La compression des modèles de Deep-Learning pour le TinyML

Quantization, Pruning, Distillation et Architecture légères pour le machine learning embarqué

LEFEVRE Antonin

Résumé

L'intelligence artificielle moderne s'est développée grâce à la puissance de calcul des serveurs et des infrastructures cloud, capables d'entraîner et d'exécuter des modèles de plus en plus complexes. Cette centralisation a permis des progrès considérables en vision, en langage et en robotique, mais elle révèle aujourd'hui ses limites : dépendance permanente à la connectivité, latence parfois incompatible avec le temps réel, consommation énergétique importante et difficulté à garantir la confidentialité des données.
Dans de nombreux scénarios, les systèmes intelligents doivent pourtant réagir immédiatement, sans recourir à une connexion réseau. C'est le cas des objets connectés, des capteurs industriels, des drones ou des robots autonomes. Pour répondre à ces contraintes, une approche nouvelle s'est imposée : le TinyML — pour Tiny Machine Learning — qui consiste à exécuter des modèles d'apprentissage automatique directement sur des dispositifs à ressources très limitées, comme les microcontrôleurs.

Le principe du TinyML est de rapprocher l'intelligence du lieu où les données sont produites. En embarquant le modèle directement sur l'appareil, le traitement devient local, rapide et économe en énergie, tout en préservant la confidentialité. Cette orientation, à la croisée de l'intelligence artificielle, de l'électronique embarquée et de l'optimisation logicielle, marque une évolution profonde dans la manière d'envisager l'IA : plus décentralisée, plus frugale et plus accessible.

Mais faire tourner un réseau de neurones sur un microcontrôleur de quelques centaines de kilo-octets de mémoire est un véritable défi. Il ne s'agit plus seulement d'entraîner des modèles performants, mais de les adapter pour qu'ils tiennent dans un espace minuscule sans sacrifier leur efficacité. C'est tout l'enjeu du TinyML : concevoir ou transformer les architectures de réseaux pour les rendre légères, efficaces et compatibles avec les contraintes matérielles de l'embarqué.

Key Points

- L'IA quitte le cloud pour s'exécuter directement sur les appareils.
- Le TinyML rend possible l'IA sur microcontrôleurs à faibles ressources.
- Optimiser les modèles devient essentiel pour le déploiement embarqué.
- Expérimentation sur robot ESP32.

Implémentation
Tous les codes sont disponibles sur le repo GitHub.

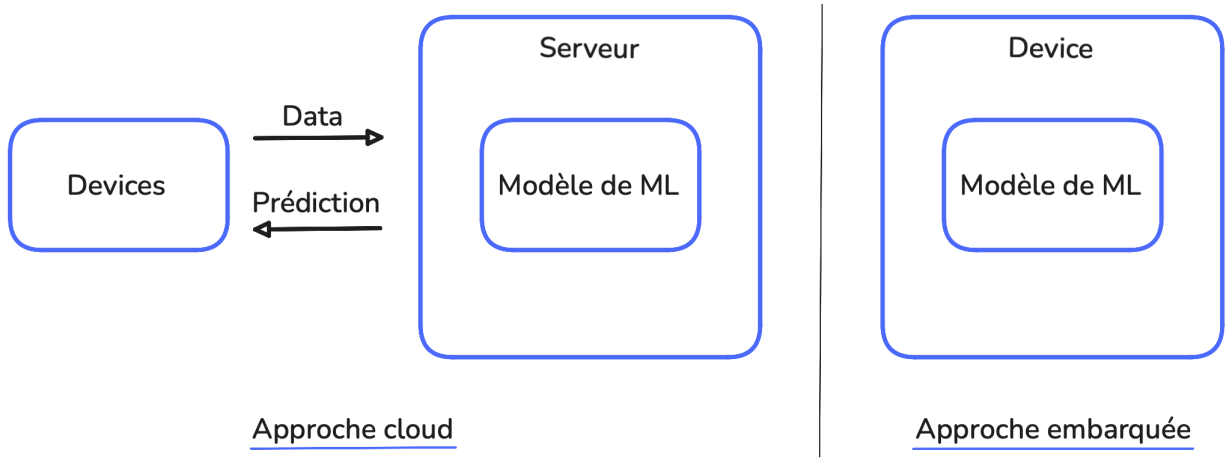
Keywords TinyML, deep learning, systèmes embarqués, Quantization, distillation, pruning

Table des matières

1. L'essor du TinyML : quand l'intelligence artificielle devient embarquée

L'intelligence artificielle contemporaine s'est développée dans l'ombre des centres de données : entraînements massifs, inférences servies par des GPU ou TPU, disponibilité « à la demande ». Cette centralisation a permis un bond qualitatif dans de nombreux domaines — vision, langage, recommandation — mais elle montre ses limites dès qu'il s'agit d'obtenir des interactions immédiates, une **autonomie énergétique** ou une **confidentialité locale**.

L'**Edge AI** a émergé comme réponse à ces contraintes : rapprocher le calcul des capteurs, traiter les données là où elles sont produites, et rendre l'intelligence **autonome** et **réactive**.



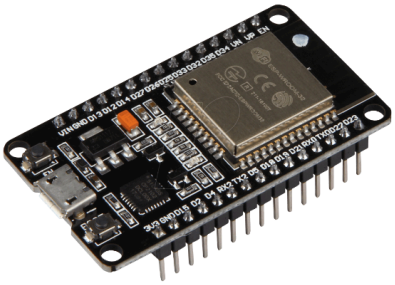
1.1. C'est quoi le TinyML ?

Le **TinyML** (Tiny Machine Learning) désigne l'ensemble des méthodes, matériels et outils permettant d'exécuter des modèles d'apprentissage **directement sur des dispositifs à très faibles ressources** — microcontrôleurs, capteurs autonomes ou objets connectés — tout en consommant **quelques milliwatts seulement**. Son objectif est de rendre possibles des usages always-on : détection de mots-clés, veille intelligente, reconnaissance de gestes ou surveillance d'environnement, **sans dépendre d'une connexion réseau**.

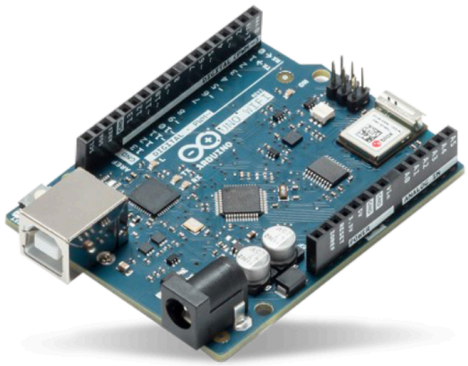
1.2. Microcontrôleurs : le cœur matériel du TinyML

Un **microcontrôleur** (ou MCU) est un mini-ordinateur intégré sur une seule puce, combinant processeur, mémoire, stockage et interfaces d'E/S permettant d'interagir avec le monde extérieur : capteurs, moteurs, boutons, LED ou modules de communication.

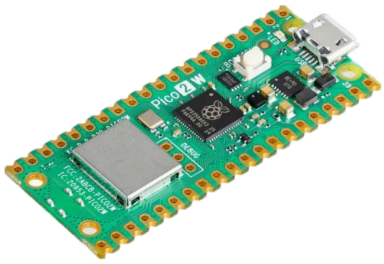
Contrairement aux microprocesseurs de PC ou de smartphones, les MCU sont conçus pour **exécuter une tâche spécifique**, avec un **coût réduit**, une **consommation minimale** et une **intégration complète** sur une carte compacte.



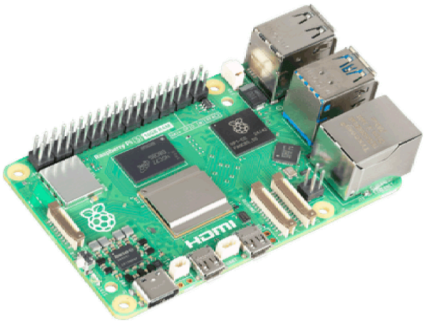
ESP32



Arduino UNO



Raspberry Pi Pico



Raspberry Pi 5

Ces cartes illustrent la diversité des plateformes : certaines, comme l'**ESP32**, disposent d'un double cœur et du Wi-Fi, tandis que d'autres (Arduino Uno, Raspberry Pi Pico) sont plus minimalistes mais idéales pour l'expérimentation. Le **Raspberry Pi**, quant à lui, occupe une position intermédiaire, à mi-chemin entre microcontrôleur et mini-ordinateur.

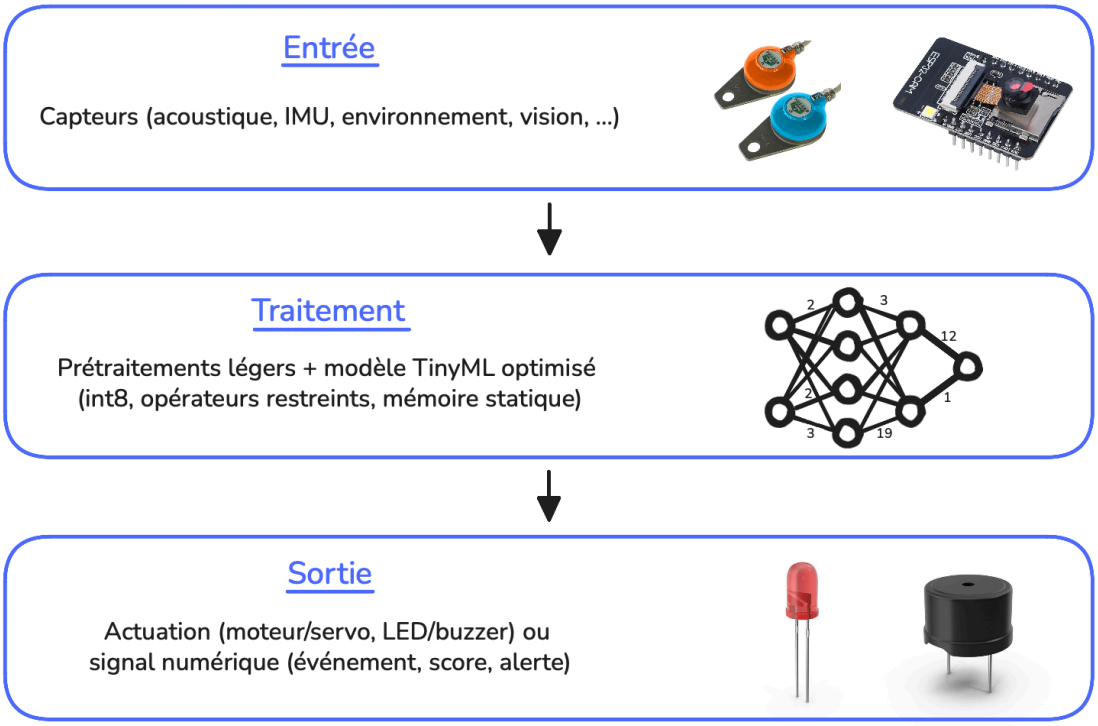
1.3. Pourquoi le TinyML émerge aujourd'hui ?

Deux tendances majeures convergent. D'une part, l'**explosion du nombre de capteurs et d'objets connectés**, dont plus de **99 % des données restent inexploitées**. D'autre part, la **maturité des techniques d'allègement des modèles** (quantization, pruning, distillation) et des **runtimes embarqués** comme TensorFlow Lite for Microcontrollers.

Cette rencontre rend aujourd'hui possible l'**interprétation locale** de signaux complexes (audio, vibration, vision) sur des MCU peu coûteux, sans dépendre d'un serveur distant.

1.4. Du capteur à l'action

Quel que soit le domaine d'application, le fonctionnement d'un système TinyML suit la même logique : une **boucle complète de perception, de décision et d'action**.



Cette structure universelle est au cœur de l'IA embarquée : elle transforme un objet initialement passif en un système **réactif, contextuel et autonome**, capable d'analyser son environnement et d'agir instantanément.

1.5. Applications emblématiques

Le TinyML est déjà déployé dans des environnements très variés. Dans l'**industrie** par exemple, il permet la **maintenance prédictive** : un capteur vibratoire fixé sur une turbine détecte une anomalie avant la panne, évitant des arrêts coûteux. Dans l'**agriculture**, il sert à identifier sur le terrain les maladies des cultures ou à mesurer en continu l'humidité et la luminosité sans infrastructure réseau. Partout où il y a un capteur, il peut y avoir une décision locale.

1.6. Rendre les modèles plus légers

Les réseaux de neurones profonds ont transformé la vision, le langage et la robotique, mais leur puissance repose souvent sur des milliards de paramètres — bien au-delà de ce qu'un microcontrôleur peut contenir. Réduire leur empreinte sans sacrifier leur intelligence est donc le cœur du **TinyML** : comment faire tenir un modèle en **quelques centaines de kilo-octets**, tout en conservant sa capacité de décision ?

Les sections suivantes présentent les principales **techniques de compression et d'optimisation** des réseaux, en abordant à la fois leur fondement théorique et leurs effets concrets sur la taille, la latence et la précision des modèles embarqués.

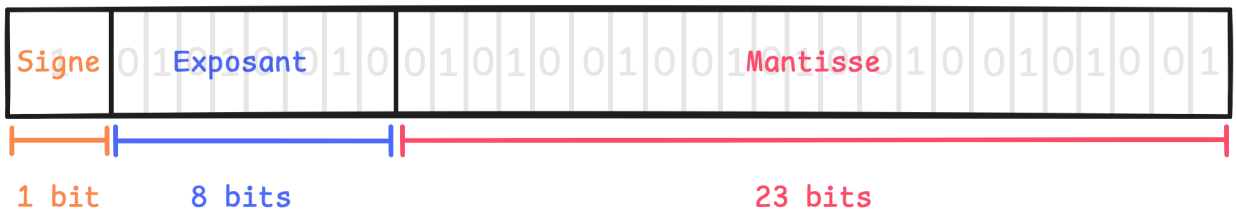
2. La quantization : réduire la précision sans perdre l'intelligence

La quantization, ou quantification, est une méthode fondamentale pour rendre les réseaux de neurones plus légers et plus rapides, sans sacrifier significativement la précision. Elle consiste à représenter les valeurs internes du modèle — poids et activations — sur un nombre réduit de bits, typiquement on passe de nombres en virgule flottante de haute précision, à des entiers de faible précision. Cette simple idée permet de réduire drastiquement la taille mémoire et le coût computationnel de l'inférence, tout en maintenant une performance très proche de celle du modèle original.

> *« Si quelqu'un vous demande l'heure, vous ne répondez pas "10 h 14 min 34 s 430 ms", mais plutôt "dix heures et quart". »*

2.1. De la virgule flottante à l'entier : représenter autrement

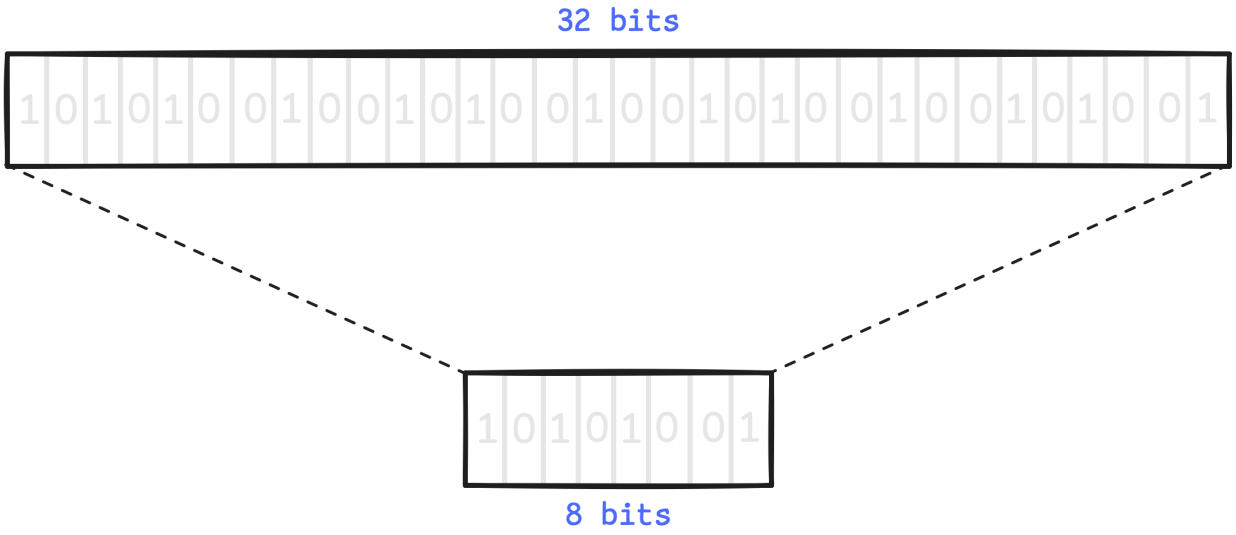
Dans un réseau de neurones classique, chaque poids et activation est souvent représenté en virgule flottante sur 32 bits (*float32*). Cette représentation suit la norme IEEE 754 : 1 bit pour le signe, 8 bits pour l'exposant et 23 bits pour la mantisse.



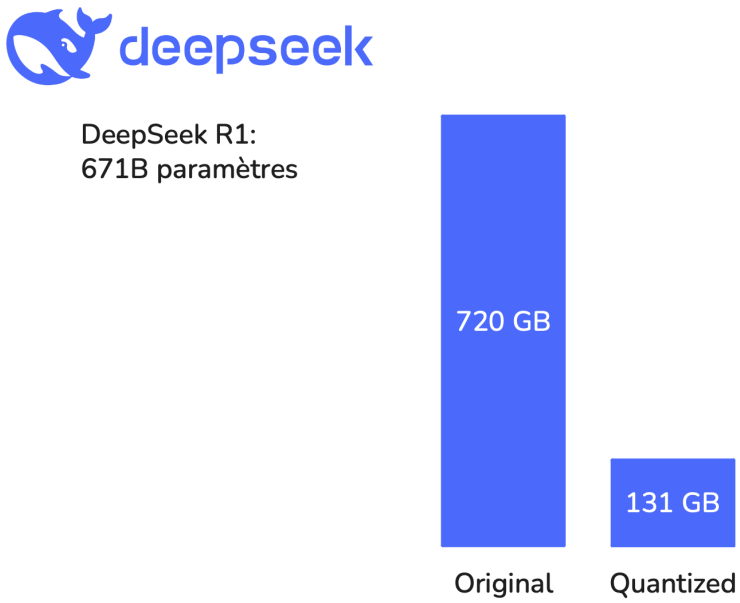
$$\text{nombre} = (-1)^{\text{Signe}} \times (1.\text{Mantisse}) \times 2^{\text{Exposant}-127}$$

Elle permet de représenter une gamme extrêmement large de valeurs, de l'ordre de 10^{-38} à 10^{38}, avec une grande précision. Mais cette précision a un coût. Chaque valeur occupe 4 octets en mémoire. Un modèle comme **ResNet-50**, avec ses **25 millions de paramètres**, requiert environ **100 Mo** rien que pour les poids, sans compter les activations intermédiaires. Pour un modèle géant comme **Llama 3 70B**, cette taille dépasse les **centaines de gigaoctets**.

La quantization propose de remplacer ces valeurs flottantes par des **entiers**, par exemple codés sur 8 bits (*int8*).



Un entier signé sur 8 bits peut prendre **256 valeurs distinctes**, comprises entre -128 et 127. La représentation devient ainsi discrète : au lieu d'un continuum de valeurs, on ne conserve que 256 crans possibles. Pour un grand modèle comme **DeepSeek R1**, la quantization permet de passer de **720 Go à environ 130 Go**, rendant possible son exécution sur **deux GPU H100** seulement.



Au-delà du gain mémoire, l'arithmétique entière est beaucoup plus efficace : les processeurs exécutent les additions et multiplications entières en un seul cycle d'horloge, contre plusieurs cycles pour les opérations flottantes qui nécessitent normalisation, alignement d'exposants et arrondi.

2.2. *Principes mathématiques de la quantization*

Mathématiquement, la quantization consiste à projeter un ensemble de valeurs réelles continues sur un ensemble discret d'entiers. Soit x une valeur réelle. Sa version quantifiée x_q s'obtient par :

$$x_q = \text{round}\left(\frac{x}{S} + Z\right) \tag{1}$$

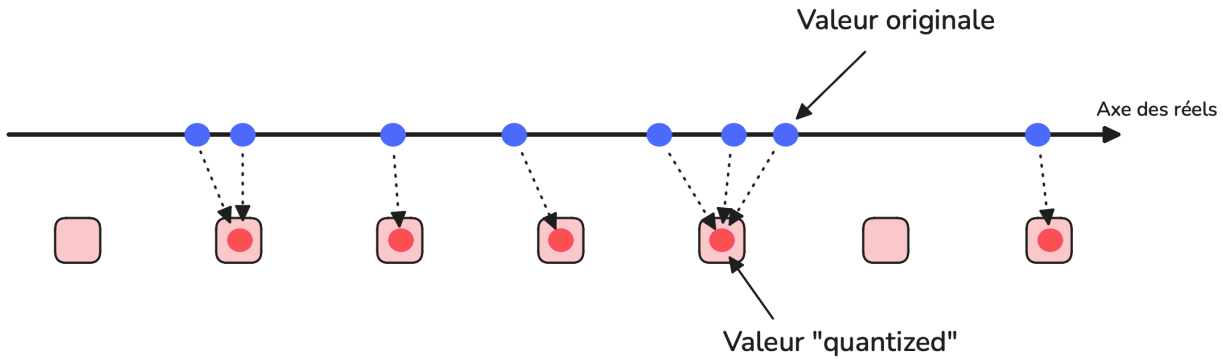
où :

- $S > 0$ est un **facteur d'échelle** (**scale**), qui définit l'espacement entre deux crans dans l'espace discret,
- Z est le **point zéro** (**zero-point**), c'est-à-dire la valeur entière correspondant à 0 dans l'espace quantifié.

Pour revenir à une valeur approchée dans l'espace réel, on applique la **déquantification** :

$$\hat{x} = S \times (x_q - Z) \tag{2}$$

La valeur $\hat{x}$ est une approximation de x. Le facteur d'échelle S contrôle directement la précision. Si S est petit, les crans sont serrés et la précision est fine, mais la plage couverte est plus restreinte (les valeurs extrêmes sont alors *clippées*), et au contraire si S est grand, la plage couverte est large, mais la quantization devient plus grossière.



L'objectif est donc de trouver un **compromis entre précision et couverture** en ajustant S et Z. Ces deux paramètres sont choisis de manière à ce que les valeurs réelles du modèle — poids ou activations — soient représentées le plus fidèlement possible dans l'espace discret des entiers.

Autrement dit, la quantization cherche à **établir une correspondance linéaire** entre deux espaces :

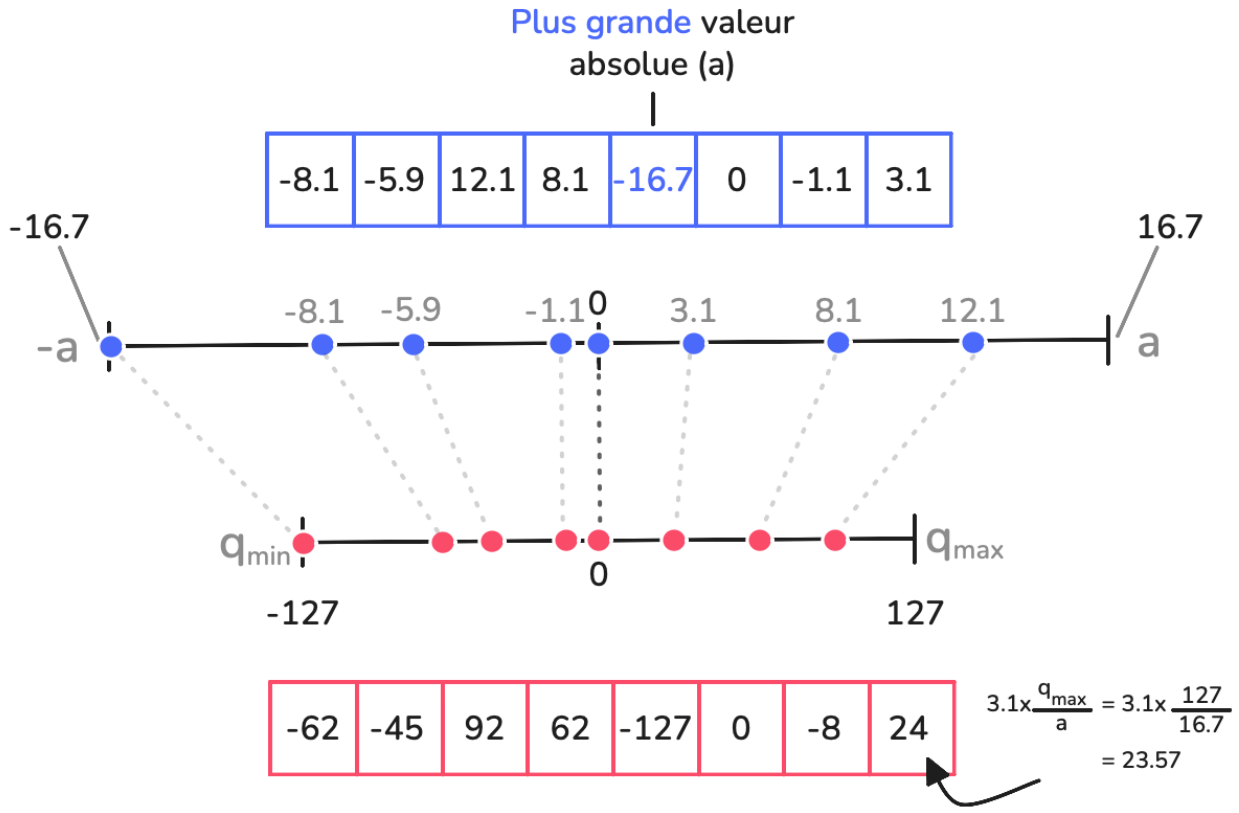
- l'espace continu des valeurs flottantes réelles, noté $[a, b]$
- et l'espace discret des entiers représentables, noté $[q_{\min}, q_{\max}]$

Selon la nature des données, cette correspondance peut être définie de deux façons principales : soit la plage réelle est **centrée autour de zéro**, soit elle est **décalée vers des valeurs positives**. C'est cette distinction fondamentale qui conduit aux deux schémas classiques de quantization : la **quantization symétrique** et la **quantization affine (asymétrique)**.

Dans la **quantization symétrique**, la plage des valeurs réelles est centrée sur zéro : $[-a, a]$. On fixe alors $Z = 0$, ce qui simplifie les calculs et accélère l'inférence, car aucune translation n'est nécessaire entre les domaines réel et entier. Si la plage entière correspondante est $[q_{\min}, q_{\max}]$, on définit :

$$S = \frac{a}{q_{\max}} \tag{3}$$

Ainsi, chaque unité entière représente un incrément de S dans l'espace réel. Dans le cas du type *int8*, on utilise généralement une plage **symétrique** $[-127, 127]$ plutôt que $[-128, 127]$, afin de garantir que le zéro soit représentable et centré. Ce schéma est particulièrement adapté aux **poids** des réseaux de neurones, qui sont souvent distribués symétriquement autour de zéro.



Dans l'exemple ci-dessus, la valeur maximale en absolu est $a = 16.7$, la valeur maximale pour passer en *int8* est 127, donc :

$$S = \frac{16.7}{127} \approx 0.1315 \tag{4}$$

Ainsi, pour $x = 3.1$:

$$x_q = \text{round}\left(\frac{3.1}{0.1315}\right) = 24 \tag{5}$$

Et la déquantification donne :

$$\hat{x} = 0.1315 \times 24 = 3.156 \tag{6}$$

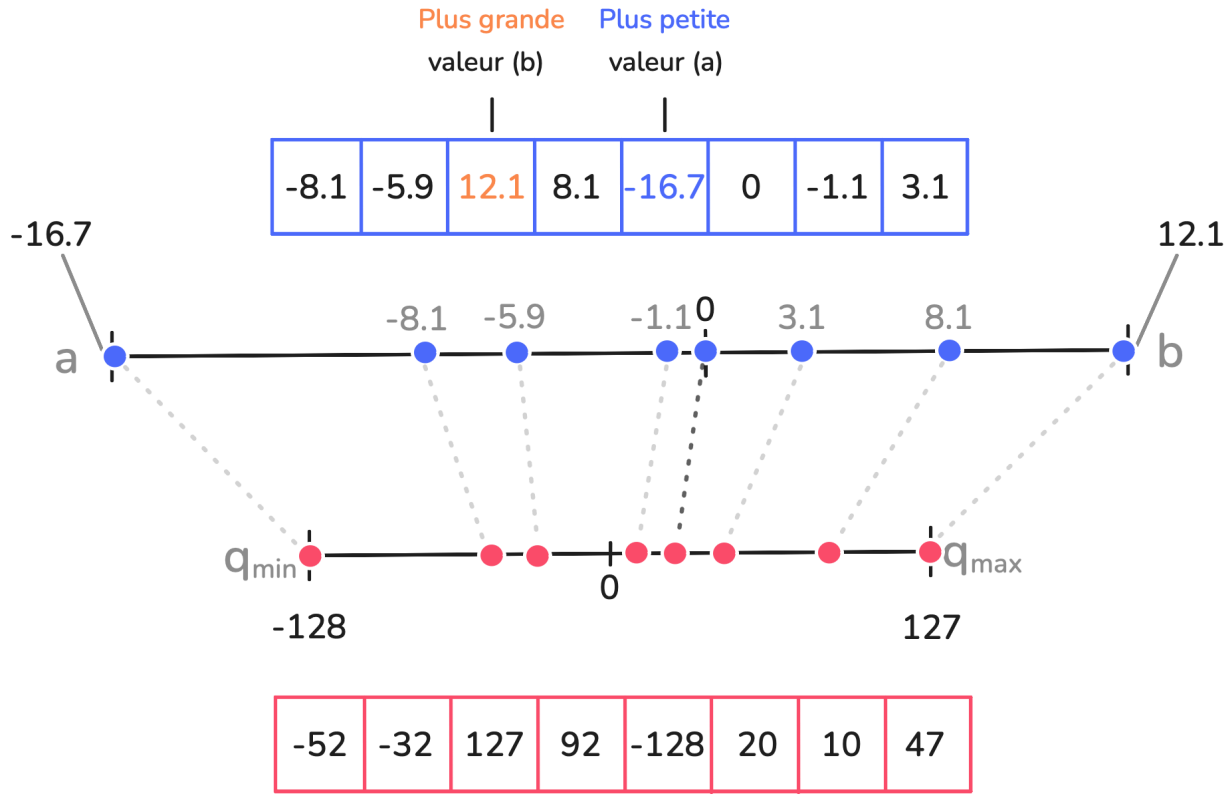
Le résultat est une approximation très proche de la valeur initiale.

Dans la **quantization affine**, la plage des valeurs réelles n'est pas centrée : $[a, b]$ avec $a \geq 0$ dans le cas d'activations issues d'une ReLU. On introduit alors un **point zéro** Z pour que la valeur réelle zéro soit représentée exactement dans l'espace des entiers. Ce décalage améliore la précision sur les distributions positives, typiques des activations. Si la plage entière est $[q_{\min}, q_{\max}]$, on calcule :

$$S = \frac{b-a}{q_{\max} - q_{\min}} \qquad \text{et} \qquad Z = \text{round}\left(q_{\min} - \frac{a}{S}\right) \tag{7}$$

où $q_{\min}, q_{\max}$ sont les bornes du type entier (*ex.* $-128, 127$ pour int8 signé).

Ce schéma permet de « translater » la plage réelle vers l'espace entier de manière plus flexible. Les valeurs extrêmes sont ensuite **clippées** dans l'intervalle $[q_{\min}, q_{\max}]$ pour éviter tout dépassement.



Dans l'exemple ci-dessus, on a $a = -16.7$ et $b = 12.1$ et on quantifie sur $[-128, 127]$, alors :

$$S = \frac{12.1 - (-16.7)}{127 - (-128)} = \frac{28.8}{255} \approx 0.1129 \quad \text{et} \quad Z = \text{round}\left(-128 - \frac{-16.7}{0.1129}\right) = 20 \tag{8}$$

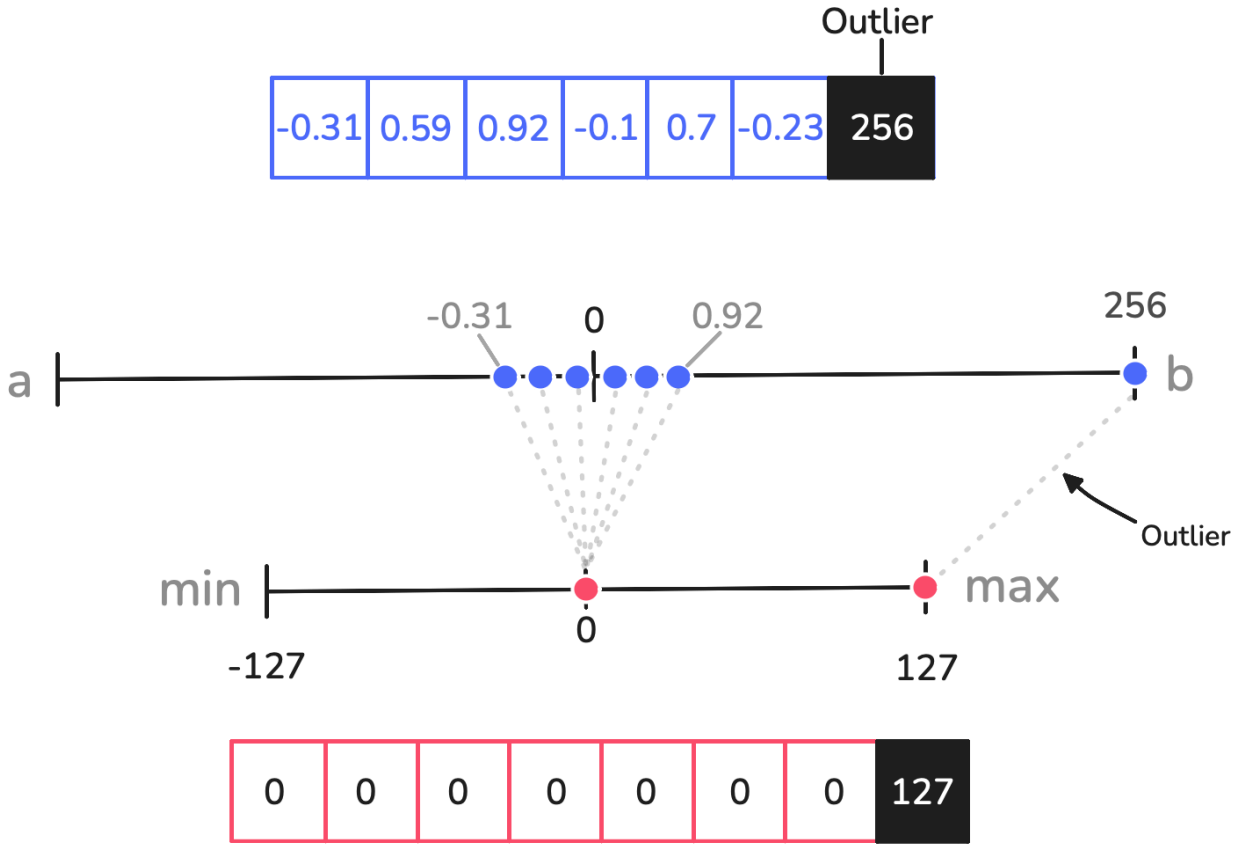
Ainsi, pour $x = 3.1$:

$$x_q = \text{round}\left(\frac{3.1}{0.1129} + 20\right) = 47 \tag{9}$$

La déquantification redonne $\hat{x} = 0.1129 \times (47 - 20) \approx 3.05$.

2.3. Calibration : choisir la bonne plage

Le choix de la plage $[a, b]$ est crucial, car il détermine directement les paramètres S et Z. Si la plage est trop large, les crans sont espacés et la précision diminue. Si elle est trop étroite, certaines valeurs réelles dépassent les bornes et sont **clippées**.



L'exemple ci-dessus illustre parfaitement ce dilemme. Ici, la majorité des valeurs sont concentrées autour de zéro, mais un unique **outlier** (valeur extrême) tire la borne supérieure très haut. Lorsqu'on calcule le facteur d'échelle S sur la base de cette plage complète, les pas de quantization deviennent si grands que toutes les petites valeurs sont ramenées vers zéro après conversion.

Résultat : presque tout le signal est perdu, et seule la valeur aberrante subsiste, saturée à la limite maximale (*int8 = 127*).

La calibration vise justement à éviter ce phénomène en estimant des bornes $[a, b]$ plus représentatives du comportement réel du modèle. Plutôt que de se baser sur les extrêmes absolus, on cherche à déterminer une plage qui capture la majorité des valeurs utiles, quitte à ignorer quelques cas rares.

Il existe plusieurs façons d'effectuer cette estimation, selon le moment où elle intervient dans le cycle de vie du modèle et le compromis souhaité entre **facilité d'application** et **précision obtenue**. On distingue donc deux grandes familles : la **Post-Training Quantization** (PTQ), regroupant notamment les modes dynamique et statique, et la **Quantization-Aware Training** (QAT), où la quantization est intégrée dès l'entraînement.

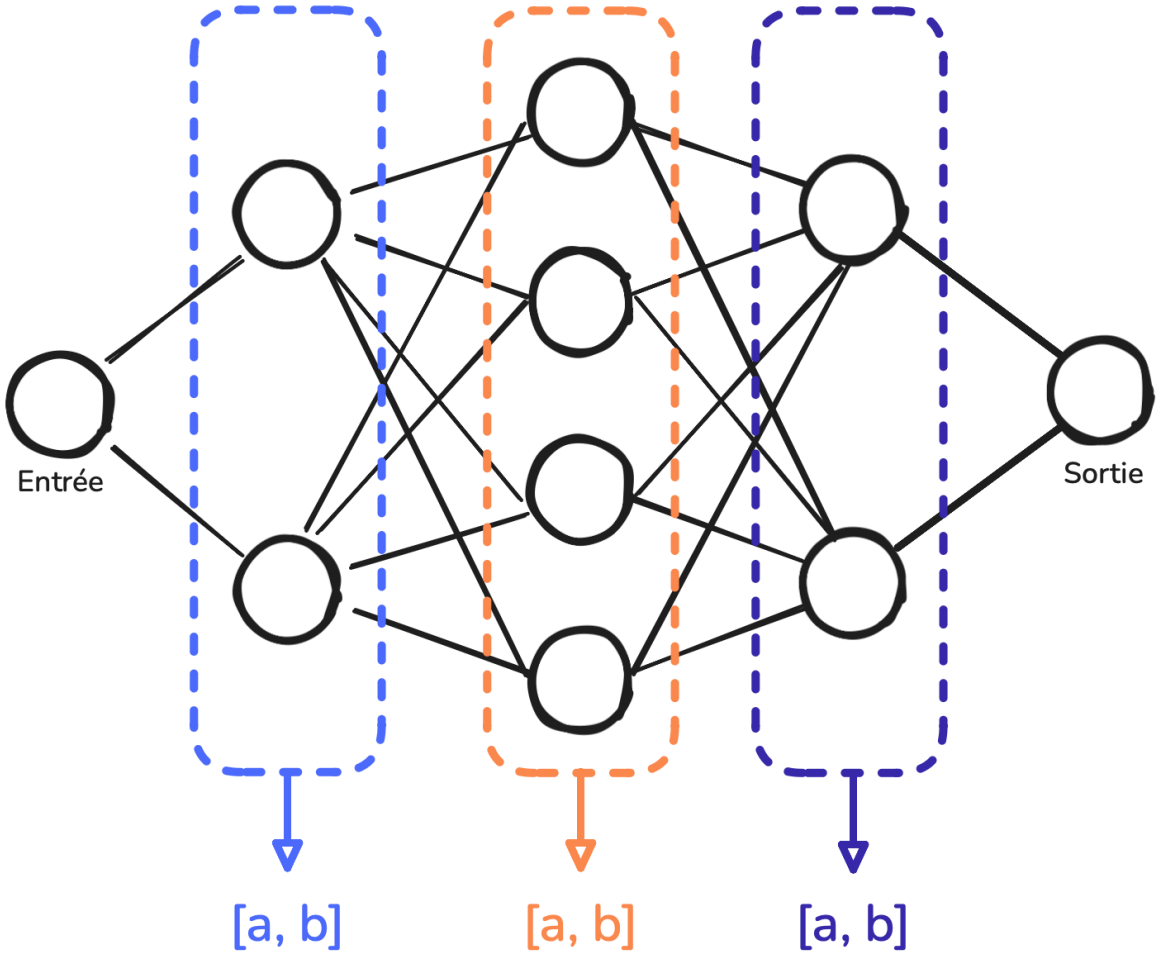
2.4. Quantization dynamique (PTQ)

La quantization dynamique est une méthode de **post-traitement** appliquée après l'entraînement du modèle. Elle se déroule en deux temps :

1. les **poids** sont d'abord quantifiés une fois pour toutes
2. les **activations** sont quantifiées **dynamiquement** à chaque passage de données pendant l'inférence.

Dans un premier temps, les poids du modèle — valeurs fixes apprises pendant l'entraînement — sont convertis en entiers (*int8*), selon un schéma **symétrique** ou **affine**. Cette étape réduit la taille mémoire du modèle tout en conservant des calculs précis.

Ensuite, lors de l'inférence, à chaque fois qu'un lot de données traverse une couche cachée, les activations produites sont analysées : on mesure leur **plage de valeurs réelles** $[a, b]$.

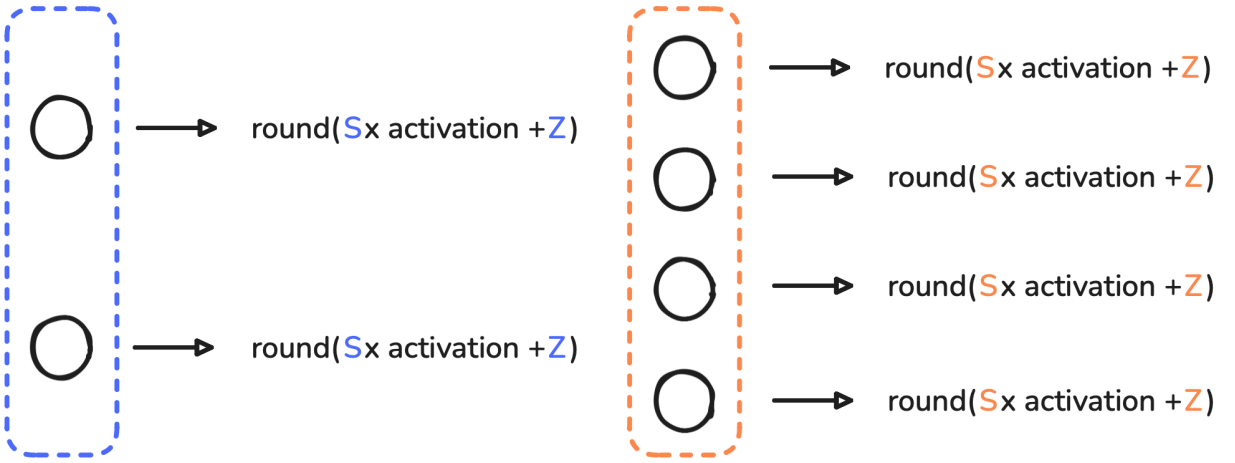


Ces bornes diffèrent d'une couche à l'autre, et même d'un lot à l'autre. Cette étape correspond à la collecte statistique illustrée ci-dessus : on observe la distribution des activations pour en extraire le **minimum** (a) et le **maximum** (b).

À partir de ces bornes, on calcule pour chaque couche les deux paramètres de quantization : le **facteur d'échelle** S et le **point zéro** Z, avec $q_{\min}, q_{\max}$ correspondant à la plage de l'espace entier, généralement $[-128, 127]$ pour l'*int8*, avec la méthode affine.

	[a, b]	[a, b]	[a, b]
$S = \frac{b - a}{q_{max} - q_{min}}$	S=	S=	S=
$Z = \text{round}(q_{min} - \frac{a}{S})$	Z=	Z=	Z=

Chaque couche possède donc ses propres valeurs S et Z, adaptées à la dynamique de ses activations. Ces paramètres sont ensuite utilisés pour **quantifier** les activations réelles en entiers, par exemple :



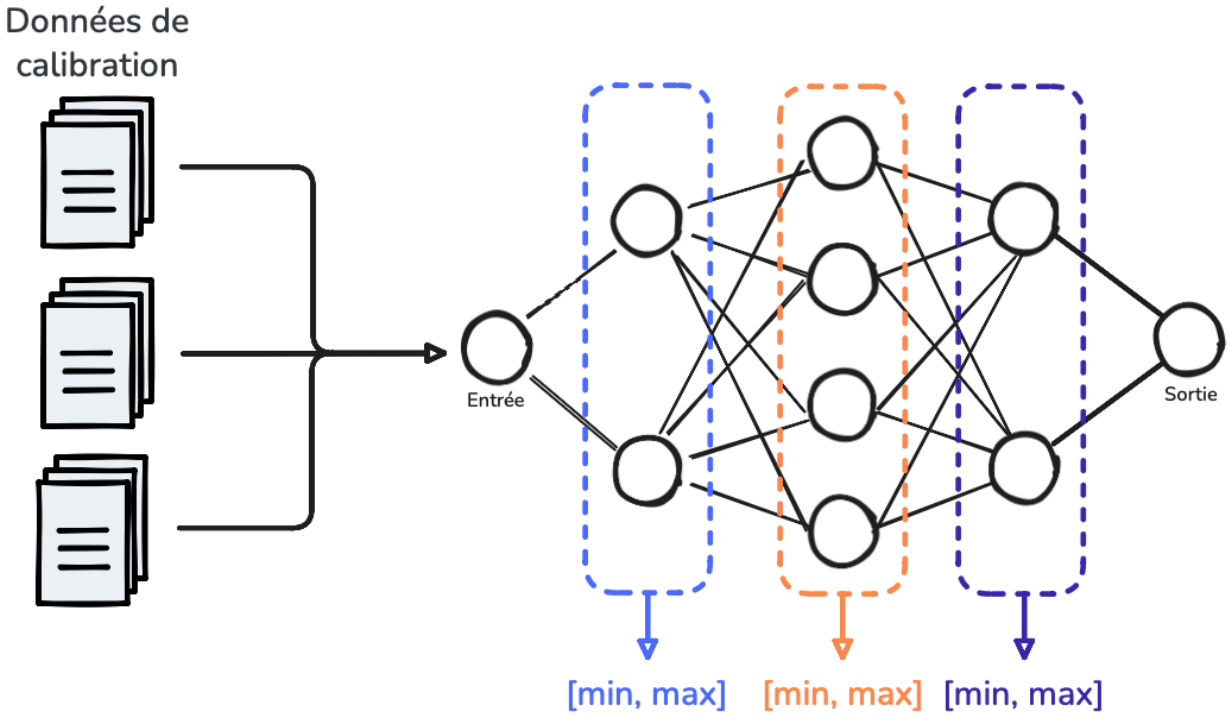
Ce processus se répète à chaque passage des données à travers le réseau, sans phase de calibration préalable. Le modèle reste donc inchangé : seules les activations sont temporairement projetées dans l'espace entier, puis reconverties. Cette approche est **simple**, **robuste** et particulièrement efficace pour les modèles séquentiels tels que **LSTM** ou **Transformers**, dont les distributions d'activations varient fortement d'un lot à l'autre.

En revanche, comme S et Z sont recalculés à chaque inférence, les conversions fréquentes entre flottants et entiers limitent les gains de performance par rapport à la quantization statique. Néanmoins, elle reste un excellent compromis entre **facilité de déploiement**, **réduction de taille** et **stabilité numérique**.

2.5. Quantization statique (PTQ)

La quantization statique se distingue de la version dynamique par le moment où les paramètres de quantization — le **facteur d'échelle** S et le **point zéro** Z — sont calculés. Ici, ces valeurs ne sont **pas déterminées pendant l'inférence**, mais **en amont**. En ce qui concerne les poids, ils sont, comme pour la Quantization dynamique, quantifiés une fois pour toutes.

Cette calibration consiste à faire passer un petit **jeu de données représentatif** dans le modèle, sans réentraînement. À chaque couche, des observers enregistrent les activations produites et collectent leurs plages de valeurs réelles $[a, b]$ — autrement dit, les minimums et maximums observés pendant cette phase.



Ces plages sont ensuite utilisées pour calculer les paramètres S et Z selon la méthode affine, comme pour la Quantization dynamique. Une fois ces paramètres fixés pour chaque couche, le modèle peut être exécuté intégralement en *int8* lors de l'inférence. Contrairement à la quantization dynamique, **les valeurs S et Z ne sont plus recalculées à chaque passage** : elles sont figées et utilisées directement pour transformer les activations et les poids dans l'espace discret.

Cette approche présente deux avantages majeurs :

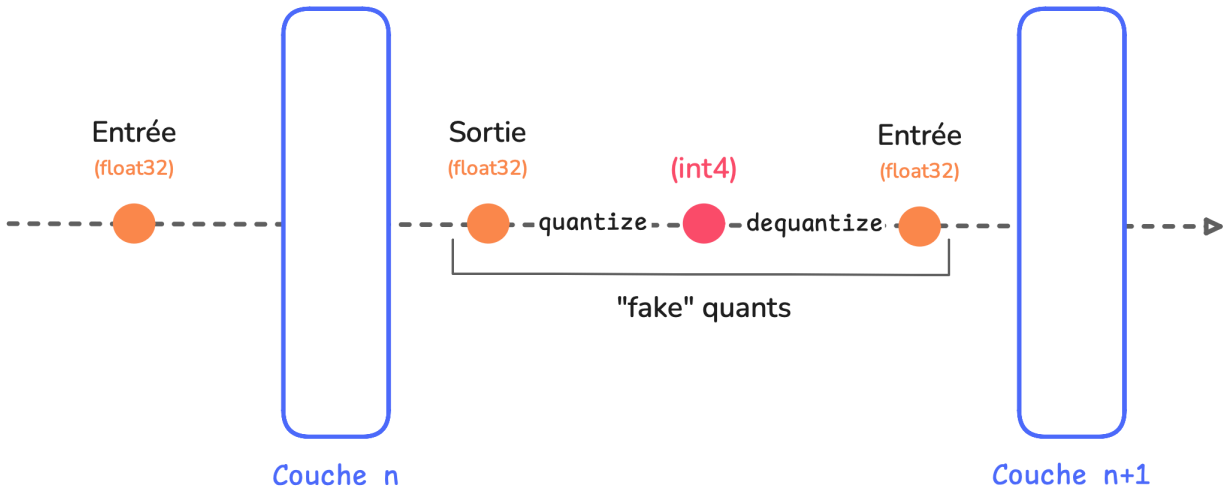
1. **Une exécution beaucoup plus rapide**, car aucune estimation en ligne des bornes $[a, b]$ n'est nécessaire.
2. **Une inférence homogène**, puisque les mêmes plages et paramètres sont appliqués pour toutes les entrées, garantissant un comportement reproductible.

En revanche, elle peut introduire une légère perte de précision, notamment si le jeu de calibration ne reflète pas parfaitement les distributions réelles rencontrées pendant l'inférence. Les valeurs extrêmes inattendues seront alors *clippées*, et certaines nuances de variation seront perdues.

2.6. Quantization-Aware Training (QAT)

Cette méthode consiste à entraîner le modèle en tenant compte de la quantization dès l'apprentissage. Au lieu d'entraîner un réseau en haute précision, puis de le compresser après coup (comme dans la PTQ), la QAT **intègre la quantization dans la boucle d'entraînement** : le réseau apprend donc à vivre avec la perte de précision.

Concrètement, on insère dans le modèle de petites opérations appelées **fausses quantizations** (fake quants). Elles imitent le passage en basse précision, sans réellement changer le type des nombres : les valeurs sont quantifiées (arrondies et limitées) comme si elles étaient en int4 ou int8, puis immédiatement "déquantifiées" pour continuer les calculs en float32.

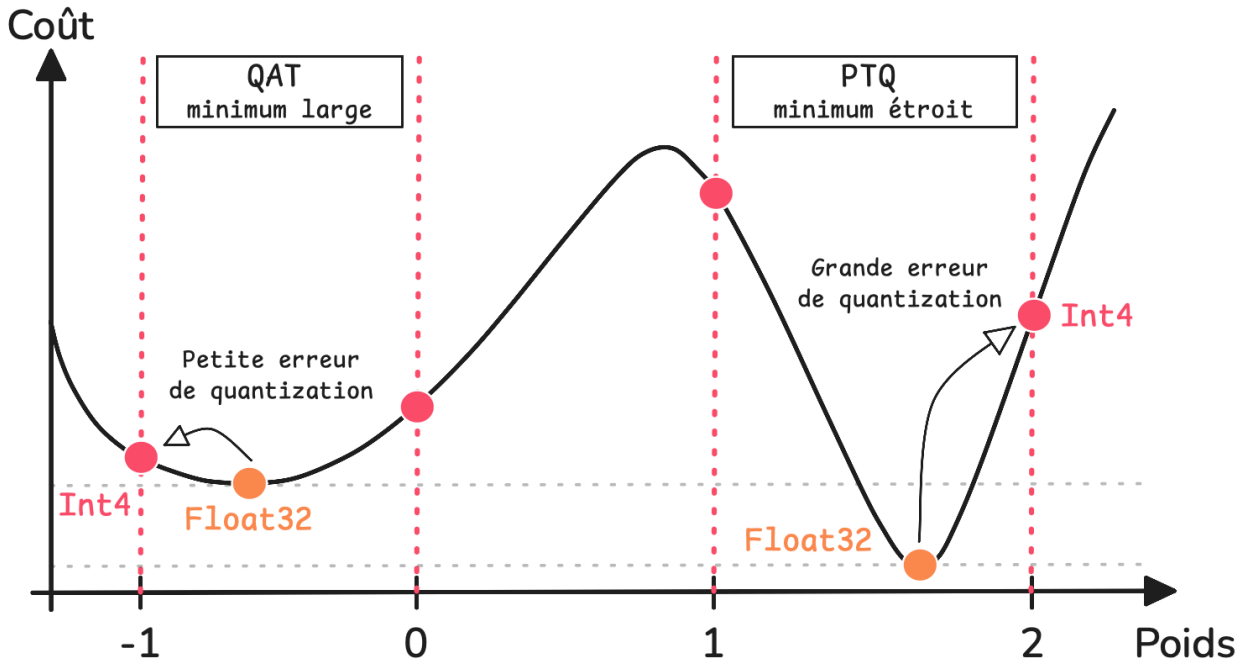


Ainsi, à chaque passage avant la fonction de coût, le modèle "voit" les effets de la quantization : il ressent les erreurs d'arrondi et de saturation qui existeront plus tard lors de l'inférence. Pendant la **rétropropagation**, ces erreurs influencent la descente de gradient — le réseau apprend donc à s'adapter à la perte de précision.

En pratique, cela signifie que la QAT ne cherche plus seulement à minimiser la perte en haute précision (FP32), mais à trouver des poids qui resteront bons même après quantization.

Dans un modèle classique, la fonction de coût peut comporter plusieurs “vallées” (ou minima) où la perte est faible. Certaines sont **étroites** : la perte augmente vite dès qu’on s’éloigne du minimum. D’autres sont **larges** : la perte reste stable même si les poids varient un peu.

Lorsqu’on quantifie les poids (par exemple en INT4), les valeurs sont arrondies : si le modèle était dans un minimum **étroit**, cet arrondi peut le faire remonter brutalement dans la perte. En revanche, un minimum **large** supporte mieux ces approximations.



La QAT apprend donc naturellement à se placer dans des minima plus larges, plus stables face à la quantization. Résultat : après compression, le modèle QAT a souvent une perte plus faible qu’un modèle PTQ, même s’il était un peu moins optimal en FP32.

Donc, la **PTQ** apprend en haute précision, puis subit la quantization — et perd un peu en performance. Et la **QAT**, elle, apprend avec la quantization — et trouve des poids qui fonctionnent bien directement en basse précision. C’est la méthode la plus coûteuse en calcul (puisqu’elle demande un réentraînement), mais c’est aussi la plus efficace pour les modèles très compacts (4 bits, 2 bits, voire 1 bit) où chaque arrondi compte.

2.7. Pour aller plus loin : vers des quantizations extrêmes

Les méthodes présentées jusqu'ici, qu'elles soient dynamiques, statiques ou intégrées à l'entraînement, utilisent généralement une précision de 8 bits (int8). Mais la recherche actuelle explore des réductions encore plus fortes, jusqu'à 6, 4, 2, voire 1 bit par poids.

Des approches récentes comme **GPTQ** (Gradient Post-Training Quantization) ou **AWQ** (Activation-aware Weight Quantization) repoussent les limites du post-training quantization. Elles cherchent à minimiser la perte de précision tout en abaissant la taille mémoire de plusieurs ordres de grandeur, permettant d'exécuter de très grands modèles sur du matériel limité (CPU, GPU grand public ou edge devices).

De leur côté, des formats comme **GGUF** ou **EXL2** se sont imposés pour le déploiement de modèles ultra-quantifiés (4 bits, parfois 2 bits), notamment dans les écosystèmes open source comme *llama.cpp* ou *Ollama*.

Ces travaux marquent une évolution majeure : la quantization n'est plus seulement une étape d'optimisation après entraînement, mais devient un principe de conception. C'est dans cette logique qu'émergent de nouvelles architectures pensées dès l'origine pour fonctionner à très faible précision.

3. PRUNING : SUPPRIMER L'INUTILE

Le pruning (élagage) des réseaux de neurones consiste à **retirer des paramètres inutiles** ou **redondants** d'un modèle afin de l'alléger pour le déploiement sur des dispositifs très contraints (mémoire, calcul, énergie). Dans le contexte du (TinyML) — microcontrôleurs, capteurs, systèmes embarqués — le pruning permet de réduire non seulement la taille du modèle, mais aussi potentiellement la latence et la consommation d'énergie, ce qui est essentiel pour qu'un modèle puisse tourner sur un MCU ou un dispositif à ressources réduites.

On distingue deux grandes familles de pruning : d'une part le **post-training pruning**, où le modèle est entraîné normalement puis l'élagage appliqué comme étape de compression après l'entraînement, d'autre part le **train-time pruning**, où l'élagage intervient pendant l'apprentissage ou via des contraintes de sparsité intégrées au processus d'entraînement.

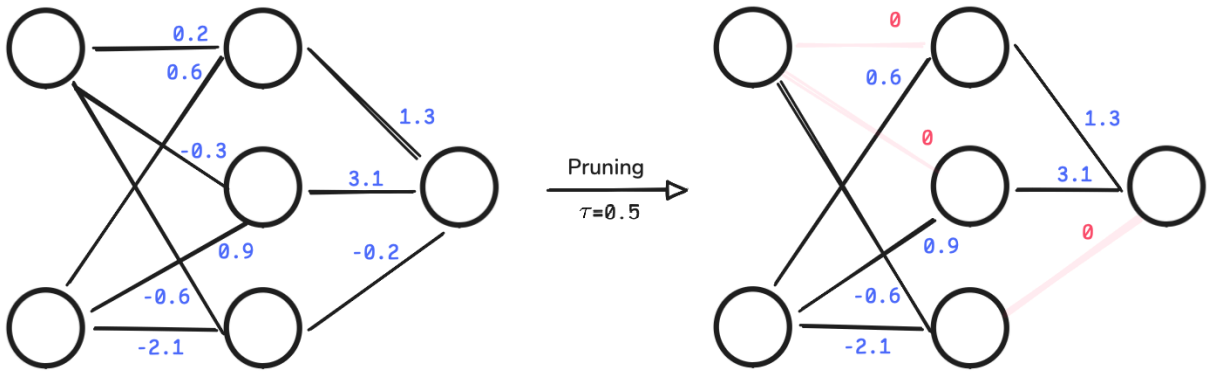
3.1. Pruning par magnitude des poids

Cette méthode [1] repose sur l'hypothèse simple : dans un réseau de neurones entraîné, les poids de faible valeur absolue (« petite magnitude ») contribuent peu à la sortie du modèle, et peuvent donc être supprimés (mis à zéro) sans trop dégrader la performance.

Concrètement, on applique un masque binaire tel que pour chaque poids w_i :

$$\widetilde{w_i} = \begin{cases} w_i \text{ si } |w_i| \geq \tau \\ 0 \text{ sinon} \end{cases} \tag{10}$$

Pour déterminer le seuil τ, on peut collecter toutes les valeurs absolues des poids (par couche ou globalement), puis calculer un seuil τ — par exemple le p-ème percentile de cette distribution — ou fixer un pourcentage p% de poids à supprimer. On obtient alors un réseau « sparse » (beaucoup de poids à zéro) tout en gardant la même topologie. Par exemple :



Ici, les poids dont la magnitude est inférieure à 0.5 sont supprimés.

Ensuite, une fois le masque appliqué (et les poids dont la magnitude est trop faible mis à zéro), on procède à un fine-tuning du modèle. Lors de ce fine-tuning, on **garde les poids mis à zéro à zéro** (dans l'idéal, ils ne sont pas réactivés) afin de conserver la structure « élaguée ». Le modèle apprend alors à ré-optimiser les poids restants pour récupérer autant que possible la précision initiale.

On peut répéter le cycle : **élagage** → **fine-tuning** → éventuellement **re-élagage** → nouveau fine-tuning, jusqu'à atteindre le taux de sparsité souhaité ou jusqu'à ce que la perte de performance devienne excessive.

3.2. pruning de filtres de convolution

Cette méthode [2] consiste à **supprimer des filtres entiers** dans les couches convolutionnelles, plutôt que des poids isolés. (ils sont retirés du modèle, pas simplement mis à zéro) L'idée est simple : chaque filtre agit comme un petit "détecteur" (de bord, texture, motif...). Certains de ces détecteurs produisent des cartes de caractéristiques (**feature maps**) très faibles ou redondantes. Ils peuvent donc être retirés sans nuire significativement aux performances du modèle.

Pour mesurer l'importance d'un filtre $\mathcal{F}_{i,j}$ dans une couche i, on calcule la somme de la valeur absolue de ses poids :

$$s_j = \sum_{l=1}^{n_i} \sum_{k \in \mathcal{F}_{ij}} |k| \tag{11}$$

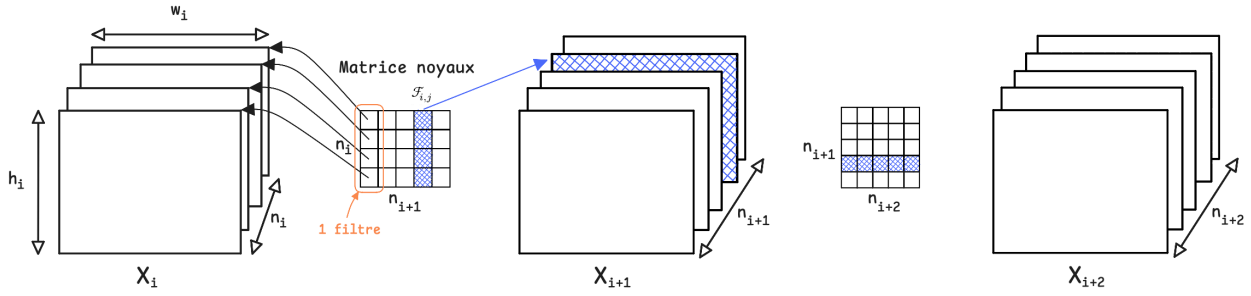
où n_i est le nombre de canaux d'entrée du filtre. Ce score s_j représente la magnitude totale du filtre : plus elle est faible, moins le filtre contribue aux activations du réseau.

Dans la pratique, on trie les filtres d'une couche selon leur score s_j. On peut ensuite :

- soit fixer un **pourcentage** de filtres à supprimer (par exemple 30 %),
- soit définir un **seuil de magnitude** : tous les filtres dont s_j est inférieur à ce seuil sont éliminés.

Ce choix peut se faire **par couche** ou de manière **globale** sur tout le réseau.

L'image ci-dessous montre une couche de convolution (matrice noyaux) avec $n_i = 4$ canaux d'entrée et $n_{i+1} = 5$ filtres (les colonnes de la matrice de noyaux). Chaque filtre est appliqué à tous les canaux d'entrée : chaque cellule de la colonne correspond à un petit noyau (par exemple 3×3). On calcule pour chaque filtre la somme des valeurs absolues de tous ses poids. Ici, le filtre d'indice 4 a le plus petit score s_j : il est supprimé, sa **feature map** devient nulle, et la **ligne correspondante** dans la matrice de la couche suivante (le canal associé) est également retirée (les zones hachurées en bleu).



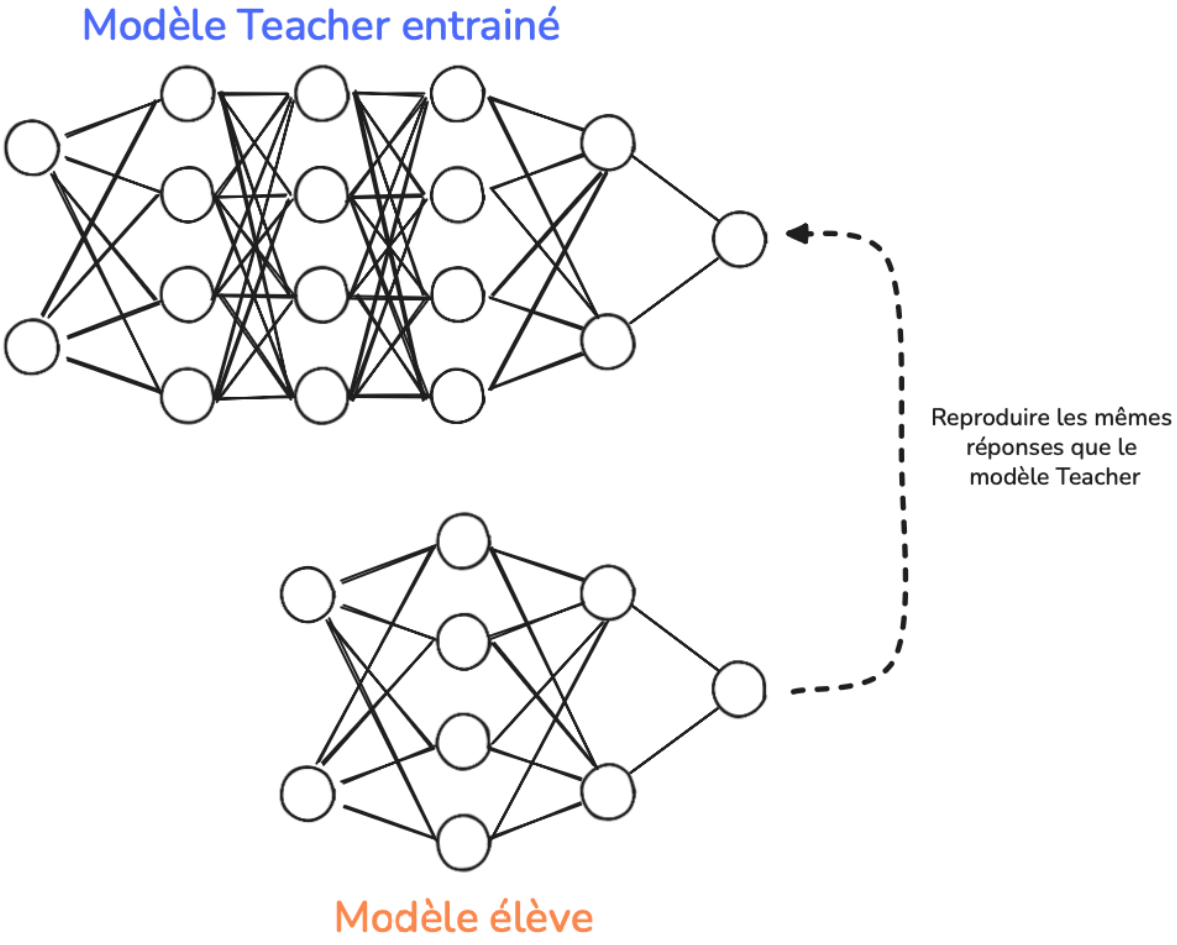
Après le pruning, le réseau est **fine-tuné** — c'est-à-dire réentraîné sur les données d'origine — afin de récupérer une grande partie de la précision perdue, tout en gardant les filtres désactivés à 0. Dans l'article original [2], cette étape permet de conserver une précision quasiment identique au modèle initial, tout en réduisant de **30 à 40 %** le nombre d'opérations de convolution (FLOPs).

En revanche, le choix du pourcentage ou du seuil de suppression influence fortement les performances, et un **fine-tuning soigné** reste indispensable pour stabiliser la précision du modèle.

4. Distillation : apprendre l'essentiel d'un grand modèle

La **distillation de connaissances** (**Knowledge Distillation**, KD) consiste à entraîner un **petit modèle** (**student**) à **imiter le comportement** d'un **grand modèle** (**teacher**). L'idée est simple : le teacher apprend la tâche complète, puis le student apprend à reproduire ses réponses, tout en étant plus léger, plus rapide et moins gourmand en ressources. C'est ce qu'on appelle la **"reponse-based knowledge distillation"** car ce sont les sorties du teacher qui sont copiées.

Comme dans une relation professeur–élève, le modèle teacher transmet son savoir au modèle student. Même si ce dernier reste plus simple, il peut, avec un bon entraînement, atteindre presque le même niveau de performance.



Cette approche est utilisée pour **réduire la taille et le coût d'exécution** des modèles, sans perdre trop de précision. Un exemple célèbre est **DistilBERT**, une version allégée de BERT, environ 40 % plus petite, tout en conservant près de 95 % de ses performances et étant 2 fois plus rapide.

Dans le contexte du **TinyML**, la distillation est essentielle : elle permet de rendre des réseaux complexes compatibles avec des **microcontrôleurs**, en transférant leur intelligence dans des architectures compactes et efficaces. La distillation s'intègre également avec la **Quantization** et le **Pruning** : ensemble, ils forment la boîte à outils complète du déploiement de l'IA sur l'edge.

4.1. Response-based KD

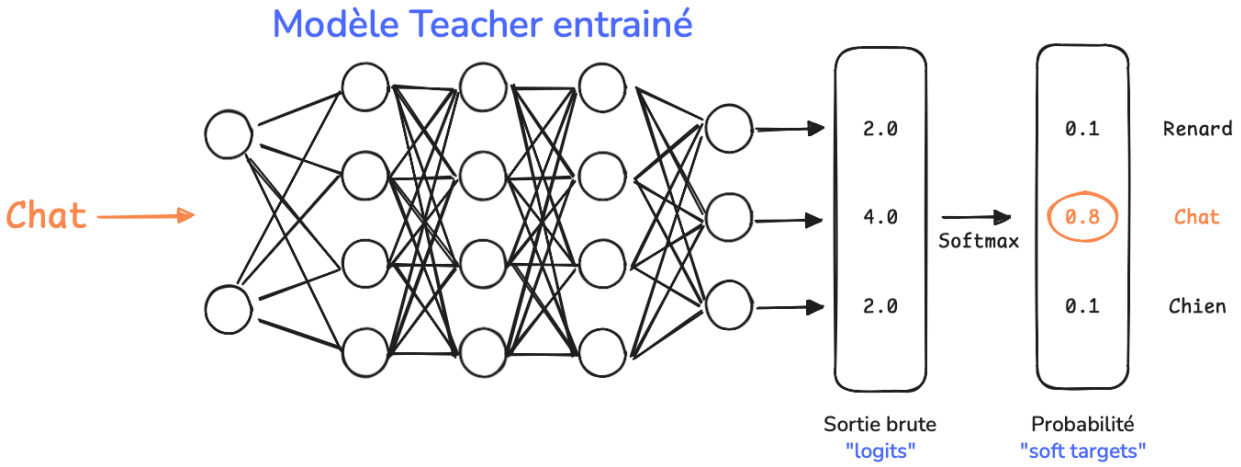
La distillation basée sur les réponses (**Response-Based KD**) est la forme la plus simple et la plus utilisée de la distillation. Elle a été introduite par **Geoffrey Hinton et al. (2015)** dans leur article *"Distilling the Knowledge in a Neural Network"* [3]. L'idée est de forcer un petit modèle (**student**) à **imiter les probabilités de sortie** d'un grand modèle (**teacher**), typiquement dans des tâches de classification, plutôt que d'apprendre uniquement à partir des étiquettes du dataset.

On dispose donc d'un modèle **teacher**, déjà entraîné et précis. On entraîne ensuite un modèle **student**, plus petit, pour qu'il apprenne à reproduire les sorties du teacher.

Le student reçoit deux types d'informations :

- Les **étiquettes réelles** du dataset (la classe réelle de chaque donnée).
- Les **sorties du teacher**, appelées **soft targets**.

Ces soft targets sont calculées dans la dernière couche du réseau, souvent par une fonction **softmax**, qui transforme les **logits** (sorties non normalisées) en probabilités.

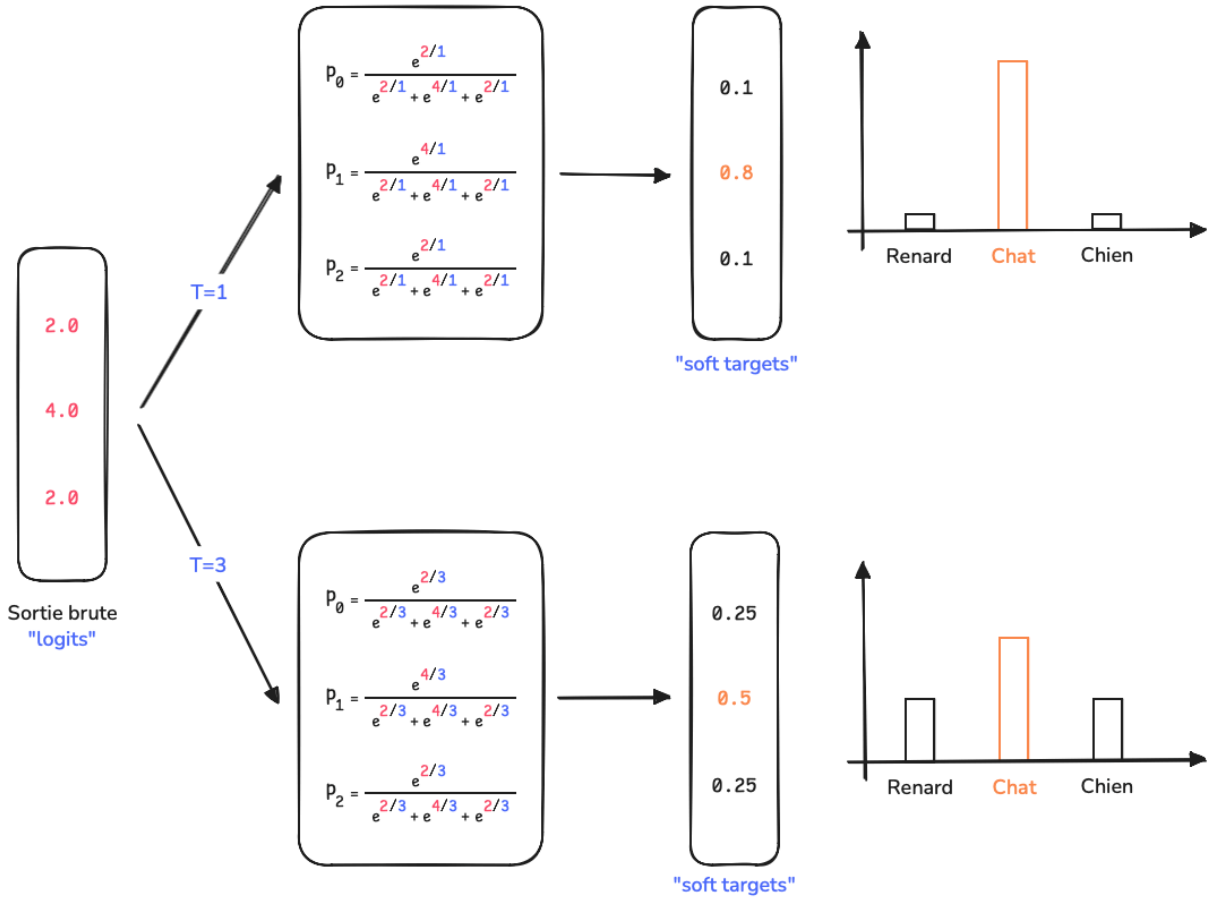


Mais on va ici introduire une **température** T pour "adoucir" cette distribution. On parle alors de **softmax à température**. La probabilité de la i-ème classe avec une température T, notée $p_i^{(T)}$ est définie par :

$$p_i^{(T)} = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}} \tag{12}$$

Ici, z_i est la i-ème sortie brute du modèle. Concernant la température, si $T = 1$, on retrouve la softmax classique : les plus grandes classes dominent, mais si $T > 1$, la distribution devient plus **plate**, plus **riche en informations**.

Par exemple, dans l'exemple ci-dessous, on voit que la température rend les petites classes plus visibles.



Maintenant, pour que le student imite le teacher, il faut que sa distribution de probabilités en sortie soit la plus proche possible du teacher. Il faut ainsi une métrique qui calcule la différence entre leurs distributions de probabilités, et la plus utilisée est la **divergence de Kullback–Leibler (KL divergence)**.

$$\mathrm{KL}(P|Q) = \sum_i P_i \log \frac{P_i}{Q_i} \tag{13}$$

Ici :

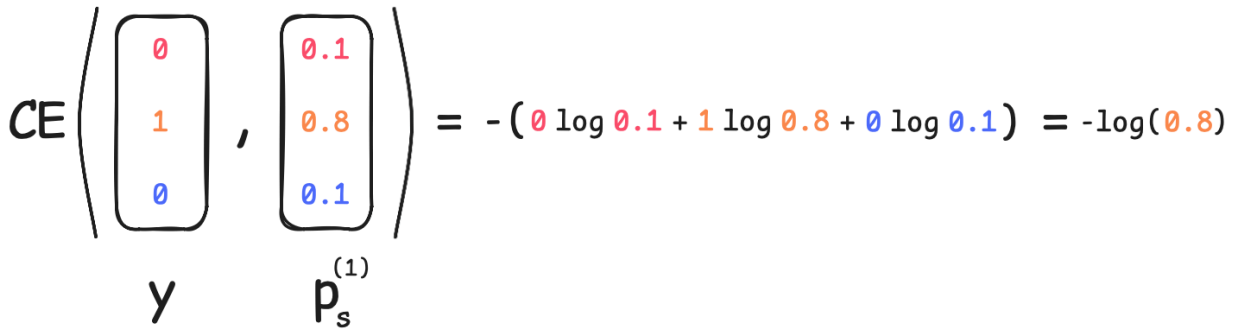
- $P = p_t^{(T)}$ sont toutes les probabilités de sortie du teacher (avec température T),
- $Q = p_s^{(T)}$ sont toutes les probabilités de sortie du student (avec la même température T).

La KL divergence mesure à quel point une distribution Q (celle du student) s'écarte d'une autre P (celle du teacher). Si Q est identique à P, la divergence vaut 0. C'est donc une mesure naturelle pour "faire ressembler" le student au teacher, qui apparaitra dans la fonction coût du processus.

Cependant, le student doit aussi apprendre à prédire la bonne classe réelle. On ajoute donc une perte classique de **Cross-Entropy (CE)** entre les vrais labels y et les sorties du student p_s avec une température $T = 1$:

$$\mathrm{CE}\left(y, p_s^{(1)}\right) = -\sum_i y_i \log p_{s,i}^{(1)} \tag{14}$$

Ici, y_i vaut 1 uniquement pour la classe correcte, et 0 pour les autres : en pratique, un seul terme de la somme contribue à la perte. Par exemple :



Cette partie agit comme un rappel de la “vérité terrain” : elle empêche le student de copier aveuglément les erreurs du teacher. Donc même si le student prédit bien la même distribution que le teacher, si la prédiction est mauvaise, alors le coût sera élevé.

Au final, la perte totale combine les deux composantes, la **Cross-Entropy** et la **KL divergence** :

$$\mathcal{L} = (1-\alpha)\mathrm{CE}\left(y, p_s^{(1)}\right) + \alpha T^2 \ \mathrm{KL}\left(p_t^{(T)} | p_s^{(T)}\right) \tag{15}$$

où :

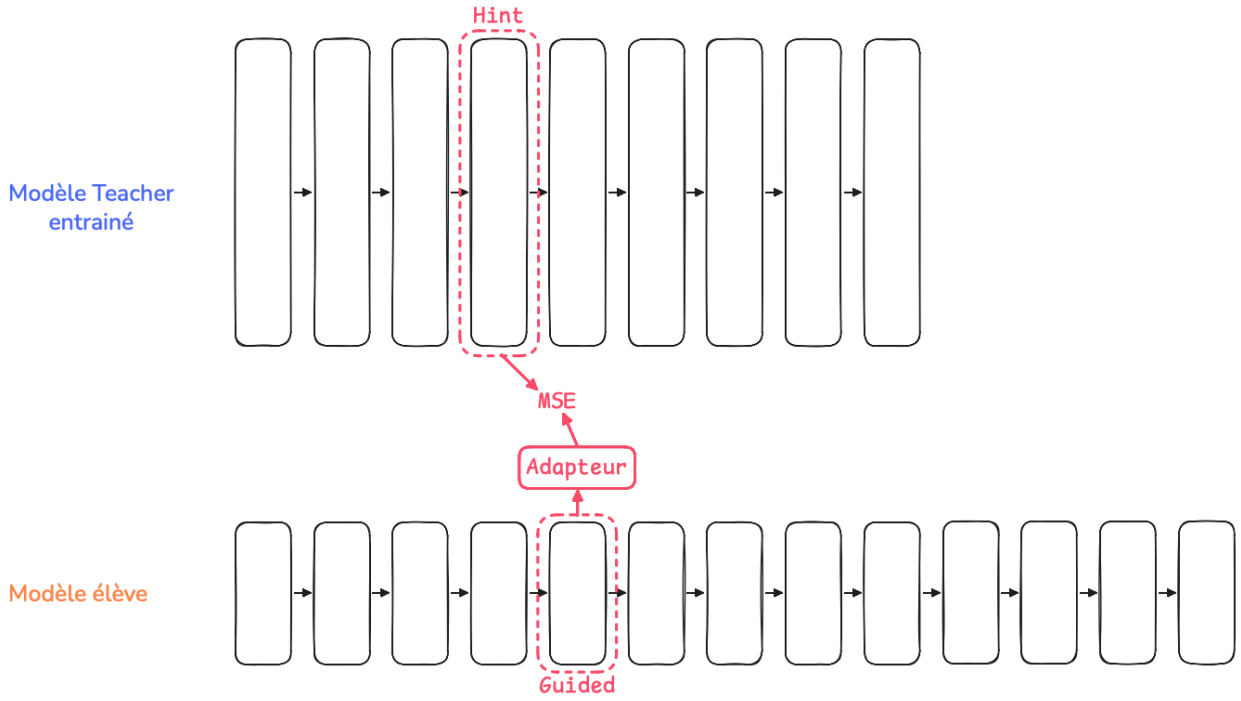
- $\alpha \in [0, 1]$ gère l’équilibre entre labels réels et imitation du teacher, typiquement 0.5
- T est la température du softmax, généralement entre 2 et 5
- le facteur T^2 est introduit pour **compenser la réduction des gradients** à haute température

Ainsi, la **Cross-Entropy** pousse le student à prédire juste, tandis que la **KL divergence** l’incite à penser comme le teacher. En combinant les deux, on obtient un modèle plus petit, mais aussi plus précis, robuste et stable.

4.2. Feature-based KD - FitNets

Après avoir vu comment un **student** peut imiter les sorties d'un **teacher** (**response-based KD**), on peut aller plus loin : lui apprendre à imiter les activations internes du teacher, c'est-à-dire les « choses que le teacher voit » avant de produire sa sortie. C'est le principe de **FitNets** [4], une méthode de distillation basée sur les caractéristiques (**Feature-based KD**) développée pour des réseaux convolutionnels (CNN) appliqués à la vision par ordinateur.

L'idée est de guider le **student** non seulement par les étiquettes réelles et les probabilités finales, mais aussi par les cartes de caractéristiques (feature maps) intermédiaires du **teacher**. Le **student**, dans le cas de FitNets, est un modèle plus **mince** que le teacher, avec possiblement moins de paramètres, mais plus **profond** que lui.



imiter les activations de la couche "hint" du teacher à l'aide d'une **perte de régression** (souvent MSE). Mais comme les dimensions des activations diffèrent (le student est plus mince), on ajoute un **adaptateur** (ou **regresseur**) pour transformer la sortie du student avant comparaison.

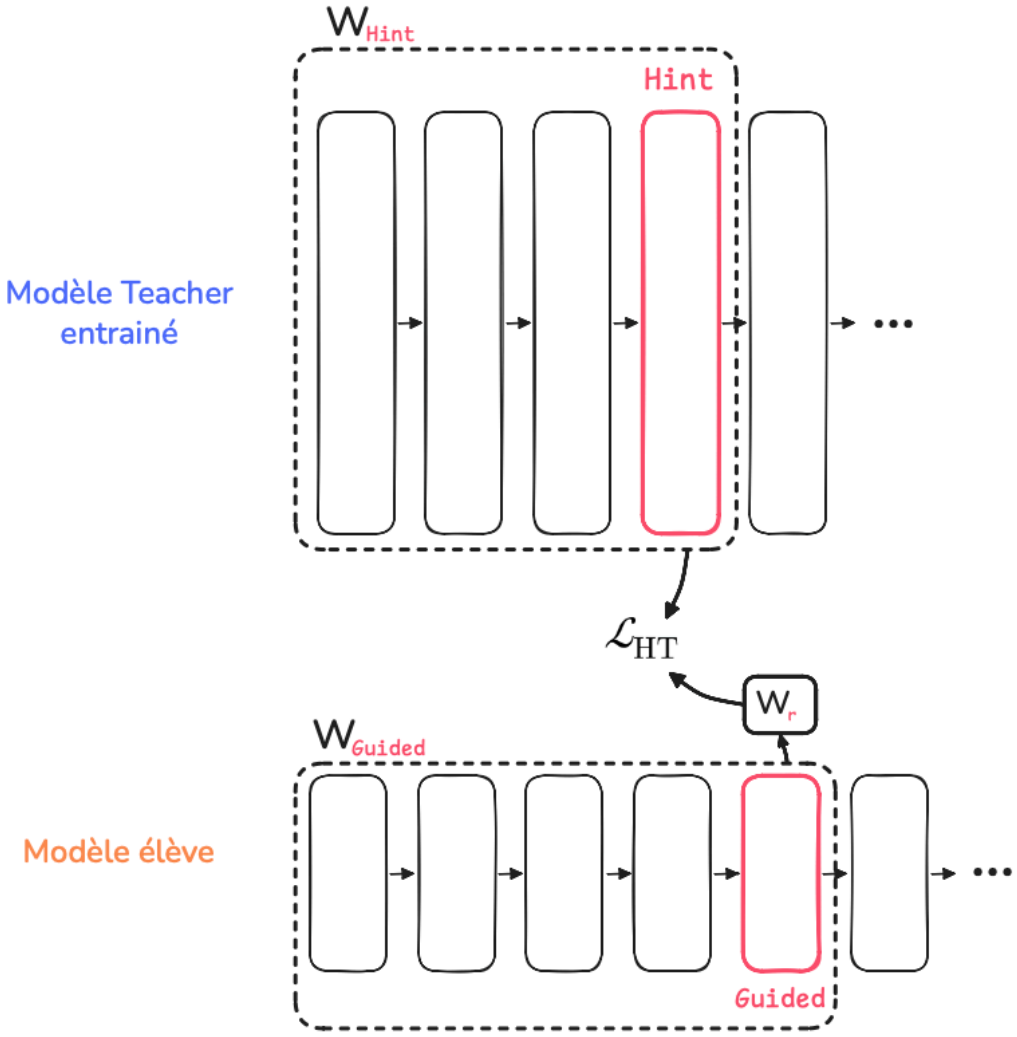
On note ensuite $u_h(x; W_{\text{hint}})$ l'ensemble des activations du teacher jusqu'à la couche **hint**, $v_g(x; W_{\text{guided}})$ l'ensemble des activations du student jusqu'à la couche **guided** et $r(v_g(x; W_{\text{guided}}), W_r)$ l'**adapteur** appliqué à la sortie intermédiaire de la couche "guided" du student.

La perte de **hint training**, notée $\mathcal{L}_{\text{HT}}$, s'écrit alors :

$$\mathcal{L}_{\text{HT}}(W_{\text{guided}}, W_r) = \frac{1}{2} \| u_h(x, W_{\text{hint}}) - r(v_g(x, W_{\text{guided}}), W_r) \|^2 \tag{16}$$

où :

- W_{hint}, W_{guided} et W_r sont les poids respectifs du teacher, du student et de l'adapteur,
- la norme $\|...\|^2$ correspond à une **erreur quadratique moyenne (MSE)** entre activations.



On minimise ensuite cette perte pendant la première étape de l'entraînement. Dans FitNets, l'apprentissage se déroule en deux temps. D'abord vient la phase de **Hint Training**, où seule la partie initiale du student (jusqu'à la couche "guided") est entraînée à reproduire les activations intermédiaires du teacher. Cette supervision agit comme une forme de pré-entraînement guidé, qui aide le réseau à construire de bonnes représentations de départ.

Une fois cette première phase terminée, on passe à la **Knowledge Distillation complète**, où le student est entraîné de bout en bout à imiter les sorties du teacher, comme dans la méthode Response-based KD présentée précédemment. Les poids appris lors de la phase de Hint Training servent alors de point de départ solide, ce qui rend l'entraînement global plus stable et plus efficace.

On peut voir cette méthode comme une forme de **curriculum learning** : au début, le **student** apprend des objectifs "simples" (reproduire une couche intermédiaire). Ensuite, il apprend des objectifs plus "complexes" (imiter les sorties finales). Ce processus progressif rend l'apprentissage plus stable et mieux régularisé. En d'autres termes, on ne demande pas au student d'imiter directement les conclusions du teacher, mais d'apprendre d'abord à "voir" le monde comme lui.

Cette double phase d'apprentissage combine **stabilité** et **efficacité**, tout en réduisant considérablement les ressources nécessaires. Et cela permet d'obtenir des gains considérables en efficacité et en taille de modèle, sans perte significative de précision. Dans l'article original FitNets [4], les auteurs évaluent leurs FitNets sur le jeu de données **CIFAR-10**, en comparant les performances d'un teacher à celles de plusieurs students plus profonds mais beaucoup plus légers. Les résultats sont les suivants :

Modèle	Nombre de couches	Nombre de paramètres	Nombre de calculs	Précision	Speed-up	Taux de compression
Teacher	5	~9M	~725M	90.18 %	1	1
FitNet 1	11	~250K	~30M	89.01 %	**13.36**	**36**
FitNet 2	11	~862K	~108M	**91.06 %**	4.64	10.44
FitNet 3	13	~1.6M	~392M	**91.10 %**	1.37	5.62
FitNet 4	19	~2.5M	~382M	**91.61 %**	1.52	3.60

Ces résultats montrent qu'il est possible d'obtenir un student jusqu'à **36 fois** plus petit et plus de **13** fois plus rapide (FitNet 1) que le teacher, tout en conservant un niveau de précision comparable. Plus remarquable encore, certaines architectures distillées (FitNet 2, 3 et 4) dépassent la précision du teacher, malgré une réduction drastique du nombre de paramètres. Cela prouve que la supervision intermédiaire par hints améliore non seulement la compacité du modèle, mais aussi sa capacité de généralisation.

5. Architectures légères pour le TinyML

Les techniques de compression — **quantization**, **distillation** et **pruning** — ont profondément transformé la manière de déployer les modèles de Deep Learning sur des appareils à ressources limitées. Mais dans de nombreux cas, même un modèle compressé reste trop lourd pour tourner sur un microcontrôleur : il faut alors repenser l'architecture elle-même.

C'est ainsi qu'est née une nouvelle génération de réseaux, conçus dès l'origine pour l'embarqué. Ces modèles "nativement légers" intègrent les mêmes principes que la compression (réduction de précision, partage de paramètres, élimination de redondances), mais **directement dans leur design structurel** : filtres séparables, blocs inversés, couches réduites, ou encore factorisation des convolutions.

Ces architectures sont aujourd'hui le socle du TinyML, capables de traiter des images, du son ou des données de capteurs — sans dépendre du cloud. Voici quelques modèles emblématiques :

5.1. MobileNetV2 – Vision par ordinateur

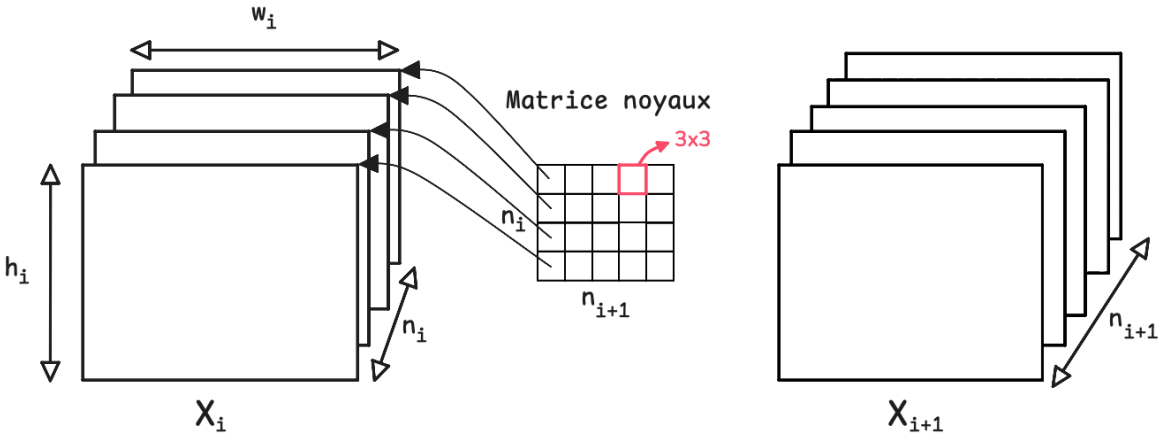
Développée par Google en 2018, MobileNetV2 [5] s'appuie sur trois idées principales : les **depthwise separable convolutions** (introduit dans MobileNetV1 [6]), les **Linear Bottlenecks** et les **inverted residuals**. Ces mécanismes réduisent drastiquement le coût de calcul et la mémoire nécessaire, tout en maintenant une bonne précision — ce qui en fait une architecture de référence pour la **vision embarquée** et le TinyML.

Les depthwise separable convolutions (rappel du principe MobileNetV1)

Avant d'entrer dans MobileNetV2, rappelons le principe général de factorisation utilisée dans MobileNetV1 : la **depthwise separable convolution**.

Dans une **convolution classique**, on applique à l'entrée $X_i \in \mathbb{R}^{h_i \times w_i \times n_i}$ un ensemble de noyaux $k \times k$ qui mélangent à la fois l'information spatiale et l'information de canaux. On peut voir le tenseur de noyaux comme un ensemble de n_{i+1} filtres, chacun étant de taille $k \times k \times n_i$ et produisant un canal de sortie. La sortie X_{i+1} est alors de taille $h_i \times w_i \times n_{i+1}$. Le coût computationnel d'une telle convolution 2D classique est :

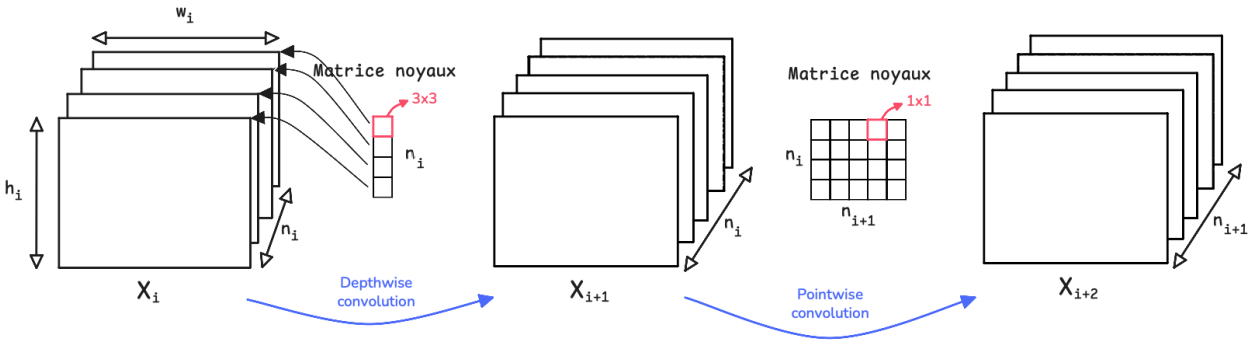
$$h_i \times w_i \times n_i \times n_{i+1} \times k^2, \tag{17}$$



Chaque filtre $k \times k \times n_i$ produit un canal de sortie parmi les n_{i+1}. Dans le schéma ci-dessus on prend $k = 3$ comme pour MobileNetV2.

Pour réduire ce coût, MobileNetV1 remplace cette opération par une **depthwise separable convolution**, qui factorise la convolution en deux étapes :

1. **Depthwise convolution** : on applique, pour chaque canal d'entrée, un filtre $3 \times 3 \times 1$ indépendant (ici $k = 3$). On obtient une carte intermédiaire $X_{i+1} \in \mathbb{R}^{h_i \times w_i \times n_i}$ et le coût associé est :
$$h_i \times w_i \times n_i \times k^2. \tag{18}$$
2. **Pointwise convolution (1×1)** : on applique ensuite des noyaux $1 \times 1 \times n_i$ pour combiner les n_i canaux et produire n_{i+1} canaux de sortie. Le coût de cette étape est :
$$h_i \times w_i \times n_i \times n_{i+1}. \tag{19}$$

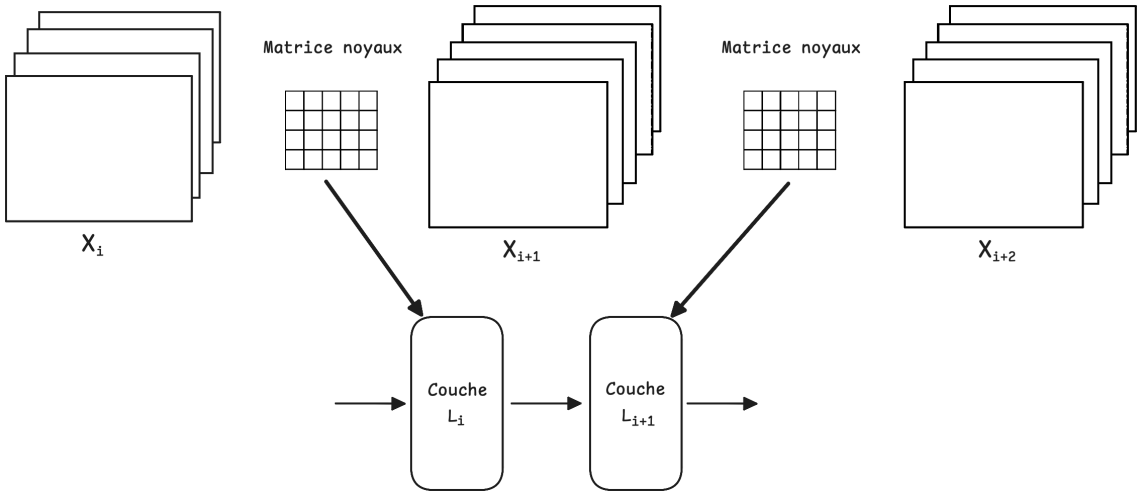


Au total, la depthwise separable convolution ne coûte plus que : $h_i \times w_i \times n_i(k^2 + n_{i+1})$ au lieu de $h_i \times w_i \times n_i \times n_{i+1} \times k^2$. Pour $k = 3$, cela conduit typiquement à une réduction d'un facteur proche de 8 à 9, pour une perte de précision très limitée.

Les Linear Bottlenecks

Dans un réseau convolutif, la matrice des noyaux correspond en fait à une couche L_i du réseau qui prend en entrée un tenseur X_i de forme $h_i \times w_i \times n_i$ où h_i est hauteur de la carte d'activation, w_i la largeur et n_i le nombre de canaux (ou profondeur).

Le tenseur de sortie $X_{\{i+1\}}$ de forme $h_{i+1} \times w_{i+1} \times n_{i+1}$ devient à son tour l'entrée de la couche suivante, etc. Ainsi, ces tenseurs contiennent en fait les activations, ce sont simplement **les sorties intermédiaires** du réseau, entre les opérations convolutives.

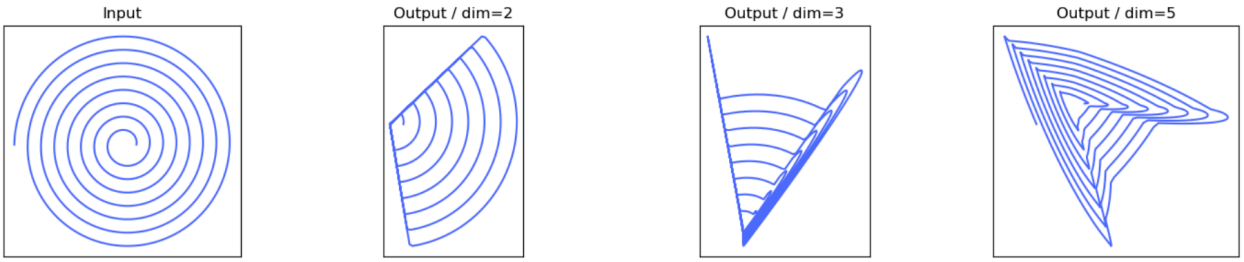


Ces activations peuvent être vues comme un ensemble de $h_i \times w_i$ "pixels", chacun étant un vecteur de dimension n_i : c'est dans cet espace que vivent les **manifolds d'intérêt** étudiés dans MobileNetV2. Dans la pratique, ces manifolds possèdent souvent une **faible dimension intrinsèque** : même si n_i est grand, l'information « utile » vit dans un sous-espace beaucoup plus petit.

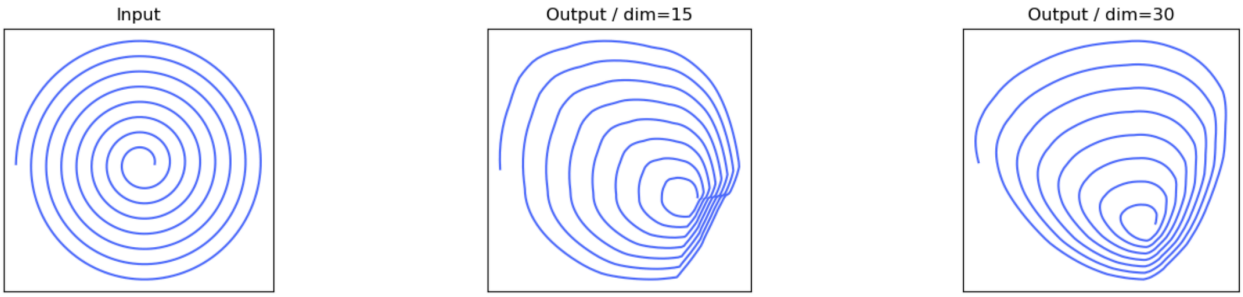
Une idée naturelle serait donc de réduire la dimension n_i (comme avec le **width multiplier** de MobileNetV1 [6]), puisque cela réduit immédiatement le coût de calcul. Mais cette intuition ignore un point crucial : les réseaux profonds appliquent des non-linéarités par canal (ReLU, ReLU6, etc).

Or, appliquer ReLU dans un espace **faible dimension** peut détruire l'information. Certains canaux **s'annulent entièrement**, des points distincts du manifold deviennent **indiscernables**, l'information perdue n'est **plus récupérable** par les couches suivantes.

Comme on le voit dans l'image ci-dessous, lorsque le manifold (la spirale d'origine) est projeté dans un espace de faible dimension (2, 3 ou 5 ici) puis soumis à ReLU, de larges portions de la spirale s'écrasent sur les axes : des segments distincts deviennent confondus, certaines directions disparaissent totalement et la géométrie est irrémédiablement aplatie. Cela illustre parfaitement la **perte d'information** provoquée par une non-linéarité appliquée dans un espace trop étroit.



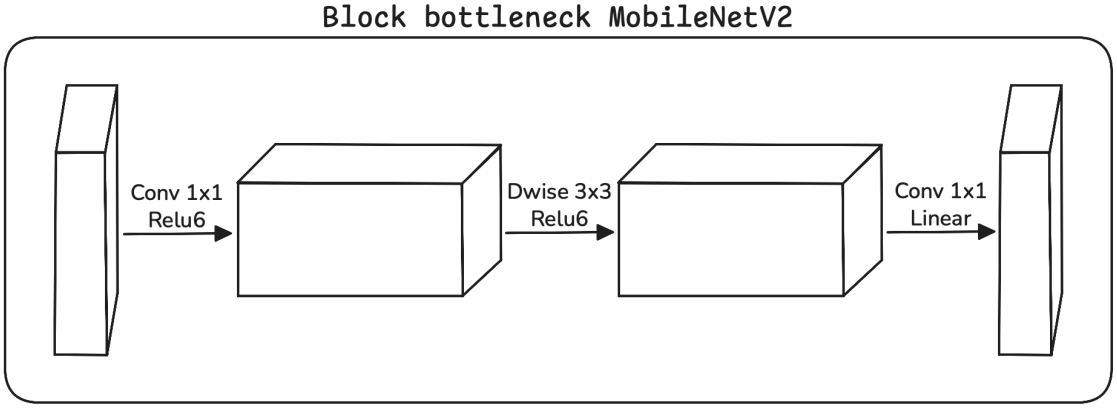
En revanche, si l'on **augmente d'abord la dimension** n_i vers un espace plus large : la non-linéarité peut déformer le manifold **sans l'écraser**, les points **restent séparés**, l'expressivité (nombre de morceaux linéaires) **augmente** et l'information peut ensuite être projetée dans un espace étroit **sans ReLU**.



Cette fois-ci, comme l'illustre l'image ci-dessus, lorsque la spirale est projetée dans un espace de dimension plus élevée (15 ou 30) avant l'application de ReLU, sa structure géométrique est largement préservée : les points restent séparés, les courbes conservent leur forme globale, et la transformation devient au contraire plus expressive et non linéaire. La non-linéarité ne détruit plus le manifold : elle le **déforme sans le faire s'effondrer**, ce qui permet ensuite de revenir à un espace étroit via une projection linéaire sans perdre l'information essentielle.

MobileNetV2 exploite précisément ce principe. Le bloc bottleneck est structuré en trois étapes :

1. **Expansion 1×1 (pointwise) + ReLU6** On étend n_i vers $t \times n_i$ (avec $t \approx 6$). Cette couche **n'est pas la pointwise du depthwise separable convolution** : c'est une expansion.
2. **Depthwise convolution 3×3 + ReLU6** C'est la seule partie du bloc correspondant directement au **depthwise** du schéma MobileNetV1. Elle capture les motifs spatiaux à faible coût.
3. **Projection linéaire 1×1 (sans ReLU)** Cette couche **n'est plus la "pointwise" du depthwise separable convolution** : elle devient le **Linear Bottleneck**, garantissant une compression fidèle du manifold.

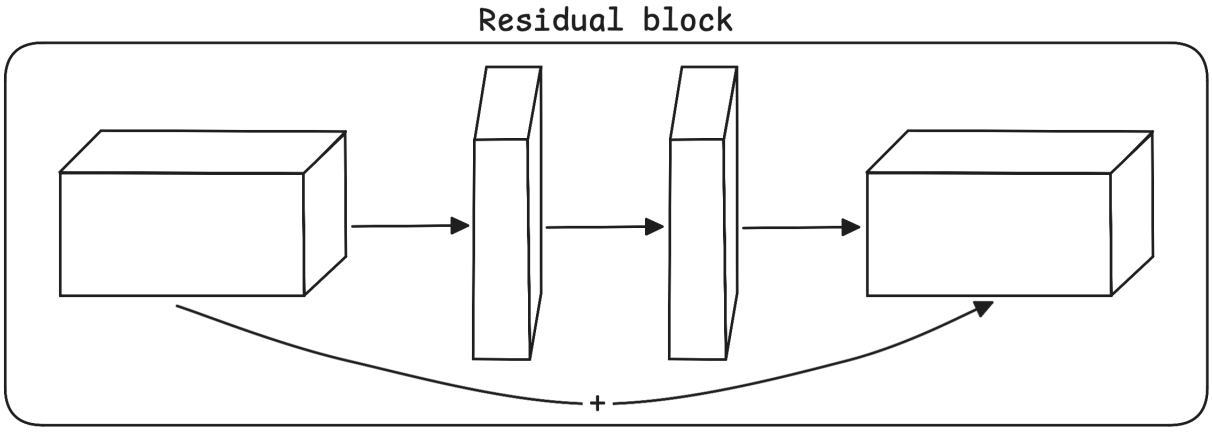


Ainsi, MobileNetV2 ne reprend pas mécaniquement la "depthwise separable convolution" du V1. Il en conserve uniquement la **depthwise 3×3**, entourée d'une expansion et d'une projection linéaire — ce qui donne un bloc **plus expressif**, **plus robuste** et mieux adapté aux contraintes du TinyML.

Les inverted residuals

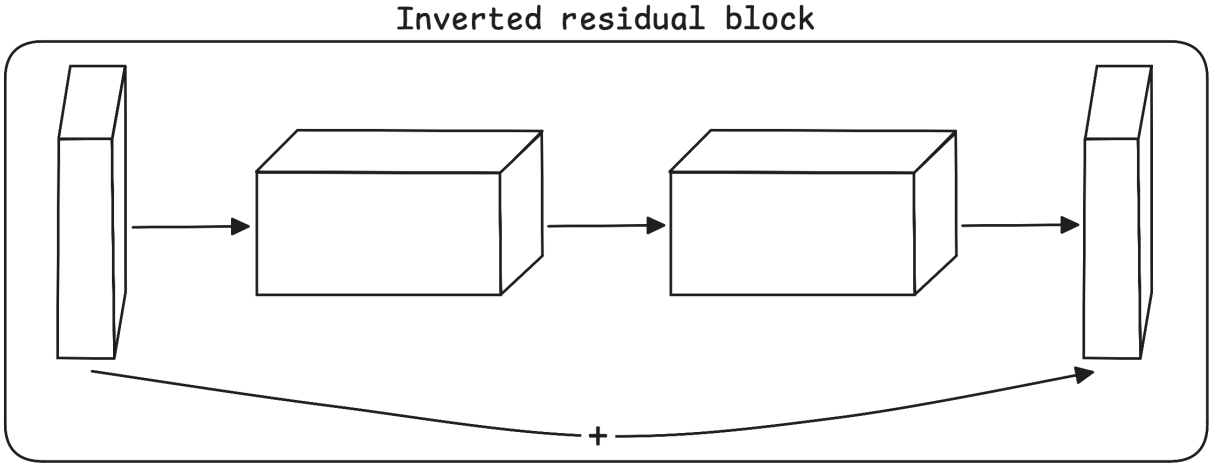
Le bottleneck de MobileNetV2 ressemble, en apparence, au bloc résiduel classique : on part d'un tenseur d'entrée étroit, on applique plusieurs transformations, puis on obtient un tenseur de sortie. La différence fondamentale réside dans **l'endroit où l'on place la connexion résiduelle**.

Dans un **residual block** traditionnel (comme dans ResNet), la connexion relie les **couches larges** : celles qui possèdent beaucoup de canaux. L'information « contourne » des transformations massives, et coûteuses à mémoriser.



Dans MobileNetV2, l'idée est inversée, le **bottleneck étroit** est l'endroit où vit réellement l'information utile (le manifold) tandis que la **couche d'expansion** ne sert pas à stocker de l'information, mais simplement d'espace intermédiaire pour appliquer les non-linéarités en toute sécurité.

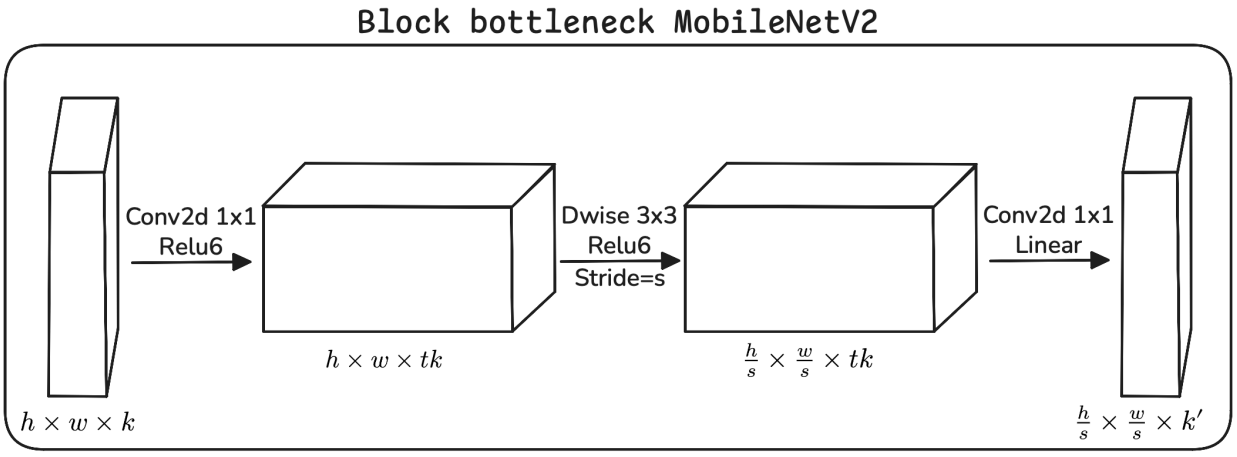
La connexion résiduelle relie donc **les couches étroites**, d'où le terme **inverted residual**. Le flux de données est inversé par rapport à ResNet : on relie les petits espaces plutôt que les grands.



Cependant, cette connexion résiduelle inversée n'est possible que si **deux conditions sont réunies** : la **résolution spatiale est inchangée** c'est à dire que la $\text{stride} = 1$, et la **dimension d'entrée est identique à la dimension de sortie** c'est à dire que $n_i = n_{i+1}$. Dans tous les autres cas (réduction de résolution, changement du nombre de canaux), le bloc ne possède pas de connexion résiduelle.

Architecture complète de MobileNetV2

Maintenant que les idées clés sont en place on peut décrire l'architecture complète de MobileNetV2. Le bloc élémentaire n'est pas une « depthwise separable convolution » comme dans MobileNetV1, mais un **bloc bottleneck réorganisé**, composé par :



Le coût en multiplications-additions pour un bloc de taille $h \times w$, facteur d'expansion t, noyau k, et dimensions d'entrée/sortie d' et d'' est :

$$h \times w \times d' \times t(d' + k^2 + d'') \tag{20}$$

L'architecture MobileNetV2 empile 19 blocs bottlenecks, organisés selon la configuration suivante :

- chaque ligne décrit une **séquence** de blocs identiques (même t, même c),
- le **premier bloc** de la séquence peut réduire la résolution (stride $s = 2$),
- les blocs suivants ont toujours $s = 1$ et utilisent une connexion résiduelle quand $n_i = n_{i+1}$.

Input	**Opérations**	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1×1	-	1280	1	1
$7^2 \times 1280$	avgpool 7×7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1×1	-	k	-	

Comme dans MobileNetV1, deux hyperparamètres permettent d'ajuster le compromis précision / complexité :

- la **résolution d'entrée** (par exemple 96 à 224 pixels) ;
- un **width multiplier** qui réduit uniformément le nombre de canaux.

Ces réglages permettent de couvrir une large gamme de budgets computationnels : de quelques millions d'opérations (TinyML très contraint) à plus de 300M d'opérations pour les versions haute précision. La taille du réseau varie alors de 1.7M à 6.9M paramètres.

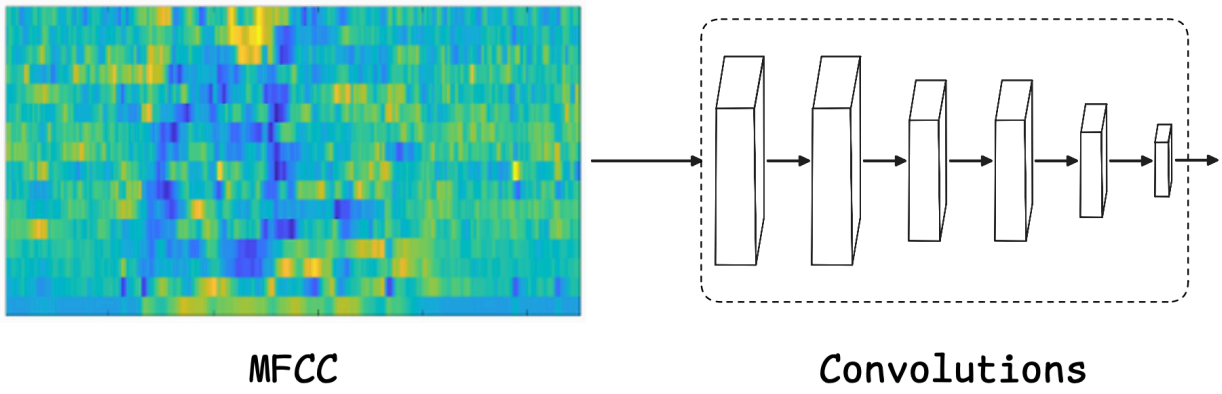
5.2. TinySpeech – Reconnaissance vocale

TinySpeech [7] est une **famille de réseaux** conçus pour la **reconnaissance de mots-clés** (keyword spotting) dans des environnements très contraints comme les microcontrôleurs ou les microprocesseurs basse consommation. L'objectif est ambitieux : réaliser une reconnaissance vocale fiable, en temps réel, sans connexion au cloud, avec moins de 10 à 15 k paramètres, et sous 8 bits.

Pour y parvenir, TinySpeech s'appuie sur deux idées complémentaires : un nouveau mécanisme de **self-attention** extrêmement compact, appelé **attention condenser**, et une exploration architecturale automatique, elle-même pilotée par machine (machine-driven design), permettant de construire des réseaux entièrement optimisés pour le TinyML.

Les attention condensers

Les architectures standard de reconnaissance vocale reposent généralement sur des **réseaux convolutifs** : on extrait des MFCC (une représentation temps-fréquence de l'audio), puis on applique des convolutions 2D pour capturer la structure locale du signal. Même dans leur version allégée, ces réseaux convolutifs restent trop coûteux lorsqu'on vise des modèles de quelques kilo octets.

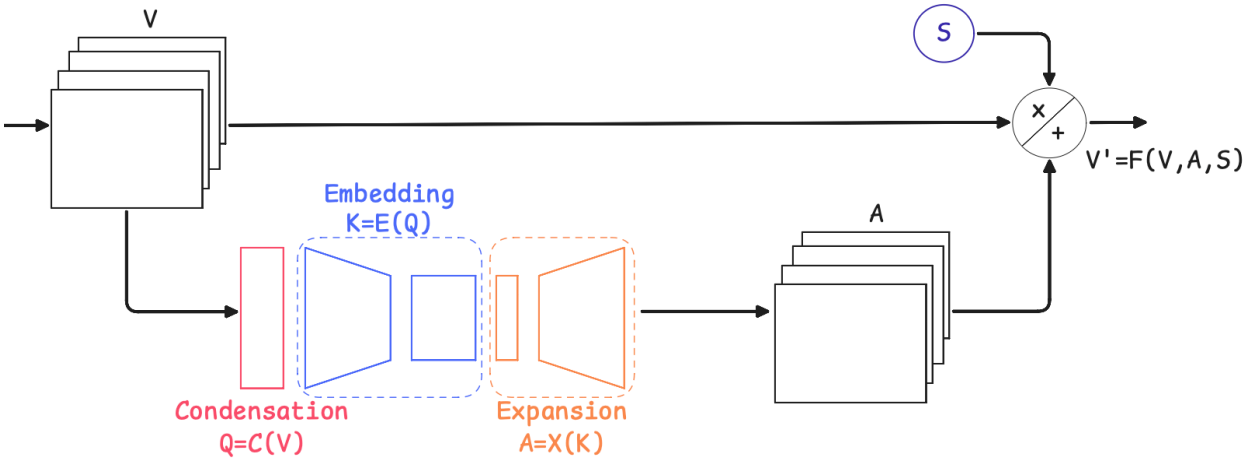


L'idée de TinySpeech est donc de remplacer les convolutions intermédiaires par un mécanisme de **self-attention léger** conçu pour modéliser à la fois les relations locales dans le temps-fréquence et les relations entre canaux. Là où les mécanismes d'attention traditionnels sont lourds, multi-têtes et difficiles à déployer sur microcontrôleurs, l'attention condenser est pensé comme un bloc minimaliste mais suffisant pour extraire l'essentiel des dépendances du signal vocal.

Un attention condenser est constitué de **quatre sous-modules** :

- Un module de **condensation** $C(V)$, qui réduit la dimension du tenseur d'activations V, issu d'une couche précédente, dans un nouveau tenseur Q.
- Un module **Embedding condensé** $E(Q)$ qui capture simultanément les relations locales dans le spectrogramme (variations temporelles et fréquentielles) et les relations inter-canaux, qui permettent de comprendre comment les différentes bandes fréquentielles interagissent.
- Un module **Expansion** $X(K)$, qui ré-étend l'embedding condensée K dans un espace plus large pour produire un tenseur A, dont les valeurs serviront à moduler finement l'activation d'entrée. Cette expansion compense la réduction initiale de dimension, tout en restant extrêmement légère computationnellement.
- Un module **Selective attention** $F(V, A, S)$ qui combine l'entrée V, les valeurs d'attention A, et un facteur de pondération S. Le facteur S permet de contrôler l'intensité de l'attention. Quand S diminue, l'effet de l'attention augmente, et lorsque $S \to 0$, la sortie se rapproche de A. Ce mécanisme offre un contrôle fin de la sélectivité du modèle.

On peut représenter cette structure par le schéma ci-dessous :



L'effet global est de remplacer ce que ferait un ensemble de convolutions (capture de motifs locaux et combinaison entre canaux) par un module bien plus compact, avec un nombre de paramètres extrêmement réduit, mais dont l'attention ciblée permet de maintenir une bonne précision.

Autrement dit : l'attention condenser joue le rôle d'un « condensateur d'information », qui extrait et renforce les motifs vocaux les plus utiles, tout en supprimant les parties non pertinentes du spectrogramme.

Machine-driven design exploration

Dans TinySpeech, le second pilier après les attention condensers est l'usage d'une méthode d'exploration architecturale automatique, appelée machine-driven design exploration. L'idée est radicale : ne pas concevoir à la main l'architecture, mais laisser une machine explorer des milliers d'architectures potentielles et en extraire celles qui respectent des contraintes très strictes de TinyML.

Les contraintes imposées dans le cas de TinySpeech sont :

- une précision minimale de 90 % sur le jeu de données Speech Commands [8],
- un nombre de paramètres inférieur à 15.000,
- un poids des paramètres quantifié en 8 bits (quantization),
- et, dans une variante spécialisée, uniquement des opérations supportées par TensorFlow Lite Micro (pour microcontrôleurs).

Ces contraintes sont strictes, ce qui pousse implicitement le générateur à utiliser massivement les attention condensers (qui sont très peu coûteux), et à éviter autant que possible les convolutions classiques.

Cette approche s'appuie sur un cadre théorique appelé generative synthesis, un procédé d'optimisation où un générateur apprend à produire des architectures répondant à des critères de performance et de contraintes définies par l'utilisateur.

Dans TinySpeech, la méthode consiste à poser le problème suivant :

$$\mathcal{G} = \max_{\mathcal{G}} \mathcal{U}(\mathcal{G}(s)) \quad \text{sous la contrainte que} \quad \mathbb{1}_r(\mathcal{G}(s)) = 1, \quad \forall s \in S \tag{21}$$

où :

- $\mathcal{G}$ = un générateur de réseaux neuronaux,
- $\mathcal{U}$ = une fonction de performance universelle,
- $\mathbb{1}_r$ = une fonction indicatrice qui impose les contraintes réelles (mémoire, précision, opérations autorisées…).
- S est un ensemble de graines de génération

Autrement dit : la machine essaie de créer des architectures, mais seules celles qui respectent toutes les contraintes sont validées, et la meilleure d'entre elles est retenue.

Architecture générale de TinySpeech

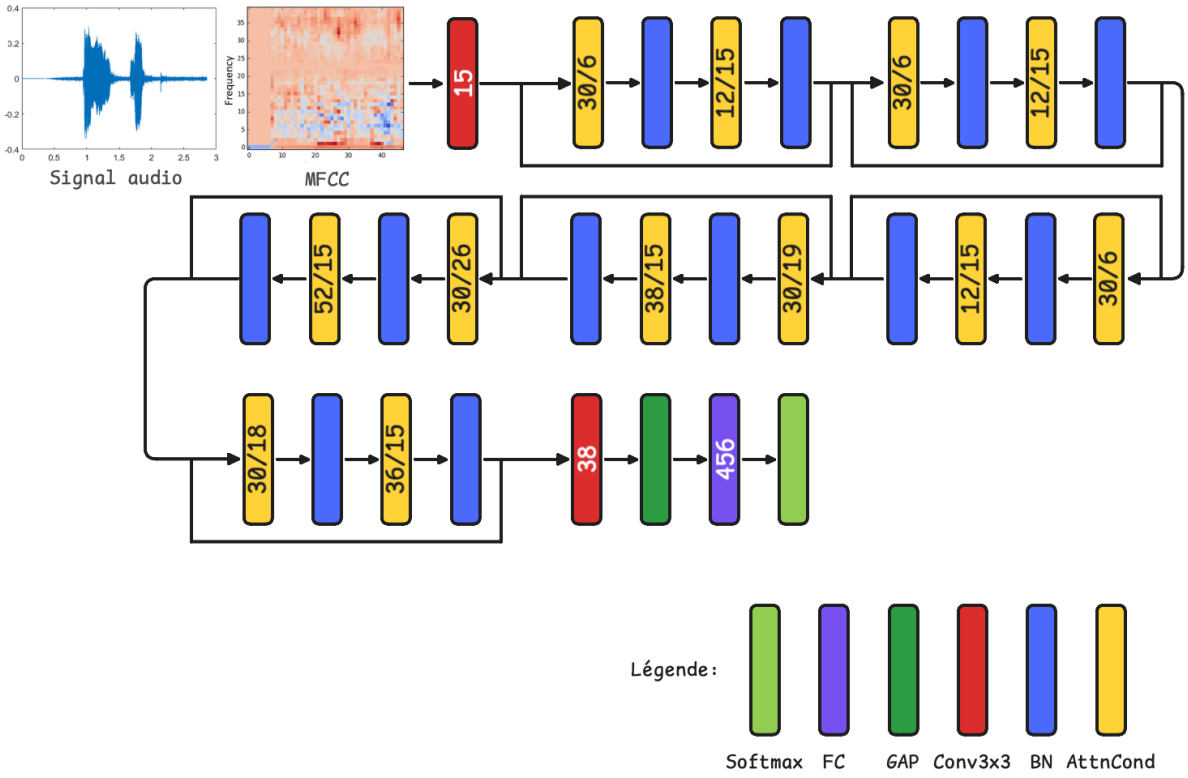
La famille TinySpeech est caractérisée par une organisation extrêmement compacte, optimisée pour les microcontrôleurs et les dispositifs TinyML. Malgré leurs différences, toutes les variantes (TinySpeech-X, -Y, -Z et -M) suivent une structure macro-architecturale commune, mise en évidence par l'exploration automatique :

1. Une première **convolution 3x3** qui sert à projeter le spectrogramme **MFCC** dans un espace de canaux suffisant pour initier les traitements d'attention.
2. Une succession de blocs **AttnCond** + **BatchNorm (BN)** (ou sans BN dans TinySpeech-M) où chaque bloc contient :
   - un Attention Condenser (capturant les relations locales + inter-canaux),
   - un BatchNorm (sauf dans la version M),
   - un raccourci résiduel (add).
3. Une seconde **convolution 3×3** qui permet une dernière combinaison spatiale en sortie des blocs d'attention avant la classification.
4. Un Global Average Pooling 1×1 (GAP) qui réduit toute la carte d'activation à un vecteur de dimension égale au nombre de canaux de sortie.
5. Une couche **Fully Connected** (FC) + **Softmax** qui produit la probabilité des classes (mots-clés).

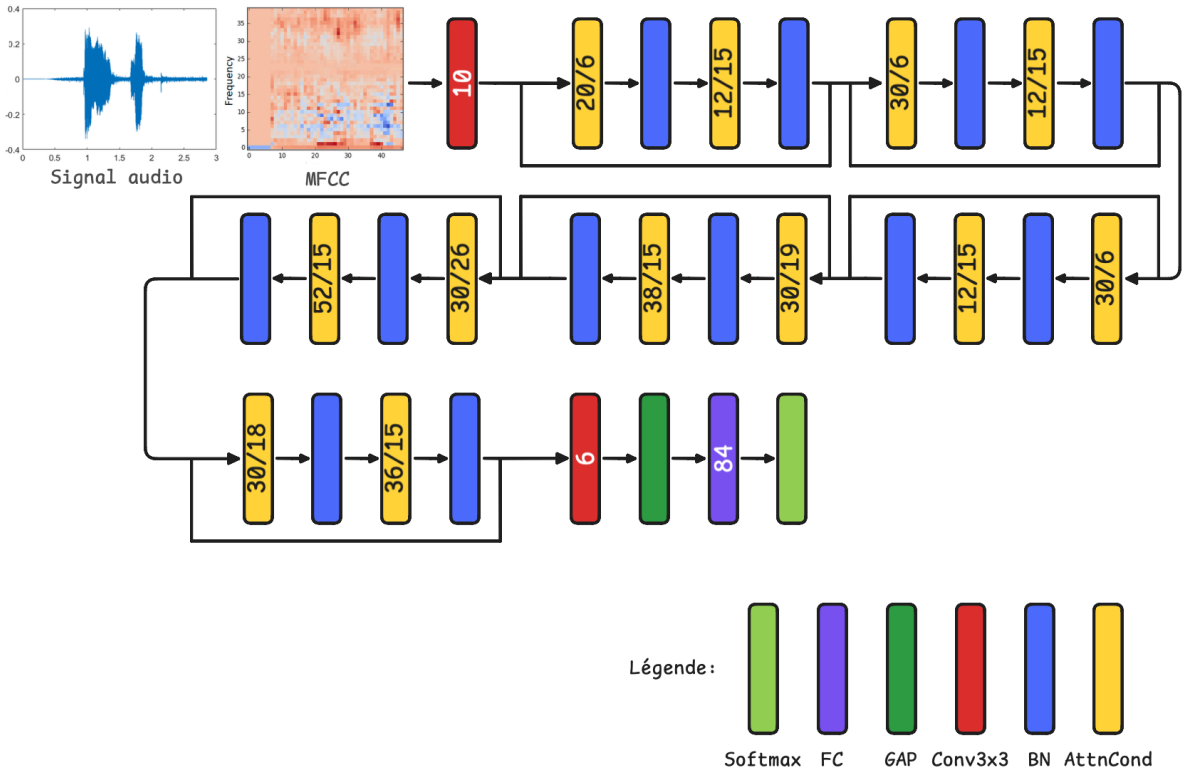
Cette organisation reste identique pour toutes les variantes; ce qui change, c'est le nombre de blocs AttnCond, les dimensions internes des attention condensers (mid/out channels), les canaux de sortie des deux convolutions, la présence ou non de BatchNorm (absent dans TinySpeech-M).

Cette diversité reflète directement le processus de machine-driven design exploration, où le générateur ajuste à la fois la micro-architecture (nombre de canaux) et la macro-architecture (nombre de blocs).

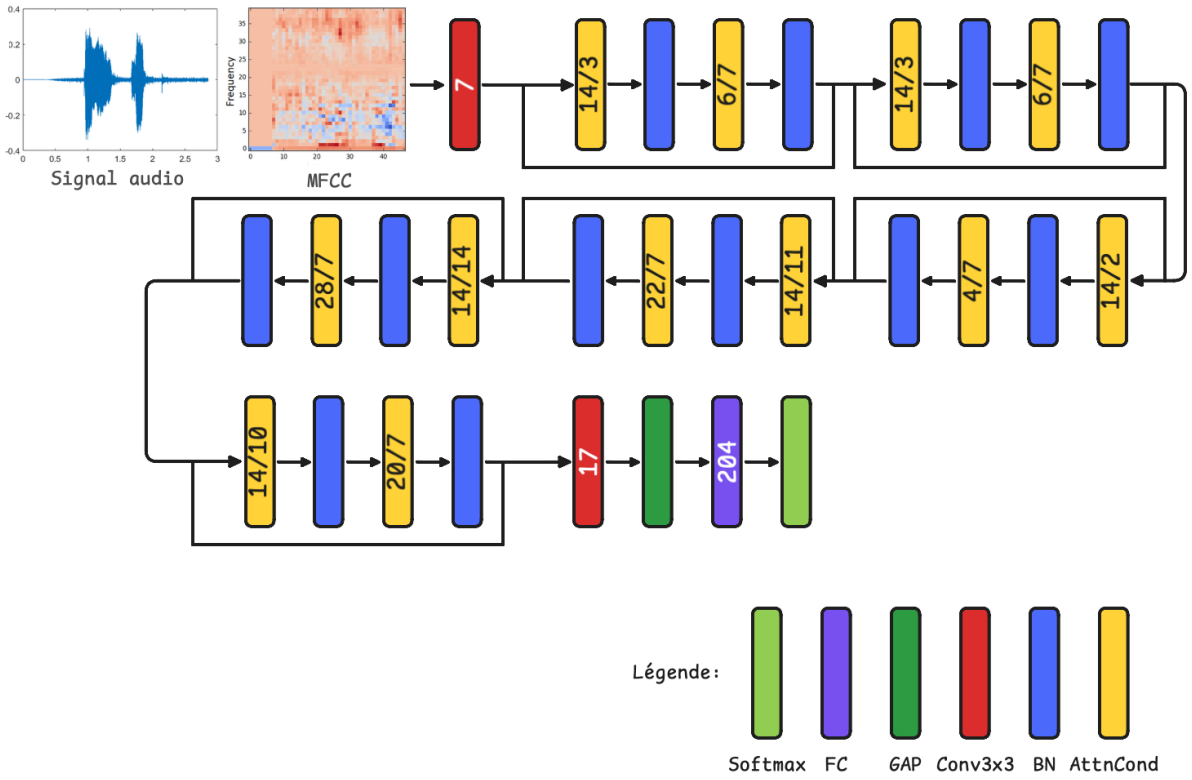
TinySpeech-X



TinySpeech-Y

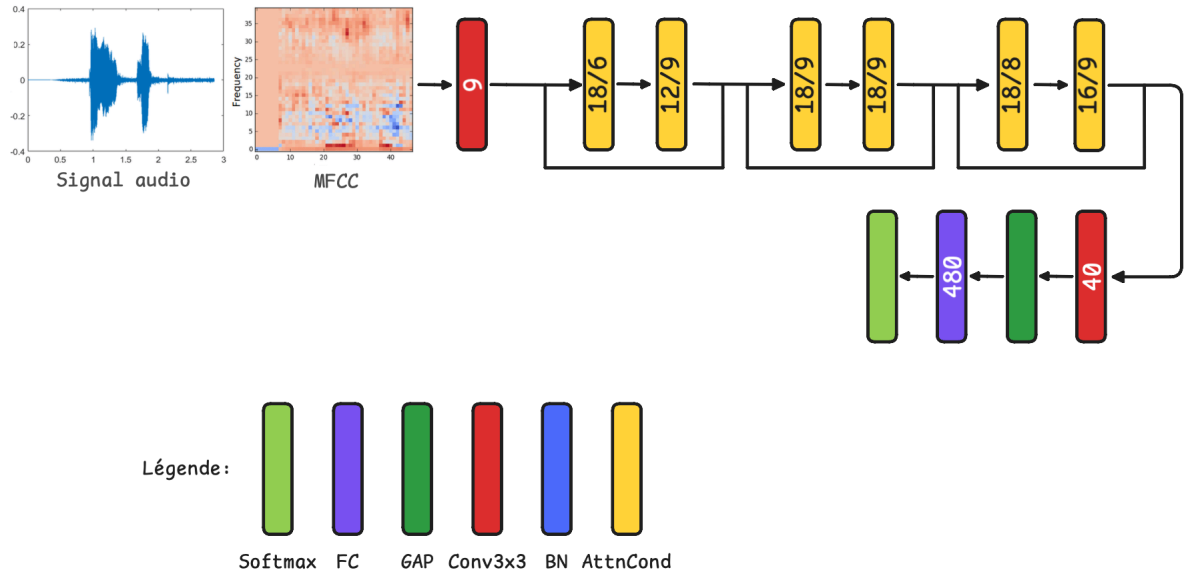


TinySpeech-Z



TinySpeech-M — Architecture *TensorFlow Lite for Microcontrollers*

TinySpeech-M est la seule variante produite sous contraintes strictes de microcontrôleurs, imposant l'usage exclusif des opérations supportées par *TensorFlow Lite for Microcontrollers*. Cela se traduit immédiatement par l'absence de BatchNorm dans les blocs; la profondeur de l'architecture se trouve réduite, et les condensers eux-mêmes adoptent des tailles internes plus modestes.



Résultats du modèle

TinySpeech a été évalué sur le **Google Speech Commands Dataset**, un corpus standard de 65 000 enregistrements d'une seconde contenant une vingtaine de mots courts (« yes », « no », « up », « down », etc.), ainsi que du bruit de fond. Tous les TinySpeech sont entrainés avec les mêmes paramètres (learning rate, epochs, batch size, ...).

Les performances sont comparées à cinq modèles de référence de la littérature, tous réputés pour leur efficacité en reconnaissance vocale embarquée. Contrairement à ces modèles de référence évalués en 32 bits, les TinySpeech sont évalués en 8 bits (faible précision), ce qui rend les comparaisons encore plus impressionnantes.

Modèle	Précision	Nombre de paramètres	Nombre de calculs
trad-fpool13 [9]	90.5%	1370K	125M
tpool2 [9]	91.7%	1090K	103M
TDNN [10]	94.2%	251K	25.1M
res15-narrow [11]	94.0%	42.6K	160M
PONAS-kws2 [12]	94.3%	131K	168M
TinySpeech-X	**94.6%**	10.8K	10.9M
TinySpeech-Y	93.6%	6.1K	6.5M
TinySpeech-Z	92.4%	**2.7K**	**2.6M**
TinySpeech-M	91.9%	4.7K	4.4M

TinySpeech-X obtient la meilleure précision de tous les modèles TinySpeech (94.6 %), avec seulement 10.8k paramètres. **TinySpeech-Z** est le plus compact de la famille (2.7k paramètres) pour une précision encore très robuste (92.4 %). **TinySpeech-M**, conçu pour microcontrôleurs, respecte des contraintes matérielles strictes (TensorFlow Lite Micro), tout en surpassant trad-fpool13 en précision avec ≈291× moins de paramètres et 28× moins d'opérations.

Au fil de cet article, nous avons progressivement construit une vision complète du TinyML moderne : réduire, comprimer, distiller et ré-architecturer l'intelligence artificielle pour qu'elle puisse s'exécuter au plus près des capteurs, avec quelques dizaines de kilooctets de mémoire et une puissance de calcul dérisoire.

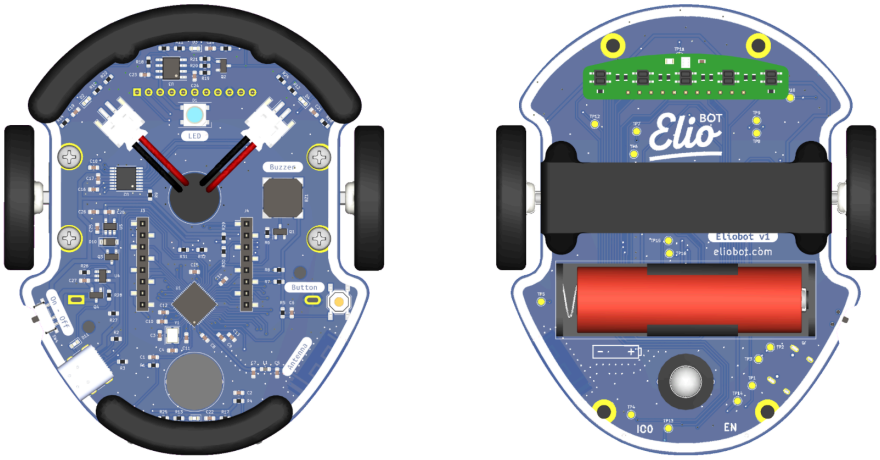
La variante TinySpeech-M, spécialement conçue sous les contraintes de *TensorFlow Lite Micro*, constitue un candidat idéal pour l'exécution sur microcontrôleur. Sa structure dépourvue de BatchNorm, sa taille compacte et son faible nombre d'opérations le rendent particulièrement adapté à un déploiement sur un robot sous ESP32, pour la commande vocale.

6. PILOTER ELIOBOT GRÂCE À LA VOIX AVEC TINYSPEECH

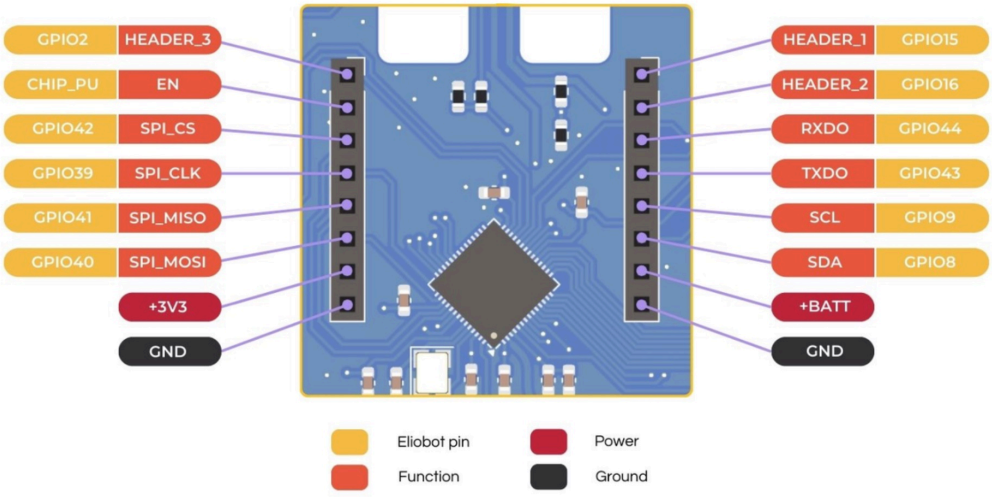
Dans cette partie pratique, nous allons mettre en œuvre le robot ElioBot, un petit robot mobile basé sur un ESP32-S2, pour lui donner une véritable capacité de commande vocale embarquée. L'objectif : capter l'audio en continu via un microphone SPH0645 MEMS en interface I2S, extraire les caractéristiques du signal, et exécuter le modèle TinySpeech directement sur le microcontrôleur pour interpréter des ordres tels que avance, stop, gauche ou droite.

6.1. Caractéristiques d'ElioBot et accès aux broches

Le robot est construit autour d'une carte ESP32-S2 intégrée au châssis, offrant un accès direct aux broches nécessaires pour les moteurs, les capteurs et les extensions. La vue supérieure et inférieure permet de repérer les connecteurs latéraux et les pins de la carte principale :



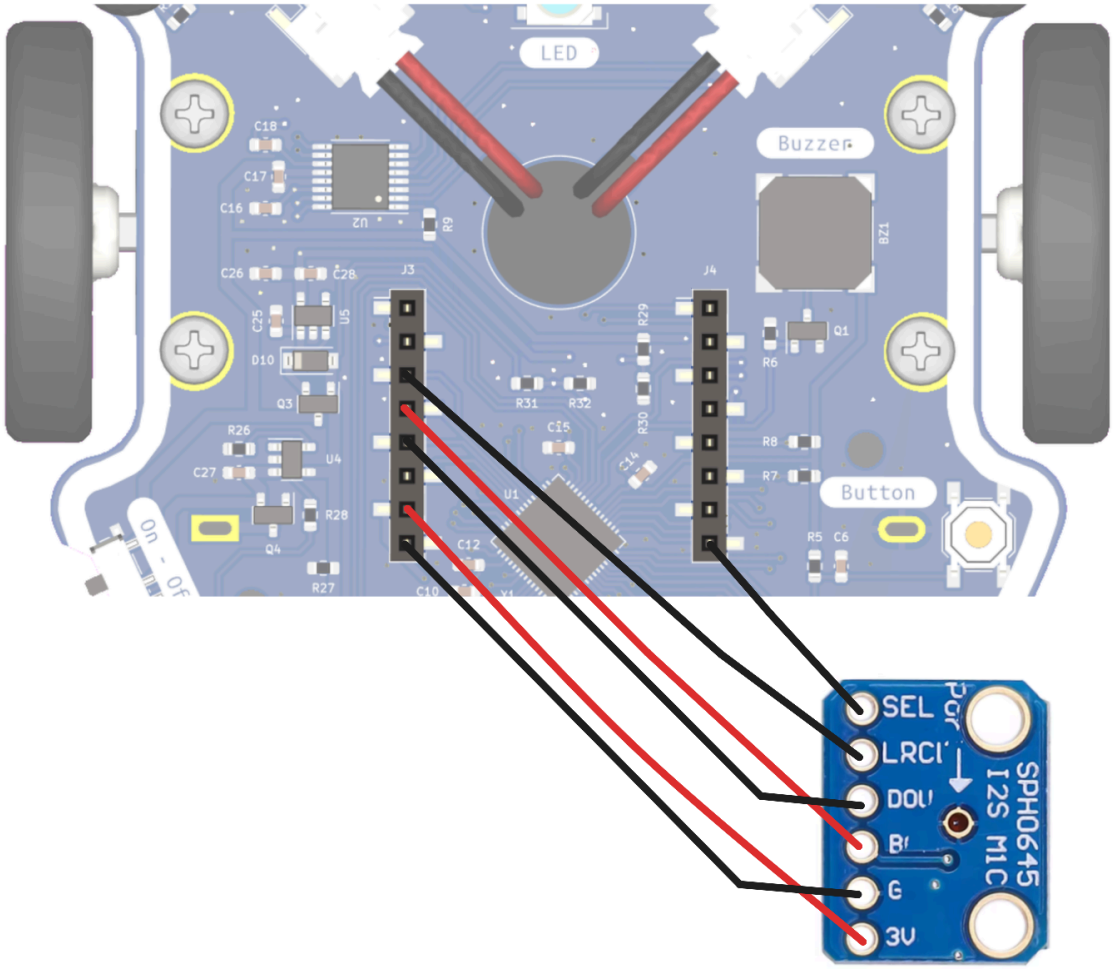
L'illustration suivante détaille précisément les broches disponibles de l'ESP32-S2 sur ElioBot. Elles nous serviront pour le câblage du microphone et l'acquisition audio *I2S* :



6.2. Branchement du microphone SPH0645 (I2S)

Le module SPH0645 intègre un microphone MEMS bottom-port, c'est-à-dire que l'entrée acoustique se trouve sous la carte. Il dispose de six broches : alimentation (3.3 V / GND), sélection du canal (SEL), et les trois signaux I2S (BCLK, LRCL et DOUT).

Voici le câblage pour une intégration propre et compatible avec l'ESP32-S2 d'ElioBot :



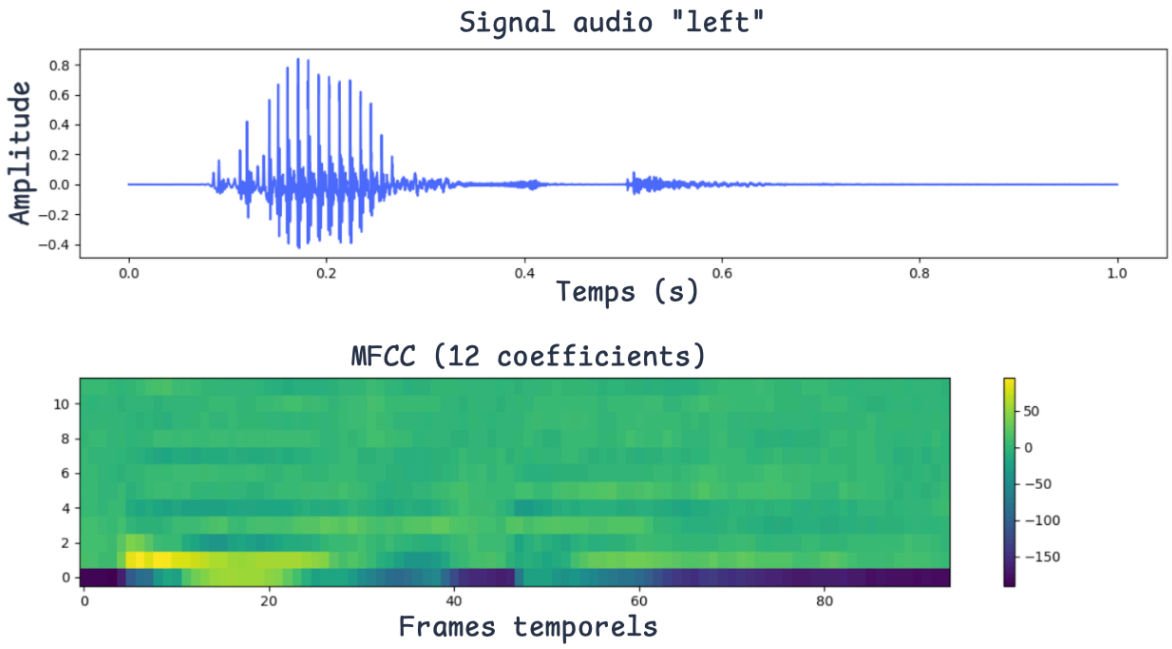
Ce branchement permet au micro de produire un flux audio numérique propre, directement exploitable pour l'extraction des MFCC et l'inférence du modèle TinySpeech.

6.3. *Les données*

Le dataset utilisé est le dataset `speech_commands` version 2 [8], publié le 11 avril 2018. Il contient $105,829$ fichiers audio d'une seconde. Les signaux sont mono, normalisés et échantillonnés à 16 kHz.

Dans le cadre de ce projet, nous ne conservons que les 10 mots utiles pour la commande vocale d'un robot : Yes, No, Up, Down, Left, Right, On, Off, Stop, Go. Chaque étiquette est représentée par environ $4,000$ échantillons, soit un total de $38,546$ signaux. On divisera les données en un jeu d'entrainement de $30,836$ données, un jeu de validation de $3,855$ données et un jeu de test de $3,855$ données.

Exemple d'un signal associé au mot `left`, accompagné de son MFCC :



6.4. *Architecture du modèle*

Pour satisfaire les contraintes très strictes d'un microcontrôleur, le modèle doit rester très léger. Nous partons donc de TinySpeech-Z, l'architecture la plus légère de la famille TinySpeech, puis nous l'allégeons davantage : les deux derniers blocs d'attention sont retirés (pour passer à quatre), et la couche fully-connected intermédiaire est réduite à 17 unités. Une dernière couche projette ensuite vers les 10 classes souhaitées.

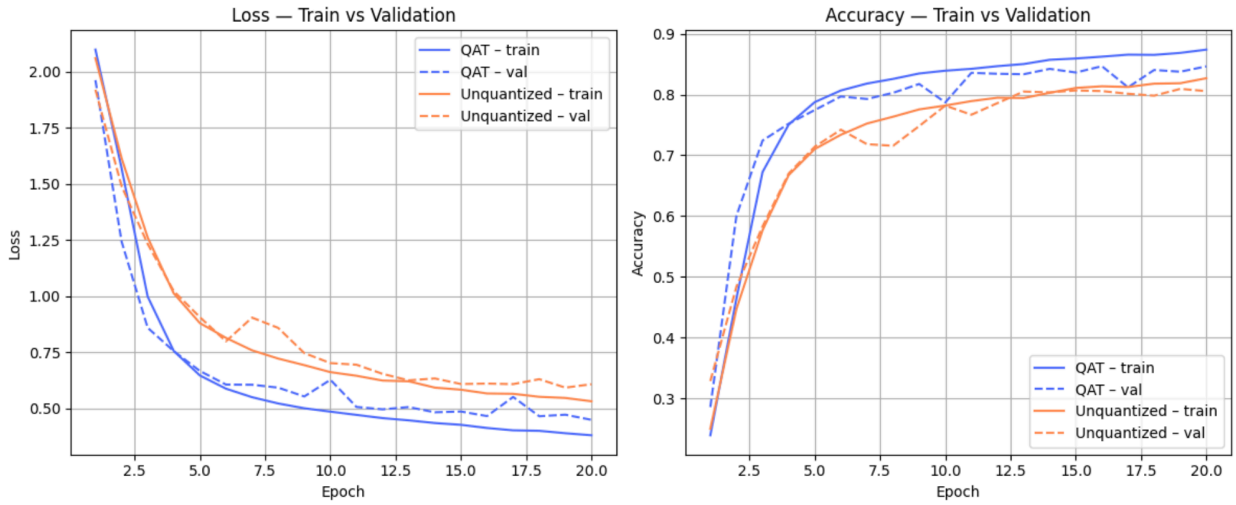
Afin d'optimiser l'exécution embarquée, nous appliquons une **Quantization-Aware Training** (QAT). Durant l'apprentissage, des opérations de FakeQuant simulent un environnement *int8* alors que les calculs restent en *float32*. Cette approche permet au réseau d'apprendre directement à être robuste à la quantification, bien avant son déploiement.

Pour situer les performances, nous entraînons aussi une version non quantifiée de TinySpeech-Z, avec l'architecture complète (6 blocs).

6.5. *Entraînement*

L'entraînement est réalisé sur un MacBook Pro M1 Pro pendant 20 epochs, avec un batch size de 64 et 4 workers. Nous utilisons SGD avec un learning rate de 0.01, un momentum de 0.9 et un weight decay de 10^{-4}. Une epoch prend environ 1 minute en mode QAT, contre 40 secondes pour la version non quantifiée.

Les courbes de Loss et d'Accuracy sur les jeux d'entraînement et de validation montrent un comportement très stable. La version quantifiée, représentée en bleu, converge même légèrement mieux que la version non quantifiée en orange :



À la fin de l'entraînement, les deux modèles présentent des profils nettement distincts.

Le modèle non quantifié (TinySpeechZ) contient $4,846$ paramètres entraînables, ce qui représente environ 18.93 KB en FP32. La version quantifiée (QTinySpeechZ, QAT) n'en compte que $3,164$, notamment grâce à la suppression de deux blocs d'attention, et occupe environ 12.36 KB en FP32 avant la quantization finale. Une fois la quantization réelle appliquée, les poids passent en int8, ce qui réduit drastiquement la mémoire : le modèle QTinySpeechZ INT8 (déployable) pèse seulement 3.09 KB, et ne nécessite pas de stockage supplémentaire pour les échelles (scales) dans ce cas précis.

Sur le plan des performances, l'effet de la quantification est visible : le modèle QAT atteint 86.3% d'accuracy sur le jeu de test, tandis que le modèle non quantifié obtient 80.7%.

6.6. *Implémentation du modèle et du moteur d'inférence embarqué*

L'objectif poursuivi est de produire un modèle de reconnaissance vocale suffisamment performant pour un usage réel, tout en respectant les contraintes d'un ESP32, dont la mémoire disponible se mesure en dizaines de kilooctets. Pour y parvenir, l'entraînement repose sur une quantification entièrement maîtrisée : un apprentissage en QAT, une conversion finale en entiers signés sur 8 bits, puis l'export d'un fichier `C` statique contenant l'ensemble des poids et des échelles de déquantification. Cette approche élimine toute dépendance à PyTorch lors de l'inférence, condition indispensable pour fonctionner sur microcontrôleur.

L'architecture utilisée, TinySpeech-Z, n'emploie pas les couches standards de PyTorch mais un ensemble de blocs spécialisés issus du projet [13], eux-mêmes inspirés des mécanismes de low-bit quantization développés dans BitNetMCU. Ces blocs assurent la quantification des activations, la mise à jour adaptative des facteurs d'échelle, et l'utilisation d'un Straight-Through Estimator pour maintenir la stabilité de l'apprentissage malgré la présence d'arrondis. Durant l'entraînement, les tenseurs restent en *float32*, mais l'ensemble des opérations simule un comportement *int8*. La quantification finale remplace ensuite réellement les poids par des entiers compris entre -128 et 127, accompagnés de leurs échelles respectives.

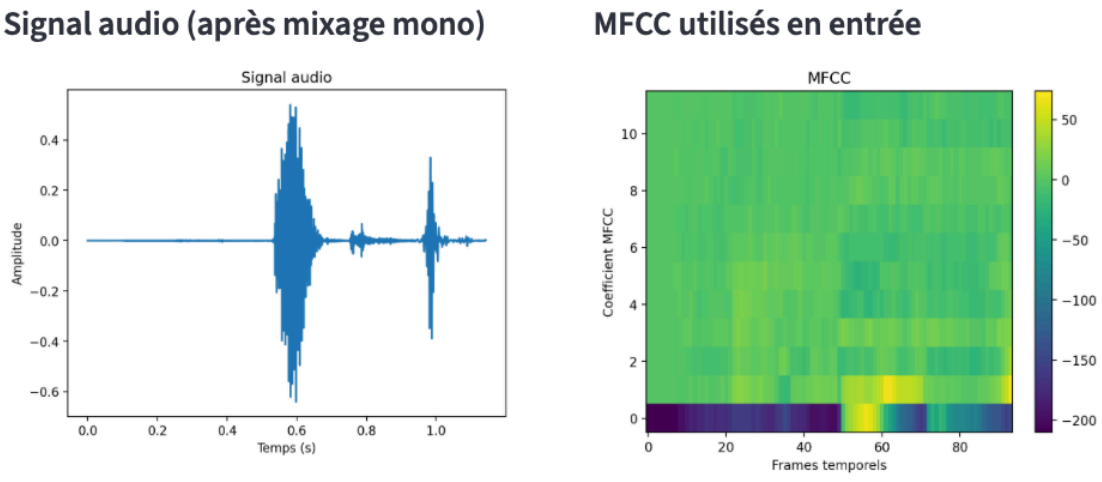
Une fois l'apprentissage terminé, un script Python génère automatiquement un fichier header rassemblant les poids quantifiés, la description des tenseurs, les dimensions et les constantes nécessaires à l'inférence. Ce header constitue l'unique source de vérité du modèle embarqué. Le moteur d'inférence en `C`, fourni par [13] et dérivé des travaux initiaux de BitNetMCU, reconstruit alors l'ensemble du réseau en exprimant manuellement les opérations fondamentales : convolutions en int8, normalisation quantifiée, fonctions d'activation, attention condensée et couche linéaire finale. L'exécution ne dépend d'aucune bibliothèque externe, ce qui permet une portabilité complète vers des plateformes à faibles ressources.

Une fois la compilation validée sur ordinateur hôte, le moteur reproduit fidèlement l'inférence du modèle PyTorch quantifié. Le modèle devient alors entièrement autonome et directement intégrable dans le firmware du robot ESP32.

6.7. Test du modèle entraîné et préparation à l'inférence temps réel

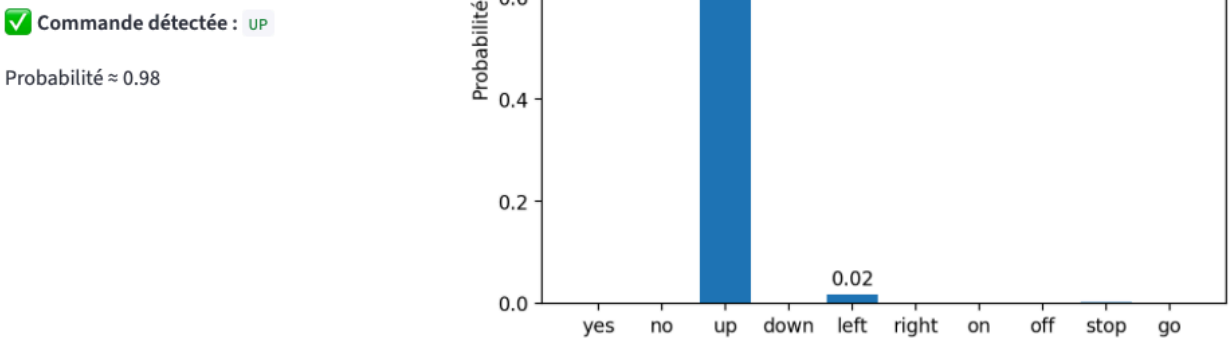
Avant d'envisager le déploiement embarqué, il est indispensable d'examiner le comportement du modèle dans un environnement contrôlé. Cette étape valide non seulement la capacité du réseau quantifié à reconnaître correctement les commandes vocales, mais permet également de définir la stratégie décisionnelle qui sera ensuite exécutée sur l'ESP32.

Pour cela, nous avons développé une interface **Streamlit** permettant d'enregistrer directement la voix via le navigateur. Une fois l'audio capturé, nous extrayons uniquement la dernière seconde du signal, puis nous calculons son MFCC, exactement comme pendant l'entraînement. Ce sera notre première logique pour tester le modèle. Cet extrait est ensuite passé au modèle pour obtenir les probabilités associées aux 10 commandes vocales.



Deux situations se présentent alors. Tant qu'aucune probabilité ne dépasse un certain seuil, le segment est considéré comme du bruit et aucune action n'est déclenchée. En revanche, dès qu'une classe dépasse ce seuil, une commande est reconnue et peut être transmise au robot. Dans cette application, un seuil de confiance de 80% a été retenu, car il représente un compromis efficace entre sensibilité et robustesse.

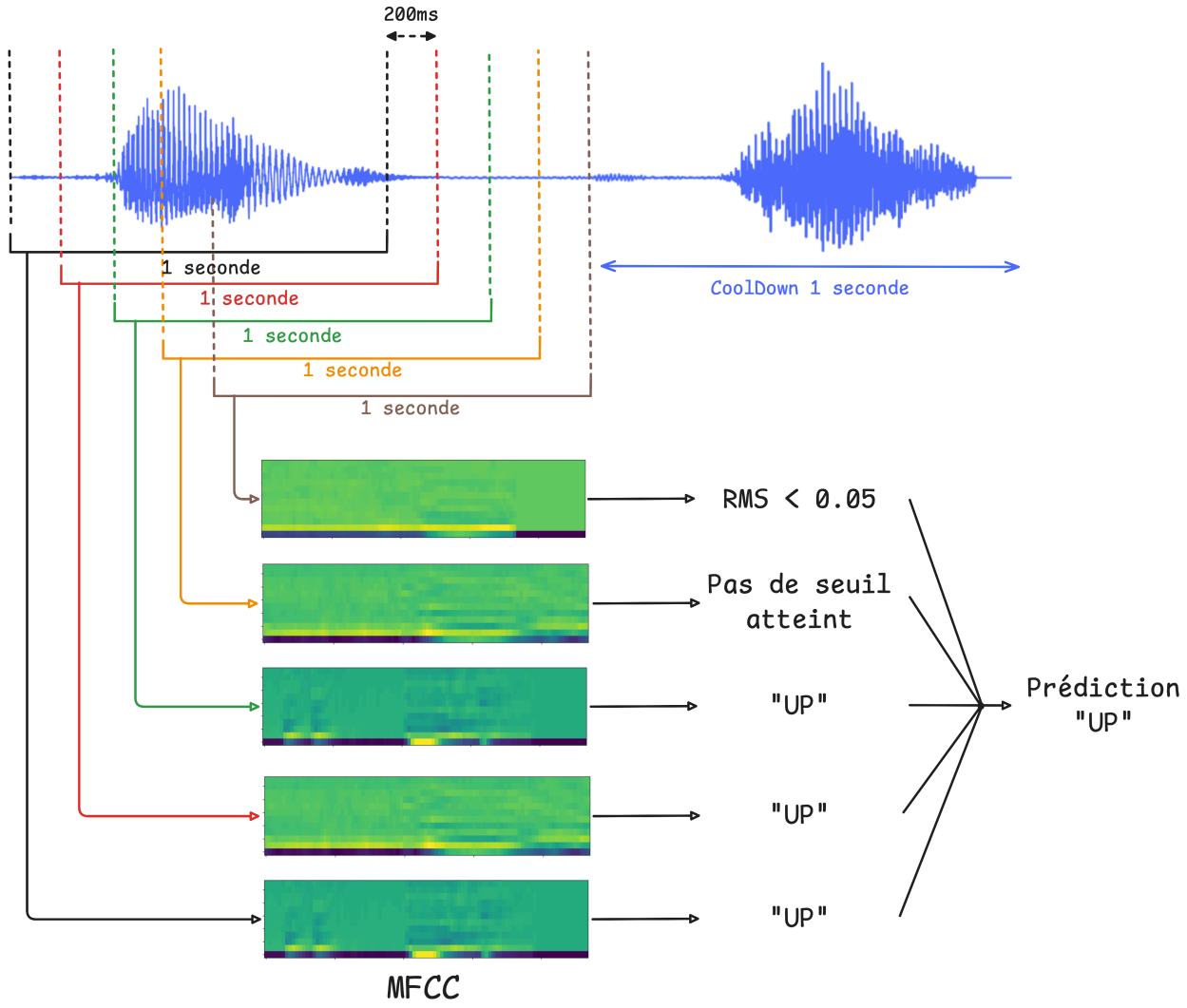
Résultat



Cette première approche permet d'observer le comportement du modèle dans un cadre stable, mais elle reste éloignée de l'usage réel. Un robot n'écoute pas des fragments figés d'une seconde : il traite un flux continu. Des fluctuations aléatoires dans le signal ou des bruits transitoires peuvent parfois produire une prédiction faussement confiante.

C'est ainsi qu'émerge la nécessité d'une logique mieux adaptée au temps réel, fondée non plus sur une décision instantanée, mais sur une analyse progressive de plusieurs prédictions successives. Le principe reste très simple. Au lieu de prendre un unique segment d'une seconde, l'audio est découpé en fenêtres de même durée, mais qui se chevauchent toutes les 200 ms. À chaque fenêtre, le signal n'est traité que si son amplitude **RMS** dépasse un seuil minimal de 0.05, afin d'éliminer les portions silencieuses et les bruits faibles. Les probabilités produites par le modèle sont ensuite enregistrées dans un petit historique de cinq fenêtres, qui permet de lisser les variations. Une commande n'est validée que si elle reste dominante dans au moins trois de ces fenêtres et si sa probabilité moyenne dépasse le seuil de 0.80.

Dès qu'une détection est confirmée, une période de cooldown d'une seconde empêche toute nouvelle prédiction, ce qui évite les déclenchements multiples et rend le comportement plus stable.



Cette logique glissante reflète bien mieux le fonctionnement attendu sur le robot. Elle permet de tester le modèle dans un environnement quasi temps réel, et prépare directement son intégration dans le firmware de l'ESP32, où cette même mécanique de fenêtres chevauchantes, de filtrage par amplitude et de stabilisation par moyennage sera appliquée à l'identique.

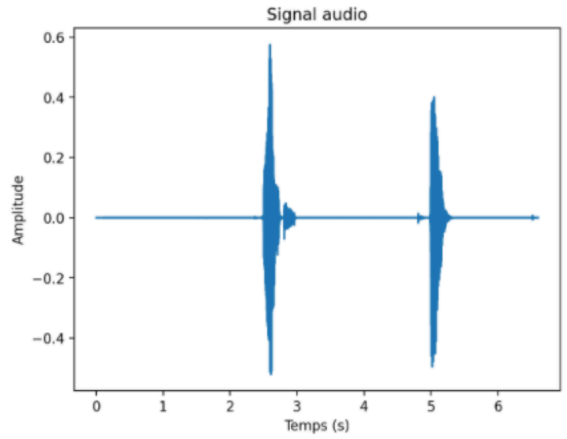
Par exemple, dans l'enregistrement suivant, deux commandes ont été prononcées — "right" puis "on" — et le système les a correctement identifiées en appliquant cette logique.

Résultats de la logique de streaming simulé

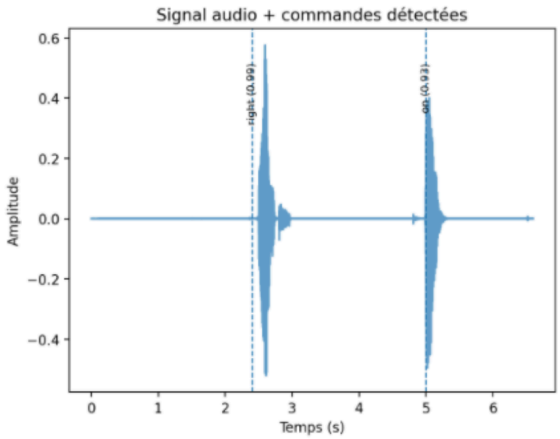
✅ Commande **RIGHT** détectée vers t ≈ 2.40 s (confiance moyenne ≈ 0.99, 4 fenêtres en accord).

✅ Commande **ON** détectée vers t ≈ 5.00 s (confiance moyenne ≈ 0.93, 5 fenêtres en accord).

Signal audio



Signal + commandes détectées



6.8. Déploiement sur le robot et Optimisations Mémoire

Le passage de la théorie (Python/Streamlit) à la pratique (Code `C` sur le firmware du robot) nous confronte au **"Mur de la Mémoire"**. Même sur un microcontrôleur performant comme l'ESP32-S3, la mémoire vive (RAM) reste une ressource critique et fragmentée. L'allocation naïve des tenseurs, telle que pratiquée implicitement dans les frameworks classiques comme PyTorch, conduit inévitablement à des **latences élevées**, voire à des **erreurs d'allocation** (`malloc failed`) lors du traitement de flux audio en temps réel.

Pour garantir une inférence fluide et robuste sur le robot, nous avons dû adapter le moteur d'inférence en nous concentrant sur trois optimisations majeures.

La gestion stricte du cycle de vie des tenseurs

En Python, le **Garbage Collector** gère la mémoire automatiquement. En `C` embarqué, chaque tenseur intermédiaire créé par une couche (par exemple la sortie d'une convolution) reste en mémoire indéfiniment s'il n'est pas explicitement libéré. Dans une architecture séquentielle comme TinySpeech, la sortie du "Bloc 1" devient l'entrée du "Bloc 2". Une fois le "Bloc 2" calculé, les données du "Bloc 1" ne sont plus utiles. Nous avons donc implémenté une gestion manuelle stricte via la fonction `free_tensor()`, appelée immédiatement après la consommation d'une donnée. Cela permet de maintenir l'empreinte mémoire (Peak RAM) à un niveau constant et bas, plutôt que de la voir s'accumuler linéairement au fur et à mesure que l'on avance dans les couches du réseau.

La fusion d'opérateurs (Operator Fusion)

C'est l'optimisation la plus impactante pour la performance. Dans une implémentation standard, une couche de BatchNormalization suivie d'une quantification effectue les étapes suivantes :

- Calcul du Batchnorm (*float*) **puis** Stockage en RAM (création d'un gros tampon temporaire).
- Division par l'échelle et arrondi **puis** Stockage en *int8* (tampon final).
- Libération du *float*.

Ce pic temporaire de mémoire est inutile et coûteux en temps d'écriture. Nous avons développé une version "fusionnée" de la couche (batchnorm2d optimisée) qui effectue le calcul flottant dans les registres du processeur et écrit directement le résultat quantifié (Int8) en mémoire. Le tampon intermédiaire n'est jamais alloué, ce qui divise par 4 la consommation mémoire instantanée de cette couche et accélère le traitement.

Attention Condenser "Zero-Copy"

Le mécanisme d'attention utilise une fonction **Sigmoïde** qui, par nature, opère sur des nombres flottants. Pour les tenseurs de masque d'attention, une conversion temporaire en float32 réclamerait une large plage de mémoire contiguë, difficile à trouver sur un système embarqué qui tourne depuis longtemps. Nous avons implémenté une fonction `apply_fused_sigmoid_attention` qui applique la sigmoïde, la multiplication par l'échelle et la pondération du résiduel en une seule passe, pixel par pixel. Cette approche "in-place" (sur place) ne nécessite aucune allocation mémoire supplémentaire, garantissant que le modèle ne plantera pas, quelle que soit la charge du système.

6.9. Résultats sur cible

Grâce à ces optimisations logicielles, couplées à la puissance de l'ESP32-S3, le modèle complet à 4 blocs s'exécute avec une latence minimale. L'inférence produit un vecteur de probabilités sur 10 classes strictement identique à celui obtenu en Python, confirmant que la quantification *int8* et les optimisations mathématiques n'ont pas altéré la précision du réseau. Le robot est ainsi capable d'analyser son environnement sonore en continu sans saturation mémoire.

À suivre ...

References

[1] X. Qian and D. Klabjan, “A Probabilistic Approach to Neural Network Pruning.” [Online]. Available: https://arxiv.org/abs/2105.10065

[2] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning Filters for Efficient ConvNets.” [Online]. Available: https://arxiv.org/abs/1608.08710

[3] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network.” [Online]. Available: https://arxiv.org/abs/1503.02531

[4] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, “FitNets: Hints for Thin Deep Nets.” [Online]. Available: https://arxiv.org/abs/1412.6550

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” [Online]. Available: https://arxiv.org/abs/1801.04381

[6] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” [Online]. Available: https://arxiv.org/abs/1704.04861

[7] A. Wong, M. Famouri, M. Pavlova, and S. Surana, “TinySpeech: Attention Condensers for Deep Speech Recognition Neural Networks on Edge Devices.” [Online]. Available: https://arxiv.org/abs/2008.04245

[8] P. Warden, “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition.” [Online]. Available: https://arxiv.org/abs/1804.03209

[9] T. N. Sainath and C. Parada, “Convolutional neural networks for small-footprint keyword spotting,” in *Interspeech 2015*, 2015, pp. 1478–1482. doi: 10.21437/Interspeech.2015-352.

[10] B. Yegnanarayana, Ed., *Interspeech 2018, 19th Annual Conference of the International Speech Communication Association, Hyderabad, India, 2-6 September 2018*. ISCA, 2018. doi: 10.21437/Interspeech.2018.

[11] R. Tang and J. Lin, “Deep Residual Learning for Small-Footprint Keyword Spotting.” [Online]. Available: https://arxiv.org/abs/1710.10361

[12] R. D. A. Anderson J. Su and D. Gregg, “Performance-oriented neural architecture search.” [Online]. Available: https://arxiv.org/pdf/2001.02976

[13] A. R. Ravikiran, “QP-TinySpeech: Extremely Low-Bit Quantized + Pruned TinySpeech-Z for low-power MCUsw.” 2024.