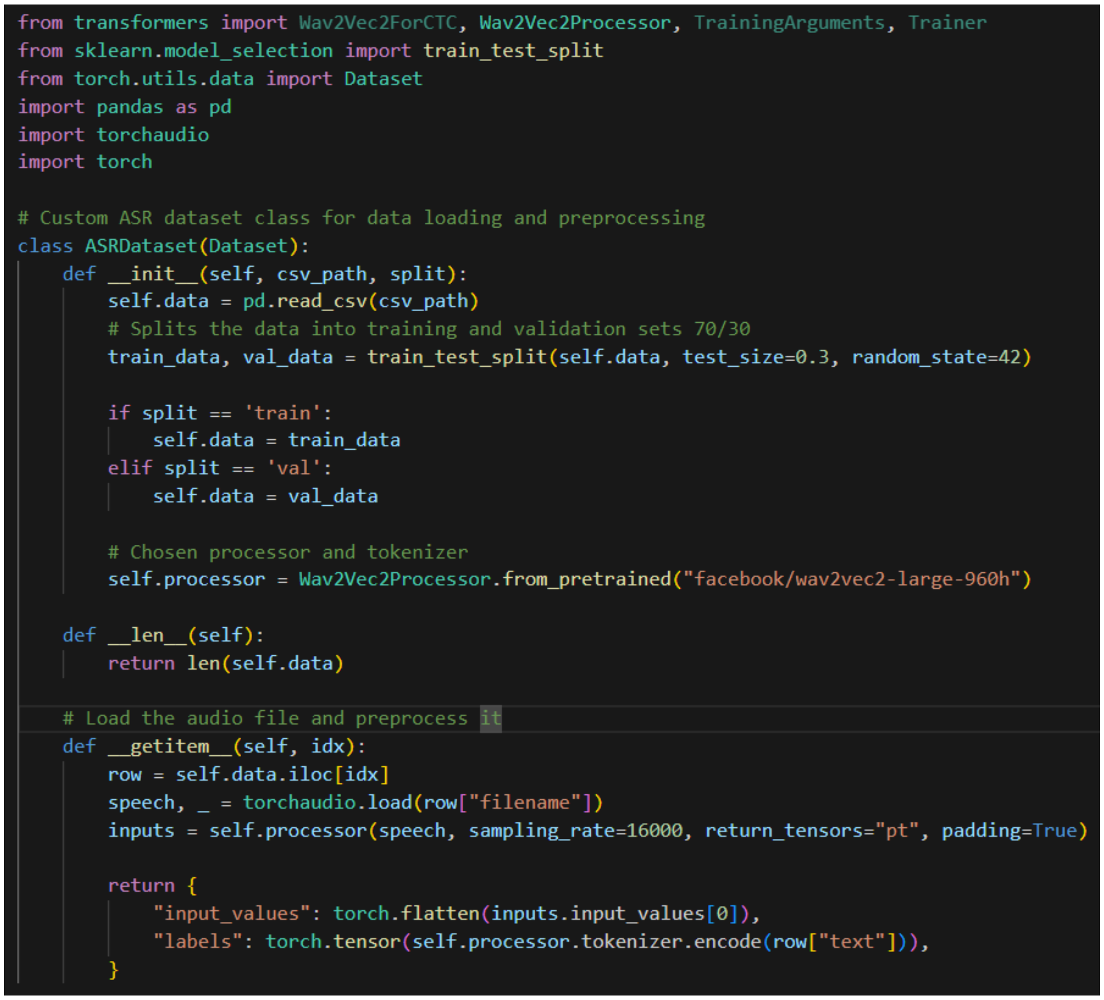
Training Report

Data Preprocessing

A custom dataset loader and preprocessing class created to load the datasets of converted audio wav files and ground truth text.

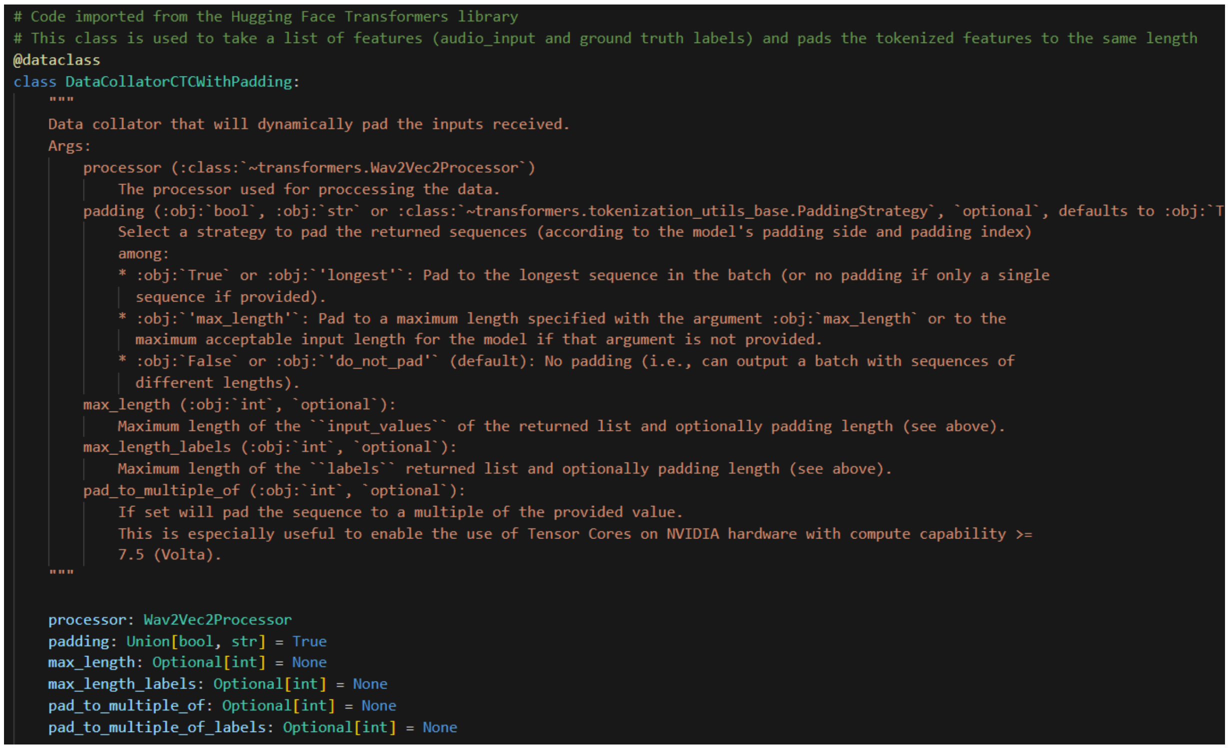
Tokenizer chosen is from the pre_trained processor “facebook/wav2vec2-large-960h".

This processor and tokenizer is chosen as it is able to normalize, extract features and tokenize in data in a way that the model is compatible with. Furthermore, we are fine-tuning our model on clean English language datasets, hence there is little need to redefine our tokenizer and vocabulary configuration.

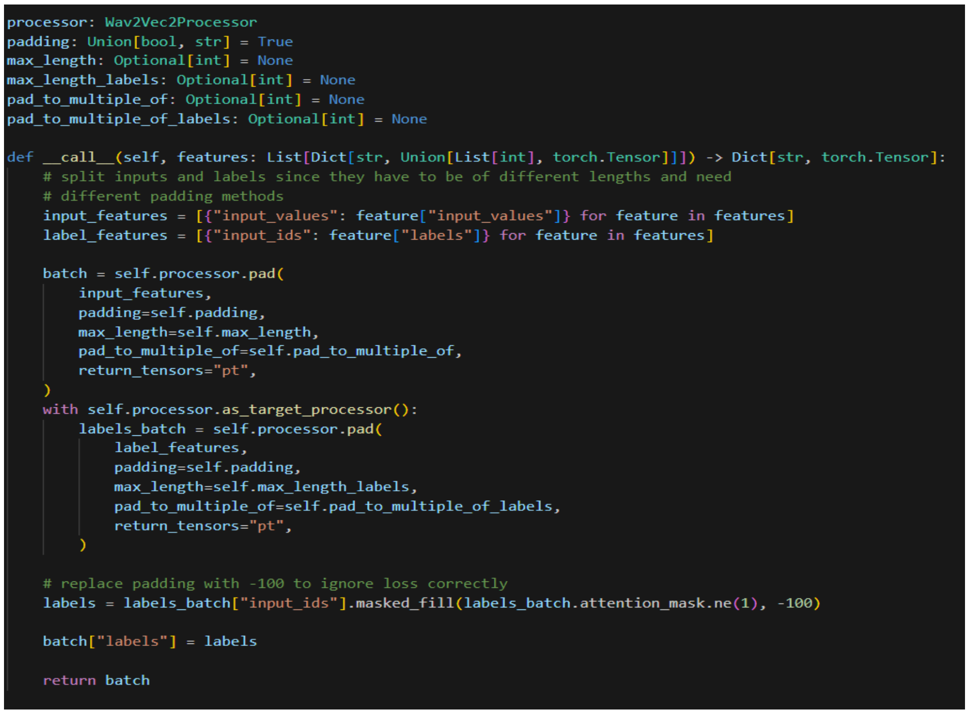


```python
from transformers import Wav2Vec2ForCTC, Wav2Vec2Processor, TrainingArguments, Trainer
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset
import pandas as pd
import torchaudio
import torch

# Custom ASR dataset class for data loading and preprocessing
class ASRDataset(Dataset):
    def __init__(self, csv_path, split):
        self.data = pd.read_csv(csv_path)
        # Splits the data into training and validation sets 70/30
        train_data, val_data = train_test_split(self.data, test_size=0.3, random_state=42)

        if split == 'train':
            self.data = train_data
        elif split == 'val':
            self.data = val_data

        # Chosen processor and tokenizer
        self.processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-large-960h")

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

    # Load the audio file and preprocess it
    def __getitem__(self, idx):
        row = self.data.iloc[idx]
        speech, _ = torchaudio.load(row["filename"])
        inputs = self.processor(speech, sampling_rate=16000, return_tensors="pt", padding=True)

        return {
            "input_values": torch.flatten(inputs.input_values[0]),
            "labels": torch.tensor(self.processor.tokenizer.encode(row["text"])),
        }
```

This batch of code is taken from HuggingFace example. The purpose of this class is to pad the tokenized input values (audio) and label (ground truth text) to have the same sequence length.



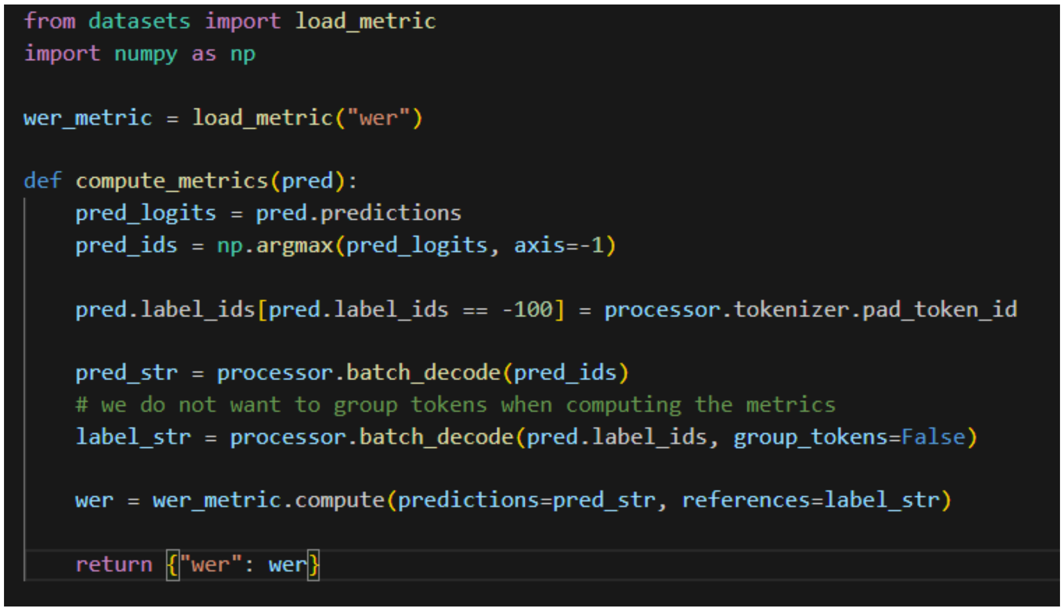
```python
# Code imported from the Hugging Face Transformers library
# This class is used to take a list of features (audio_input and ground truth labels) and pads the tokenized features to the same length
@dataclass
class DataCollatorCTCWithPadding:
    """
    Data collator that will dynamically pad the inputs received.
    Args:
        processor (:class:`~transformers.Wav2Vec2Processor`)
            The processor used for proccessing the data.
        padding (:obj:`bool`, :obj:`str` or :class:`~transformers.tokenization_utils_base.PaddingStrategy`, `optional`, defaults to :obj:`T
            Select a strategy to pad the returned sequences (according to the model's padding side and padding index)
            among:
            * :obj:`True` or :obj:`'longest'`: Pad to the longest sequence in the batch (or no padding if only a single
              sequence if provided).
            * :obj:`'max_length'`: Pad to a maximum length specified with the argument :obj:`max_length` or to the
              maximum acceptable input length for the model if that argument is not provided.
            * :obj:`False` or :obj:`'do_not_pad'` (default): No padding (i.e., can output a batch with sequences of
              different lengths).
        max_length (:obj:`int`, `optional`):
            Maximum length of the ``input_values`` of the returned list and optionally padding length (see above).
        max_length_labels (:obj:`int`, `optional`):
            Maximum length of the ``labels`` returned list and optionally padding length (see above).
        pad_to_multiple_of (:obj:`int`, `optional`):
            If set will pad the sequence to a multiple of the provided value.
            This is especially useful to enable the use of Tensor Cores on NVIDIA hardware with compute capability >=
            7.5 (Volta).
    """

    processor: Wav2Vec2Processor
    padding: Union[bool, str] = True
    max_length: Optional[int] = None
    max_length_labels: Optional[int] = None
    pad_to_multiple_of: Optional[int] = None
    pad_to_multiple_of_labels: Optional[int] = None
```



```python
processor: Wav2Vec2Processor
padding: Union[bool, str] = True
max_length: Optional[int] = None
max_length_labels: Optional[int] = None
pad_to_multiple_of: Optional[int] = None
pad_to_multiple_of_labels: Optional[int] = None

def __call__(self, features: List[Dict[str, Union[List[int], torch.Tensor]]]) -> Dict[str, torch.Tensor]:
    # split inputs and labels since they have to be of different lengths and need
    # different padding methods
    input_features = [{"input_values": feature["input_values"]} for feature in features]
    label_features = [{"input_ids": feature["labels"]} for feature in features]

    batch = self.processor.pad(
        input_features,
        padding=self.padding,
        max_length=self.max_length,
        pad_to_multiple_of=self.pad_to_multiple_of,
        return_tensors="pt",
    )
    with self.processor.as_target_processor():
        labels_batch = self.processor.pad(
            label_features,
            padding=self.padding,
            max_length=self.max_length_labels,
            pad_to_multiple_of=self.pad_to_multiple_of_labels,
            return_tensors="pt",
        )

    # replace padding with -100 to ignore loss correctly
    labels = labels_batch["input_ids"].masked_fill(labels_batch.attention_mask.ne(1), -100)

    batch["labels"] = labels

    return batch
```

Evaluation Metric

For the purpose of this model fine-tuning, Word Error Rate (WER) is chosen to be a metric in evaluating the ratio of incorrect words by the model's inference to the ground truth.

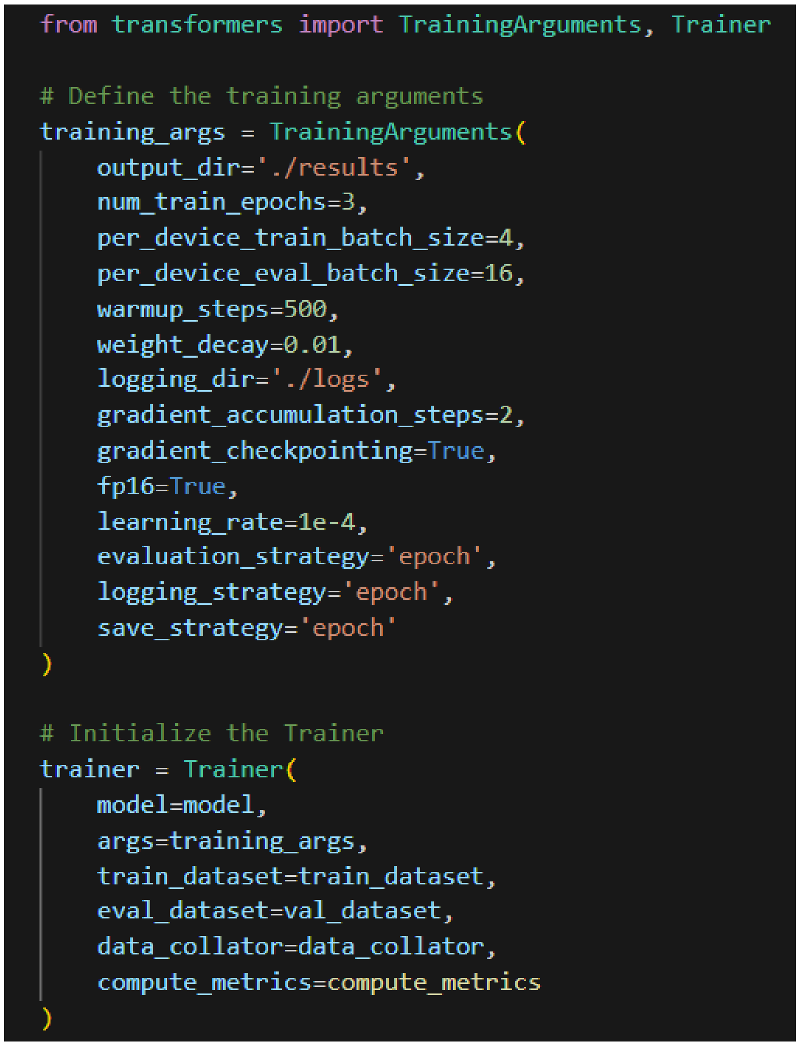


```python
from datasets import load_metric
import numpy as np

wer_metric = load_metric("wer")

def compute_metrics(pred):
    pred_logits = pred.predictions
    pred_ids = np.argmax(pred_logits, axis=-1)

    pred.label_ids[pred.label_ids == -100] = processor.tokenizer.pad_token_id

    pred_str = processor.batch_decode(pred_ids)
    # we do not want to group tokens when computing the metrics
    label_str = processor.batch_decode(pred.label_ids, group_tokens=False)

    wer = wer_metric.compute(predictions=pred_str, references=label_str)

    return {"wer": wer}
```

Training Arguments and Hyperparameter

A trainer class from HuggingFace's transformers is used to intialize hyperparameters of training arguments. Due to the lack of computation power of my device, the training epoch is set to 3 and a small training batch size of 4 is chosen, this greatly reduced the load on my GPU.

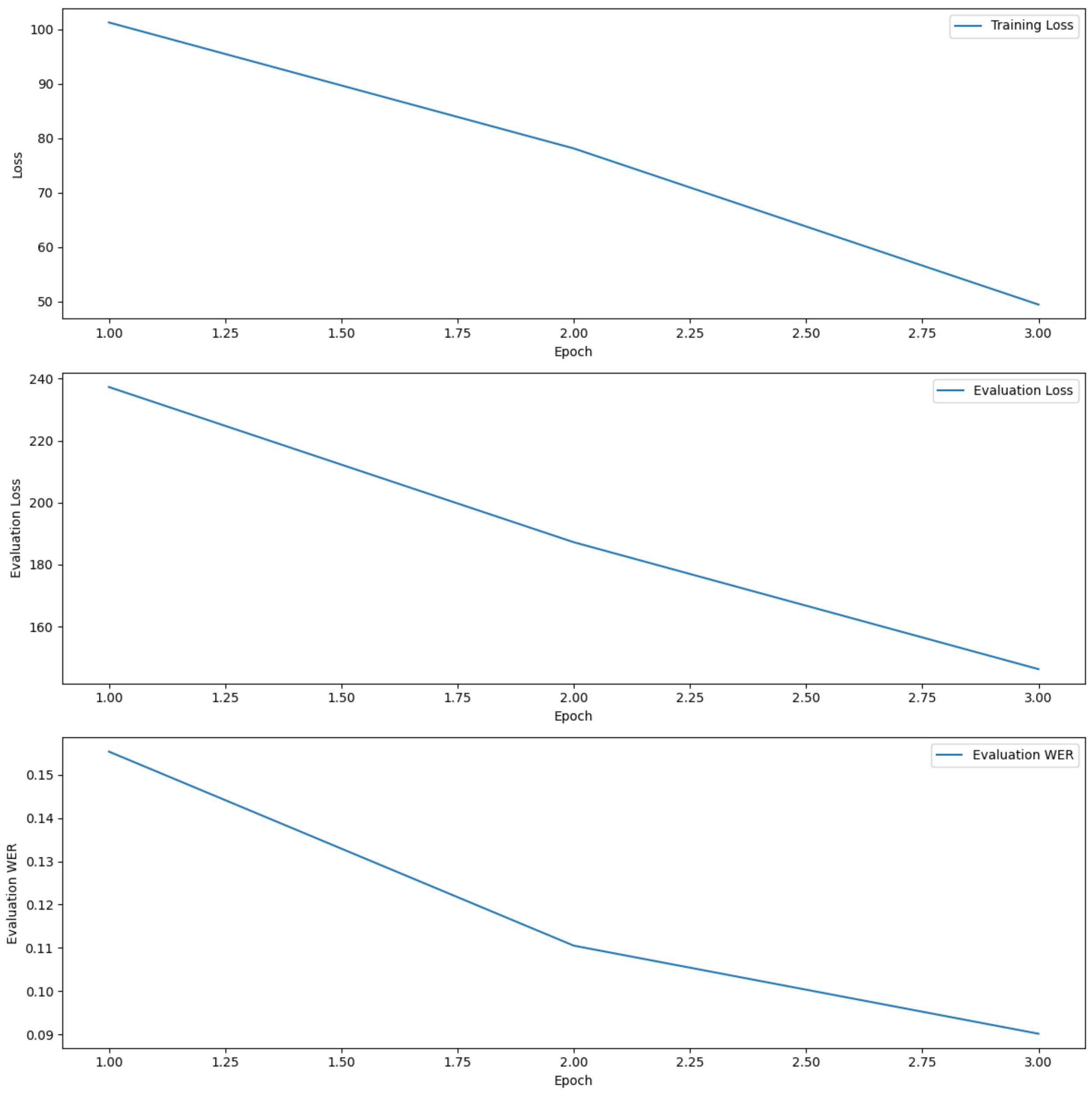
An evaluation strategy of "epoch" is chosen defines that the model is evaluated against the training and validation data as well as WER function every epoch. This is chosen instead of every fixed "steps" as evaluation is computationally demanding, hence evaluating 3 times throughout the model training eases GPU usage.



```python
from transformers import TrainingArguments, Trainer

# Define the training arguments
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir='./logs',
    gradient_accumulation_steps=2,
    gradient_checkpointing=True,
    fp16=True,
    learning_rate=1e-4,
    evaluation_strategy='epoch',
    logging_strategy='epoch',
    save_strategy='epoch'
)

# Initialize the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    data_collator=data_collator,
    compute_metrics=compute_metrics
)
```

Training Results

It can be observed from the training loss and evaluation loss that the loss has yet to converge. This is due to the lack of training steps as maximum epoch is only set to 3. This also shows that the model can be further train to achieve lower loss and higher accuracy.

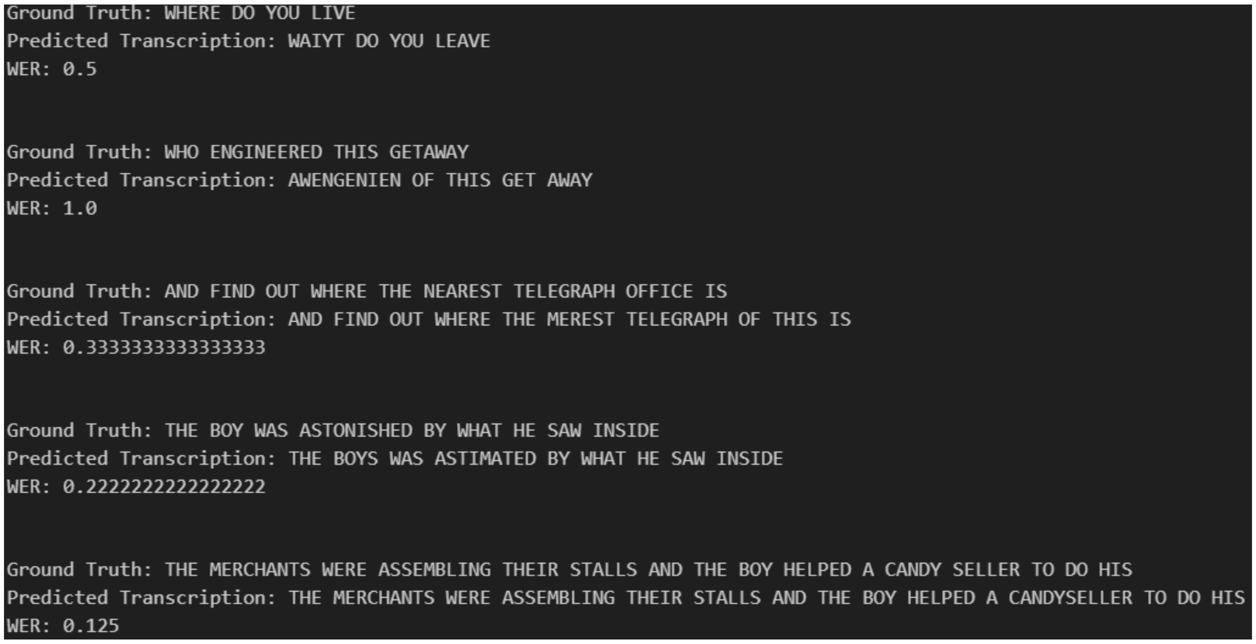


Fine-tuned Model Evaluation

Against Test Dataset:

After saving and loading the final fine-tuned and newly trained model, I begin to run inference on the test dataset mp3 files to generate predicted transcription and calculate WER against the ground truth. It is found that the mean WER of the fine-tuned model against the test dataset is: **0.1**.

Examples of incorrectly predicted transcriptions:

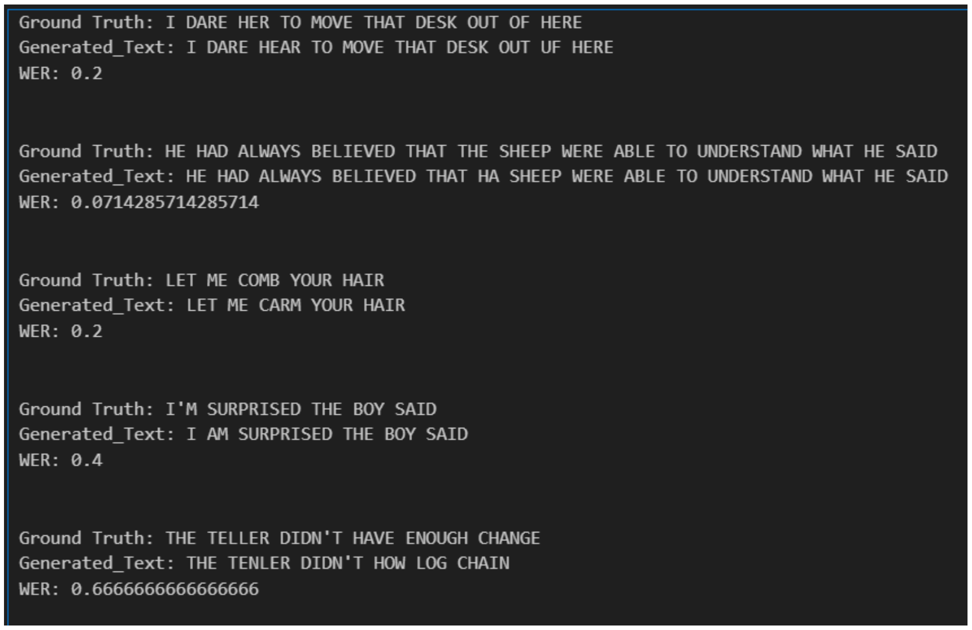


```
Ground Truth: WHERE DO YOU LIVE
Predicted Transcription: WAIYT DO YOU LEAVE
WER: 0.5


Ground Truth: WHO ENGINEERED THIS GETAWAY
Predicted Transcription: AWENGENIEN OF THIS GET AWAY
WER: 1.0


Ground Truth: AND FIND OUT WHERE THE NEAREST TELEGRAPH OFFICE IS
Predicted Transcription: AND FIND OUT WHERE THE MEREST TELEGRAPH OF THIS IS
WER: 0.3333333333333333


Ground Truth: THE BOY WAS ASTONISHED BY WHAT HE SAW INSIDE
Predicted Transcription: THE BOYS WAS ASTIMATED BY WHAT HE SAW INSIDE
WER: 0.2222222222222222


Ground Truth: THE MERCHANTS WERE ASSEMBLING THEIR STALLS AND THE BOY HELPED A CANDY SELLER TO DO HIS
Predicted Transcription: THE MERCHANTS WERE ASSEMBLING THEIR STALLS AND THE BOY HELPED A CANDYSELLER TO DO HIS
WER: 0.125
```

Comparing against “facebook/wav2vec2-large-960h" Model:

Against the cv-valid-dev dataset of audio files the mean WER of the original “facebook/wav2vec2-large-960h" model is found to be: **0.117**.

Examples of incorrectly predicted transcriptions by “facebook/wav2vec2-large-960h":



```
Ground Truth: I DARE HER TO MOVE THAT DESK OUT OF HERE
Generated_Text: I DARE HEAR TO MOVE THAT DESK OUT UF HERE
WER: 0.2


Ground Truth: HE HAD ALWAYS BELIEVED THAT THE SHEEP WERE ABLE TO UNDERSTAND WHAT HE SAID
Generated_Text: HE HAD ALWAYS BELIEVED THAT HA SHEEP WERE ABLE TO UNDERSTAND WHAT HE SAID
WER: 0.07142857142857142


Ground Truth: LET ME COMB YOUR HAIR
Generated_Text: LET ME CARM YOUR HAIR
WER: 0.2


Ground Truth: I'M SURPRISED THE BOY SAID
Generated_Text: I AM SURPRISED THE BOY SAID
WER: 0.4


Ground Truth: THE TELLER DIDN'T HAVE ENOUGH CHANGE
Generated_Text: THE TENLER DIDN'T HOW LOG CHAIN
WER: 0.6666666666666666
```

Against the same dataset the fine-tuned model “wav2vec2-large-960h-cv" performed surprisingly better with a mean WER of: **0.104.**

Areas for improvement

The accuracy of the fine-tuned model showed an improvement in accordance with the WER metric. However, as seen above, the training loss and validation loss has not converged, indicating that the model can be further trained to better accuracy. To further enhance the accuracy of the training model, we can employ these steps:

Increasing the training data

Due to limitation of my personal hardware, I am unable to train the model on the full 195,776 dataset of audio files and labels. Instead, I trained it against 10,000 data.

Increasing the training dataset on a better performing machine will likely increase the accuracy of model.

Increasing training steps/epochs:

Increasing the number of training epochs means that the model will have more iterations over the entire dataset to learn from. This can potentially improve the accuracy of the model. However, training over too many iterations can result in overtraining where the model overfits to the training data and performs worse on unseen or validation data. It is important to save checkpoint weights at different training iteration or epoch and visualize the loss curve in order to retrieve the weights that have the best result on both the training and validation data. An early stopping method can also be used to stop the training of the model in an event where the validation loss fails to improve.

Data Augmentation:

Similar to increasing training data, data augmentation increases the diversity of the data set by making slight modifications like adding noise to the audio data. This ensures that during training, the model is exposed to a wider variety of dataset to help the model generalize and perform better.