Modèles pré-entraînés pour la reconnaissance automatique de la parole

Dans cette section, nous verrons comment utiliser le `pipeline()` pour tirer parti des modèles pré-entraînés pour la reconnaissance automatique de la parole. Dans l'[unité 2](.. /chapter2/asr_pipeline), nous avons introduit le `pipeline()` comme un moyen facile d'exécuter des tâches de reconnaissance de la parole, avec tout le prétraitement et le post-traitement gérés sous le capot et la flexibilité d'expérimenter rapidement avec n'importe quel *checkpoint* pré-entraîné disponible sur le *Hub*. Dans cette unité, nous irons plus loin et explorerons les différents attributs des modèles de reconnaissance automatique de la parole et comment nous pouvons les utiliser pour aborder une gamme de tâches différentes.

Comme détaillé dans l'unité 3, le modèle de reconnaissance automatique de la parole se divise généralement dans l'une des deux catégories suivantes :

1. Modèle avec Classification temporelle connexionniste (CTC) : modèles avec que l'encodeur du *transformer* avec une tête de classification linéaire sur le dessus
2. Modèle de séquence à séquence (Seq2Seq) : modèles encodeur-décodeur, avec un mécanisme d'attention croisée entre l'encodeur et le décodeur

Avant 2022, la variante avec CTC était la plus populaire des deux architectures, avec des modèles tels que Wav2Vec2, HuBERT et XLSR réalisant des percées dans le paradigme de pré-entraînement / *finetuning* de la parole. De grandes entreprises, telles que Meta et Microsoft, ont pré-entraîné l'encodeur sur de grandes quantités de données audio non étiquetées pendant plusieurs jours ou semaines. Les utilisateurs pouvent ensuite prendre un *checkpoint* pré-entraîné et le finetuner avec une tête CTC sur à peine **10 minutes** de données audio étiquetées pour obtenir de solides performances sur une tâche en aval de reconnaissance automatique de la parole. Cependant, les modèles CTC ont des lacunes. L'ajout d'une simple couche linéaire à un encodeur donne un petit modèle global rapide, mais peut être sujet à des fautes d'orthographe phonétiques. Nous allons le démontrer ci-dessous pour le modèle Wav2Vec2.

Sonder les modèles CTC

Chargeons un petit extrait du jeu de données LibriSpeech ASR pour démontrer les capacités de transcription de Wav2Vec2 :

```python
from datasets import load_dataset

dataset = load_dataset(
    "hf-internal-testing/librispeech_asr_dummy", "clean", split="validation"
)
dataset
```

Sortie :

```
Dataset({
    features: ['file', 'audio', 'text', 'speaker_id', 'chapter_id', 'id'],
    num_rows: 73
})
```

Nous pouvons choisir l'un des 73 échantillons audio et en inspecter l'audio ainsi que la transcription :

```python
from IPython.display import Audio

sample = dataset[2]

print(sample["text"])
Audio(sample["audio"]["array"], rate=sample["audio"]["sampling_rate"])
```

Sortie :

```
HE TELLS US THAT AT THIS FESTIVE SEASON OF THE YEAR WITH CHRISTMAS AND ROAST BEEF LOOMING BEFORE US SIMILES DRAWN FROM EATING
AND ITS RESULTS OCCUR MOST READILY TO THE MIND
```

Noël et rôti de bœuf, ça sonne bien! 🎄 Après avoir choisi un échantillon de données, nous chargeons maintenant un *checkpoint finetuné* dans le `pipeline()`. Pour cela, nous utiliserons le *checkpoint* officiel [base Wav2Vec2] (facebook/wav2vec2-base-100h) *finetuné* sur 100 heures de données LibriSpeech :

```python
from transformers import pipeline

pipe = pipeline("automatic-speech-recognition", model="facebook/wav2vec2-base-100h")
```



```python
pipe(sample["audio"].copy())
{
```

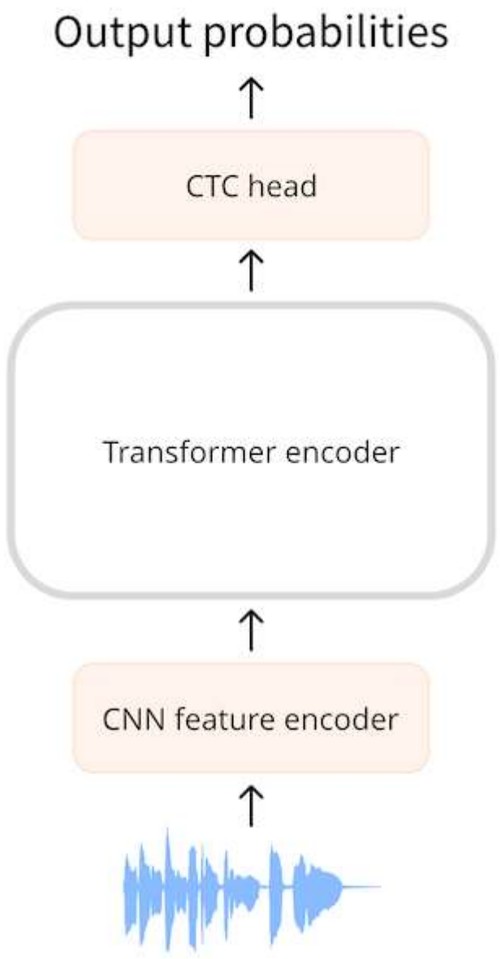
```
    "text": "HE TELLS US THAT AT THIS FESTIVE SEASON OF THE YEAR WITH CHRISTMAUS AND ROSE BEEF LOOMING BEFORE US SIMALYIS DRAWN
}
```


Nous pouvons voir que le modèle Wav2Vec2 fait un assez bon travail pour transcrire cet échantillon. A première vue, il semble généralement correct. Mettons la cible et la prédiction côte à côte et mettons en évidence les différences:

```
Target:      HE TELLS US THAT AT THIS FESTIVE SEASON OF THE YEAR WITH CHRISTMAS AND ROAST BEEF LOOMING BEFORE US SIMILES
DRAWN FROM EATING AND ITS RESULTS OCCUR MOST READILY TO THE MIND
Prediction:  HE TELLS US THAT AT THIS FESTIVE SEASON OF THE YEAR WITH **CHRISTMAUS** AND **ROSE** BEEF LOOMING BEFORE US
**SIMALYIS** DRAWN FROM EATING AND ITS RESULTS OCCUR MOST READILY TO THE MIND
```

En comparant le texte cible à la transcription prévue, nous pouvons voir que tous les mots *sonnent* correctement mais que certains ne sont pas orthographiés avec précision. Par exemple :

- *CHRISTMAUS* vs. *CHRISTMAS*
- *ROSE* vs. *ROAST*
- *SIMALYIS* vs. *SIMILES* Cela met en évidence les lacunes d'un modèle CTC. Un modèle CTC est essentiellement uniquement un modèle « acoustique» : il se compose d'un encodeur qui forme des représentations d'états cachés à partir des entrées audio, et d'une couche linéaire qui associe les états cachés aux caractères :



Cela signifie que le système base presque entièrement sa prédiction sur l'entrée acoustique qui lui a été donnée (les sons phonétiques de l'audio), et a donc tendance à transcrire l'audio de manière phonétique. Il donne moins d'importance au contexte de modélisation linguistique des lettres précédentes et successives, et est donc sujet aux fautes d'orthographe phonétique. Un modèle plus intelligent identifierait que *CHRISTMAUS* n'est pas un mot valide dans le vocabulaire anglais, et le corrigerait en *CHRISTMAS* lors de ses prédictions. Il nous manque également deux grandes fonctionnalités dans notre prédiction, la casse et la ponctuation, ce qui limite l'utilité des transcriptions du modèle aux applications réelles.

Passage à Seq2Seq

Comme indiqué dans l'unité 3, les modèles Seq2Seq sont formés d'un codeur et d'un décodeur reliés par un mécanisme d'attention croisée. L'encodeur joue le même rôle qu'auparavant, calculant des représentations d'états cachés des entrées audio, tandis que le décodeur joue le rôle d'un **modèle de langage**. Le décodeur traite toute la séquence de représentations d'états cachés de l'encodeur et génère les transcriptions de texte correspondantes. Avec le contexte global de l'entrée audio, le décodeur est capable d'utiliser le contexte du modèle de langage lorsqu'il fait ses prédictions, corrigeant les fautes d'orthographe à la volée et contournant ainsi le problème des prédictions phonétiques. Les modèles Seq2Seq présentent deux inconvénients :

1. Ils sont intrinsèquement plus lents lors du décodage puisqu'il se produit une étape à la fois, plutôt que tout à la fois 2. Ils nécessitent beaucoup plus de données d'entraînement pour converger

Le besoin de grandes quantités de données d'apprentissage a été un goulot d'étranglement dans l'avancement des architectures Seq2Seq pour la parole. Les données audios étiquetées sont difficiles à obtenir, les plus grands jeux de données annotées à l'époque ne totalisant que 10 000 heures. Tout cela a changé en 2022 avec la sortie de **Whisper**. Whisper est un modèle pré-entraîné pour la reconnaissance automatique de la parole publié en septembre 2022 par Alec Radford et al. d'OpenAI. Contrairement à ses prédécesseurs CTC, qui étaient entièrement pré-entraînés sur des données audio **non étiquetées**, Whisper est pré-entraîné sur une grande quantité de données de transcription audio **étiquetées**, 680 000 heures pour être précis. Il s'agit d'un ordre de grandeur de données plus grand que les données audio non étiquetées utilisées pour entraîner Wav2Vec 2.0 (60 000 heures). De plus, 117 000 heures de ces données de pré-entraînement sont des données multilingues. Il en résulte des *checkpoints* qui peuvent être appliqués à plus de 96 langues, dont beaucoup sont considérées comme *à faible ressource*. Lorsqu'ils sont passés à l'échelle, les modèles Whisper démontrent une forte capacité à généraliser sur de nombreux jeux de données et domaines. Obtenant des résultats compétitifs de pointe, avec un taux d'erreur de mots (WER) de près de 3% sur le sous-ensemble de test LibriSpeech et de 4,7% WER sur le sous-ensemble de test TED-LIUM (*cf.* Tableau 8 du papier de Whisper). La capacité de Whisper à gérer de longs échantillons audio, sa robustesse au bruit et sa capacité à prédire les transcriptions en casse et ponctuées revêtent une importance particulière. Cela en fait un candidat viable pour les systèmes de reconnaissance automatique de la parole du monde réel. Le reste de cette section vous montrera comment utiliser les modèles Whisper pré-entraînés pour la reconnaissance automatique de la parole à l'aide de 🤗 *Transformers*. Dans de nombreuses situations, les *checkpoints* pré-entraînés sont extrêmement performants et donnent d'excellents résultats, nous vous encourageons donc à essayer de les utiliser comme première étape pour résoudre tout problème de reconnaissance automatique de la parole. Grâce à un *finetuning*, les *checkpoints* peuvent être adaptés à des jeux de données et à des langues spécifiques afin d'améliorer encore ces résultats. Nous montrerons comment faire cela dans la section sur le *finetuning*. Whisper est disponible en cinq tailles différentes de modèles. Les quatre plus petits sont entraînés soit sur des données en anglais soit sur des données multilingues. Le *checkpoint* le plus grand est uniquement multilingue. Les neuf *checkpoints* pré-entraînés sont disponibles sur le *Hub*. Ils sont résumés dans le tableau suivant. « VRAM » indique la mémoire GPU requise pour exécuter le modèle avec une taille de batch minimale de 1. « *Rel Speed* » est la vitesse relative d'un *checkpoint* par rapport au plus grand modèle. Sur la base de ces informations, vous pouvez sélectionner le *checkpoint* le plus adapté à votre matériel.

Taille	Paramètres	VRAM / Go	*Rel Speed*	Anglais	Multilingue
tiny	39 M	1.4	32	✓	✓
base	74 M	1.5	16	✓	✓
small	244 M	2.3	6	✓	✓
medium	769 M	4.2	2	✓	✓
large	1550 M	7.5	1	x	✓

Chargeons le Whisper Base, qui est de taille comparable au Wav2Vec2 que nous avons utilisé précédemment. Anticipant notre passage à la reconnaissance automatique de la parole multilingue, nous allons charger la variante multilingue du *checkpoint* de base. Nous chargons également le modèle sur le GPU s'il est disponible, ou sur le CPU dans le cas contraire. Le `pipeline()` se chargera ensuite de déplacer toutes les entrées / sorties du CPU vers le GPU selon les besoins:

```
import torch
from transformers import pipeline
```

```
device = "cuda:0" if torch.cuda.is_available() else "cpu"
pipe = pipeline(
    "automatic-speech-recognition", model="openai/whisper-base", device=device
)
```

Bien, maintenant, transcrivons l'audio comme avant. Le seul changement que nous apportons est de passer un argument supplémentaire, `max_new_tokens`, qui indique au modèle le nombre maximum de *tokens* à générer lors de sa prédiction :

```
pipe(sample["audio"], max_new_tokens=256)
```

Sortie :

```
{'text': ' He tells us that at this festive season of the year, with Christmas and roast beef looming before us, similarly is drawn from eating and its results occur most readily to the mind.'}
```

La première chose que vous remarquerez est la présence de la casse et de la ponctuation. Cela rend immédiatement la transcription plus facile à lire par rapport à la transcription non casée et non ponctuée de Wav2Vec2. Mettons la transcription côte à côte avec la cible :

```
Target:     HE TELLS US THAT AT THIS FESTIVE SEASON OF THE YEAR WITH CHRISTMAS AND ROAST BEEF LOOMING BEFORE US SIMILES DRAWN FROM EATING AND ITS RESULTS OCCUR MOST READILY TO THE MIND
Prediction: He tells us that at this festive season of the year, with **Christmas** and **roast** beef looming before us, **similarly** is drawn from eating and its results occur most readily to the mind.
```

Whisper a fait un excellent travail pour corriger les erreurs phonétiques que nous avons vues avec Wav2Vec2 : *Christmas* et *roast* sont orthographiés correctement. Nous voyons que le modèle a encore du mal avec *SIMILES*, étant incorrectement transcrit comme *similarly*, mais cette fois la prédiction est un mot valide du vocabulaire anglais. L'utilisation d'un Whisper plus grand peut aider à réduire davantage les erreurs de transcription, au détriment d'un calcul plus important et d'un temps de transcription plus long. On nous a promis un modèle capable de gérer 96 langues, alors passons à de la reconnaissance automatique multilingue 🌎 ! Le jeu de données Multilingual LibriSpeech (MLS) est l'équivalent multilingue du jeu de données LibriSpeech, avec des données audio étiquetées en six langues. Nous allons charger un échantillon de l'échantillon espagnol de MLS, en utilisant le mode *streaming* afin de ne pas avoir à télécharger l'ensemble de données :

```
dataset = load_dataset(
    "facebook/multilingual_librispeech", "spanish", split="validation", streaming=True
)
sample = next(iter(dataset))
```

Encore une fois, nous allons inspecter la transcription du texte et écouter le segment audio:

```
print(sample["text"])
Audio(sample["audio"]["array"], rate=sample["audio"]["sampling_rate"])
```

Sortie :

```
entonces te delelitarás en jehová y yo te haré subir sobre las alturas de la tierra y te daré á comer la heredad de jacob tu padre porque la boca de jehová lo ha hablado
```

C'est le texte cible que nous visons avec notre transcription Whisper. Bien que nous sachions maintenant que nous pouvons probablement faire mieux, puisque notre modèle va aussi prédire la ponctuation et la casse, qui ne sont pas présents dans la référence. Transmettons l'échantillon audio au pipeline pour obtenir notre prédiction de texte. Une chose à noter est que le pipeline *consomme* le dictionnaire des entrées audio que nous entrons, ce qui signifie que le dictionnaire ne peut pas être réutilisé. Pour contourner ce problème, nous allons passer une *copie* de l'échantillon audio, afin de pouvoir réutiliser le même échantillon audio dans les exemples de code suivants :

```
pipe(sample["audio"].copy(), max_new_tokens=256, generate_kwargs={"task": "transcribe"})
```

Sortie :

```
{'text': ' Entonces te deleitarás en Jehová y yo te haré subir sobre las alturas de la tierra y te daré a comer la heredad de Jacob tu padre porque la boca de Jehová lo ha hablado.'}
```

Super, cela ressemble énormément à notre texte de référence (sans doute mieux car il a une ponctuation et une casse !). Vous remarquerez que nous avons transféré `"task"` en tant que *generate kwarg*. Définir la `"task"` à `"transcribe"` oblige Whisper à effectuer la tâche de reconnaissance de la parole, où l'audio est transcrit dans la même langue que le discours a été prononcé. Whisper est également capable d'effectuer la tâche étroitement liée de traduction de la parole, où l'audio en espagnol peut être traduit en texte en anglais. Pour y parvenir, nous définissons la `"task"` sur `"translate"` :

```
pipe(sample["audio"], max_new_tokens=256, generate_kwargs={"task": "translate"})
```

Sortie :

```
{'text': ' So you will choose in Jehovah and I will raise you on the heights of the earth and I will give you the honor of Jacob to your father because the voice of Jehovah has spoken to you.'}
```

Maintenant que nous savons que nous pouvons basculer entre la reconnaissance automatique de la parole et la traduction de la parole, nous pouvons choisir notre tâche en fonction de nos besoins. Soit nous reconnaissons l'audio dans la langue X vers le texte dans la même langue X (par exemple, l'audio espagnol vers le texte espagnol), soit nous traduisons de l'audio dans n'importe quelle langue X vers du texte en anglais (par exemple, l'audio espagnol vers le texte anglais). Pour en savoir plus sur la façon dont l'argument `"task"` est utilisé pour contrôler les propriétés du texte généré, reportez-vous à la carte de modèle pour le modèle Whisper base.

Longue transcription et horodatage

Jusqu'à présent, nous nous sommes concentrés sur la transcription de courts échantillons audio de moins de 30 secondes. Nous avons mentionné que l'un des attraits de Whisper était sa capacité à travailler sur de longs échantillons audio. Nous allons nous attaquer à cette tâche ici ! Créons un long fichier audio en concaténant des échantillons successifs à partir du jeu de données MLS. Étant donné que MLS est organisé en divisant de longs enregistrements de livres audio en segments plus courts, la concaténation d'échantillons est un moyen de reconstruire des passages de livres audio plus longs. Par conséquent, l'audio résultant doit être cohérent sur l'ensemble de l'échantillon. Nous allons définir notre durée audio cible sur 5 minutes et arrêter de concaténer des échantillons une fois que nous aurons atteint cette valeur :

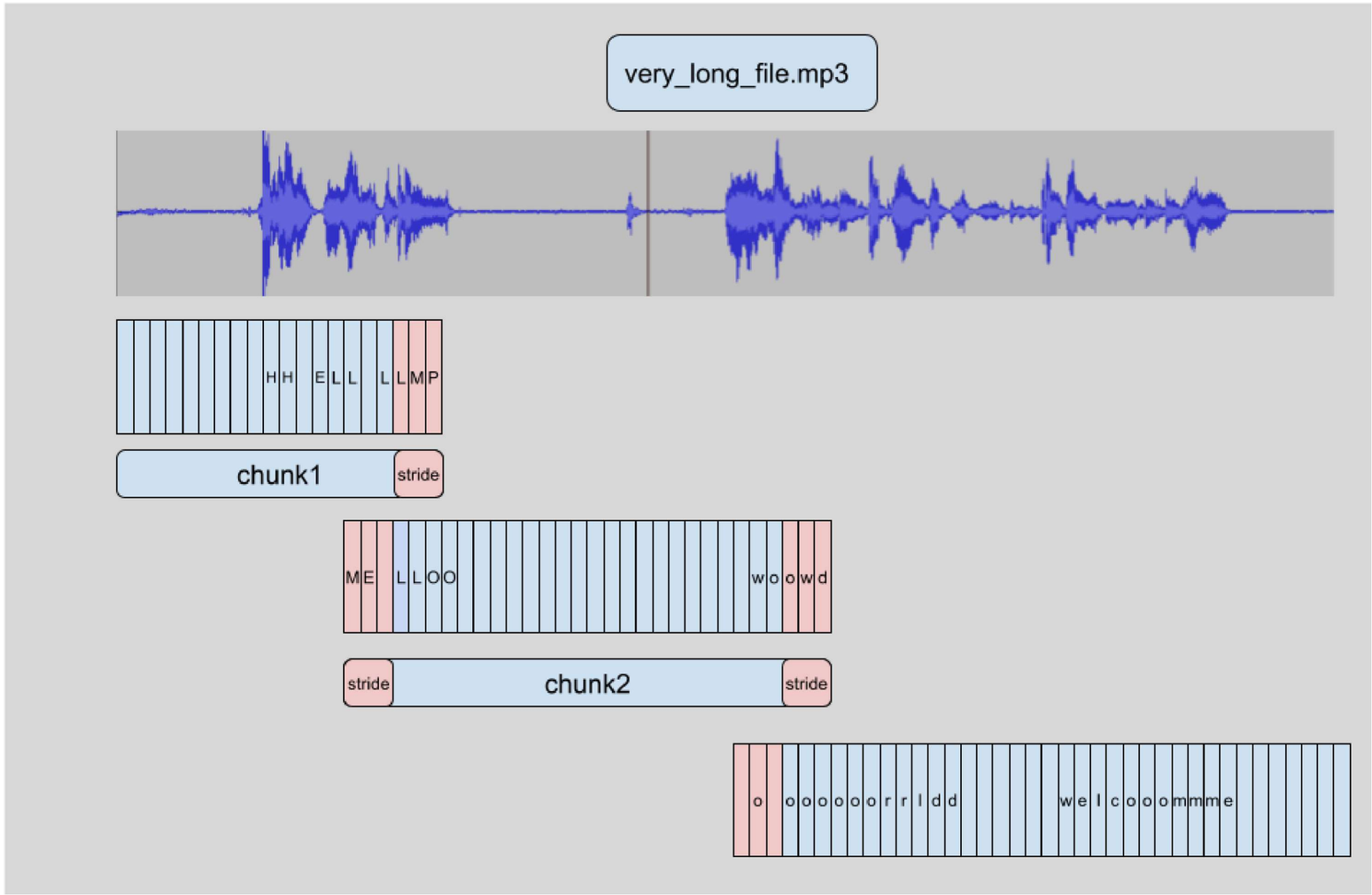
```
import numpy as np

target_length_in_m = 5

# convertir les minutes en secondes (* 60) en nombre d'échantillons (* taux d'échantillonnage)
sampling_rate = pipe.feature_extractor.sampling_rate
target_length_in_samples = target_length_in_m * 60 * sampling_rate

# itérer sur notre jeu de données en streaming, en concaténant les échantillons jusqu'à ce que nous atteignions notre cible
long_audio = []
for sample in dataset:
    long_audio.extend(sample["audio"]["array"])
    if len(long_audio) > target_length_in_samples:
        break

long_audio = np.asarray(long_audio)

# Résultat
seconds = len(long_audio) / 16000
minutes, seconds = divmod(seconds, 60)
print(f"Length of audio sample is {minutes} minutes {seconds:.2f} seconds")
```

Sortie :

```
Length of audio sample is 5.0 minutes 17.22 seconds
```

5 minutes et 17 secondes d'audio à transcrire. Le transfert de ce long échantillon audio directement vers le modèle pose deux problèmes :

1. Whisper est intrinsèquement conçu pour fonctionner avec des échantillons de 30 secondes. Tout ce qui est inférieur à 30s est rembourré à 30s avec du silence, et tout ce qui dépasse 30s est tronqué à 30s en coupant l'audio excédentaire. Donc si nous passons notre audio directement, nous n'obtiendrons la transcription que pour les 30 premières secondes
2. La mémoire dans un *transformer* évolue quadratiquement avec la longueur de séquence a : doubler la longueur d'entrée quadruple le besoin en mémoire, de sorte que le passage de fichiers audio très longs entraînera inévitablement une erreur de mémoire insuffisante (OOM : out-of-memory) La transcription de longs audio fonctionne dans 🤗 *Transformers* en segmentant l'audio d'entrée en segments plus petits et plus faciles à gérer. Chaque segment a un petit chevauchement avec le précédent. Cela nous permet de recoudre avec précision les segments aux bornes, car nous pouvons trouver le chevauchement entre les segments et fusionner les transcriptions en conséquence :



L'avantage de segmenter les échantillons est que nous n'avons pas besoin du résultat du bloc \(i \) pour transcrire le morceau suivant \(i + 1 \). La couture est effectuée après que nous ayons transcrit tous les morceaux aux bornes des morceaux. Donc peu importe l'ordre dans lequel nous transcrivons les morceaux. L'algorithme est entièrement **sans états**, donc nous pouvons même faire du découpage sur \(i + 1 \) en même temps que du découpage sur \(i \) ! Cela nous permet de former des *batch* de morceaux et de les exécuter dans le modèle en parallèle, offrant une grande accélération de calcul par rapport à leur transcription séquentielle. Pour en savoir plus sur la segmentation d'audio dans 🤗 *Transformers*, vous pouvez vous référer à cet [article de blog] (https://huggingface.co/blog/asr-chunking) (en anglais). Pour activer les longues transcriptions, nous devons ajouter un argument supplémentaire lorsque nous appelons le pipeline. Cet argument, `chunk_length_s` , contrôle la longueur des segments divisés en secondes. Pour Whisper, des morceaux de 30 secondes sont optimaux, car cela correspond à la longueur d'entrée attendue par Whisper. Pour activer le traitement par batch, nous devons passer l'argument `batch_size` au pipeline. En mettant tout cela ensemble, nous pouvons transcrire le long échantillon audio comme suit:

```
pipe(
    long_audio,
    max_new_tokens=256,
    generate_kwargs={"task": "transcribe"},
    chunk_length_s=30,
```

```
    batch_size=8,
)
```

Sortie :

```
{'text': ' Entonces te deleitarás en Jehová, y yo te haré subir sobre las alturas de la tierra, y te daré a comer la
heredad de Jacob tu padre, porque la boca de Jehová lo ha hablado. nosotros curados. Todos nosotros nos descarriamos
como bejas, cada cual se apartó por su camino, mas Jehová cargó en él el pecado de todos nosotros...
```

Nous n'affichons pas toute la sortie ici car elle est assez longue (312 mots au total). Sur un GPU V100 de 16 Go, vous pouvez vous attendre à ce que la ligne ci-dessus prenne environ 3,45 secondes à s'exécuter, ce qui est assez bon pour un échantillon audio de 317 secondes. Sur un CPU, attendez-vous à plus de 30 secondes. Whisper est également capable de prédire les *horodatages* au niveau du segment pour les données audio. Ces horodatages indiquent l'heure de début et de fin d'un court passage audio et sont particulièrement utiles pour aligner une transcription avec l'audio d'entrée. Supposons que nous voulions fournir des sous-titres pour une vidéo. Nous avons besoin de ces horodatages pour savoir quelle partie de la transcription correspond à un certain segment de vidéo, afin d'afficher la transcription correcte pour cette heure. L'activation de la prédiction d'horodatage est simple, il suffit de définir l'argument `return_timestamps=True` . Les horodatages sont compatibles avec les méthodes de segmentation et de traitement par batchs que nous avons utilisées précédemment. Nous pouvons donc simplement ajouter l'argument timestamp à notre code précédent :

```
pipe(
    long_audio,
    max_new_tokens=256,
    generate_kwargs={"task": "transcribe"},
    chunk_length_s=30,
    batch_size=8,
    return_timestamps=True,
)["chunks"]
```

Sortie :

```
[{'timestamp': (0.0, 26.4),
  'text': ' Entonces te deleitarás en Jehová, y yo te haré subir sobre las alturas de la tierra, y te daré a comer la heredad
de Jacob tu padre, porque la boca de Jehová lo ha hablado. nosotros curados. Todos nosotros nos descarriamos como bejas, cada
cual se apartó por su camino,'},
 {'timestamp': (26.4, 32.48),
  'text': ' mas Jehová cargó en él el pecado de todos nosotros. No es que partas tu pan con el'},
 {'timestamp': (32.48, 38.4),
  'text': ' hambriento y a los hombres herrantes metas en casa, que cuando vieres al desnudo lo cubras y no'},
 ...
```

Et le tour est joué ! Nous avons notre texte prédit ainsi que les horodatages correspondants.

Résumé

Whisper est un modèle pré-entraîné solide pour la reconnaissance automatique de la parole et la traduction. Par rapport à Wav2Vec2, il a une plus grande précision de transcription, avec des sorties qui contiennent la ponctuation et la casse. Il peut être utilisé pour transcrire la parole en anglais ainsi que dans 96 autres langues, à la fois sur des segments audio courts et des segments plus longs par le biais de la segmentation. Ces attributs en font un modèle viable pour de nombreuses tâches de reconnaissance automatique de la parole et de traduction sans avoir besoin d'être *finetuné*. La méthode `pipeline()` fournit un moyen facile d'inférer en une ligne de code avec un contrôle sur les prédictions générées. Bien que le modèle Whisper fonctionne extrêmement bien sur de nombreuses langues à ressources élevées, il a une précision de transcription et de traduction plus faible sur les langues à faibles ressources, c'est-à-dire celles pour lesquelles les données d'apprentissage sont moins facilement disponibles. Les performances varient également selon les accents et dialectes de certaines langues, y compris une précision moindre pour les locuteurs de différents sexes, races, âges ou autres critères démographiques (*cf.* le papier de Whisper). Pour améliorer les performances sur les langues, les accents ou les dialectes à faibles ressources, nous pouvons prendre le modèle Whisper pré-entraîné et l'entraîner sur un petit corpus de données sélectionnées de manière appropriée, dans un processus appelé *finetuning*. Nous montrerons qu'avec seulement dix heures de données supplémentaires, nous pouvons améliorer les performances du modèle Whisper de plus de 100% sur une langue à faibles ressources. Dans la section suivante, nous aborderons le processus de sélection d'un jeu de données à *finetuner*.