# CTC and Wave2Vec for ASR

## Introduction

The aim of this project is to implement and experiment with a pre-trained Wave2Vec 2.0 model using Connectionist Temporal Classification (CTC) approach. We need to load audio data, process them through the model and compare different algorithms such as Greedy Decoding and Beam Search Decoding.

## Wave2Vec Model Description

Wave2Vec is a self-supervised model used for Automatic Speech Recognition (ASR). It is composed of three main components:

- Feature Encoder
- Quantization Model
- Contextual Transformer Network

The model is trained on large amounts of unlabeled audio data. During pre-training, parts of the audio input are masked and the model learns to predict them based on surrounding context. After pre-training, the model is tuned on a labeled database for ASR task. The output is a sequence of tokens.
The pre-trained model is loaded from torchaudio pipelines.The model includes the necessary components to perform ASR. After that, the model processes the audio waveform to generate tokens emission, which are probability distributions of possible output tokens.
Then we adopt two different strategies:

- Greedy Algorithm
  Greedy Algorithm converts the predicted tokens into a transcript. This approach selects the most likely token at each time step but can miss some context information.

- Beam Search Decoding
  Beam Search improves Greedy Algorithm by considering multiple hypothesis, multiple possible sequences of tokens, selecting the best one using also language model's probabilities.
  The algorithm begins with a list of beams, each beam represents a possible sequence of tokens and its corresponding score.In each iteration, the algorithm adjusts each beam by predicting the next possible tokens using the model's emissions.The algorithm then selects the top candidate tokens for each beam. For each selected token, the algorithm appends it to the existing sequence in the beam, updates the beam's state, and adds the updated beam to a new list. Once all beams have been completed, the algorithm sorts the new beams by their scores and considers only the top beams. The process continues iteratively until no more tokens can be added. Beam Search

efficiently explores the  search space, making it a useful tool for sequence generation and decoding tasks.

## Questions

### How to improve the Greedy Algorithm

The greedy algorithm selects the most probable token at each time step. It is not optimal since it just considers the immediate next token and not the surrounding context and the coherence of the sequence. To improve greedy algorithms one can use sampling. Using sampling we introduce a sort of randomness in the process. I consider top-k sampling choosing only the most probable tokens limiting the process selection, reducing the impact of very low-probability tokens but maintaining some level of diversity.

Using top-3 sampling, the result is the same obtained with the greedy algorithm. This occurs because sampling doesn't directly improve greedy algorithm, but it just adds some element of randomness to introduce some diversity.

```
class KSamplingDecoder(torch.nn.Module):
    def __init__(self, labels, blank_position=0, k=3):
        super().__init__()
        self.labels = labels
        self.blank = blank_position
        self.k = k

    def forward(self, emission: torch.Tensor) -> str:
        # Ensure emission has 3 dimensions: (batch_size, time_steps, vocab_size)
        # If emission is (time_steps, vocab_size), add batch dimension
        if emission.dim() == 2:
            emission = emission.unsqueeze(0)

        # Get the top k probabilities and their indices along the vocab_size dimension
        topk_probs, topk_indices = torch.topk(F.log_softmax(emission, dim=-1), self.k, dim=-1)

        # Select the highest probability for each time step
        sampled_indices = topk_indices[:, :, 0]  # Shape: (batch_size, time_steps)

        # Remove consecutive duplicates across time_steps
        sampled_indices = torch.unique_consecutive(sampled_indices, dim=-1)

        # Filter out the blank token and convert indices to labels
        sampled_indices = [i.item() for i in sampled_indices[0] if i != self.blank]

        return "".join([self.labels[i] for i in sampled_indices])
```

```
labels = ctc_preTrained_object.get_labels()
greedy_decoder = GreedyAlgorithmDecoder(labels=labels)
k_sampling_decoder = KSamplingDecoder(labels=labels, k=3)

# Run the decoders
greedy_transcript = greedy_decoder(pred_tokens[0].to(device))
k_sampling_transcript = k_sampling_decoder(pred_tokens[0].to(device))
```

### Different beam search decoder configurations

To better understand the beam search decoder, different configurations have been setted. In particular, I focus on beam size and n_best parameters.

The beam size controls how many sequences are considered at each step. I try to change the beam size to 150 and 3000. A smaller beam size makes the decoding faster but might reduce accuracy. On the other hand, increasing beam size could lead to a greater accuracy but might require more computational resources and time.

The n_best parameter determines how many of the sequences should be returned. I try to change these parameters to 5 and 10. Changing the number of sequences might be useful if the user wants to choose between several alternatives.

```
# Define the configurations
configs = [
    {'beam_size': 150, 'n_best': 5},
    {'beam_size': 150, 'n_best': 10},
    {'beam_size': 3000, 'n_best': 5},
    {'beam_size': 3000, 'n_best': 10}
]

# Placeholder to store results for comparison
results = {}

# Iterate over each configuration and perform decoding
for config in configs:
    beam_search_decoder = ctc_decoder(
        lexicon=files.lexicon,
        tokens=files.tokens,
        lm=files.lm,
        nbest=config['n_best'],
        beam_size=config['beam_size']
    )

    # Perform the decoding
    beam_search_result = beam_search_decoder(pred_tokens)
```

In this case, the results are the same across all configurations.

```
Configuration: beam_size=150, n_best=5
Transcript: i had that curiosity beside me at this moment
Error Rate: 0.0
--------------------------------------------------------------------------------
Configuration: beam_size=150, n_best=10
Transcript: i had that curiosity beside me at this moment
Error Rate: 0.0
--------------------------------------------------------------------------------
Configuration: beam_size=3000, n_best=5
Transcript: i had that curiosity beside me at this moment
Error Rate: 0.0
--------------------------------------------------------------------------------
Configuration: beam_size=3000, n_best=10
Transcript: i had that curiosity beside me at this moment
Error Rate: 0.0
--------------------------------------------------------------------------------
```