantization** and **pruning** reduces model size and inference time, making TTS suitable for real-time applications.
- **Prosody and Naturalness**: Maintaining natural prosody remains a challenge, with many models producing flat or robotic speech due to inadequate prosody control.

Future Improvements

- **Enhanced Prosody and Emotion Control**: Improved prosody modeling and emotion embeddings can enhance the naturalness and expressiveness of synthesized speech.
- **Real-Time Deployment Optimization**: Further reductions in model latency and memory usage are needed for real-time and edge deployment.
- **Low-Resource Language Support**: Leveraging transfer learning can improve TTS models' performance for underrepresented languages.
- **Advanced Evaluation Metrics**: Developing refined evaluation methods can better assess the naturalness and expressiveness of TTS outputs, guiding further enhancements.