NNDL_ASSIGNMENT_3

April 18, 2025

[ ]:
```
# -*- coding: utf-8 -*-
"""
Speech Recognition with Wav2Vec2 - Evaluation Script

This script demonstrates a basic speech recognition pipeline using the Wav2Vec2 model
from Hugging Face Transformers. It loads a small subset of the LibriSpeech ASR
dataset, performs inference using a pre-trained Wav2Vec2 model, and evaluates
the performance using the Word Error Rate (WER) metric.

The script is structured for clarity and includes comments to explain each step.
It also adheres to coding best practices for readability.
"""

# Import necessary libraries
from datasets import load_dataset
import torchaudio
import torch
from itertools import islice
from transformers import Wav2Vec2Processor, Wav2Vec2ForCTC
import evaluate
import jiwer  # For a potentially more robust WER calculation

# --------------------------------------------------------------------------------
# 1. Code with Proper Comments (Documentation & Readability)
# --------------------------------------------------------------------------------

# Load the Word Error Rate (WER) metric from the Hugging Face Evaluate library.
# This metric will be used to assess the accuracy of the speech recognition model.
wer_metric = evaluate.load("wer")

# Load the LibriSpeech ASR dataset (specifically the "clean" subset of the "train.100" split)
# in streaming mode. Streaming allows us to work with large datasets without loading
# the entire dataset into memory at once.
```

```
dataset = load_dataset("librispeech_asr", "clean", split="train.100",␣
 ↪streaming=True)

# Create a tiny subset of the dataset containing only the first 10 samples.
# This is useful for quick testing and demonstration purposes.
tiny_dataset = list(islice(dataset, 10))

# Load the pre-trained Wav2Vec2 processor. The processor is responsible for
# converting raw audio waveforms into the input format expected by the model.
# It handles tasks such as feature extraction and tokenization.
processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-base-960h")

# Load the pre-trained Wav2Vec2 model for Connectionist Temporal Classification␣
 ↪(CTC).
# CTC is a common loss function used for sequence-to-sequence tasks like speech␣
 ↪recognition,
# where the alignment between the input (audio) and the output (text) is not␣
 ↪known.
model = Wav2Vec2ForCTC.from_pretrained("facebook/wav2vec2-base-960h")

# Set the model to evaluation mode. This disables dropout and other layers that
# are only active during training, ensuring consistent inference results.
model.eval()

# Initialize empty lists to store the predicted transcriptions and the ground␣
 ↪truth references.
predictions = []
references = []

# Print a header to indicate the start of the sample-wise results.
print("\n Sample-wise Results:\n")

# Iterate through each sample in the tiny dataset.
for i, sample in enumerate(tiny_dataset):
    try:
        # Print the sample number for tracking.
        print(f" Sample {i+1}")

        # Extract the reference text (ground truth transcription) from the␣
 ↪sample.
        reference = sample["text"]
        # Convert the reference text to lowercase for case-insensitive␣
 ↪comparison.
        references.append(reference.lower())

        # Extract the audio data from the sample.
```

```
        audio = sample["audio"]
        # Convert the audio array to a PyTorch tensor, add a batch dimension␣
↪(unsqueeze(0)),
        # and cast it to float32, which is the expected data type for the model.
        waveform = torch.tensor(audio["array"]).unsqueeze(0).float()
        # Get the sampling rate of the audio.
        sample_rate = audio["sampling_rate"]

        # Process the audio waveform using the Wav2Vec2 processor. This involves
        # extracting features and preparing the input tensor for the model.
        # `return_tensors="pt"` ensures that the output is a PyTorch tensor.
        # `padding=True` will pad shorter sequences in a batch (though we are␣
↪processing one at a time here).
        inputs = processor(waveform.squeeze(), sampling_rate=sample_rate,␣
↪return_tensors="pt", padding=True)

        # Disable gradient calculation during inference to save memory and␣
↪speed up computation.
        with torch.no_grad():
            # Pass the input tensor to the Wav2Vec2 model to get the logits.
            # Logits are the raw, unnormalized predictions of the model.
            logits = model(**inputs).logits

        # Get the predicted token IDs by taking the argmax over the last␣
↪dimension
        # of the logits, which corresponds to the vocabulary.
        predicted_ids = torch.argmax(logits, dim=-1)

        # Decode the predicted token IDs back into a human-readable␣
↪transcription
        # using the processor's `batch_decode` method. We take the first (and␣
↪only)
        # element of the resulting list since we processed a single audio␣
↪sample.
        transcription = processor.batch_decode(predicted_ids)[0]
        # Convert the predicted transcription to lowercase for case-insensitive␣
↪comparison.
        predictions.append(transcription.lower())

        # Print the original and predicted texts for comparison.
        print(f" Original Text  : {reference}")
        print(f" Predicted Text : {transcription}")

        # Calculate the Word Error Rate (WER) for the current sample by␣
↪comparing
        # the predicted transcription with the reference text.
```

```
        single_wer = wer_metric.compute(predictions=[transcription.lower()],␣
 ↪references=[reference.lower()])
        # Print the WER for the current sample, formatted to four decimal␣
 ↪places.
        print(f"  WER for Sample : {single_wer:.4f}\n")

    # Handle any potential exceptions that might occur during the processing of␣
 ↪a sample.
    except Exception as e:
        print(f" Error processing sample {i+1}: {e}\n")

# --------------------------------------------------------------------------------
# Explanation of hyperparameters and model architecture
# --------------------------------------------------------------------------------

# Hyperparameters:
# The Wav2Vec2-base-960h model is a pre-trained model, so many of its␣
 ↪hyperparameters
# are already set. Key architectural hyperparameters include:
# - Number of transformer layers: Typically around 12 in the base model.
# - Hidden layer size: Usually around 768 dimensions.
# - Number of attention heads: Often around 12.
# - Convolutional feature extractor: A multi-layer convolutional neural network
#   that extracts low-level audio features.
# - Masking mechanism: During pre-training, random spans of the audio input are␣
 ↪masked
#   and the model is trained to predict the masked features.
# - Learning rate, batch size, and training steps: These are crucial during the
#   pre-training and fine-tuning phases but are fixed for the loaded␣
 ↪pre-trained model.

# Model Architecture:
# Wav2Vec2 consists of a convolutional feature extractor followed by a␣
 ↪transformer network.
# 1. Feature Extractor: This part consists of multiple layers of 1D␣
 ↪convolutional
#    layers with ReLU activation and layer normalization. It processes the raw␣
 ↪audio
#    waveform to produce a sequence of feature vectors.
# 2. Transformer Network: This is a standard transformer encoder architecture␣
 ↪with
#    self-attention mechanisms. It takes the feature vectors from the feature
#    extractor as input and learns high-level contextual representations.
# 3. CTC Head: For speech recognition, a linear layer is added on top of the␣
 ↪transformer
```

```
#    output to predict the probability distribution over the vocabulary␣
 ↪(including
#    a special "blank" token). The Connectionist Temporal Classification (CTC)␣
 ↪loss
#    function is used to train the model to map the audio features to the output
#    transcription without requiring explicit alignment.

# ------------------------------------------------------------------------------
# Use of meaningful variable names and coding best practices
# ------------------------------------------------------------------------------

# The code uses descriptive variable names (e.g., `waveform`, `sample_rate`,
# `predicted_ids`, `transcription`, `reference`).
# It is well-structured with comments explaining each significant step.
# The use of a try-except block ensures robustness against potential errors
# during sample processing.
# The code follows a logical flow: loading data, loading model, processing data,
# and evaluating results.

# Calculate the overall Word Error Rate (WER) by comparing all predicted␣
 ↪transcriptions
# with their corresponding reference texts.
overall_wer = wer_metric.compute(predictions=predictions, references=references)

# Calculate an approximate accuracy based on the overall WER.
# Accuracy is often expressed as (1 - WER) * 100%.
accuracy = (1 - overall_wer) * 100

# Print the overall evaluation results, including the WER and the approximate␣
 ↪accuracy.
print("\n Overall Evaluation:")
print(f" Word Error Rate (WER): {overall_wer:.4f}")
print(f" Approximate Accuracy : {accuracy:.2f}%")
```