```python
import torch
import torchaudio

print(torch.__version__)
print(torchaudio.__version__)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(device)
```

```
2.3.1+cu121
2.3.1+cu121
cuda
```

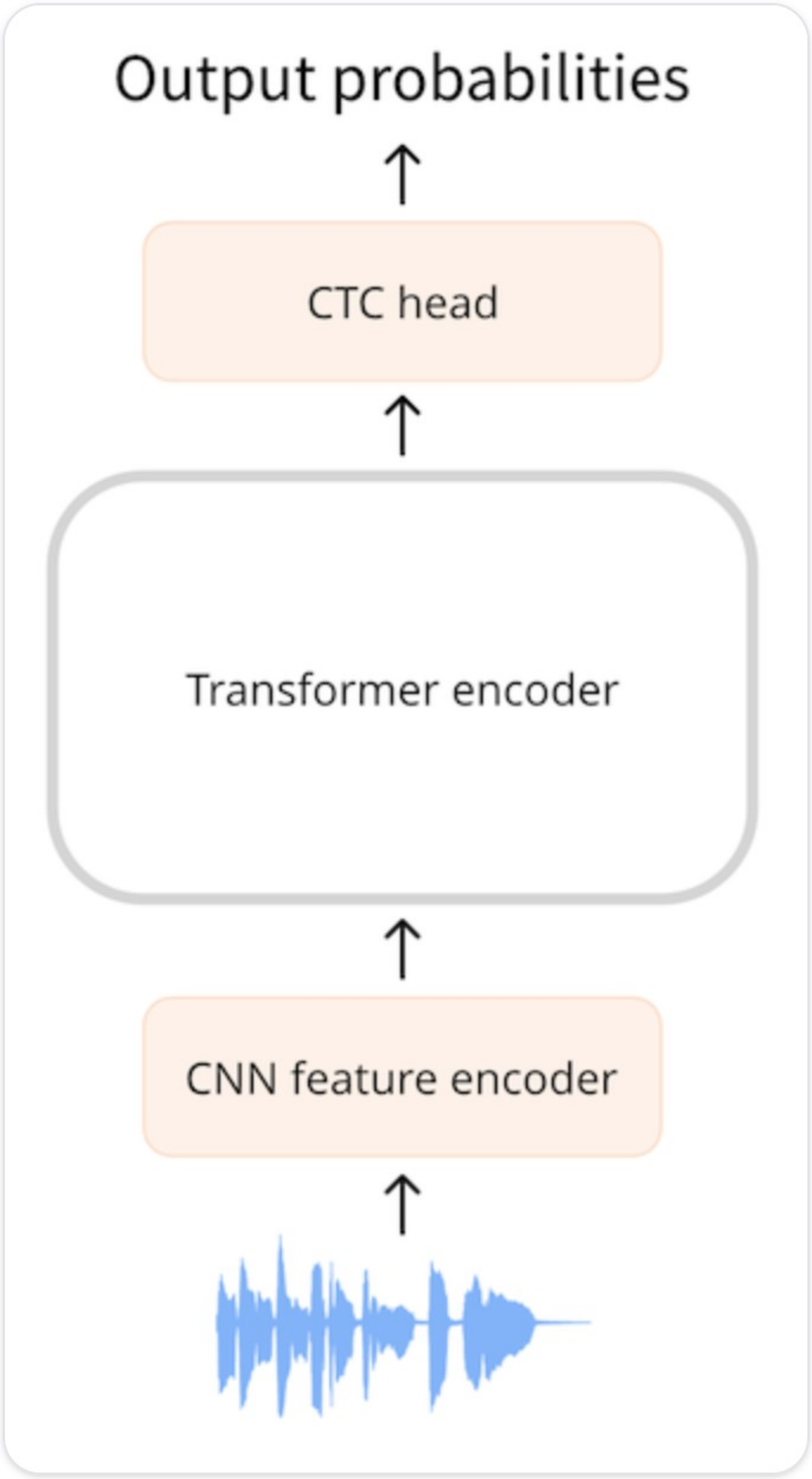
```python
import IPython
import matplotlib.pyplot as plt
from torchaudio.utils import download_asset

sound_voice = download_asset("tutorial-assets/Lab41-SRI-VOiCES-src-sp0307-ch127535-sg0042.wav")
```

```
100%|██████████| 106k/106k [00:00<00:00, 171kB/s]
```

# Using Connectionist Temporal Classification (CTC) to classify characters

Basic architectures for characters classification use an encoder-only transformer to read the input sequence (the audio waveform) and encode it into a sequence of hidden-states, i.e.,embeddings. However, alignment is lost...

In Wave2Vec (one of the existing models, read https://arxiv.org/pdf/2006.11477.pdf) first CNN layers downsample the input to produce a sequence of 50 hidden-states. Each hidden-state corresponds to about 25ms (with a small overlapping between consecutive embeddings - about 5ms). After this the transformer encoder receives the 50 hidden-states and produces an encoding of such states into 50 embeddings of dimensionality 768. Finally, we pass these embeddings to a linear layer to map them into the possible tokens of a very small (typically a 32 items or less) vocabulary (output is a (50,32) tensor). For each token the corresponding input is 20ms (plus 5ms overlapping) of audio, hence each of them correspond to roughly a character. However, we predict 50 characters in (50*20=1000ms) one second audio. So surely if a charater is pronounced in more than 20ms we will have duplicates. CTC algorithm introduced two special characters the "black" (_) and the space (|) to easy word classification.


Output probabilities
CTC head
Transformer encoder
CNN feature encoder

In torchaudio, we have pre-trained versions of many CTC models. A pre-trained WAV2VEC 2.0 is also available. Let's load it.

In torchaudio we have trained models embedded into objects including additional information, such as the "sample rate", the "labels" (i.e., the dictionaries tokens) and so so forth - read https://pytorch.org/audio/stable/generated/torchaudio.pipelines.Wav2Vec2ASRBundle.html#torchaudio.pipelines.Wav2Vec2ASRBundle.

The model has been trained with a specific audio sample rate, which is avaiable in the object. Considered labels are 29!

```
ctc_preTrained_object = torchaudio.pipelines.WAV2VEC2_ASR_BASE_960H

print("This is the model sample rate:",
ctc_preTrained_object.sample_rate)

print("These are the considered tokens:",
ctc_preTrained_object.get_labels())
```

```
This is the model sample rate: 16000
These are the considered tokens: ('-', '|', 'E', 'T', 'A', 'O', 'N',
'I', 'H', 'S', 'R', 'D', 'L', 'U', 'M', 'W', 'C', 'F', 'G', 'Y', 'P',
'B', 'V', 'K', "'", 'X', 'J', 'Q', 'Z')
```

This object contains also the trained model, let's load it into our GPU.

```
model = ctc_preTrained_object.get_model().to(device)
```

```
Downloading:
"https://download.pytorch.org/torchaudio/models/wav2vec2_fairseq_base_
ls960_asr_ls960.pth" to
/root/.cache/torch/hub/checkpoints/wav2vec2_fairseq_base_ls960_asr_ls9
60.pth
100%|██████████| 360M/360M [00:06<00:00, 56.6MB/s]
```

Let's now import our previously developed audio manipulation functions. Remember: we sent the model to our device (GPU), so also the input data must be loaded there. Also, make sure to resample input to the model sampling rate - the model expects a (1,seconds*16000) tensor.

```
import torch
import torchaudio
from IPython.display import Audio

# Load the sample audio
waveform, sample_rate = torchaudio.load(sound_voice)
waveform = waveform.to(device)  # Send data to the device

# Check the sample audio
Audio(waveform.cpu().numpy(), rate=sample_rate)
```

```
print("Sample rate:", sample_rate)

# Resample if necessary
if sample_rate != ctc_preTrained_object.sample_rate:
    waveform = torchaudio.functional.resample(waveform, sample_rate,
ctc_preTrained_object.sample_rate)

# Ensure waveform is resampled correctly
waveform = waveform if sample_rate ==
ctc_preTrained_object.sample_rate else
torchaudio.functional.resample(waveform, sample_rate,
ctc_preTrained_object.sample_rate)
```

```
Sample rate: 16000
```

```
waveform.shape
```

```
torch.Size([1, 54400])
```

```
with torch.inference_mode(): #this is to disable grad reasonings and
improve performance
    pred_tokens, _ = model(waveform)
print(pred_tokens.size())
#we have 169 identified characters each associated with a vector with
29 values (logits)
#(i.e., the strenght of believing that a word is one of the 29 tokens
of the dictionary)
```

```
torch.Size([1, 169, 29])
```

```
class GreedyAlgorithmDecoder(torch.nn.Module):
    def __init__(self, labels, blank_position=0): #we can specify the
position of blank value in the label list - in this case pos 0
        super().__init__()
        self.labels = labels
        self.blank = blank_position

    def forward(self, emission: torch.Tensor) -> str:
        indices = torch.argmax(emission, dim=-1)  # grab the highest
logit for each sequence in the tensor
        #print(indices.shape)
        #if a character is spelled for more than 20ms
        #we will have consecutive duplicates, so we need to remove
them
        indices = torch.unique_consecutive(indices)
        indices = [i for i in indices if i != self.blank] #now remove
blank character - remove index 0
        return "".join([self.labels[i] for i in indices]) #grab token
from the dictionary using indexes
```

```
greedy_CTC_decoder =
GreedyAlgorithmDecoder(labels=ctc_preTrained_object.get_labels())
transcript = greedy_CTC_decoder(pred_tokens[0]) #it's a tensor with
one channel and 169 words mapped to 29 tokens

print(transcript)
```

```
I|HAD|THAT|CURIOSITY|BESIDE|ME|AT|THIS|MOMENT|
```

## Advanced!

Beam Search is a more advanced algorithm to identify the best sequences according to the model emissions. Basic idea is that it explores multiple possible sequences of tokens and selects the most likely ones based on a scoring function. Beam search is particularly useful in decoding the output of models trained with sequence generation techniques like recurrent neural networks (RNNs) or transformers.

Basic strategy is as follows:

1. The algorithm initializes with a list of beams, each containing a sequence of tokens and its corresponding state (whether they are conclusive or not). For starters, we will have only initial tokens.
2. In each iteration, the algorithm expands each beam by generating the next set of tokens using the model. It then selects the top k (where k is the maximum size for a beam at at single step) tokens as candidates for each beam.
3. For each candidate token, the algorithm appends it to the beam's sequence, updates the beam's state based on the candidate token, and adds the updated beam to a new list of beams.
4. Once all beams are processed, the algorithm sorts the new beams by their scores (e.g., log probabilities or other evaluation) and selects the top beam_width beams to continue the search.
5. The process repeats until the maximum sequence length is reached or no more sequences are active (i.e., all beams are completed or stopped).

Beam search efficiently explores the search space by maintaining a limited number of active sequences (beams) at each step, thereby balancing between exploration and exploitation. The beam_width parameter controls the trade-off between exploration (higher values) and exploitation (lower values). Larger beam widths increase the computational cost but may yield better results by exploring more diverse sequences.

Example:

Vocabulary: ['a', 'b', 'c']

And let's perform beam search to generate sequences of length 2 with a beam width of 2.

Initialization: At the beginning, the token state is simply the start token, indicating the start of the sequence.

Token state: [<start>] ... any symbol here

Step 1:

Beam 1: [<start>, 'a'], Score: 1 Beam 2: [<start>, 'b'], Score: 1

Token states for each beam:

Beam 1: Token state: [<start>] → [<start>, 'a']

Beam 2: Token state: [<start>] → [<start>, 'b']

Step 2:

Beam 1: [<start>, 'a', 'a'], Score: 2

Beam 1: [<start>, 'a', 'b'], Score: 2

Beam 2: [<start>, 'b', 'a'], Score: 2

Beam 2: [<start>, 'b', 'b'], Score: 2

Token states for each beam:

Beam 1: Token state: [<start>, 'a'] → [<start>, 'a', 'a'] and [<start>, 'a'] → [<start>, 'a', 'b']

Beam 2: Token state: [<start>, 'b'] → [<start>, 'b', 'a'] and [<start>, 'b'] → [<start>, 'b', 'b']

At this stage, the sequences are completed, and we stop the search.

```
!pip install flashlight-text
```

```
Collecting flashlight-text
  Downloading flashlight_text-0.0.7-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (10 kB)
Downloading flashlight_text-0.0.7-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/1.3 MB ? eta -:--:--
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 45.4 MB/s eta
0:00:00
```

```
from torchaudio.models.decoder import download_pretrained_files

files = download_pretrained_files("librispeech-4-gram")

print(files)
print(files.tokens)

f = open(files.tokens, "r")

print(f.read())
```

```
100%|██████████| 4.97M/4.97M [00:01<00:00, 3.11MB/s]
100%|██████████| 57.0/57.0 [00:00<00:00, 148kB/s]
100%|██████████| 2.91G/2.91G [03:44<00:00, 13.9MB/s]

PretrainedFiles(lexicon='/root/.cache/torch/hub/torchaudio/decoder-
assets/librispeech-4-gram/lexicon.txt',
tokens='/root/.cache/torch/hub/torchaudio/decoder-assets/librispeech-
4-gram/tokens.txt', lm='/root/.cache/torch/hub/torchaudio/decoder-
assets/librispeech-4-gram/lm.bin')
/root/.cache/torch/hub/torchaudio/decoder-assets/librispeech-4-gram/
tokens.txt
-
|
e
t
a
o
n
i
h
s
r
d
l
u
m
w
c
f
g
y
p
b
v
k
'
x
j
q
z
```

The lexicon is a mapping from words to their corresponding tokens sequence, and is used to restrict the search space of the decoder to only words from the lexicon. The expected format of the lexicon file is a line per word, with a word followed by its space-split tokens.

```
f = open(files.lexicon, "r")
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
```

```
a	a |

a''s	a ' ' s |

a'body	a ' b o d y |

a'court	a ' c o u r t |
```

```
files.lm
```

{"type":"string"}

Language model can be used to improve ASR performance by estimating a probability of having a particular sequence as output. In files we have a 4-gram KenLM trained using LibriSpeech.

https://kheafield.com/code/kenlm/

From the flashlight library: https://arxiv.org/pdf/2201.12465.pdf

```
from torchaudio.models.decoder import ctc_decoder


beam_search_decoder = ctc_decoder(
    lexicon=files.lexicon,
    tokens=files.tokens,
    lm=files.lm,
    nbest=3, #how many sequences we want to return
    beam_size=1500
)

# Ensure pred_tokens is on the correct device (CPU) before decoding
pred_tokens = pred_tokens.cpu()

# Perform beam search decoding
beam_search_result = beam_search_decoder(pred_tokens)
print(beam_search_result)

# Extract the words from the beam search result
beam_search_transcript = " ".join(beam_search_result[0]
[0].words).strip()

# Define the real transcript
real_transcript = "i had that curiosity beside me at this
moment".split()

# Calculate the edit distance (number of edits required)
```

```
distance = torchaudio.functional.edit_distance(real_transcript,
beam_search_result[0][0].words)

# Calculate the error rate
beam_search_error = distance / len(real_transcript)

# Print the results
print(f"Transcript: {beam_search_transcript}")
print(f"Error: {beam_search_error}")
```

```
[[CTCHypothesis(tokens=tensor([ 1,  7,  1,  8,  4, 11,  1,  3,  8,  4,
3,  1, 16, 13, 10,  7,  5,  9,
         7,  3, 19,  1, 21,  2,  9,  7, 11,  2,  1, 14,  2,  1,  4,
3,  1,  3,
         8,  7,  9,  1, 14,  5, 14,  2,  6,  3,  1,  1]), words=['i',
'had', 'that', 'curiosity', 'beside', 'me', 'at', 'this', 'moment'],
score=2489.6167145967484, timesteps=tensor([  0,  31,  34,  37,  39,
41,  43,  45,  47,  49,  50,  52,  56,  61,
         63,  67,  74,  79,  83,  86,  89,  91,  95,  98, 102, 109,
111, 113,
        115, 118, 120, 123, 126, 127, 128, 130, 132, 134, 136, 138,
142, 146,
        149, 152, 153, 154, 156, 170], dtype=torch.int32)),
CTCHypothesis(tokens=tensor([ 1,  7,  1,  8,  4, 22,  2,  1,  3,  8,
4,  3,  1, 16, 13, 10,  7,  5,
         9,  7,  3, 19,  1, 21,  2,  9,  7, 11,  2,  1, 14,  2,  1,
4,  3,  1,
         3,  8,  7,  9,  1, 14,  5, 14,  2,  6,  3,  1,  1]),
words=['i', 'have', 'that', 'curiosity', 'beside', 'me', 'at', 'this',
'moment'], score=2483.0585545301437, timesteps=tensor([  0,  31,  34,
37,  39,  41,  42,  43,  45,  47,  49,  50,  52,  56,
         61,  63,  67,  74,  79,  83,  86,  89,  91,  95,  98, 102,
109, 111,
        113, 115, 118, 120, 123, 126, 127, 128, 130, 132, 134, 136,
138, 142,
        146, 149, 152, 153, 154, 156, 170], dtype=torch.int32)),
CTCHypothesis(tokens=tensor([ 1,  7,  1,  8,  4, 11,  1,  3,  8,  2,
1, 16, 13, 10,  7,  5,  9,  7,
         3, 19,  1, 21,  2,  9,  7, 11,  2,  1, 14,  2,  1,  4,  3,
1,  3,  8,
         7,  9,  1, 14,  5, 14,  2,  6,  3,  1,  1]), words=['i',
'had', 'the', 'curiosity', 'beside', 'me', 'at', 'this', 'moment'],
score=2481.070281147957, timesteps=tensor([  0,  31,  34,  37,  39,
41,  43,  45,  47,  49,  51,  56,  61,  63,
         67,  74,  79,  83,  86,  89,  91,  95,  98, 102, 109, 111,
113, 115,
        118, 120, 123, 126, 127, 128, 130, 132, 134, 136, 138, 142,
146, 149,
        152, 153, 154, 156, 170], dtype=torch.int32))]]
```

```
Transcript: i had that curiosity beside me at this moment
Error: 0.0
```

## Report Suggestions:

1. Understand the Wave2Vec 2.0 model and discuss about the solution.
2. Improve greedy algorithm performance (can we boost it? How?)
3. Work with the BeamSearch algorithm and try different configuration, what can you see?

```
#Try to boost greedy algorthm using k sampling of 3
import torch
import torch.nn.functional as F

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))

print("Greedy Transcript:", greedy_transcript)
print("K-Sampling Transcript (k=3):", k_sampling_transcript)
```

```
Greedy Transcript: I|HAD|THAT|CURIOSITY|BESIDE|ME|AT|THIS|MOMENT|
K-Sampling Transcript (k=3): I|HAD|THAT|CURIOSITY|BESIDE|ME|AT|THIS|
MOMENT|
```

```
from torchaudio.models.decoder import ctc_decoder

# 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)

    # Extract the top result (best sequence)
    best_transcript = " ".join(beam_search_result[0][0].words).strip()
    best_error_rate =
torchaudio.functional.edit_distance(real_transcript,
beam_search_result[0][0].words) / len(real_transcript)

    # Store the result
    results[f"beam_size={config['beam_size']},
n_best={config['n_best']}"] = {
        'transcript': best_transcript,
        'error_rate': best_error_rate,
        'beam_search_result': beam_search_result
    }

# Compare the results
for key, value in results.items():
    print(f"Configuration: {key}")
    print(f"Transcript: {value['transcript']}")
    print(f"Error Rate: {value['error_rate']}")
    print("-" * 80)
```

```
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
----------------------------------------------------------------------
----------
```

```
print(f"Real Transcript: {real_transcript}")
print(f"Best Transcript: {best_transcript}")
```

```
Real Transcript: ['i', 'had', 'that', 'curiosity', 'beside', 'me',
'at', 'this', 'moment']
Best Transcript: i had that curiosity beside me at this moment
```