ESP32forth et l'Intelligence Artificielle

Marc PETREMANN

Version 1.0 - 22/02/26

Table des matières

Préambule

Ce manuel est destiné à la prise en main de fonctions IA avec ESP32forth et une carte ESP32-S3.

ESP32-S3 : Le Microcontrôleur taillé pour l'Edge AI

L'ESP32-S3 n'est pas une simple mise à jour ; c'est le premier SoC d'Espressif à intégrer des **instructions vectorielles** (extensions SIMD). Ces instructions permettent d'accélérer massivement les opérations mathématiques de base des réseaux de neurones (multiplications-accumulations), propulsant les performances d'inférence bien au-delà de ses prédécesseurs.

Le code d'origine ESP-NN se trouve ici : https://github.com/espressif/esp-nn

🛠️ Atouts Matériels Clés

- **Accélération IA :** Instructions dédiées pour le calcul de tenseurs et le traitement de signal.
- **Mémoire Flexible :** Supporte jusqu'à **16 Mo de PSRAM** externe (indispensable pour charger des modèles de vision ou d'audio).
- **Architecture :** Dual-core Xtensa® LX7 à 240 MHz, permettant de dédier un cœur à l'acquisition de données et l'autre à l'inférence.

📚 Références des Librairies Essentielles

Pour transformer ce silicium en intelligence, vous devrez vous appuyer sur l'un de ces trois piliers logiciels, selon votre niveau de contrôle souhaité.

ESP-DL (Espressif Deep Learning)

C'est la bibliothèque **native et optimisée** par le fabricant. Elle est conçue pour tirer le maximum des instructions vectorielles du S3.

- **Usage :** Inférence de modèles haute performance (détection de visages, reconnaissance d'objets).
- **Lien :** espressif/esp-dl
- **Le "Plus" :** Contient un "Model Zoo" avec des modèles déjà optimisés (Face Detection, Gesture Recognition).

ESP-NN

C'est la couche de bas niveau (back-end) utilisée par ESP-DL et TensorFlow Lite Micro pour l'ESP32-S3.

- **Usage :** Si vous développez votre propre moteur d'inférence et avez besoin de fonctions de convolution ou d'activation ultra-rapides.
- **Lien :** espressif/esp-nn

TensorFlow Lite for Microcontrollers (TFLM)

Le standard de Google, adapté à l'écosystème Espressif via un composant spécifique.

- **Usage :** Portabilité maximale. Si vous entraînez vos modèles sur TensorFlow/Keras, c'est la voie la plus simple pour le déploiement.
- **Lien :** espressif/tflite-micro-esp-examples

ESP-Skainet (Reconnaissance Vocale)

Framework dédié aux interfaces vocales hors-ligne.

- **Usage :** Détection de mots clés ("Wake Word") et commandes vocales (jusqu'à 200 commandes personnalisables sans réentraînement).
- **Lien :** espressif/esp-skainet

🚀 Domaines d'Applications Prometteurs

Grâce à ces outils, l'ESP32-S3 excelle dans trois domaines de l'IA embarquée (TinyML) :

Domaine	Exemple concret	Librairie conseillée
Vision	Compteur de personnes, lecture de compteurs.	**ESP-WHO** (basée sur ESP-DL)
Audio	Assistant domotique local, détection de bruits.	**ESP-Skainet**
Sensation	Maintenance prédictive (vibrations), analyse ECG.	**Edge Impulse** (outil No-Code compatible)

Installation de la librairie ESP-NN dans ARDUINO IDE 2.x

Dans l'IDE Arduino, il n'existe pas de bibliothèque nommée "ESP-NN" que vous pouvez trouver via le *Library Manager* (Gestionnaire de bibliothèques).

La raison est simple : **ESP-NN est déjà intégrée nativement** dans le "core" ESP32 d'Espressif pour Arduino.

Pour l'utiliser dans votre code (et donc dans les macros `userwords.h`), vous n'avez rien à installer de plus.

Le calcul tensoriel

Pour comprendre les tenseurs, il suffit d'arrêter de les voir comme des objets mathématiques abstraits et de les voir comme des **conteneurs de données**.

L'idée de base est simple : c'est une question de **dimensions** (ou d'étages).

Imaginons comment organiser des informations. On va monter en complexité par étapes.

Les rangs de tenseurs

Le scalaire : c'est juste **un nombre unique**.

- *Exemple :* la température (37°C). C'est un point isolé, il n'y a pas de direction, juste une valeur.

Le vecteur : c'est une **liste de nombres**.

- *Exemple :* les coordonnées GPS (Latitude, Longitude). Ici, l'ordre compte et les nombres ensemble décrivent quelque chose de plus précis qu'un seul chiffre. C'est une ligne de données.

La matrice : c'est un **tableau de nombres** (des lignes et des colonnes).

- *Exemple :* Une photo en noir et blanc. Chaque case du tableau correspond à un pixel, et le chiffre dedans dit si c'est gris, noir ou blanc. C'est une grille de données.

Le vrai tenseur. C'est là que ça devient intéressant. Imaginons un **cube de nombres**, ou une pile de tableaux.

- *Exemple :* Une photo en couleur. Pour chaque pixel, on a trois couches : une pour le Rouge, une pour le Vert, une pour le Bleu (RVB). On empile donc trois matrices les unes sur les autres.

Si on entend parler de "Tenseurs", c'est souvent à cause de l'**Intelligence Artificielle** (comme avec la bibliothèque *TensorFlow*).

Pourquoi ? Parce que pour qu'une IA comprenne le monde, elle a besoin de manipuler des données énormes et complexes.

- Une vidéo est un tenseur de rang 4 : une pile d'images (3D) qui défilent dans le temps (la 4ème dimension).

ESP-NN avec ESP32forth

nn.add_elementwise_s8

Ce mot nécessite 15 paramètres.

C'est le couteau suisse de l'addition dans le monde de l'IA embarquée. Il ne se contente pas d'additionner deux nombres. Il effectue une **normalisation en temps réel** pour que le résultat tienne toujours dans un octet signé (`int8_t`).

Voici le détail de chaque paramètre selon la signature du SDK ESP-NN :

Les Pointeurs de Données

- `input1_data` **&** `input2_data` : Les adresses des deux tableaux (vecteurs) à additionner. Ils doivent être alignés sur 16 octets pour profiter de l'accélération S3.
- `output` : L'adresse où le résultat sera stocké.

La Quantification d'Entrée (Offsets)

- `input1_offset` **&** `input2_offset` : En IA, le "zéro" réel n'est pas toujours le `0` binaire (c'est le principe du *Zero Point*). Ces offsets sont ajoutés à chaque élément avant l'opération : (x+offset). Si vos données sont déjà centrées sur 0, on met `0`.

La Mise à l'Échelle (Multipliers & Shifts)

C'est ici que la magie opère pour éviter de perdre en précision.

- `input1_mult` **&** `input2_mult` : Des multiplicateurs entiers.
- `input1_shift` **&** `input2_shift` : Des décalages vers la droite (divisions par 2n).
- `left_shift` : Un décalage global vers la gauche appliqué au début pour augmenter la précision intermédiaire des calculs.

La Quantification de Sortie

- `out_offset` : La valeur ajoutée au résultat final pour le "re-shifter" vers une plage spécifique.
- `out_mult` **&** `out_shift` : Utilisés pour redimensionner le résultat de l'addition (qui peut dépasser 8 bits) afin qu'il rentre à nouveau dans un `int8_t`.

La Sécurité (Clamping)

- `activation_min` **&** `activation_max` : Définissent les bornes du résultat.
    - Pour une addition standard, on utilise `-128` et `127`.

- Si vous voulez un **ReLU** (ne garder que le positif), vous pouvez mettre `0` pour `activation_min`. Tout résultat négatif sera alors transformé en `0` instantanément sans coût CPU supplémentaire.

La Taille

- `size` : Le nombre total d'éléments (octets) à traiter.

Pour chaque élément i, la fonction calcule approximativement :

$$out[i]=clamp([2^{shift1}(in1[i]+off1)\cdot mult1+2^{shift2}(in2[i]+off2)\cdot mult2]\cdot scaleout+offout,min,max)$$

Dans un réseau de neurones, les couches n'ont pas toutes la même "échelle" de valeurs. Si vous additionnez une couche qui varie entre -1 et 1 avec une couche qui varie entre -10 et 10, vous devez les réaligner (via les `mult` et `shift`) pour que l'addition ait un sens mathématique, tout en restant en nombres entiers pour la vitesse.

Exemple :

```
0 value input1
    align here to input1
    10 c, 20 c, 30 c,

0 value input2
    align here to input2
    5 c, 5 c, 5 c,

0 value output
    align here to output
    0 c, 0 c, 0 c,

input1 input2
0 0                     \ n13, n12 : offsets
1073741824 1073741824   \ n11, n10 : mult au max
0 0 1                   \ n9, n8, n7 : left_shift = 1
output
0                       \ n5 : out_offset
2147483647              \ n4 : out_mult au max
0                       \ n3 : out_shift
-128 127                \ n2, n1 : bornes
3                       \ n0 : size
```

Ceci correspond aux 16 paramètres nécessaires pour exécuter l'addition des données `input1` et `input2` :

```
nn.add_elementwise_s8

output 0 nn.get_element_s8 .    \ affiche 15
output 1 nn.get_element_s8 .    \ affiche 25
output 2 nn.get_element_s8 .    \ affiche 35
```

On a bien une somme octet par octet des données de `input1` et `input2`.

ATTENTION : pour obtenir ce résultat, on a un peu triché sur les paramètres pour arriver à ce résultat. Dans les faits, le mot `nn.add_elementwise_s8` effectue en réalité une somme pondérée pour éviter le débordement sur 8 bits dans l'intervalle -128..127.

nn.mul_elementwise_s8

Le mot `nn.mul_elementwise_s8` est une primitive de haut niveau qui exploite les instructions accélérées de l'ESP32 pour effectuer des calculs de réseaux de neurones. Contrairement à une multiplication classique, elle gère la **quantification**, c'est-à-dire le passage d'un monde mathématique complexe (flottants) à un monde matériel rapide (entiers 8 bits).

Voici le rôle de chacun de ses 11 paramètres pour que tu puisses les maîtriser :

Les Entrées et Sorties (Pointeurs)

- `input1_data` **(n10) &** `input2_data` **(n9)** : Les adresses mémoires de tes deux vecteurs sources.
- `output_pointer` **(n6)** : L'adresse où le résultat sera écrit.

La Logique de Quantification (Le "Scaling")

C'est ici que se joue la précision du calcul. Pour transformer une multiplication de grands nombres en un résultat qui tient dans un seul octet (`s8`), la fonction utilise cette formule :

$$Res = clamp([(In1 + Off1) \times (In2 + Off2) \times Mult] >> (31 + Shift) + OffOut)$$

- `input_offset` **(n8, n7)** : Ajoute une valeur à tes entrées avant de multiplier. Très utile si tes données `s8` sont asymétriques.
- `out_offset` **(n5)** : Ajoute une valeur au résultat final. C'est ce que nous avons utilisé pour corriger ton "49" en "50".
- `out_mult` **(n4) &** `out_shift` **(n3)** : Travaillent ensemble. Le multiplicateur est en **Q31** (un nombre où 231 représente 1.0). Le shift décale ensuite les bits vers la droite pour réduire la valeur.

La Protection (Clamping)

- `activation_min` **(n2) &** `activation_max` **(n1)** : Définissent les barrières de sécurité. Si le résultat dépasse `max`, il est forcé à `max`. Cela simule des fonctions d'activation comme **ReLU**.
- `size` **(n0)** : Le nombre d'éléments à traiter en une seule fois.

Sur un ESP32, ce mot utilise des instructions **SIMD** (Single Instruction, Multiple Data). Au lieu de multiplier les nombres un par un, le processeur peut en traiter plusieurs

simultanément dans ses registres, ce qui est crucial pour faire de l'intelligence artificielle en temps réel.

Exemple :

```
3 value dataSize

0 value input1
    align here to input1
    10 c, 20 c, 30 c,

0 value input2
    align here to input2
    5 c, 4 c, 3 c,

0 value output
    align here to output
    0 c, 0 c, 0 c,

input1          \ n10 : Adresse source 1
input2          \ n9  : Adresse source 2
0               \ n8  : Offset entrée 1 (souvent 0)
0               \ n7  : Offset entrée 2 (souvent 0)
output          \ n6  : Adresse de destination
0               \ n5  : Offset de sortie (souvent 0)
2147483647      \ n4  : out_mult (Facteur d'échelle en Q31, ici 0.5 pour
l'exemple)
0               \ n3  : out_shift (Décalage binaire)
-128            \ n2  : Activation MIN (Clamping bas s8)
127             \ n1  : Activation MAX (Clamping haut s8)
dataSize        \ n0  : Nombre d'éléments (SIZE)
nn.mul_elementwise_s8
```

Et voici le résultat de la multiplication matricielle :

```
output 0 nn.get_element_s8 .    \ affiche 49
output 1 nn.get_element_s8 .    \ affiche 79
output 2 nn.get_element_s8 .    \ affiche 89
```

Les données restituées ne sont pas celles attendues qui devraient être respectivement **50**, **80** et **90**.

Ceci est le résultat des mécanismes de quantification et d’arrondis.

nn.get_element_s8 (addr index -- n)

Récupère une valeur 8 bits dans une zone de données pointée par addr et augmentée de la valeur de index. Pour une zone contenant n données 8 bits, la valeur de index est valide dans l’intervalle 0..n-1.

nn.set_element_s8 (addr index size value -- n)

Ici, on stocke une valeur huit bits à l'adresse **`addr+index`**. La valeur size est la taille maximale de la zone **`addr`**. Définir la taille de la zone cible permet d'assurer une sécurité en évitant de déborder au-delà de la taille maximale de la zone de données. Exemple :

```
input1 0 3 24 nn.set_element_s8
```

Index lexical