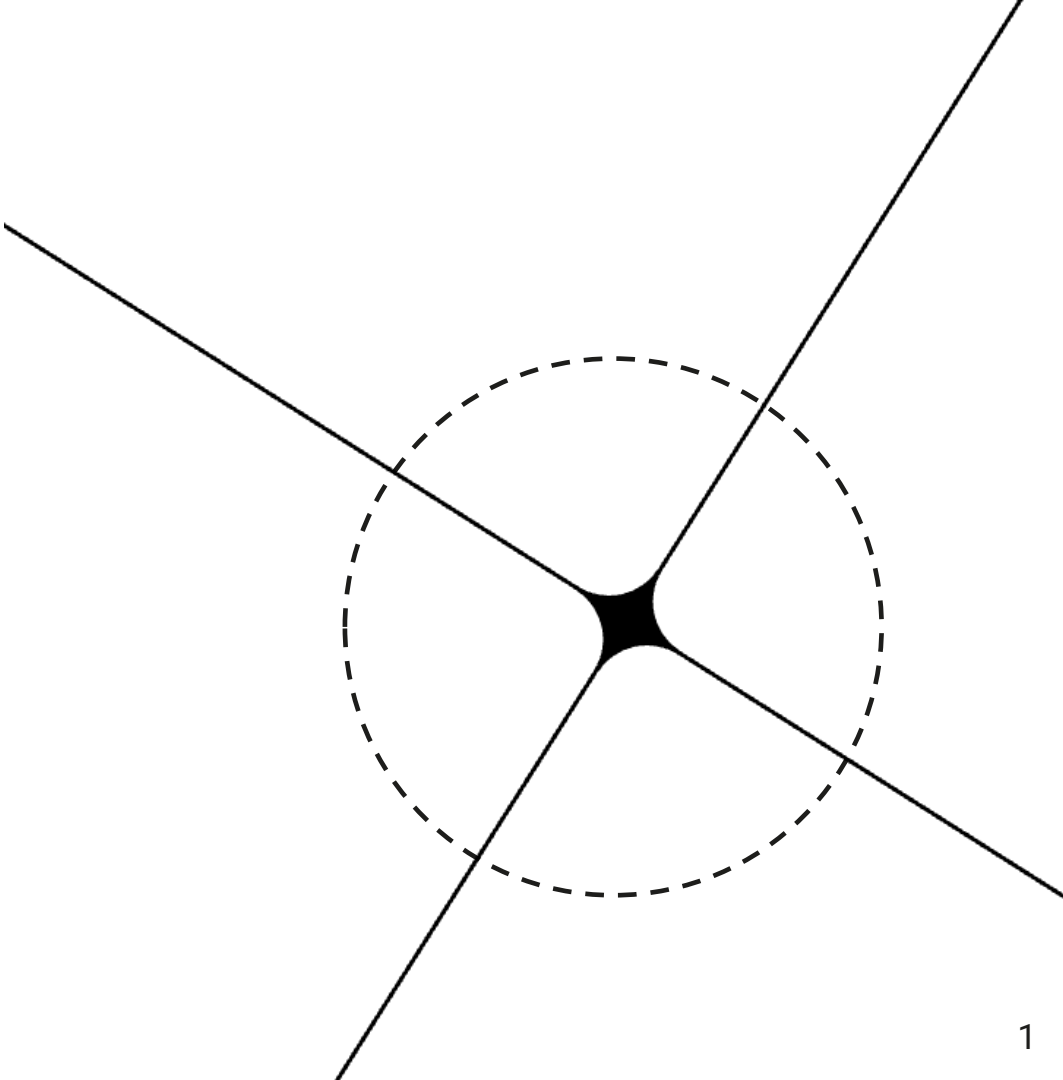
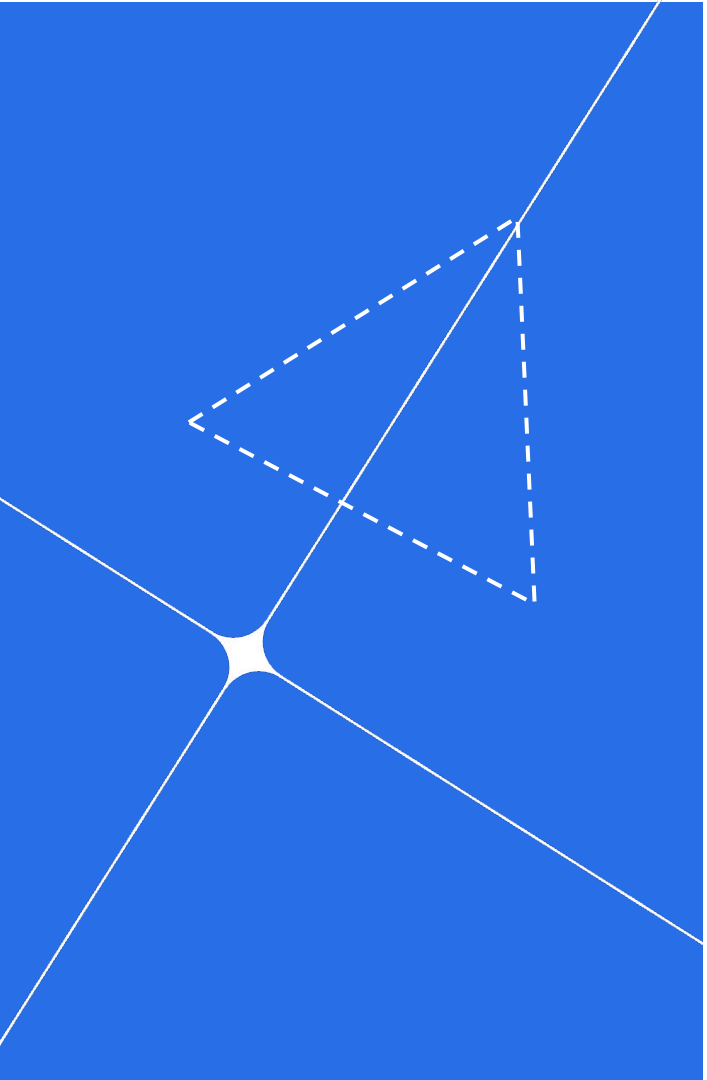
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis



Deep learning pour l'audio

Julien Romero



Motivation & panorama

Pourquoi le deep learning audio maintenant ?

- Audio = canal “naturel” (parole, ambiance, musique), souvent plus accessible que texte
- Explosion des usages : assistants vocaux, call centers, médias, IoT, santé, voiture
- L’audio apporte du *contexte* absent du texte : intonation, émotion, bruit, identité vocale
- Gains DL vs méthodes classiques : robustesse, end-to-end, transfert (foundation models)
- Convergence multimodale : audio + texte + image/vidéo dans les systèmes produits
- Objectif du cours : comprendre les représentations, familles de modèles, ASR/TTS, mise en prod

Cas d'usage "produit" : 6 exemples concrets

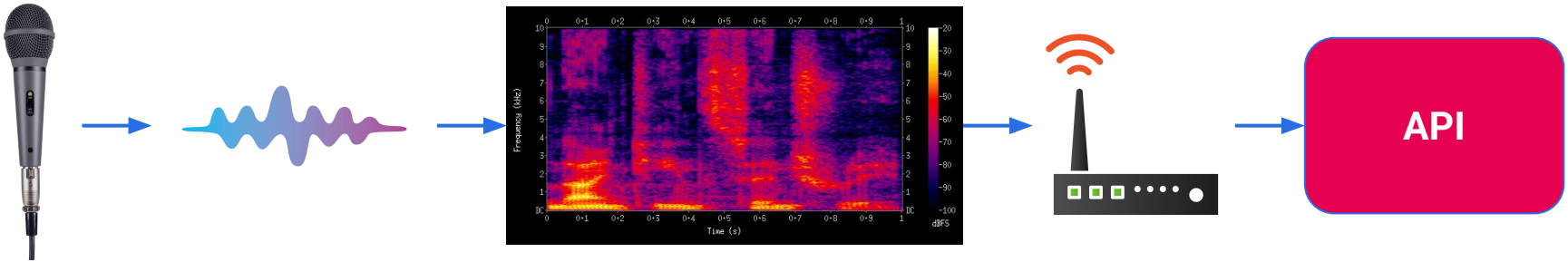
- **ASR** (Automatic Speech Recognition) : dictée, sous-titres, recherche dans podcasts
- **TTS** (Text-To-Speech) : voice assistant, lecture d'articles, accessibilité, doublage
- **Classification** : langue, intention, spam vocal, "quality monitoring" call center
- **Audio event detection** : sirène, verre brisé, toux, machine en défaut, "smart home"
- **Speaker tasks** : identification (who), vérification (is it X?), diarization (who spoke when)
- **Music** : tagging, recommandation, séparation de sources (voix/instruments)

Pourquoi c’est plus dur que l’image ?

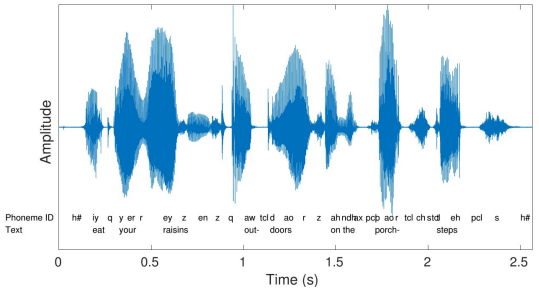
- Signal 1D long : 16 kHz × 10 s = **160k** échantillons (séquences très longues)
- Variabilité extrême : accent, débit, micro, pièce, bruit, overlap speakers
- Alignement temporel non trivial : “mots” ≠ fenêtres fixes
- Forte sensibilité aux prétraitements (normalisation, resampling, silence trimming)
- Évaluation dépend du contexte : WER (ASR), MOS (TTS), F1 (events), EER (speaker)
- Données coûteuses : labels précis + privacy (voix = donnée personnelle)

Pipeline mental : de l'onde au produit

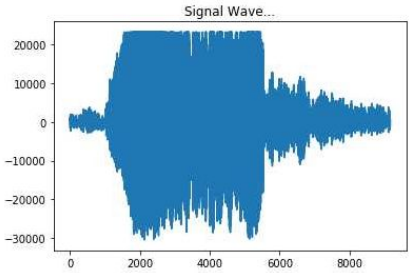
- Acquisition : micro / fichier / stream, fréquence d'échantillonnage (sample rate)
- Prétraitements : mono/stéréo, resampling, normalisation, VAD (Voice Activity Detection)
- Représentation : waveform brute ou time-frequency (spectrogramme, mel)
- Modèle : CNN/RNN/Transformer, ou **SSL** (Self-Supervised Learning) + head
- Post-traitement : décodage (ASR), vocoder (TTS), agrégation temporelle (classification)
- Intégration : latence, coût GPU/CPU, streaming, monitoring, drift



Mini-révision signal audio

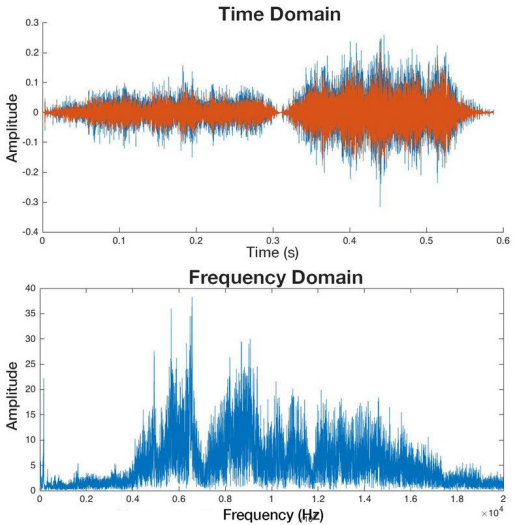


- **Signal audio (temps continu)** : fonction x(t) (pression acoustique, ou tension micro) en fonction du temps t.
- **Waveform / forme d'onde** : représentation de x(t) (ou x[n]) dans le **domaine temporel**.
- **Échantillonnage** : passage au discret $x[n]=x(nT_s)$ avec **période d'échantillonnage** Ts.
  - **Fréquence d'échantillonnage** $F_s=1 / T_s$ (ex. 16 kHz, 44.1 kHz).
  - **Nyquist** : fréquence max représentable $F_s/2$. Au-delà ⇒ **aliasing** (repliement spectral).
- **Quantification / bit depth** : l'amplitude est discrétisée sur un nombre de bits (ex. 16 bits). Plus de bits ⇒ moins de bruit de quantification, meilleure **dynamique**.
- **Amplitude** : "hauteur" du signal.
  - **RMS** (Root Mean Square) : mesure d'énergie moyenne (liée au "volume" perçu).
  - **dB** : échelle logarithmique. En audio numérique : **dBFS** (Decibels relative to Full Scale, 0 dBFS = saturation).
  - **Clipping** : écrêtage quand $|x[n]|$ dépasse la plage représentable (distorsion).
- **Fréquence (Hz)** : nombre d'oscillations par seconde.
  - **Période** $T=1/f_T$.
  - **Pitch** : perception de la hauteur.



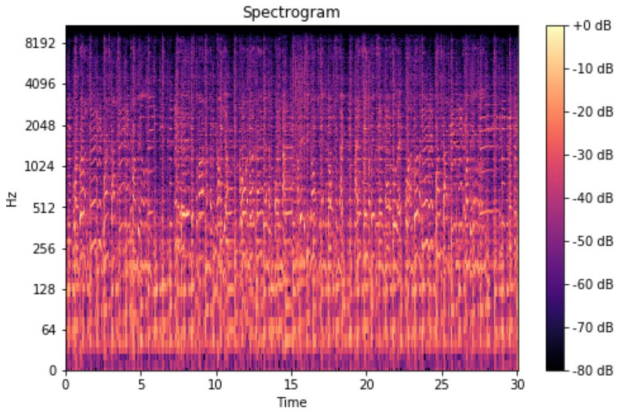
Mini-révision signal audio

- **Transformée de Fourier (spectre)** : décompose le signal en sinusoïdes.
- **DFT / FFT** (discrete/fast Fourier transform) : version discrète et calcul rapide. Donne un spectre X[k] (complexe) :
  - **Magnitude** |X| (énergie par fréquence) et **phase**.
  - La **phase** décrit **où on en est dans un cycle d'oscillation** à un instant donné, par rapport à une référence.
- **Fenêtrage (windowing)** : on coupe le signal en tranches courtes (frames) et on applique une **fenêtre** (Hann, Hamming…) pour réduire la **fuite spectrale** (spectral leakage).
  - **win_length** : durée de la tranche (ex. 25 ms).
  - **hop_length** : pas entre deux tranches (ex. 10 ms).
  - **overlap** : recouvrement entre tranches (= win_length − hop_length).
- **STFT (Short-Time Fourier Transform)** : Fourier "locale" sur chaque tranche ⇒ représentation **temps–fréquence**.
  - **n_fft** : taille FFT (résolution fréquentielle).
  - **Trade-off temps/fréquence** : fenêtre courte ⇒ bonne résolution temporelle, moins bonne en fréquence (et inversement).



Mini-révision signal audio

- **Échelle Mel** : échelle fréquentielle “perceptive” (plus fine dans les basses fréquences).
  - **Filterbank Mel** : banque de filtres triangulaires qui projette le spectre sur **n_mels** bandes.
  - **Log-Mel spectrogram** : log(Mel($|\mathrm{STFT}|^2$)) = entrée standard pour beaucoup de modèles.
- **MFCC** (Mel-Frequency Cepstral Coefficients) : DCT (discrete cosine transform) des log-mel (compaction), très utilisé en “audio classique”, moins incontournable en deep learning moderne.
- **Pourquoi on révise ça ?** Parce que les “hyperparamètres” de pré-traitement (F_s, win_length, hop_length, n_fft, n_mels, log…) changent directement l’information donnée au modèle, le coût calcul, et la robustesse (bruit, micro, domaine).



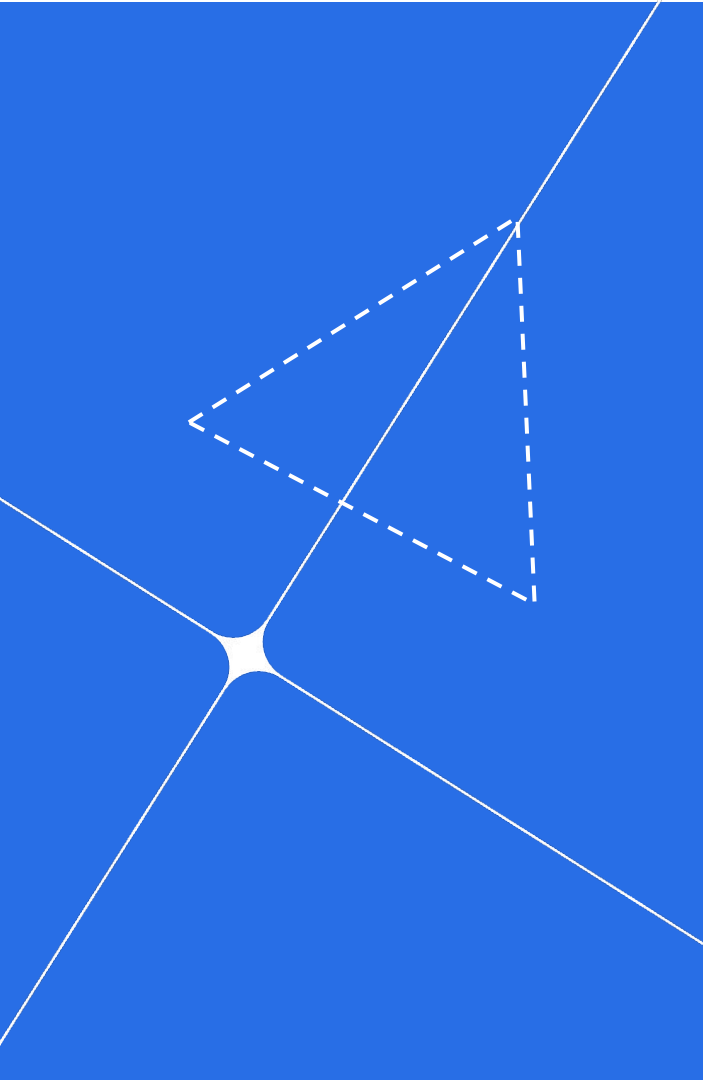
Représentations usuelles (et quand les choisir)

- **Waveform brute** : end-to-end possible, mais séquences très longues ; besoin d'énormes données
- **Spectrogramme magnitude** : standard pour classification/ASR ; proche "image" 2D
- **Mel-spectrogramme** : compact, aligné perception ; très courant pour TTS (conditionnement vocoder)
- **MFCC** (Mel-Frequency Cepstral Coefficients) : compact, historique ; utile baseline léger
- Features "prosodiques" : pitch (F0), energy, duration (souvent utiles en TTS/expressif)
- Règle pratique : si "produit rapide" + petit dataset → mel/mfcc + modèle pré-entraîné

Exemple code (ultra générique) : charger et passer en mel

```
import torchaudio

wav, sr = torchaudio.load("sample.wav")      # wav: [channels, time]
wav = wav.mean(dim=0, keepdim=True)          # mono
wav = torchaudio.functional.resample(wav, sr, 16000)
mel = torchaudio.transforms.MelSpectrogram(
    sample_rate=16000, n_fft=400, hop_length=160, n_mels=80
)(wav)                                       # mel: [1, n_mels, frames]
logmel = (mel + 1e-6).log()
print(wav.shape, logmel.shape)
```

Lecture : n_fft=400 (~25 ms à 16 kHz), hop=160 (~10 ms), 80 bandes mel



Représentations & prétraitements

Représentations & prétraitements

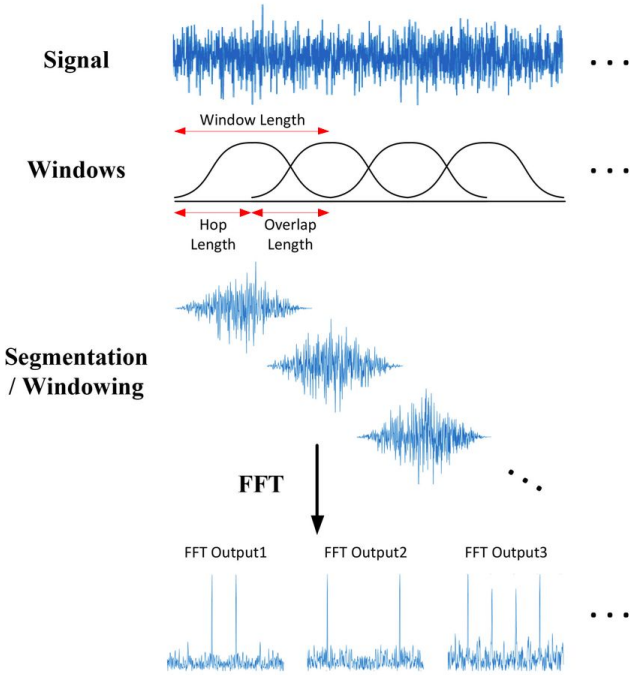
- Pourquoi une représentation :
  - invariances utiles (bruit, micro)
  - stabilité apprentissage
  - coût compute
- Objectif : transformer un signal long et instable en features “apprenables”
- Deux grandes familles : **waveform** (1D) vs **time–frequency** (2D)
- Même modèle, résultats très différents si (sr, n_fft, hop, n_mels, log) changent
- Bon réflexe produit : choisir représentation en fonction (latence, précision, données)
- On vise ici : classification, ASR, TTS

Waveform vs spectrogramme : ce qu'on gagne / perd

- **Waveform** : garde phase + détails fins ; séquences très longues (mémoire/temps)
- **Spectrogramme** : compresse la dimension temporelle en frames ; exploitable en CNN/Transformer
- Phase souvent ignorée en entrée (on garde |STFT|) : OK pour beaucoup de tâches de reconnaissance
- Pour TTS/vocoder, la phase redevient critique (reconstruction du signal)
- Heuristique : petit dataset + robustesse souhaitée → spectro/log-mel
- Heuristique : gros modèle end-to-end + beaucoup de data → waveform possible

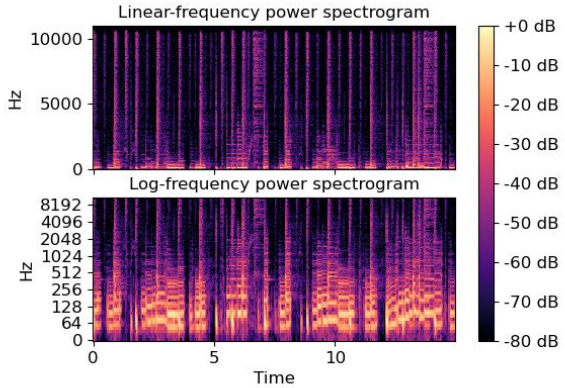
STFT : paramètres qui changent tout

- **STFT** (Short-Time Fourier Transform) : FFT sur frames (fenêtres) successives
- **n_fft** : résolution fréquentielle (plus grand = plus fin en fréquence, plus cher)
- **win_length** : durée de frame (ex. 25 ms voix) ; influence le flou temporel
- **hop_length** : pas entre frames (ex. 10 ms) ; plus petit = plus de frames = plus cher
- **Overlap** = win_length − hop_length : améliore continuité, augmente coût
- Bon point de départ speech : 16 kHz, win≈25 ms, hop≈10 ms (à ajuster selon tâche)



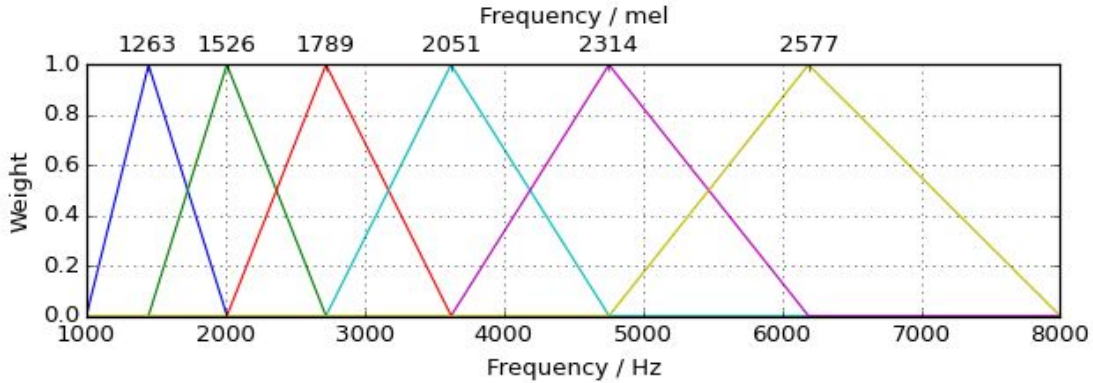
Magnitude, power, log : dynamique et stabilité

- STFT donne un spectre **complexe** : magnitude $|X|$ et phase
- **Magnitude spectrogram** : $|X|$; **Power spectrogram** : $|X|^2$
- Échelle linéaire très “étendue” : quelques fréquences dominent, gradients instables
- **Log** : compresse la dynamique (proche perception), améliore apprentissage
- Typique : log(x + eps) avec eps pour éviter log(0)
- Attention : la normalisation (mean/std) doit être cohérente train/inference



Mel filterbank : compacter sans (trop) dégrader

- **Mel** : échelle fréquentielle perceptive (plus de résolution dans les basses)
- **Filterbank mel** : projette le spectre sur n_mels bandes (ex. 64/80/128)
- Gain : moins de dimensions, plus robuste, souvent meilleur sur petits datasets
- Coût : perte de précision fréquentielle fine (peut gêner certaines tâches musique)
- Log-mel = entrée standard pour beaucoup de modèles (classification, TTS)
- Paramètres à documenter : sample_rate, n_mels, f_min, f_max



MFCC : utile, mais à utiliser avec intention

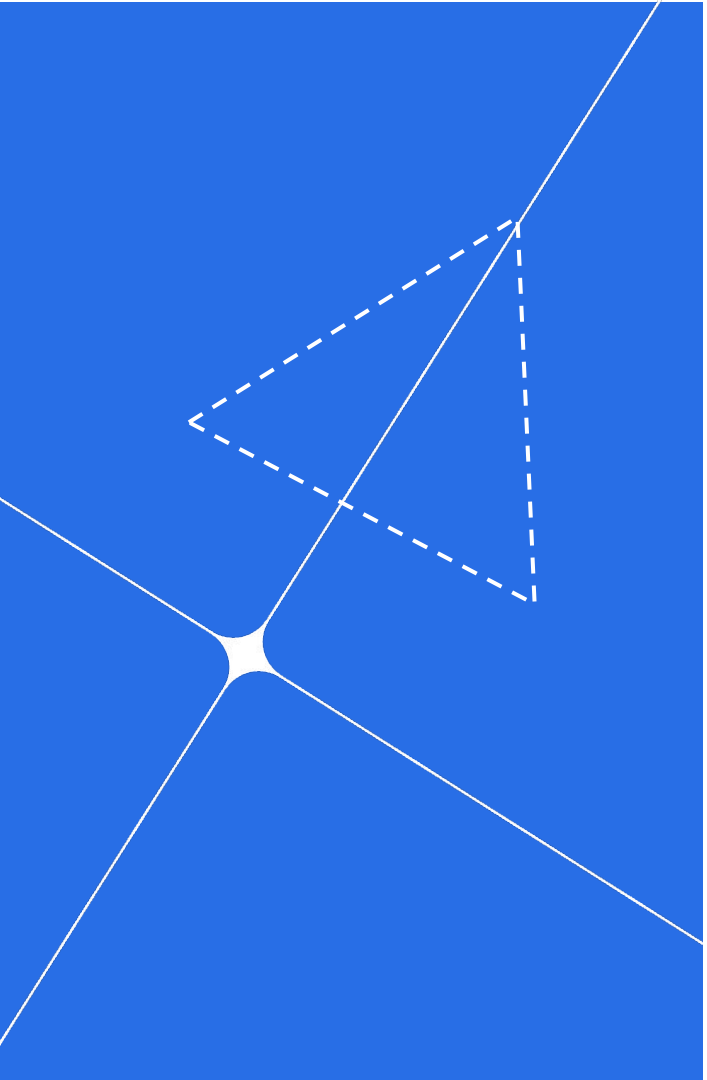
- **MFCC** (Mel-Frequency Cepstral Coefficients) : DCT des log-mel (compression + décorrélation)
- Avantages : très compact, rapide, bon baseline “classique”
- Limites : perd de l’information fine ; parfois inférieur aux log-mel pour DL moderne
- Pratique : MFCC si contraintes fortes (edge/CPU) ou baseline rapide
- Pratique : log-mel si modèle moderne (CNN/Transformer) + besoin de performance
- Ne pas mélanger MFCC/log-mel sans raison (features différentes, inductive bias différent)

Prétraitements “prod” qui évitent des catastrophes

- **Resampling** : imposer un sample_rate unique (sinon features incohérentes)
- **Mono** : convertir stéréo → mono (sauf cas musique spatial)
- **Normalisation amplitude** : éviter que le gain micro devienne un “label caché”
- **Trim silence** (avec prudence) : utile pour classification, dangereux pour ASR (contexte)
- **VAD** (Voice Activity Detection) : détecter segments de voix (réduit coût, latence)
- Traçabilité : stocker la config feature-extractor avec le modèle (contrat d’entrée)

Augmentations audio : robustesse sans “tricher”

- Objectif : simuler la variabilité réelle (micro, pièce, bruit) sans changer la classe
- Baselines : ajout de bruit, reverb, gain aléatoire, time shift
- **SpecAugment** : masquage en temps + en fréquence sur log-mel (on “cache” des bandes/segments)
- Attention : augmentation trop forte = label noise (dégrade perf)
- Règle : commencer simple, mesurer impact, versionner la politique d’augmentation
- Exemple attendu : modèle qui généralise mieux à un autre micro / autre pièce



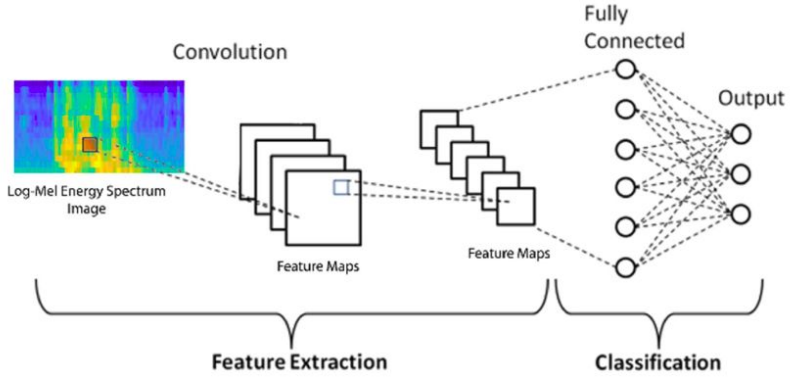
Encoders audio modernes

Pourquoi autant de modèles audio ?

- Même entrée (log-mel) peut servir à des tâches très différentes : keyword spotting, events, ASR, TTS
- Audio = signal temporel : dépendances locales (phonèmes) + globales (mots, phrases, contexte)
- Besoin de compromis : **qualité** vs **latence** vs **coût** vs **robustesse** (bruit, micro)
- Familles de modèles = réponses à des contraintes différentes (données, compute, streaming)
- Stratégie “produit” fréquente :
  - **pré-entraîné** (encoder) + **tête** (head) adaptée
  - Comme pour les images
- Ici : panorama des biais inductifs (CNN / RNN / Transformer / Conformer / SSL)

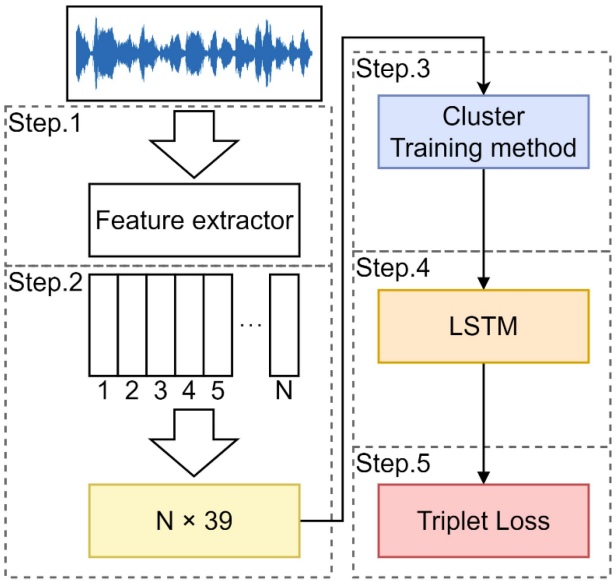
CNN 2D sur log-mel : le baseline industriel solide

- On traite le spectrogramme comme une image
- Hypothèse : motifs locaux temps-fréquence ⇒ convolution efficace
- Très bon pour **classification** (events, keyword spotting, intent acoustique)
- Avantages : rapide, stable à entraîner, bon sur petits datasets, facile à quantifier
- Limites :
  - contexte long difficile
  - pour ASR longue phrase, pas suffisant seul
- Variantes : ResNet, MobileNet-like, CRNN (CNN + RNN), pooling temporel
- Heuristique : si “je veux un modèle robuste demain” ⇒ CNN + log-mel



RNN (LSTM/GRU) : historique, encore utile en séquence

- Rappels :
  - **RNN** (Recurrent Neural Network) : état caché qui se propage dans le temps
  - **LSTM/GRU** : mécanismes pour réduire le “vanishing gradient” sur longues séquences
- Points forts :
  - modélise naturellement la chronologie
  - simple pour séquences modestes
- Points faibles
  - parallélisation limitée (lent)
  - long contexte reste difficile
- Usage actuel : souvent en “composant” (ex. après CNN) plutôt qu’architecture principale
- Bon réflexe : comprendre RNN pour lire des systèmes legacy / embarqués

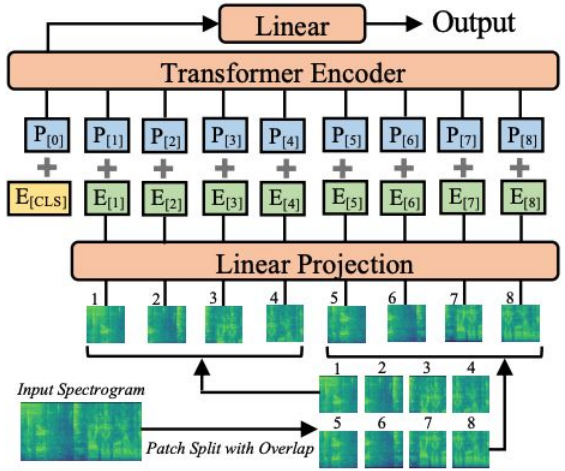


Transformers : contexte global + parallélisation

- **Transformer** : self-attention sur une séquence de frames (ou de patches)
- Points forts :
  - capte dépendances longues
  - s'entraîne efficacement sur GPU
- Point faible :
  - coût attention ~ $O(T^2)$ en longueur de séquence (T = nb frames)
- Contournements : downsampling (strides), pooling, attention "efficiente", chunking
- Pour audio, attention seule manque parfois de biais local (d'où Conformer)
- En pratique : très bon pour ASR, embeddings audio, multimodal audio-texte

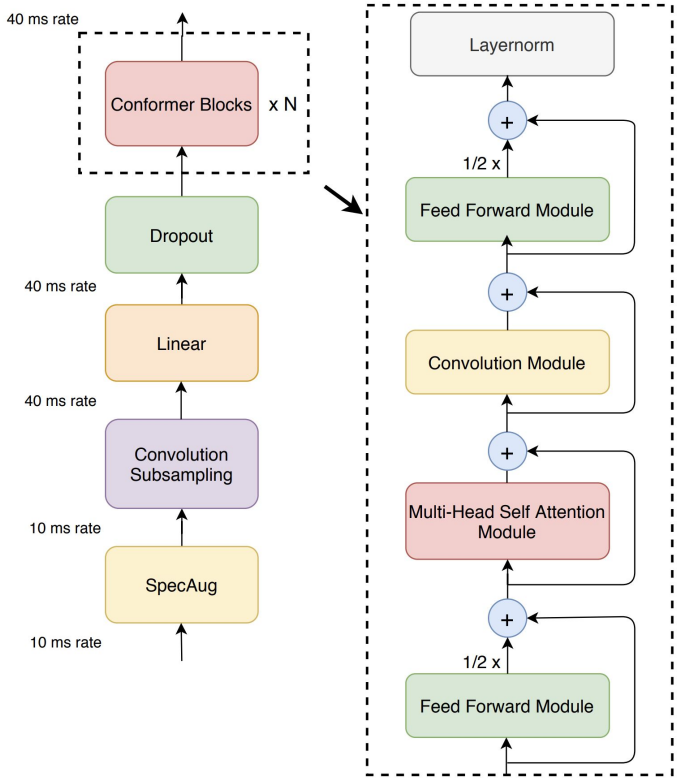
Audio Spectrogram Transformer (AST)

- Entrée : spectrogramme découpé en **patches** (comme ViT en vision)
- Chaque patch devient un token ; positional embeddings sur temps/fréquence
- Avantages : réutilise recettes ViT, bon pour classification / tagging
- Limites : dépend fortement de la représentation (patch size, normalisation)
- Produit : utile quand on veut un modèle “general audio tagging” prêt à l’emploi
- À retenir : patching = manière simple de réduire T (moins de tokens)



Conformer : Transformer + convolution pour la parole

- **Conformer** = architecture hybride : self-attention (global) + convolution (local)
- Motivation : parole contient structure locale forte (formants, transitions phonémiques)
- Résultat : meilleur compromis précision/robustesse que Transformer “pur” sur speech
- Typique dans ASR moderne (souvent avec CTC, classification temporelle connexionniste, ou transducer en décodage)
- Conformer peut être décliné “streaming” avec contraintes de causalité
- À retenir : “local + global” est une idée directrice en audio



Self-Supervised Learning (SSL) : encoder pré-entraîné + head léger

- **SSL** (Self-Supervised Learning) : apprendre des représentations sans labels (mask/predict)
  - Similaire à l'entraînement des LLMs, mais adapté à l'audio
  - LLM prédit des tokens de texte
  - SSL audio apprend à reconstruire/predire des segments audio masqués (souvent via pseudo-tokens), pour produire un encodeur généraliste réutilisable.
- Output typique : embeddings frame-level ou segment-level réutilisables
- Pattern produit : geler l'encoder, entraîner une petite tête (linear / MLP) sur votre tâche
- Avantages :
  - performant sur petits datasets
  - meilleure généralisation cross-domain
- Limites :
  - dépendance au domaine (speech vs environmental)
  - coût d'inférence parfois élevé
- Exemples de familles : wav2vec2/HuBERT/WavLM (speech), autres pour "general audio"

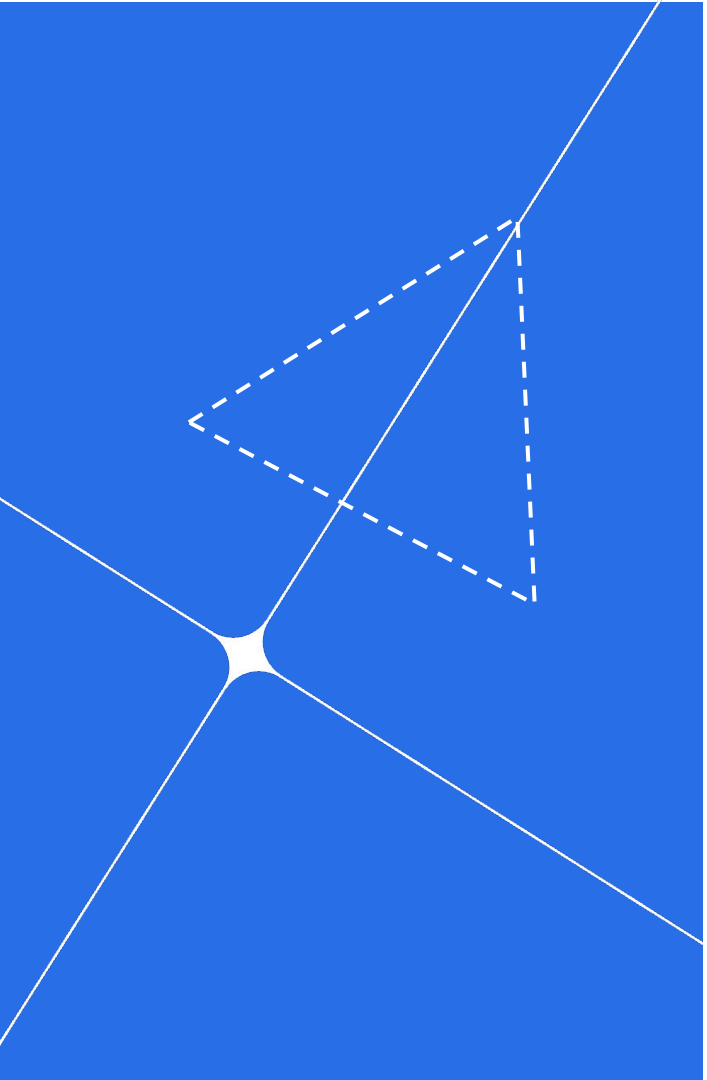
Exemple code générique : encoder pré-entraîné + head classification

```python
import torch
from transformers import AutoProcessor, AutoModel

processor = AutoProcessor.from_pretrained("facebook/wav2vec2-base")   # example
encoder = AutoModel.from_pretrained("facebook/wav2vec2-base")

# wav: torch.Tensor shape [batch, time] sampled at expected sample_rate
inputs = processor(wav, sampling_rate=16000, return_tensors="pt", padding=True)
with torch.no_grad():
    h = encoder(**inputs).last_hidden_state          # [B, T, D]

# simple pooling + linear head (illustrative)
x = h.mean(dim=1)                                     # [B, D]
logits = torch.nn.Linear(x.size(-1), num_classes)(x)  # [B, C]
```

- Lecture : encoder = gros coût ; head = petit coût ; pooling = choix d'agrégation temporelle

Choisir une famille en pratique : heuristiques “produit”

- Classification/events, petit dataset, latence faible : **CNN** + log-mel
- Speech embeddings / transfert : **SSL encoder** + head, fine-tune si besoin
- ASR offline haute qualité : **Transformer/Conformer** + décodage adapté
- ASR streaming : modèle **causal** (ou chunked) + VAD (Voice Activity Detection) + gestion “partial results”
- TTS : modèle texte→acoustique + **vocoder** (qualité vs RTF)
- Toujours documenter : budget latence, hardware cible, taille modèle, métrique principale



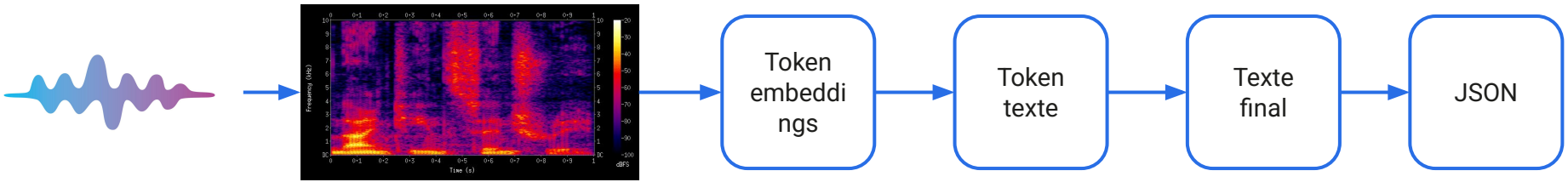
ASR (Audio-to-Text)

Pourquoi l'ASR est un problème "produit" (pas juste un modèle)

- **ASR** (Automatic Speech Recognition) : convertir audio parole → texte exploitable
- Valeur business immédiate : recherche, sous-titres, CRM/call-center, assistants vocaux
- Contraintes réelles : bruit, accents, micro, overlap, latence, coûts d'inférence
- Le "système ASR" inclut : prétraitements, segmentation, décodage, post-traitement
- Objectif ingénieur : optimiser **qualité + latence + robustesse**, pas uniquement WER
- Un bon ASR doit produire : texte, timestamps, langue, confiance (confidence) si possible

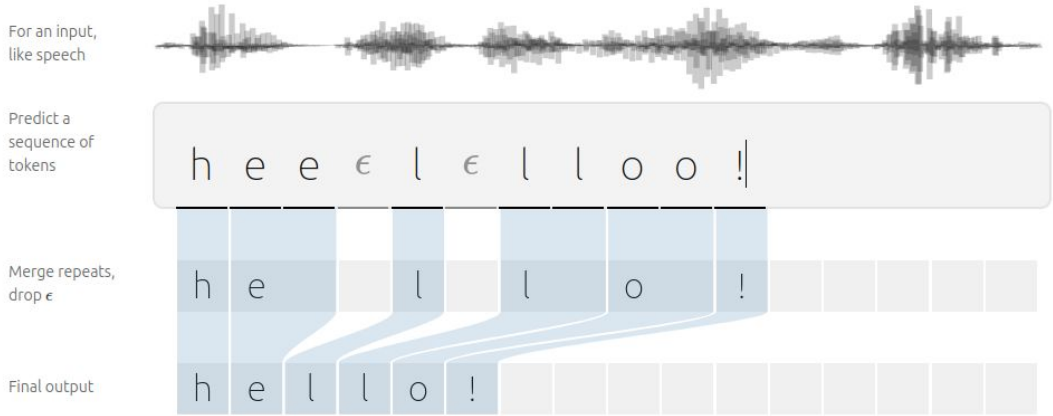
Pipeline ASR : de l'audio aux tokens texte

- Entrée : waveform normalisée (sample rate contractuel, typiquement 16 kHz pour speech)
- Optionnel : **VAD** (Voice Activity Detection) pour détecter les segments de voix
- Feature extraction : log-mel ou encodeur direct waveform (selon modèle)
- Encodeur : transforme frames audio → représentations latentes (embeddings)
- Décodeur : convertit représentations → **tokens** (subwords / BPE)
- Post-traitement : ponctuation, normalisation, formatage (nombres, dates), filtrage PII



Deux paradigmes majeurs : CTC vs Seq2Seq

- **CTC** (Connectionist Temporal Classification) : alignement implicite audio↔texte sans segmentation
  - CTC produit une séquence “frame-level” avec token spécial **blank** ; décodage ensuite
  - Avantages CTC : simple, efficace, bien adapté au streaming et aux encodeurs SSL
  - Limites CTC : modélise moins bien dépendances linguistiques sans LM (Language Model)
- **Seq2Seq** (encoder–decoder) : génère tokens texte conditionnés par l’audio + contexte précédent
  - Avantages Seq2Seq : meilleure modélisation langage, ponctuation/style ; limites : latence, complexité



CTC génère une lettre ou un blank pour chaque frame

Modèles “prêts à l’emploi” : ce qu’ils donnent (et ce qu’ils cachent)

- Whisper-like / large seq2seq : robuste multi-langues, souvent bon “out of the box”
- Produisent parfois : détection langue, timestamps, segmentation automatique
- Qualité dépend fortement : domaine (call center, médical), bruit, débit, vocabulaire
- Attention : “prompting”/context peut biaiser (noms propres, jargon, style)
- Coût : plus le modèle est gros, plus la latence et le coût GPU montent
- Bon réflexe : mesurer WER/CER sur échantillon représentatif avant toute conclusion

Modèle / famille (exemples)	Offline vs streaming	Langues	Timestamps	Coût relatif	Forces typiques	Limites / “ce que ça cache”
Whisper-like (seq2seq encoder–decoder)	Surtout offline (streaming via chunking côté système)	Multilingue	Souvent oui (segments/chunks)	Moyen → élevé	Robustesse “out of the box”, accents/bruit mieux gérés que baselines, usage simple	Latence + coût GPU, erreurs sur jargon métier, timestamps parfois approximatifs, “prompt/context” peut biaiser le texte
Conformer + CTC (ASR speech-focused)	Très compatible streaming (causal/chunked)	Souvent mono ou quelques langues (selon entraînement)	Oui (frame-level)	Moyen	Bon compromis précision/latence, architecture optimisée pour speech, contrôle fin du pipeline	Nécessite souvent tuning (VAD, LM), perf dépend du domaine, qualité chute si mismatch micro/bruit
wav2vec2/HuBERT/WavLM encoder + head CTC	Offline et streaming possibles (selon architecture/implémentation)	Variable (souvent pré-entraîné large, fine-tune langue)	Oui (via CTC/frames)	Faible → moyen	Excellent transfert, efficace sur petits datasets avec fine-tuning, embeddings réutilisables	Peut nécessiter fine-tuning + décodage soigné, sensibilité aux prétraitements, dépendance au domaine d’entraînement

Modèle / famille (exemples)	Offline vs streaming	Langues	Timestamps	Coût relatif	Forces typiques	Limites / “ce que ça cache”
RNN-T / Transducer (streaming ASR)	Streaming natif (low latency)	Variable	Oui (selon impl.)	Moyen → élevé	Latence très faible, stable en streaming, bon pour assistants vocaux	Entraînement/serving plus complexe, décodage spécifique, moins “plug-and-play”
Services / APIs ASR (cloud)	Les deux (selon offre)	Souvent multilingue	Souvent oui	Coût financier récurrent	Time-to-market rapide, scalabilité, features annexes (langue, diarization parfois)	Dépendance fournisseur, coût, contraintes privacy/compliance, faible contrôle sur erreurs et mises à jour

Décodage : là où se cache une partie de la performance

- Sortie modèle = distributions sur tokens ; il faut choisir une séquence finale
- **Greedy decoding** : rapide, parfois moins bon (choix local)
- **Beam search** : explore plusieurs hypothèses ; meilleur mais plus cher (beam width)
- Optionnel : **LM** (Language Model) externe pour guider vers texte plausible (surtout CTC)
- Normalisation : casing, ponctuation, chiffres (ex. “vingt-deux” vs “22”)
- Pour timestamps : alignement via frames (CTC) ou via tokens alignés (selon modèle)

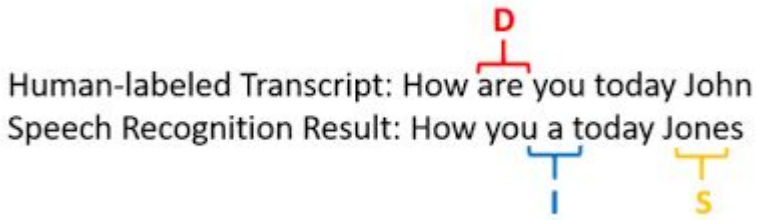
Exemple code générique : ASR inference (Transformers)

```python
from transformers import pipeline

asr = pipeline(
    task="automatic-speech-recognition",
    model="openai/whisper-small",   # exemple : modèle prêt à l'emploi
    device="cuda"
)

out = asr("sample.wav", return_timestamps=True)
print(out["text"])
print(out.get("chunks", [])[:2])    # timestamps si disponibles
```

- *pipeline* gère : feature extractor, batching, décodage (mais “boîte noire”)
- À instrumenter côté produit : temps total, temps “first token”, longueur audio, device

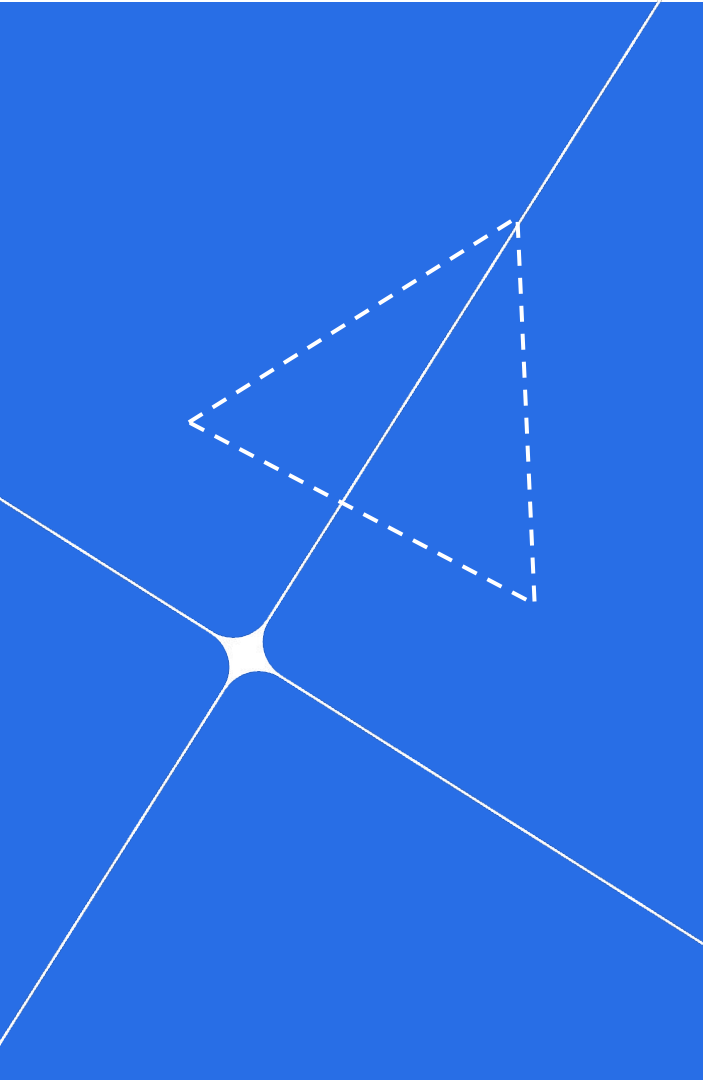
Évaluation : WER/CER et pièges classiques

- **WER** (Word Error Rate) = (S+D+I)/N : substitutions, suppressions, insertions
- **CER** (Character Error Rate) utile si tokenisation mots ambiguë (langues, ponctuation)
- Normalisation avant scoring : minuscules, suppression ponctuation (selon objectif produit)
- Attention aux chiffres, abréviations, noms propres : erreurs coûteuses “business”
- Dataset d’éval doit refléter : bruit, micro, accents, vocabulaire cible, durées
- Ne pas confondre amélioration WER globale et amélioration sur cas critiques (tail risk)



ASR en production : streaming, overlap, privacy

- Streaming : chunking + hypothèses partielles
  - KPI = latence + stabilité du texte (flicker)
- **VAD** crucial
  - réduit compute, améliore UX
  - risques : couper des phonèmes / mots
- **Diarization** (qui parle quand) nécessaire en réunions/call centers
  - overlap difficile
- Sécurité & privacy
  - audio = donnée sensible
  - minimiser stockage, anonymiser si besoin
- Monitoring : dérive micro/bruit, drift vocabulaire, chutes de confiance, taux d'échec
- Plan B : fallback (modèle plus petit), file d'attente, mode dégradé CPU



TTS (Text-to-Speech)

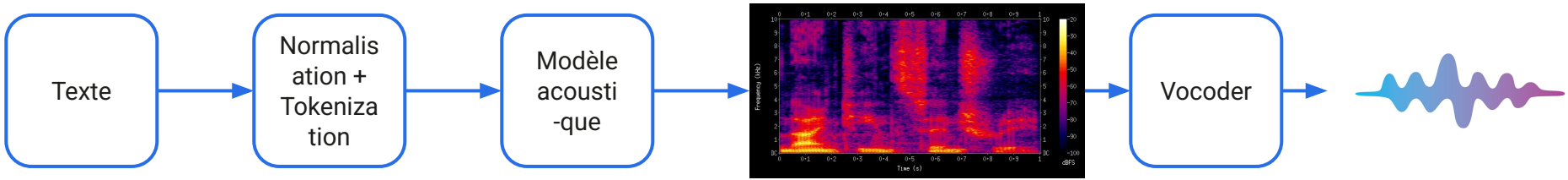
Pourquoi la TTS est un problème “produit” (pas juste de la génération)

- **TTS** (Text-To-Speech) : transformer du texte en audio naturel et intelligible
- Valeur : accessibilité, assistants vocaux, support client, narration, contenus dynamiques
- Le vrai objectif n’est pas “une voix jolie”
  - **intelligibilité + prosodie + latence**
- Contraintes : temps de réponse (first audio), stabilité, gestion ponctuation/nombres
- Variabilité : styles (neutre, enthousiaste), voix (speaker), langues/accents
- Risques : usurpation / deepfake vocal ; gouvernance et garde-fous requis

Demo: https://styletts.github.io/

Pipeline TTS moderne : texte → acoustique → waveform

- Étape 1 : texte → tokens (caractères, phonèmes, BPE), normalisation (nombres, abréviations)
- Étape 2 : modèle “acoustic” prédit une représentation audio (souvent **mel-spectrogram**)
- Étape 3 : **vocoder** reconstruit la waveform à partir du mel (ou d’un latent)
- Deux familles d’architectures : autoregressive vs non-autoregressive (vitesse)
- Conditionnements possibles : speaker embedding, style, émotion, vitesse, pitch
- Le pipeline complet doit être cohérent : tokenizer + features + vocoder compatibles



Représentation texte : graphemes, phonèmes, et normalisation

- **Graphemes** : lettres ; simple, mais ambigu (orthographe ≠ prononciation)
- **Phonèmes** : symboles de prononciation
  - meilleure qualité, nécessite lexique/G2P (grapheme-to-phoneme)
- **Text normalization** : “22/02/2026” → “vingt-deux février deux mille vingt-six”
- Ponctuation : contrôle prosodique (pauses, intonation), mais source d’erreurs si bruitée
- Multilingue : choix du tokenizer + règles de normalisation par langue
- Produit : la qualité TTS dépend souvent plus du texte “propre” que du modèle

Modèle acoustique : prédire le mel, contrôler la prosodie

- Objectif : produire un **mel-spectrogram** aligné temporellement avec le texte
- Contrôle prosodie : durée (durations), énergie, pitch (hauteur), pauses
- Non-autoregressif souvent préféré en produit : plus stable, plus rapide
- Attention au **long-form** : paragraphes → gestion de la respiration/pauses
- Conditionnement “style” : vecteur style, tokens de style, ou prompts (selon approche)
- La robustesse passe par : bon alignement + normalisation + métriques adaptées

Vocoders : la qualité finale (et la latence) se joue ici

- **Vocoder** : modèle qui convertit mel/latent en waveform (reconstruction audio)
- Autoregressive (WaveNet-like) : qualité historique, latence élevée
- GAN vocoders (HiFi-GAN-like) : rapides, souvent temps-réel, artefacts possibles
- Diffusion vocoders : qualité élevée, coût variable selon nb steps
- KPI produit : **RTF** (Real-Time Factor) = temps de génération / durée audio
- Latence perçue : “time-to-first-audio” + streaming waveform (chunked synthesis)

Évaluation TTS : MOS et métriques (avec prudence)

- **MOS** (Mean Opinion Score) : note subjective moyenne (panel), référence industrie
- Intelligibilité : tests de compréhension, ou proxies (ASR sur sortie TTS) avec limites
- Artefacts : buzz, metallic, instabilité pitch, “robotic”, coupures
- Mesures objective possibles : PESQ/STOI (Perceptual Evaluation of Speech Quality/Short-Time Objective Intelligibility pour la voix) mais corrélation imparfaite
- Évaluer sur scripts variés : nombres, dates, acronymes, noms propres, phrases longues
- Toujours séparer : qualité “voix” vs qualité “texte” (normalisation/prononciation)

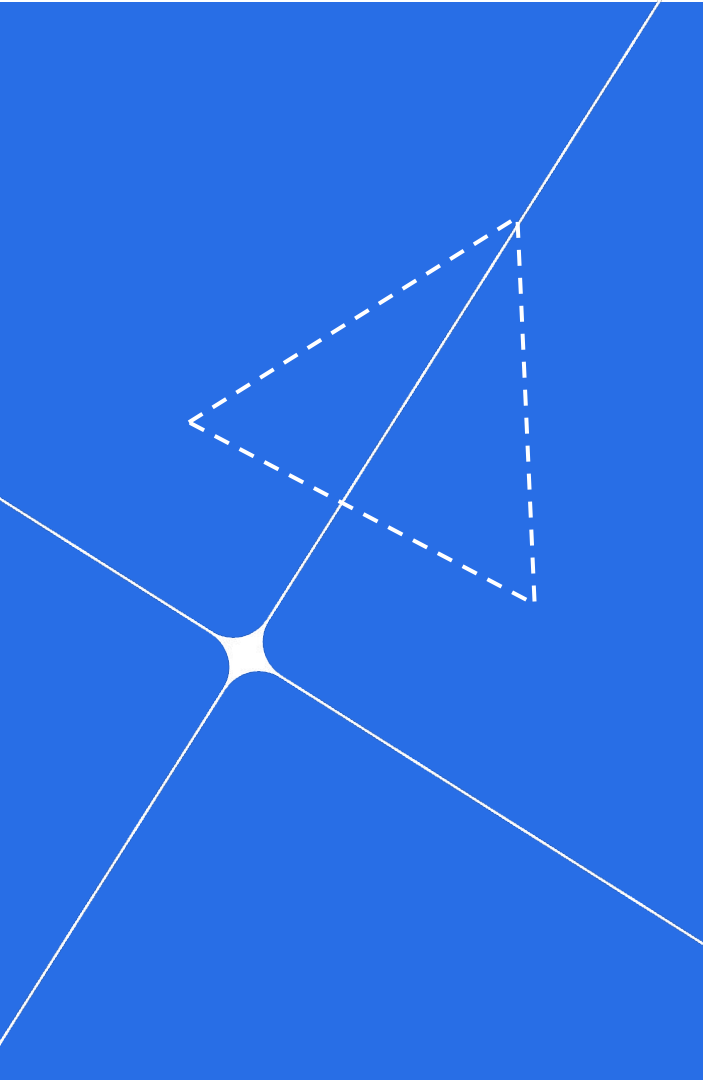
Exemple code générique : TTS prêt à l'emploi

```python
from transformers import pipeline

tts = pipeline(
    task="text-to-speech",
    model="facebook/mms-tts-fra",    # exemple : TTS FR
    device="cuda"
)

out = tts("Bonjour, votre rendez-vous est confirmé pour le 22 février à 14h30.")
audio = out["audio"]          # numpy array
sr = out["sampling_rate"]
print(audio.shape, sr)
```

- "Prêt à l'emploi" ≠ "prêt pour prod" : normalisation texte + latence + monitoring restent
- Toujours loguer : texte input, durée output, RTF, erreurs de synthèse

TTS en production : contrôle, sécurité, et modes dégradés

- Contrôle : vitesse, pitch, style → attention aux valeurs extrêmes (instabilité)
- Cohérence voix :
  - speaker embedding stable
  - éviter dérive sur long contexte
- Streaming TTS : générer par segments (phrases) + cross-fade pour éviter ruptures
- Sécurité : détecter demandes de “voice cloning” non autorisées
  - watermarking possible
- Privacy : texte peut contenir PII
  - logs et rétention doivent être minimisés
- Mode dégradé : basculer vers voix moins coûteuse si surcharge GPU



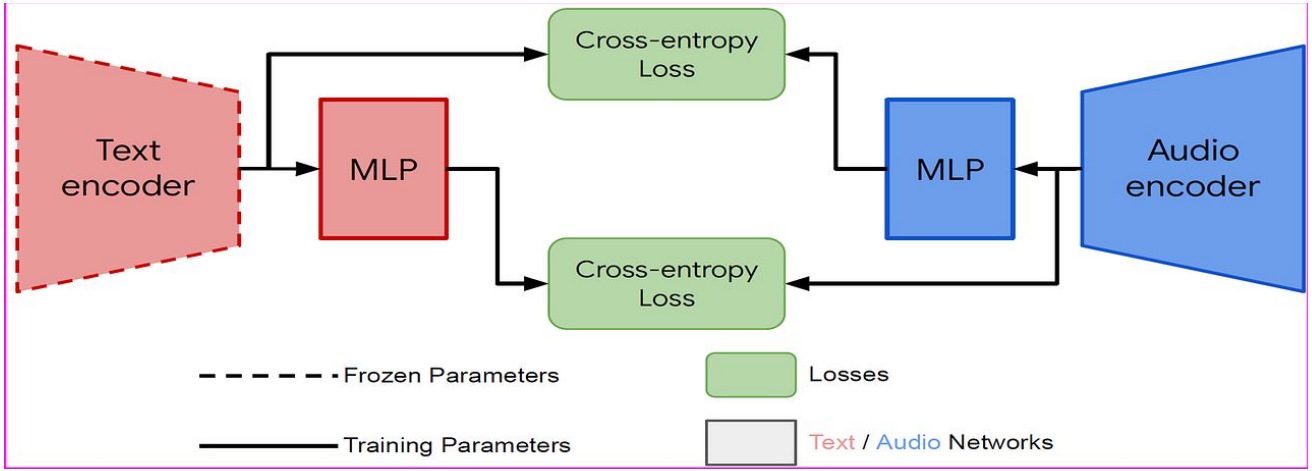
Multimodal audio–texte & retrieval

Motivation : au-delà de ASR/TTS, “comprendre” l’audio par le texte

- Beaucoup de produits veulent : “trouver / taguer / filtrer” un audio sans transcription parfaite
- Recherche sémantique : “un rire”, “une perceuse”, “applaudissements”, “voix énervée”
- L’ASR ne suffit pas : sons non verbaux, musique, environnement, bruit de fond
- Approche moderne : aligner audio et texte dans un **même espace d’embeddings**
- Résultat : zero-shot classification, retrieval, tagging, captioning (selon supervision)
- Ici : principe contrastif “CLIP-like”, métriques retrieval, risques produit

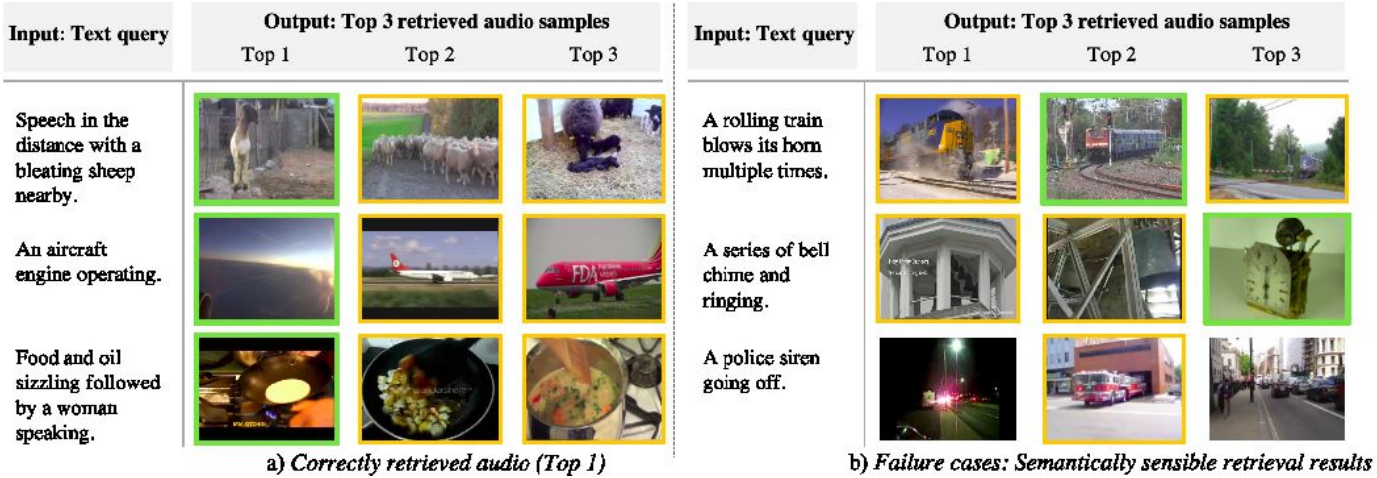
Embeddings alignés audio–texte : principe “CLIP-like”

- Deux encodeurs : **audio encoder** + **text encoder** (Transformer côté texte)
- Sortie : vecteurs normalisés ; similarité par cosine similarity
- Apprentissage : rapprocher paires (audio, texte) correspondantes, éloigner les autres
- Supervision : captions, tags, métadonnées, transcripts, weak labels
- Avantage : généralise mieux que classifier “fermée” sur N classes fixes
- Limite : dépend fortement de la qualité des textes associés (bruit, biais)



Retrieval et zero-shot classification : comment ça marche concrètement

- **Retrieval** : requête texte → embedding texte → top-K audios les plus similaires
- **Zero-shot classification** : labels décrits en texte (“sound of siren”) → choisir le plus proche
- “Prompt engineering” léger : formulations de labels influencent fortement les scores
- Calibration : scores cosine ne sont pas des probabilités
  - Il faut apprendre des seuils
- Bon usage produit : discovery, tri, pre-filtering, human-in-the-loop
- Mauvais usage : décision finale avec haut enjeux sans calibration/éval dédiée



a) *Correctly retrieved audio (Top 1)* b) *Failure cases: Semantically sensible retrieval results*

Audio Retrieval with Natural Language Queries, Oncescu et al.

Cas d’usage produit typiques (et pattern d’architecture)

- Indexation de médias : tagging automatique de podcasts/vidéos, chapitrage
- Monitoring industriel : “anomaly sounds” vs descriptions d’incidents
- Recherche “par description” : sound libraries, effets sonores, archives
- Modération : détecter cris, violence, sirènes (attention biais + faux positifs)
- Pattern : embeddings → **vector DB** (Chroma-like) → top-K → reranker optionnel
- Intégration : batching, cache embeddings, déduplication, latence top-K

Exemple code générique : embeddings + recherche vectorielle

```
# pseudo-code
audio_emb = audio_encoder(logmel)          # [D]
text_emb  = text_encoder("sound of rain")  # [D]

audio_emb = audio_emb / audio_emb.norm()
text_emb  = text_emb  / text_emb.norm()

score = (audio_emb * text_emb).sum()       # cosine (si normalisé)
topk = vectordb.search(text_emb, k=10)     # retourne ids + scores
```

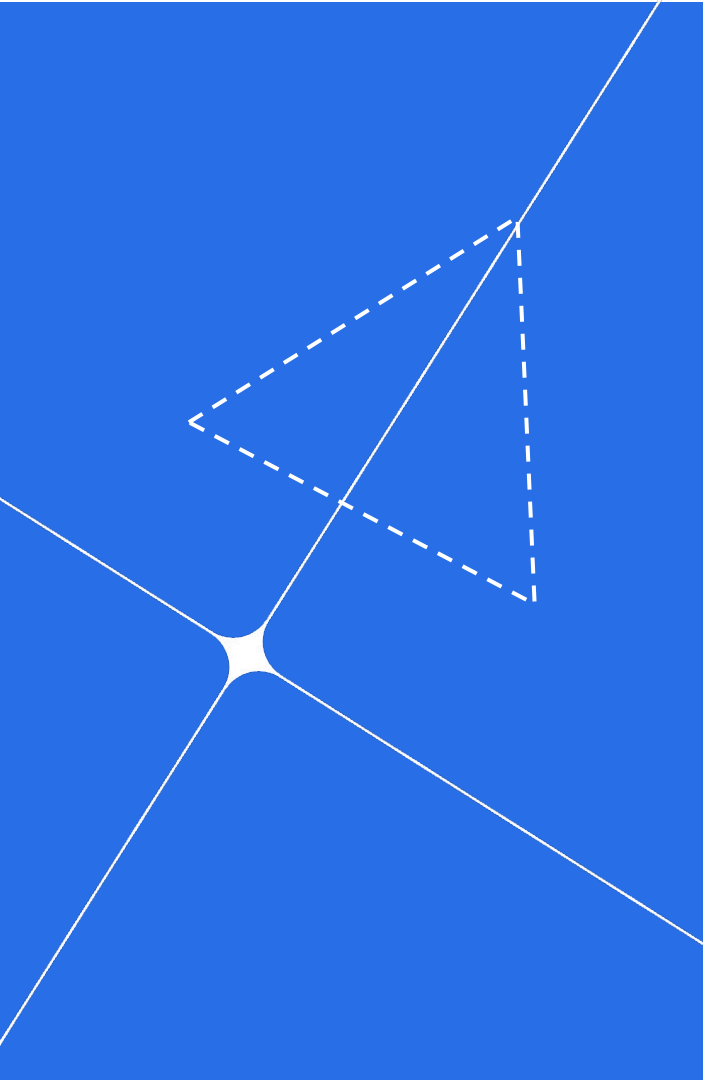
- Point clé : mêmes normalisations, même version d’encodeurs en prod
- Stocker : id, embedding, metadata (durée, source, droits) pour filtrage

Évaluation retrieval : métriques et protocoles

- Objectif : mesurer “retrouver les bons audios” plutôt que “classer parfaitement”
- **Recall@K** : proportion de requêtes où un bon résultat apparaît dans top-K
- **mAP** (mean Average Precision) : qualité de ranking sur toute la liste
- Dataset : requêtes texte + set de “relevants” (souvent multi-positifs)
- Attention aux fuites : mêmes enregistrements/variantes dans train et test
- Analyse qualitative indispensable : erreurs dues à ambiguïtés linguistiques

Risques et pièges (multimodal en prod)

- Biais des textes : tags incomplets, descriptions subjectives, langues mélangées
- Domain shift : modèle entraîné “web audio” ≠ usine / hôpital / voiture
- Sons rares : long tail, manque de supervision, faux positifs coûteux
- Sécurité : requêtes “adversariales” (ambiguës) + calibration faible
- Gouvernance : droits d’auteur (musique), consentement, données sensibles
- Mitigation : filtrage metadata, seuils, human review, active learning



Mise en production d’un système audio DL

Motivation : l'audio "casse" facilement en prod

- En prod, l'audio vient de vrais micros : gain variable, codecs, réverb, bruit, coupures
- Le pipeline audio a plus de paramètres à régler qu'en vision : sampling rate, VAD, chunking, normalisation, decode
- KPI produits multi-objectifs : qualité (WER/MOS) + latence + coût + robustesse
- Les erreurs sont parfois silencieuses : dégradation progressive (drift) difficile à voir
- Contraintes fortes : streaming, temps réel, concurrence, GPU scheduling (Slurm)
- Objectif : définir un **contrat d'entrée**, instrumenter, et prévoir des modes dégradés

Contrat d'entrée : la checklist qui évite 80% des bugs

- Fixer sample_rate (ex. 16 kHz speech), mono/stéréo, dtype, amplitude range
- Définir normalisation : RMS target / peak norm / loudness (et la version exacte)
- Définir segmentation : taille chunk, overlap, silence trimming, VAD on/off
- Définir feature extractor : n_fft, hop, n_mels, log, mean/std, f_min/f_max
- Versionner ce contrat avec le modèle (config + hash)
    - refuser inputs non conformes
- Tests automatiques : “golden files” (audio→features) + tolérances numériques

Serving : offline batch vs temps réel streaming

- Offline batch : throughput élevé, batching GPU, latence moins critique
- Temps réel : priorité à latence (P95/P99), “time-to-first-token/audio”
- Stratégie streaming : chunking (ex. 0.5–1 s) + overlap, outputs partiels stables
- Accélération : torch.compile / onnx / TensorRT selon stack, mais attention parity
- Gestion charge : file d’attente, autoscaling, quotas, backpressure
- Caching : embeddings audio (retrieval), modèles/pipelines initialisés (warm start)

Mesurer correctement : latence, coût, qualité (et leurs compromis)

- Latence : P50/P95/P99, séparée en (I/O, préproc, modèle, décodage, postproc)
- Coût : GPU-seconds par minute audio, mémoire VRAM, coût stockage/transfert
- Qualité : WER/CER (ASR), MOS/RTF (TTS), Recall@K (retrieval), F1 (events)
- Mettre en place des “canaries” : petits sets audio représentatifs rejoués régulièrement
- Surveiller la dérive : bruit, micro, accent, domaine, longueur moyenne des inputs
- Toujours relier à l’usage : erreurs critiques (noms propres, chiffres, intents)

Monitoring : ce qu'on logue (sans exploser la privacy)

- Logs techniques : sampling rate détecté, durée, niveau RMS, clipping rate, SNR (Signal Noise Ratio) proxy, device
- Logs modèle : confidence, entropie tokens, taux d'échec, longueur output, RTF (real-time throughput)
- Samples audio : éviter par défaut
  - si nécessaire, échantillonnage + consentement + rétention courte
- PII : voix = biométrie potentielle
  - transcripts peuvent contenir données personnelles
- Solutions : hashing IDs, rédaction, stockage chiffré, séparation des accès
- Alertes : chute confidence, hausse latence, drift de distribution (audio stats)

Robustesse : bruits, codecs, et domain shift

- Domain shift typique : WAV propre train vs AAC/Opus/telephone en prod
- Contre-mesures : augmentation réaliste (bruit, reverb, codecs), fine-tune ciblé
- VAD : améliore coût/latence mais peut couper mots
  - tester sur cas limites
- Overlap speakers : dégrade ASR
  - diarization/overlap handling si besoin
- Long-form : segmentation intelligente (phrases, pauses) pour stabilité
- A/B testing : comparer versions sur trafic réel avec garde-fous qualité/latence

Optimisation inference : quantization, distillation, et “smaller is better”

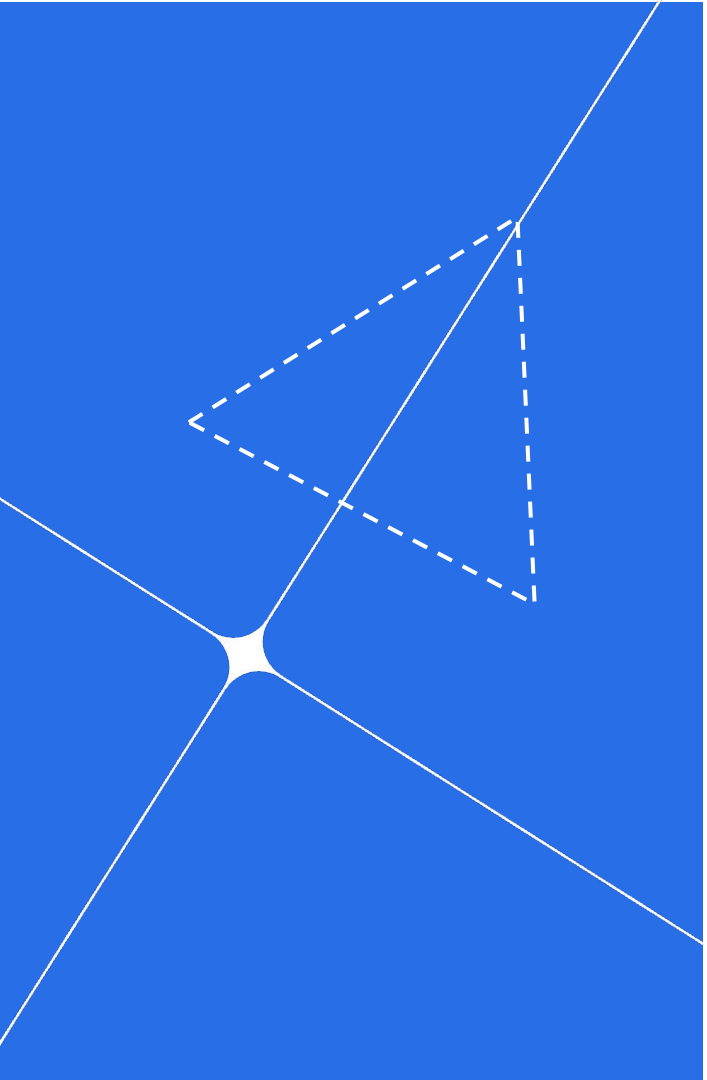
- Quantization (INT8/FP16) : réduire coût, parfois petite perte qualité
- Distillation : modèle petit imite gros (gain latence/cout), utile pour edge
- Pruning : réduction structurelle, mais gains dépendants du runtime
- Décodage : beam width impact énorme
  - tuner beam vs WER vs latence
- Spécifique audio : réduire T (downsampling, patching) diminue coût fortement
- Stratégie produit : commencer robuste (gros) → profiler → réduire progressivement

Sécurité & conformité : points non négociables

- Abuse TTS : voice cloning non autorisé, imitation
  - prévoir policy + contrôles
- Abuse ASR : injection audio (commande cachée), transcription de données sensibles
- Conformité : RGPD (finalité, minimisation, rétention), consentement si enregistrement
- Licences : datasets et voix (droits), usage commercial parfois interdit
- Red teaming : tester accents, bruits, mots interdits, tentatives de contournement
- Documentation : model card, data card, limites, cas d'échec, décisions prises

Tooling : stack typique

- Feature extraction : torchaudio, ffmpeg (codecs), normalisation contrôlée
- Modèles : Hugging Face transformers, accelerate, poids en cache partagé serveur
- Serving : FastAPI/gRPC, batching, GPU pinning, scripts Slurm pour jobs d’éval
- Vector search (retrieval) : Chroma-like
  - métadonnées + filtres
- Observabilité : logs structurés, traces latence, dashboards (Prometheus/Grafana)
- Reproductibilité : versions (pip/conda), seeds, artefacts, configs exportées



En route vers le TP !