In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
output_file_path = "/content/drive/MyDrive/quantization_output.txt"
```

In [ ]:

```
!cat /proc/cpuinfo >> $output_file_path
```

In [ ]:

```
%%capture
BRANCH = 'develop'
!python -m pip install git+https://github.com/speechbrain/speechbrain.git@$BRANCH
```

In [ ]:

```
%%capture
!pip install https://github.com/kpu/kenlm/archive/master.zip
!pip install pygtrie
```

In [ ]:

```
import functools
import gc
import numpy as np
import os
import sentencepiece
import speechbrain
import time
import torch
import torch.nn as nn
import tqdm

from copy import deepcopy
```

In [ ]:

```
%%capture
!mkdir librispeech_dev_clean
!wget https://www.openslr.org/resources/12/dev-clean.tar.gz -P /content
!tar -xvf dev-clean.tar.gz -C librispeech_dev_clean
```

In [ ]:

```
from speechbrain.inference.ASR import EncoderASR

asr_model = EncoderASR.from_hparams(
    source="speechbrain/asr-wav2vec2-commonvoice-14-en",
    savedir="/content/pretrained_ASR/asr-wav2vec2-commonvoice-14-en",
)
```

In [ ]:

```
from speechbrain.dataio.dataio import read_audio

# Retrieve the downloaded speech data as a list of audio-reference pairs
def get_samples(root):
    audios = []
    references = []
    for book in os.listdir(root):
        for chapter in os.listdir(f"{root}/{book}"):
            for file in os.listdir(f"{root}/{book}/{chapter}"):
```

```python
                if file.endswith("txt"):
                    with open(f"{root}/{book}/{chapter}/{file}", "r") as f:
                        for line in f.readlines():
                            audio_path, reference = line.split(" ", 1)
                            full_audio_path = f"{root}/{book}/{chapter}/{audio_path}.fla
c"
                            audios.append(read_audio(full_audio_path))
                            references.append(reference)
    return audios, references
```

In [ ]:

```python
from operator import itemgetter

def random_choice(items, n, seed=None):
    if seed is not None:
        np.random.seed(seed)
    indices = np.random.choice(len(items), n)
    return list(itemgetter(*indices)(items))
```

In [ ]:

```python
audios, references = get_samples("/content/librispeech_dev_clean/LibriSpeech/dev-clean")
```

In [ ]:

```python
np.random.seed(1337)
calibration_samples = random_choice(audios, 10)
```

In [ ]:

```python
def get_module(model, module_string):
    curr = model.mods
    for attr in module_string.split("."):
        curr = getattr(curr, attr)
    return curr

def set_module(model, module_string, new_module):
    curr = model.mods
    attrs = module_string.split(".")
    for attr in attrs[:-1]:
        curr = getattr(curr, attr)
    setattr(curr, attrs[-1], new_module)
```

In [ ]:

```python
from torch.ao.quantization import QuantStub, DeQuantStub

class StaticQuant(nn.Module):
    def __init__(self, model):
        super().__init__()
        self.quant = QuantStub()
        self.model = model
        self.dequant = DeQuantStub()

    def __getattr__(self, name):
        if name in self.__dict__:
            return self.__dict__[name]
        elif name in self.__dict__['_modules']:
            return self.__dict__['_modules'][name]
        else:
            return getattr(self.__dict__['_modules']['model'], name)

    def forward(self, x, *args, **kwargs):
        x = self.quant(x)
        x = self.model(x, *args, **kwargs)
        if isinstance(x, tuple):
            return tuple(self.dequant(output) for output in x)
        else:
            return self.dequant(x)
```

In [ ]:

```
def custom_quantize(
    model,
    dynamic_modules=None,
    static_modules=None,
    calibration_samples=None,
    dynamic_targets=None,
    dynamic_dtype=torch.qint8,
    static_qconfig=torch.ao.quantization.default_qconfig,
):
    #################################################
    # Dynamic Quantization                          #
    #################################################
    if dynamic_modules is not None and len(dynamic_modules) > 0:
        if dynamic_targets is None:
            dynamic_targets = {
                nn.LSTM,
                nn.GRU,
                nn.RNNCell,
                nn.GRUCell,
                nn.LSTMCell,
                nn.Linear,
            }

        for module in dynamic_modules:
            torch.quantization.quantize_dynamic(
                model=get_module(model, module),
                qconfig_spec=dynamic_targets,
                dtype=dynamic_dtype,
                inplace=True,
            )

    #################################################
    # Static Quantization                           #
    #################################################
    if static_modules is not None and len(static_modules) > 0:
        if calibration_samples is None or len(calibration_samples) == 0:
            raise Exception("No calibration samples provided for static quantization.")

        for module in static_modules:
            set_module(
                model,
                module,
                StaticQuant(get_module(model, module)),
            )
            get_module(model, module).qconfig = static_qconfig

        torch.ao.quantization.prepare(model=model, inplace=True)

        for sample in calibration_samples:
            model.transcribe_batch(sample.unsqueeze(0), torch.tensor([1.0]))

        torch.ao.quantization.convert(module=model, inplace=True)
```

In [ ]:

```
def levenshtein(x, y):
    prev = list(range(len(y) + 1))
    curr = [0] * (len(y) + 1)
    for i in range(1, len(x) + 1):
        curr[0] = i
        for j in range(1, len(y) + 1):
            if x[i - 1] == y[j - 1]:
                curr[j] = prev[j - 1]
            else:
                curr[j] = 1 + min(
                    curr[j - 1],   # Insertion
                    prev[j],       # Deletion
                    prev[j - 1]    # Substitution
                )
```

```python
        prev = curr.copy()
    return curr[len(y)]
```

In [ ]:

```python
def compute_wer(references, hypotheses):
    if isinstance(references, str):
        references = [references.split()]
    else:
        references = [ref.split() for ref in references]
    if isinstance(hypotheses, str):
        hypotheses = [hypotheses.split()]
    else:
        hypotheses = [hyp.split() for hyp in hypotheses]
    if len(references) != len(hypotheses):
        raise Exception("Number of references is not equal to the number of hypotheses")
    total_error = 0
    total_length = sum(len(reference) for reference in references)
    for reference, hypothesis in zip(references, hypotheses):
        total_error += levenshtein(reference, hypothesis)
    return total_error / total_length * 100
```

In [ ]:

```python
class Wrapper(nn.Module):
    def __init__(self, model):
        super().__init__()
        self.model = model

    def __getattr__(self, name):
        if name in self.__dict__:
            return self.__dict__[name]
        elif name in self.__dict__["_modules"]:
            return self.__dict__["_modules"][name]
        else:
            return getattr(self.__dict__["_modules"]["model"], name)
```

In [ ]:

```python
class EncoderASRWrapper(Wrapper):
    def preprocess_input(self, input):
        with torch.no_grad():
            wavs = input.unsqueeze(0)
            wav_lens = torch.tensor([1.0])
            wavs = wavs.float()
            wavs, wav_lens = wavs.to(self.model.device), wav_lens.to(self.model.device)
        return wavs, wav_lens

    def generate(self, predictions):
        is_ctc_text_encoder_tokenizer = isinstance(
            self.model.tokenizer, speechbrain.dataio.encoder.CTCTextEncoder
        )
        if isinstance(self.model.hparams.decoding_function, functools.partial):
            if is_ctc_text_encoder_tokenizer:
                predicted_words = [
                    "".join(self.model.tokenizer.decode_ndim(token_seq))
                    for token_seq in predictions
                ]
            else:
                predicted_words = [
                    self.model.tokenizer.decode_ids(token_seq)
                    for token_seq in predictions
                ]
        else:
            predicted_words = [hyp[0].text for hyp in predictions]
        return predicted_words

    def forward(self, input):
        with torch.no_grad():
            wavs, wav_lens = self.preprocess_input(input)
            encoder_out = self.model.mods.encoder(wavs, wav_lens)
```

```python
            predictions = self.model.decoding_function(encoder_out, wav_lens)
            predicted_words = self.generate(predictions)
        return predicted_words[0]

    def timed_transcribe(self, input):
        with torch.no_grad():
            wavs, wav_lens = self.preprocess_input(input)
            start = time.time()
            encoder_out = self.model.mods.encoder(wavs, wav_lens)
            end = time.time()
            duration = end - start
            predictions = self.model.decoding_function(encoder_out, wav_lens)
            predicted_words = self.generate(predictions)
        return predicted_words[0], duration
```

In [ ]:

```python
def benchmark(model, samples, references):
    total_audio_length = sum([sample.shape[0] / 16000 for sample in samples])
    total_cpu_time = 0
    outputs = []

    if isinstance(model, EncoderASR):
        wrapper = EncoderASRWrapper(model)
    elif isinstance(model, EncoderDecoderASR):
        wrapper = EncoderDecoderASRWrapper(model)
    else:
        raise NotImplementedError

    for sample in tqdm.tqdm(samples[:10], desc="warming up"):
        wrapper.timed_transcribe(sample)

    for sample in tqdm.tqdm(samples, desc="evaluating"):
        output, duration = wrapper.timed_transcribe(sample)
        outputs.append(output)
        total_cpu_time += duration

    wer = compute_wer(references, outputs)
    rtf = total_cpu_time / total_audio_length
    return wer, rtf
```

In [ ]:

```python
n = 100
audio_subset = audios[:n]
ref_subset = references[:n]
```

In [ ]:

```python
# Deepcopy the original model to avoid propagating unwanted changes
original_model = deepcopy(asr_model)
```

In [ ]:

```python
original_model.eval()
wer, rtf = benchmark(original_model, audio_subset, ref_subset)
with open(output_file_path, "a+") as f:
    f.write(f"Original Model\nWER(%): {wer}\nRTF: {rtf}\n\n")
del original_model
gc.collect()
```

In [ ]:

```python
modules = [
    "encoder.wav2vec2.model.feature_projection",
    "encoder.wav2vec2.model.encoder.layers",
    "encoder.enc",
    "encoder.ctc_lin",
]

for module in modules:
```

```
    m = deepcopy(asr_model)
    custom_quantize(m, dynamic_modules=[module])
    m.eval()
    wer, rtf = benchmark(m, audio_subset, ref_subset)
    with open(output_file_path, "a+") as f:
        f.write(f"dynamic {module}\nWER(%): {wer}\nRTF: {rtf}\n\n")
    del m
    gc.collect()
```

In [ ]:

```
modules = [
    "encoder.wav2vec2.model.feature_extractor",
    "encoder.wav2vec2.model.feature_projection",
    "encoder.ctc_lin",
]

for module in modules:
    m = deepcopy(asr_model)
    custom_quantize(m, static_modules=[module], calibration_samples=calibration_samples)
    m.eval()
    wer, rtf = benchmark(m, audio_subset, ref_subset)
    with open(output_file_path, "a+") as f:
        f.write(f"static {module}\nWER(%): {wer}\nRTF: {rtf}\n\n")
    del m
    gc.collect()
```

In [ ]:

```
dynamic_modules = [
    "encoder.wav2vec2.model.encoder.layers",
    "encoder.enc",
]
static_modules = [
    "encoder.wav2vec2.model.feature_projection",
    "encoder.wav2vec2.model.feature_extractor",
]
```

In [ ]:

```
quantized_model = deepcopy(asr_model)
```

In [ ]:

```
custom_quantize(
    model=quantized_model,
    dynamic_modules=dynamic_modules,
    static_modules=static_modules,
    calibration_samples=calibration_samples,
)
```

In [ ]:

```
quantized_model.eval()
wer, rtf = benchmark(quantized_model, audio_subset, ref_subset)
with open(output_file_path, "a+") as f:
    f.write(f"Quantized Model (dynamic layers, enc; static proj, extract)\nWER(%): {wer}
\nRTF: {rtf}\n\n")
del quantized_model
gc.collect()
```

In [ ]:

```
from speechbrain.inference.ASR import EncoderDecoderASR

crdnn = EncoderDecoderASR.from_hparams(
    source="speechbrain/asr-crdnn-commonvoice-14-en",
    savedir="/content/pretrained_ASR/asr-crdnn-commonvoice-14-en",
)
```

In [ ]:

```
class EncoderDecoderASRWrapper(Wrapper):
    def preprocess_input(self, input):
        with torch.no_grad():
            wavs = input.unsqueeze(0)
            wav_lens = torch.tensor([1.0])
            wavs = wavs.float()
            wavs, wav_lens = wavs.to(self.model.device), wav_lens.to(self.model.device)
        return wavs, wav_lens

    def generate(self, encoder_out, wav_lens):
        if self.model.transducer_beam_search:
            inputs = [encoder_out]
        else:
            inputs = [encoder_out, wav_lens]
        predicted_tokens, _, _, _ = self.model.mods.decoder(*inputs)
        predicted_words = [
            self.model.tokenizer.decode_ids(token_seq) for token_seq in predicted_tokens
        ]
        return predicted_words, predicted_tokens

    def forward(self, input):
        with torch.no_grad():
            wavs, wav_lens = self.preprocess_input(input)
            encoder_out = self.model.mods.encoder(wavs, wav_lens)
            predicted_words = self.generate(encoder_out, wav_lens)[0]
        return predicted_words[0]

    def timed_transcribe(self, input):
        with torch.no_grad():
            wavs, wav_lens = self.preprocess_input(input)
            start = time.time()
            encoder_out = self.model.mods.encoder(wavs, wav_lens)
            end = time.time()
            duration = end - start
            predicted_words = self.generate(encoder_out, wav_lens)[0]
        return predicted_words[0], duration
```

In [ ]:

```
crdnn
```

In [ ]:

```
original_model = deepcopy(crdnn)
original_model.eval()
wer, rtf = benchmark(original_model, audio_subset, ref_subset)
with open(output_file_path, "a+") as f:
    f.write(f"\n=======\nCRDNN\n\nOriginal Model\nWER(%): {wer}\nRTF: {rtf}\n\n")
del original_model
gc.collect()
```

In [ ]:

```
modules = [
    "encoder.model.RNN.rnn",
    "encoder.model.DNN",
    "decoder.dec",
    "decoder.fc.w",
]

for module in modules:
    m = deepcopy(crdnn)
    custom_quantize(m, dynamic_modules=[module])
    m.eval()
    wer, rtf = benchmark(m, audio_subset, ref_subset)
    with open(output_file_path, "a+") as f:
        f.write(f"dynamic {module}\nWER(%): {wer}\nRTF: {rtf}\n\n")
    del m
    gc.collect()
```

In [ ]:

```
modules = [
    "encoder.model.CNN",
    "decoder.fc.w",
]

for module in modules:
    m = deepcopy(crdnn)
    custom_quantize(m, static_modules=[module], calibration_samples=calibration_samples)
    m.eval()
    wer, rtf = benchmark(m, audio_subset, ref_subset)
    with open(output_file_path, "a+") as f:
        f.write(f"static {module}\nWER(%): {wer}\nRTF:{rtf}\n\n")
    del m
    gc.collect()
```

In [ ]:

```
dynamic_modules = [
    "encoder.model.RNN.rnn",
    "encoder.model.DNN",
    "decoder.dec",
    "decoder.fc.w",
]
static_modules = [
    "encoder.model.CNN",
]

quantized_model = deepcopy(crdnn)

custom_quantize(
    model=quantized_model,
    dynamic_modules=dynamic_modules,
    static_modules=static_modules,
    calibration_samples=calibration_samples,
)

quantized_model.eval()
wer, rtf = benchmark(quantized_model, audio_subset, ref_subset)
with open(output_file_path, "a+") as f:
    f.write(f"Quantized Model (dynamic rnn, dnn, dec, fc; static cnn)\nWER(%): {wer}\nRT
F: {rtf}\n\n")
del quantized_model
gc.collect()
```