ID: 20A179FS

Field: Ingénerie Informatique

City: Université de Ngaoundéré

Professor: Prof. Dr. Ing. FENDJI JEAN LOUIS KEDIENG EBONGUE

TinyML for Bean Cooking Time Prediction

MBAIGOLMEM DANG-DANG THIERY [1]

4 septembre 2025

1. dev.thiery.dangdang@gmail.com

Table des matières

Table des figures

Liste des tableaux

1 Introduction générale

1.1 Contexte et justification

L'optimisation des procédés de transformation agroalimentaire constitue un enjeu stratégique à l'échelle mondiale. Elle conditionne à la fois la qualité et la sécurité des produits, la réduction des coûts de production et la lutte contre le gaspillage alimentaire, dans un contexte marqué par la croissance démographique et les pressions sur les ressources naturelles. Parmi ces procédés, la cuisson des légumineuses — et particulièrement celle des haricots — occupe une place centrale. Consommés dans de nombreuses régions du monde, les haricots représentent une source essentielle de protéines, de fibres et de micronutriments, contribuant à la sécurité nutritionnelle et à la souveraineté alimentaire de plusieurs pays [1]. Le temps de cuisson détermine de manière décisive la texture, la saveur et la valeur nutritionnelle des produits, tout en influençant leur acceptabilité par les consommateurs [2].

Traditionnellement, ce temps est estimé à partir de méthodes empiriques ou destructives, telles que les tests de cuisson ou la mesure de l'absorption d'eau. Bien que fiables, ces techniques présentent d'importants inconvénients : elles sont chronophages, coûteuses et difficilement applicables à un contrôle en ligne. Elles génèrent par ailleurs une dépense énergétique non négligeable, problématique dans un contexte où la maîtrise des consommations énergétiques est devenue prioritaire. Ces limites sont particulièrement contraignantes pour les petites et moyennes unités de production agroalimentaire, souvent dépourvues d'infrastructures lourdes ou d'équipements sophistiqués.

L'émergence de l'intelligence artificielle (IA) et des technologies embarquées ouvre de nouvelles perspectives pour surmonter ces obstacles. Le *Tiny Machine Learning* (TinyML) constitue en particulier une innovation de rupture : il permet de déployer des modèles d'apprentissage automatique directement sur des dispositifs embarqués à faible consommation énergétique et à ressources limitées. Grâce à des architectures légères telles que *MobileNetV2* ou des réseaux de neurones convolutionnels compacts, il devient possible de traiter des données visuelles localement, sur microcontrôleurs ou smartphones, sans recourir à une infrastructure informatique lourde ni à une connexion internet permanente [3]. Cette approche rend l'IA accessible dans des environnements contraints, et s'avère particulièrement pertinente pour les zones rurales et les petites unités de transformation, qui constituent une part importante de la chaîne de valeur agroalimentaire.

Dans ce contexte, l'intégration de la vision par ordinateur et du TinyML dans l'agri-

culture et l'agroalimentaire a déjà montré son efficacité pour l'évaluation de la maturité des fruits, la détection précoce de maladies ou le suivi de la qualité de denrées périssables [4]. L'application de ces avancées à l'estimation du temps de cuisson des haricots représente un enjeu à la fois scientifique et socio-économique : elle permettrait non seulement d'améliorer la maîtrise des procédés industriels, mais aussi de renforcer l'autonomie technologique des producteurs, de réduire les coûts énergétiques et de proposer aux consommateurs des produits de qualité constante. Ce travail s'inscrit ainsi dans une dynamique d'innovation technologique au service du développement durable et de la sécurité alimentaire.

1.2 Problématique

Malgré les progrès réalisés, la prédiction précise et non destructive du temps de cuisson des légumineuses reste un défi technique et scientifique. Les méthodes conventionnelles souffrent de plusieurs limites majeures :

— **Temps et coût** : leur mise en œuvre est longue et inadaptée à un contrôle en ligne ou à une production de grande échelle.

— **Manque de portabilité** : les dispositifs existants nécessitent des équipements encombrants et onéreux, peu accessibles aux petites structures.

— **Contraintes matérielles et énergétiques** : l'absence de solutions légères freine l'adoption de technologies avancées dans les zones rurales et au sein des petites unités de production.

Dès lors, la question centrale qui guide ce mémoire est la suivante :

> *Comment concevoir et déployer un système de prédiction du temps de cuisson des haricots, fondé sur l'analyse d'images, qui soit à la fois précis, portable et compatible avec les contraintes matérielles et énergétiques du TinyML ?*

1.3 Objectifs et contributions

1.3.1 Objectif général

Concevoir, développer et valider un modèle TinyML capable de prédire le temps de cuisson des haricots à partir d'images, de façon non destructive, fiable et en quasi temps réel.

1.3.2 Objectifs spécifiques

1. Constituer et prétraiter un jeu de données d'images de haricots annotées avec leur temps de cuisson.

2. Concevoir, entraîner et optimiser des modèles légers adaptés au déploiement embarqué (CNN compacts, MobileNetV2, EfficientNet-lite, etc.).
3. Évaluer les performances des modèles en termes de précision, de consommation mémoire et de temps d'inférence.
4. Intégrer et tester le modèle final sur une plateforme embarquée (smartphone).

1.3.3 Contributions attendues

— Une méthodologie reproductible et généralisable pour la prédiction visuelle du temps de cuisson des légumineuses.

— Un modèle optimisé, validé expérimentalement et conforme aux contraintes du TinyML.

— Une démonstration fonctionnelle sur dispositif embarqué, attestant de la faisabilité et de l'impact potentiel de l'approche dans des contextes réels.

1.4 Organisation du mémoire

La structure du mémoire a été pensée pour guider progressivement le lecteur, du cadre conceptuel aux contributions pratiques et scientifiques. Elle se décline comme suit :

— **Abstract** : une présentation synthétique du sujet, de la méthodologie, des résultats principaux et des apports du travail.

— **Chapitre 1 — Introduction générale** : mise en contexte scientifique et socio-économique, formulation de la problématique, présentation des objectifs et contributions attendues.

— **Chapitre 2 — Revue de la littérature** : analyse critique des travaux existants liés à la prédiction du temps de cuisson des aliments, au TinyML et aux applications de la vision par ordinateur dans l'agroalimentaire.

— **Chapitre 3 — Méthodologie et conception** : description de la démarche méthodologique, du protocole expérimental et de la conception du système proposé.

— **Chapitre 4 — Implémentation et déploiement** : présentation des choix technologiques, de l'implémentation logicielle et du déploiement sur plateformes embarquées.

— **Chapitre 5 — Résultats et discussion** : analyse et interprétation des résultats, confrontation avec l'état de l'art et discussion des limites.

— **Chapitre 6 — Conclusion et perspectives** : synthèse des contributions majeures et identification de pistes de recherche futures.

2 Revue de la littérature

2.1 Introduction

La revue de littérature a pour objectif de positionner le présent travail dans le paysage scientifique actuel, en identifiant les contributions majeures, les lacunes existantes et les perspectives d'amélioration. Dans le cadre de ce mémoire, l'accent est mis sur deux axes complémentaires : (i) les recherches liées à la prédiction du temps de cuisson des légumineuses et travaux connexes dans l'alimentation, et (ii) l'émergence du *Tiny Machine Learning* (TinyML) comme solution au traitement embarqué de données avec des contraintes strictes en mémoire et en énergie. L'articulation de ces deux domaines ouvre la voie à des applications innovantes dans le secteur agroalimentaire, en particulier pour la cuisson et la qualité des aliments.

2.2 Intelligence artificielle et secteur agroalimentaire

L'intelligence artificielle (IA) a profondément transformé le secteur agroalimentaire en introduisant des outils pour l'automatisation, la classification et l'optimisation des procédés [5]. La vision par ordinateur, notamment, est devenue un outil incontournable pour analyser l'apparence des fruits et légumes, estimer leur maturité, évaluer leur qualité ou encore contrôler certains paramètres de cuisson [6].
Les applications sont multiples : détection de maladies végétales, suivi de la croissance, reconnaissance automatique des produits sur les chaînes de production, ou encore estimation de la fermeté des denrées après cuisson. Les récents progrès en IA embarquée démontrent que la combinaison de modèles légers et de microcontrôleurs permet de réaliser de la surveillance en temps réel, directement sur le terrain [7, 8]. Ainsi, l'agroalimentaire bénéficie à la fois des avancées de la vision par ordinateur et des développements de l'IA embarquée.

2.3 Prédiction du temps de cuisson des légumineuses et travaux connexes

La cuisson des légumineuses est un processus complexe, influencé par la variété, la taille, la teneur en eau et les conditions de stockage [9, 10]. Traditionnellement, la prédiction du temps de cuisson repose sur des méthodes expérimentales basées sur des mesures physico-chimiques (dureté, humidité, structure cellulaire) ou sur des techniques spectroscopiques telles que le proche infrarouge. Ces approches, bien que précises, restent coûteuses, lentes et difficilement transposables en conditions réelles. Des travaux plus récents exploitent la vision par ordinateur et les réseaux de neurones pour corréler les caractéristiques visuelles (forme, couleur, texture de surface) à la tendreté et au temps de cuisson des haricots [11]. Ces approches présentent l'avantage

d'être non destructives et automatisables, ce qui constitue un atout considérable pour une intégration dans des systèmes embarqués.
Les recherches connexes sur d'autres aliments confirment la faisabilité de cette approche. Par exemple, [12] ont étudié la texture du riz après cuisson, tandis que [13] se sont intéressés à la fermeté des pâtes. De même, des travaux sur les fruits et légumes montrent que l'IA peut prédire la maturité ou la tendreté à partir d'images [14]. Ces études suggèrent que l'apparence visuelle contient des informations suffisamment discriminantes pour estimer la texture et le degré de cuisson, ce qui renforce la pertinence de l'application aux légumineuses.
Néanmoins, malgré ces avancées, les études spécifiques aux haricots restent rares et fragmentaires. Les méthodes existantes sont soit limitées par la taille des échantillons, soit inadaptées au déploiement sur des dispositifs contraints. Cette lacune souligne la nécessité de développer des modèles optimisés pour le TinyML, capables de prédire en temps réel le temps de cuisson à partir d'images.

2.4 TinyML : principes, avantages, cas d'usage et limites

Le *Tiny Machine Learning* (TinyML) désigne l'exécution de modèles d'apprentissage automatique directement sur des dispositifs embarqués à ressources limitées (microcontrôleurs, capteurs intelligents) [15]. L'objectif est de réaliser des inférences en temps réel localement, avec une faible empreinte mémoire et énergétique, rendant l'IA accessible même dans des environnements à faible connectivité.

2.4.1 Avantages et cas d'utilisation

Les principaux avantages du TinyML sont :

- **Traitement local** : réduction de la latence et de la dépendance au cloud.
- **Efficacité énergétique** : consommation minimale adaptée aux dispositifs alimentés par batterie.
- **Confidentialité accrue** : les données ne quittent pas le dispositif, limitant les risques de fuite.
- **Accessibilité économique** : microcontrôleurs peu coûteux adaptés à une large diffusion.

Les applications sont variées :

- **Agriculture** : détection de maladies, estimation de la qualité des récoltes, suivi en temps réel [7, 8].
- **Agroalimentaire** : classification des produits, prédiction de la texture et du temps de cuisson.

— **Santé** : suivi des signaux physiologiques sur dispositifs portables.

— **IoT industriel** : capteurs intelligents embarqués dans des systèmes connectés.

2.4.2 Contraintes et limites

Malgré son potentiel, le TinyML se heurte à plusieurs limites :

— Mémoire et puissance de calcul limitées, restreignant la taille et la complexité des modèles.

— Nécessité de recourir à des optimisations pouvant induire une perte de précision.

— Disponibilité limitée de jeux de données spécialisés, notamment dans le domaine de la cuisson des légumineuses.

2.4.3 Techniques d'optimisation

Pour rendre l'exécution des modèles possible sur microcontrôleurs, plusieurs techniques sont utilisées :

— **Quantification** : conversion des poids du modèle en entiers (INT8) ou en flottants réduits (FP16) pour diminuer l'empreinte mémoire et accélérer l'inférence [16].

— **Pruning** : suppression des connexions et paramètres redondants afin de réduire la taille du réseau tout en préservant ses performances [17].

— **Distillation de connaissances** : entraînement d'un petit modèle (*student*) à partir d'un modèle plus complexe (*teacher*), pour conserver les performances tout en allégeant l'architecture.

Ces optimisations assurent un compromis entre efficacité computationnelle et précision, indispensable pour l'implémentation en contexte embarqué.

2.5 Modèles légers pour dispositifs contraints

La conception de modèles adaptés au TinyML repose sur des architectures légères et optimisées :

— **MobileNet** : basé sur les convolutions séparables en profondeur, permettant de réduire drastiquement le nombre de paramètres [18].

— **EfficientNet** : propose une mise à l'échelle équilibrée en profondeur, largeur et résolution pour maximiser l'efficacité [19].

— **NASNetMobile** : utilise la recherche automatique d'architectures (NAS) pour identifier les modèles optimaux pour mobiles [20].

Ces architectures, associées à la quantification et au pruning, permettent d'obtenir des modèles performants et économes en ressources, adaptés aux capteurs IoT agricoles ou culinaires [7].

2.6 Synthèse des travaux antérieurs

TABLE 2.1 – Synthèse des travaux pertinents pour la revue de littérature

Réf.	Année	Titre	Objectif	Méthodologie	Résultats	Limites	Pertinence
[5]	2018	Deep learning in agriculture	Survol IA agriculture	Revue	IA utile classification	Peu focus cuisson	Contexte agroalimentaire
[6]	2020	Food cooking estimation	Estimer temps cuisson via images	Analyse d'images	Corrélation détectée	Échantillon limité	Lien cuisson direct
[3]	2021	TinyML principles	Définir TinyML	Discussion	TinyML possible MCU	Manque implémentations	Cadre TinyML
[18]	2017	MobileNet CNN	CNN léger mobiles	Architecture	Bonne précision faible coût	Coûteux MCU	Base modèles légers
[19]	2019	EfficientNet scaling	Optimiser mise à l'échelle CNN	Architecture optimisée	Meilleure efficacité	Toujours lourd TinyML	Optimisation CNN
[20]	2018	NASNetMobile	Optimisation via NAS	Recherche auto	Architecture optimisée	Complexité NAS	Modèles mobiles
[16]	2018	Quantization NN	Réduire taille modèles	Quantification INT8/FP16	Réduction mémoire x4	Perte précision possible	Compression mémoire
[17]	2016	Pruning deep nets	Réduction paramètres	Pruning	Réduction paramètres x10	Perte précision si trop pruning	Réduction mémoire
[21]	2019	Grain classification	Classifier variétés grains	CNN grains	Bonne précision	Peu généralisable	Classification grains/haricots
[14]	2020	Fruit maturity estimation	Prédire maturité/ tendreté	CNN fruits/-légumes	Prédictions fiables	Dataset limité	Lien cuisson/texture
[22]	2021	Food defect detection	Détection défauts	Analyse images	Détection robuste	Applicabilité restreinte	Qualité alimentaire
[23]	2019	Texture prediction food	Corréler images et texture	Vision + spectro	Corrélations confirmées	Peu d'applications concrètes	Lien cuisson/texture
[7]	2025	TinyML micronutrient sensing	TinyML détection micronutriments	Revue PRISMA	Modèles hybrides performants	Reporting incomplet	Insight TinyML agri
[8]	2024	Domain-Adaptive TinyML	Détection maladies TinyML	2D-CNN + adaptation domaine	Performance correcte	Chute conditions réelles	Adaptabilité TinyML agri
[24]	2023	Survey TinyML	Taxonomie TinyML	Revue PRISMA	Mapping optimisation	Peu d'applis agricoles	Ressources méthodologiques
[25]	2025	TinyML on-device inference	Avantages TinyML embarqué	Revue appli	Gain latence et vie privée	Peu exemples agro	Justifier inférence locale
[26]	2025	From TinyML to TinyDL	Optimisation TinyDL	Comparatif architectures	Framework global	Peu applis cuisson	Inspiration optimisation modèles

2.7 Conclusion

La littérature montre que l'IA et la vision par ordinateur offrent un potentiel considérable pour la prédiction du temps de cuisson et la qualité des aliments. Toutefois, les approches existantes présentent des limites : échantillons restreints, complexité des modèles et absence d'adaptation aux dispositifs contraints. De plus, la majorité des travaux portent sur d'autres aliments que les légumineuses, laissant un champ de recherche encore peu exploré.

Le TinyML apparaît comme une réponse pertinente à ces limitations. Grâce à ses techniques d'optimisation et à ses architectures légères, il permet d'envisager des systèmes intelligents, embarqués et autonomes, adaptés à des contextes réels. Le présent mémoire s'inscrit dans cette perspective en proposant une approche originale : exploiter le TinyML pour prédire le temps de cuisson des haricots à partir d'images, comblant ainsi un vide identifié dans la littérature et ouvrant la voie à des applications pratiques dans l'agroalimentaire.

3 Méthodologie et Conception du Système

3.1 Introduction

La conception d'un système capable de prédire le temps de cuisson des haricots à partir d'images s'inscrit dans une démarche méthodologique rigoureuse alliant vision par ordinateur, apprentissage automatique ultra-léger (TinyML) et pratiques expérimentales éprouvées. En combinant l'analyse non destructrice d'images à l'intégration dans des dispositifs à ressources limitées, ce travail vise à proposer une solution à la fois scientifiquement solide et technologiquement réalisable.

Des études antérieures, utilisant l'imagerie hyperspectrale et la spectroscopie proche infrarouge (Vis/NIRS), ont démontré la capacité de modèles statistiques tels que la régression PLS à prédire avec une précision raisonnable le temps de cuisson de haricots trempés et non trempés (prédiction corrélée de $R_{\text{pred}} \approx 0,886$ avec erreur standard SEP $\approx 7,9$ min) [1]. Par ailleurs, l'adoption de systèmes TinyML dans des contextes industriels ou agroalimentaires a montré son potentiel pour des applications réactives, locales et éco-énergétiques [27, 28].

L'approche adoptée ici repose sur plusieurs étapes clés. D'abord, la collecte et le prétraitement d'un jeu de données d'images de haricots — incluant une variabilité de tailles, de textures et de conditions de cuisson — servent de fondation à la modélisation. Ensuite, nous évaluons des architectures adaptées aux contraintes embarquées, en partant d'un réseau convolutionnel léger jusqu'à des modèles préentraînés efficients tels que MobileNetV2. L'objectif est de réduire la lourdeur computationnelle tout en conservant une précision prédictive satisfaisante, à l'instar du modèle RecipeSnap, qui propose une réduction de plus de 70% de la mémoire sans altérer la performance [29].

L'entraînement des modèles est conduit en tenant compte des bonnes pratiques : optimisation via Adam, régularisation (dropout, early stopping) et mesures objectives (MAE, RMSE, R^2, etc.). Enfin, une conversion en TensorFlow Lite (avec quantification en float16, int8) est effectuée pour obtenir une empreinte mémoire réduite, tout en testant l'inférence sur smartphone android — les travaux fondateurs de TensorFlow Lite Micro apportent un soutien technique précieux à cette étape [30].

Cette méthodologie a pour ambition de proposer un pipeline complet, reproductible, allant de la capture d'images à la prédiction en temps réel sur smartphone android, afin de démontrer la faisabilité et l'efficacité de la vision par ordinateur embarquée dans le domaine agroalimentaire local.

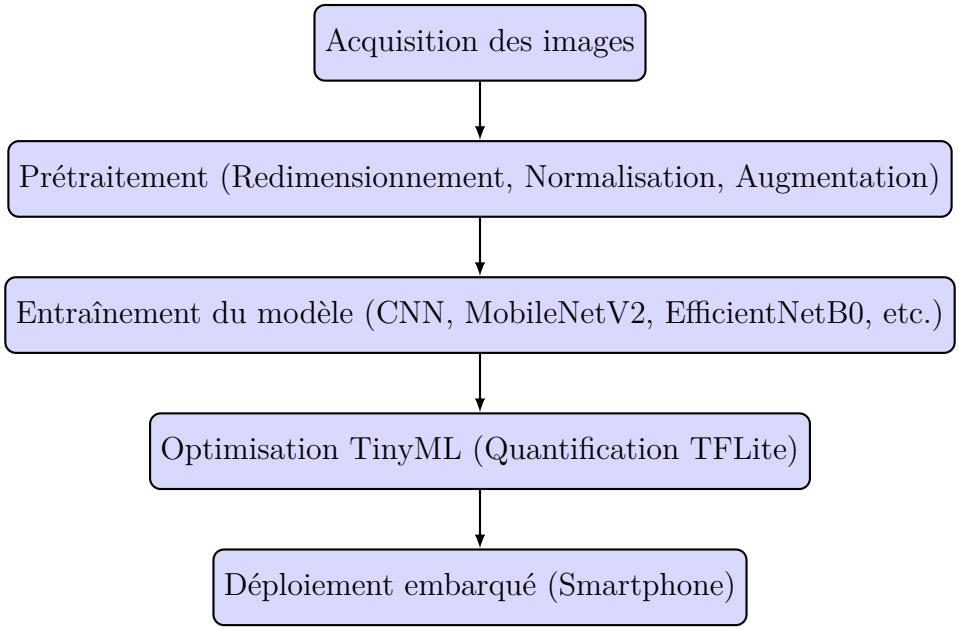

FIGURE 3.1 – Pipeline méthodologique proposé pour la prédiction du temps de cuisson des haricots basé sur TinyML.

3.2 Protocole expérimental — Prétraitement et préparation des jeux de données

Cette section détaille de façon exhaustive le protocole de prétraitement appliqué aux images et aux annotations de temps de cuisson (T_c), et précise la manière dont les jeux *train/validation/test* sont construits et stockés pour l'entraînement et le déploiement TinyML.

3.2.1 Description synthétique du dataset source

Le jeu de données original comprend **11 200** images couleur au format JPEG, réparties équitablement sur **8** variétés de haricots. Chaque variété est représentée par **1 400** images de haute résolution (3000×4000 pixels), répartie en **7** sous-variétés contenant **200** images chacune. Chaque image est associée à une annotation continue T_c (temps de cuisson en minutes), mesurée expérimentalement par chronométrage selon un protocole standardisé (arrêt au critère *fork-tender*), avec une incertitude de mesure liée à l'opérateur estimée à $\approx \pm 1$ minute.

L'exploration préliminaire de la distribution des T_c révèle une variabilité importante. Trois variétés présentent un pic de distribution autour de 120 minutes, tandis que d'autres couvrent une plage de cuisson très large (approximativement de 70 à 230 minutes). Deux variétés en particulier montrent des profils de T_c particulièrement étendus, ce qui souligne la nécessité d'une stratification rigoureuse lors du partitionnement des données pour garantir la représentativité de cette diversité dans chaque

sous-ensemble.

3.2.2 Analyse Exploratoire et Statistiques Descriptives

Afin de caractériser quantitativement la distribution des données et d'orienter les choix de modélisation, une analyse exploratoire a été conduite [31]. Cette analyse se concentre sur la distribution des temps de cuisson (T_c) à la fois globalement et au sein de chaque variété.

Statistiques descriptives par variété

Le tableau 3.1 résume les principaux descripteurs statistiques des temps de cuisson pour chaque variété.

TABLE 3.1 – Statistiques descriptives des temps de cuisson (T_c en minutes) par variété.

Variété	**Moyenne (μ)**	**Écart-type (σ)**	**Min**	**Max**	**Effectif**
Dor701	145.86	72.67	62	280	1400
Escapan021	159.00	77.47	65	287	1400
GPL190C	160.00	74.19	70	295	1400
GPL190S	144.57	66.65	58	270	1400
Macc55	237.43	103.99	88	410	1400
NIT4G16187	118.86	45.18	68	210	1400
Senegalais	155.71	78.38	70	300	1400
TY339612	148.14	77.85	51	286	1400

L'analyse de ces statistiques révèle plusieurs points cruciaux :

— **Forte hétérogénéité inter-variétés :** La moyenne des temps de cuisson varie considérablement, allant de 118.86 min (NIT4G16187) à 237.43 min (Macc55). Cette différence de plus de 100% confirme que la variété est un facteur prédictif majeur du temps de cuisson.

— **Dispersion variable :** L'écart-type (σ), qui mesure la dispersion des valeurs autour de la moyenne, est également très hétérogène. La variété `Macc55` présente la plus forte dispersion ($\sigma \approx 104$ min), indiquant que ses temps de cuisson sont très étalés. À l'opposé, `NIT4G16187` est la plus homogène ($\sigma \approx 45$ min). Cette information est vitale : le modèle d'apprentissage pourrait avoir plus de difficultés à prédire avec précision le T_c pour des variétés à forte variance.

— **Distribution globale :** La plage globale des temps de cuisson s'étend de 51 à 410 minutes, ce qui représente un défi de régression considérable pour un modèle TinyML contraint en ressources.

L'histogramme de la Figure 3.2 visualise la distribution globale des temps de cuisson, confirmant la présence de plusieurs modes et une asymétrie à droite.

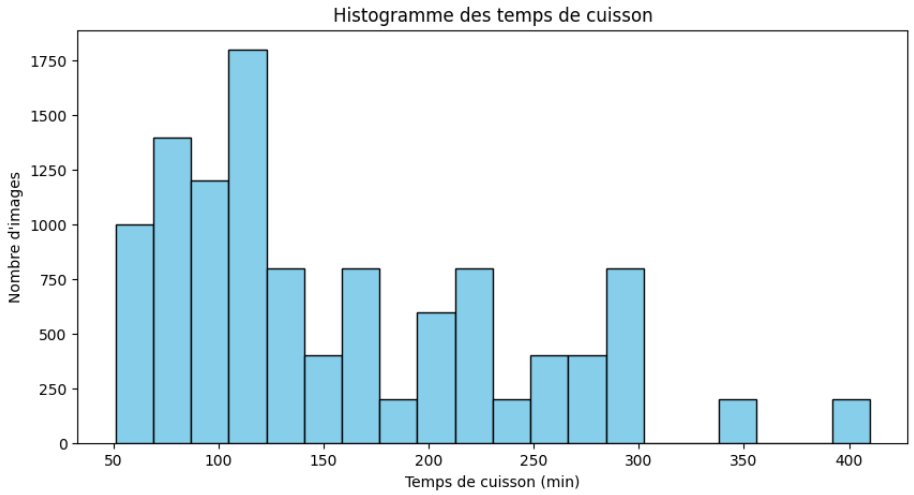

FIGURE 3.2 – Distribution globale des temps de cuisson (T_c) pour l'ensemble des variétés. La distribution est multimodale, avec un pic principal autour de 120-150 minutes.

Analyse de la variance intra-variété

Pour affiner l'analyse de la dispersion, la variance ($s^2 = \sigma^2$) a été calculée pour chaque variété (Tableau 3.2). La variance quantifie la dispersion quadratique moyenne et accentue les différences de variabilité.

TABLE 3.2 – Variance intra-variété des temps de cuisson.

Variété	**Variance Intra-variété (s^2)**
Dor701	5280.47
Escapan021	6002.00
GPL190C	5503.93
GPL190S	4442.28
Macc55	10813.97
NIT4G16187	2041.58
Senegalais	6143.16
TY339612	6060.17

La Figure 3.3 (diagramme en boîtes à moustaches) offre une représentation visuelle comparative de ces variances. Elle met en évidence la médiane, l'intervalle interquartile (IQR) et l'étendue des données pour chaque catégorie, facilitant l'identification des distributions symétriques, asymétriques et des valeurs potentiellement aberrantes.

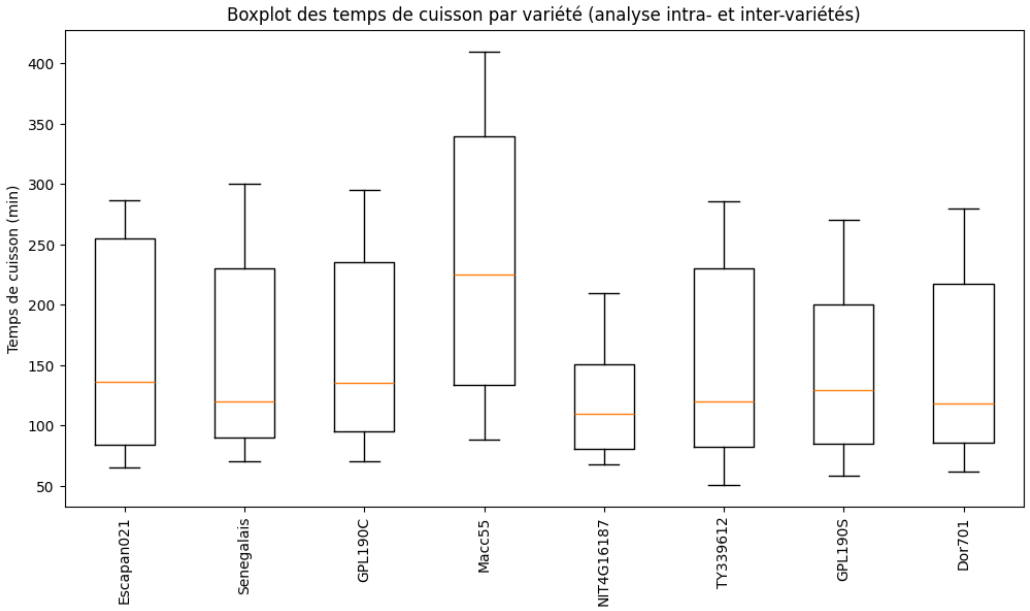

FIGURE 3.3 – Diagramme en boîtes à moustaches illustrant la distribution des temps de cuisson par variété. La variabilité extrême de la variété `Macc55` et l'homogénéité de `NIT4G16187` sont clairement visibles.

L'examen de la variance et du boxplot confirme que la variété `Macc55` est un cas d'étude particulièrement difficile en raison de sa dispersion interne massive. Des facteurs non contrôlés, tels qu'une plus grande hétérogénéité génétique ou des conditions de croissance/stockage variables au sein de cette variété, pourraient expliquer ce phénomène. Du point de vue de la modélisation, cela implique que les caractéristiques visuelles extraites des images de `Macc55` doivent être particulièrement discriminantes pour permettre une régression précise.

Équilibre et distribution des données

La performance et l'impartialité d'un modèle d'apprentissage profond dépendent fortement de l'équilibre du jeu de données. La Figure 3.4 montre la répartition des échantillons par variété.
Le jeu de données est **parfaitement équilibré**, avec 200 images par variété. Cet équilibre est un atout majeur car il prévient tout biais du modèle en faveur des classes sur-représentées. La performance évaluée sera donc une juste réflexion de la capacité du modèle à généraliser sur toutes les variétés.

3.2.3 Pipeline de prétraitement

Le prétraitement appliqué aux images et aux annotations suit le pipeline illustré par la Figure 3.5. Seules les opérations jusqu'à l'enregistrement des jeux HDF5 sont

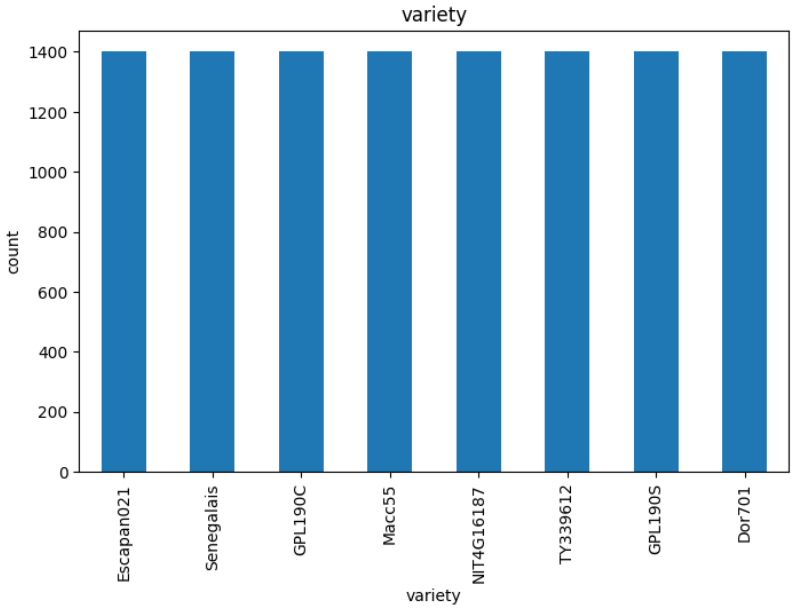

FIGURE 3.4 – Distribution du nombre d'images par variété. Le jeu de données est parfaitement équilibré, chaque classe contenant 200 échantillons.

incluses (préparation **offline** avant entraînement).

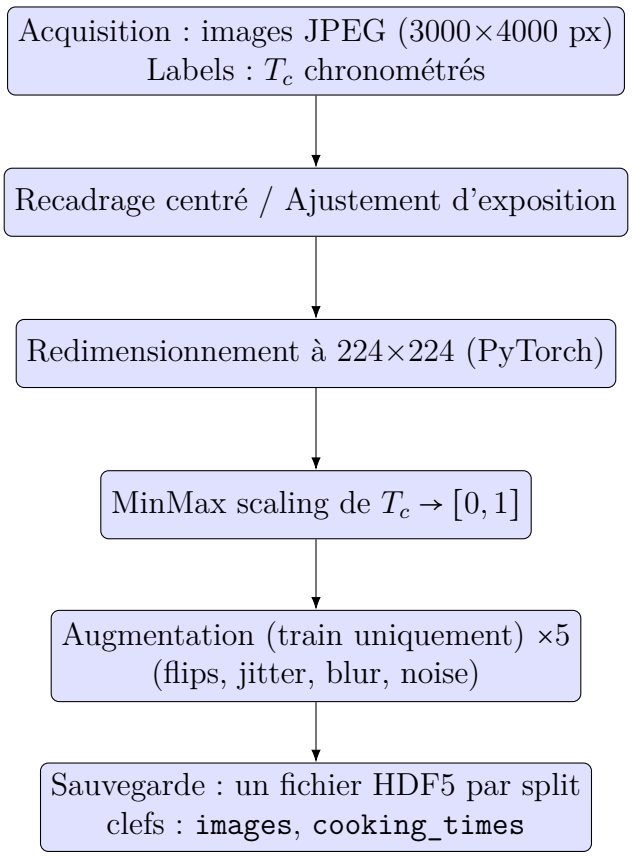

FIGURE 3.5 – Pipeline de prétraitement des images et des labels T_c.

3.2.4 Redimensionnement et outils

Le redimensionnement est réalisé via `torchvision.transforms` (PyTorch) en appliquant un `CenterCrop` suivi d'un `Resize(224)` puis d'une normalisation des canaux si nécessaire pour des modèles pré-entraînés. Le choix 224×224 découle d'un compromis entre fidélité des motifs et coût mémoire d'inférence (MobileNetV2 / EfficientNet-B0 compatible) [32, 33].

3.2.5 Augmentation (train uniquement) — paramètres

L'ensemble d'entraînement est étendu par un facteur **5** via des transformateurs aléatoires appliqués uniquement au split *train* :

- **Flips horizontal et vertical** aléatoires ($p = 0.5$),
- **RandomResizedCrop** (zoom et recadrage aléatoire),
- **ColorJitter** (variation de luminosité, contraste, saturation),
- **GaussianBlur** (flou gaussien léger),
- **Bruit Gaussien Additif**.

Aucune rotation n'est utilisée pour préserver les caractéristiques morphologiques directionnelles des grains. Ces transformations sont codées en PyTorch et appliquées stochastiquement pour générer quatre variantes supplémentaires par image d'entraînement initiale (1 image originale + 4 augmentées = facteur 5).

3.3 Normalisation des temps de cuisson (T_c)

Avant sauvegarde, les valeurs T_c (en minutes) sont mises à l'échelle par la transformation *Min–Max* pour produire une cible dans l'intervalle $[0, 1]$. Formellement, les paramètres de la transformation sont calculés **uniquement** sur l'ensemble d'entraînement $\mathcal{D}_{\text{train}}$:

$$T_{c,\min} = \min_{y \in \mathcal{D}_{\text{train}}} y, \qquad T_{c,\max} = \max_{y \in \mathcal{D}_{\text{train}}} y,$$

$$\tilde{y} = \frac{y - T_{c,\min}}{T_{c,\max} - T_{c,\min}} \quad \in [0, 1].$$

Les mêmes $T_{c,\min}$ et $T_{c,\max}$ sont ensuite utilisés pour normaliser les ensembles de validation et de test, évitant ainsi toute fuite d'information du futur vers le modèle. Les valeurs prédites $\tilde{y}$ sont restaurées dans l'échelle d'origine (en minutes) par la transformation inverse $y = \tilde{y}\,(T_{c,\max} - T_{c,\min}) + T_{c,\min}$ lors de l'interprétation finale des résultats.

3.4 Conception du système proposé

La problématique adressée consiste à estimer, à partir d'une image unique d'un haricot (ou d'un petit lot représenté sur l'image), le temps de cuisson T_c exprimé en minutes. Le système doit être d'abord compétitif en termes de précision prédictive, puis contrainte à des ressources embarquées (smartphone et/ou microcontrôleur) via des techniques TinyML. La présente section décrit l'architecture générale et le pipeline de traitement, les modèles testés (deux CNN personnalisés et trois modèles préentraînés adaptés au mobile), les optimisations TinyML appliquées, ainsi que le workflow et l'architecture logicielle menant au déploiement.

3.4.1 Motivation et choix d'approche

L'approche par vision (images) est motivée par la possibilité d'extraire des indices visuels corrélés au temps de cuisson — couleur, taille, texture, proportion d'imperfections, etc. — sans recourir à des capteurs supplémentaires. Le passage à TinyML est justifié par l'objectif d'un service embarqué (hors-connexion) : confidentialité, latence faible et coût énergétique réduit. Les techniques d'optimisation (quantification, pruning, distillation) sont des bonnes pratiques largement documentées pour adapter des réseaux convolutifs classiques à des plateformes contraignantes [34, 17].

3.4.2 Architecture générale du modèle et pipeline

Le pipeline se décompose en quatre grandes phases (Figure 3.6) :

1. **Prétraitement & chargement** : lecture des HDF5, batch sampling, conversion en float32, normalisation canaux (mean/std) compatible avec les poids pré-entraînés ; augmentation appliquée uniquement au jeu d'entraînement.
2. **Entraînement / Fine-tuning** : entraînement des CNN personnalisés depuis zéro et fine-tuning des modèles pré-entraînés (MobileNetV2, EfficientNet-B0, NASNetMobile) sur la tâche de régression continue (prévision du T_c normalisé).
3. **Optimisation TinyML** : quantification (post-training et/ou quantization-aware training), pruning graduel, éventuellement distillation depuis un « teacher » plus large vers un « student » compact.
4. **Déploiement & mesures** : conversion en TFLite, mesures de latence et consommation sur smartphone cible (mesures empiriques, idéalement sur plusieurs appareils).

La fonction apprise s'écrit :

$$\hat{T}_c = f_\theta\big(\mathcal{A}(I)\big)$$

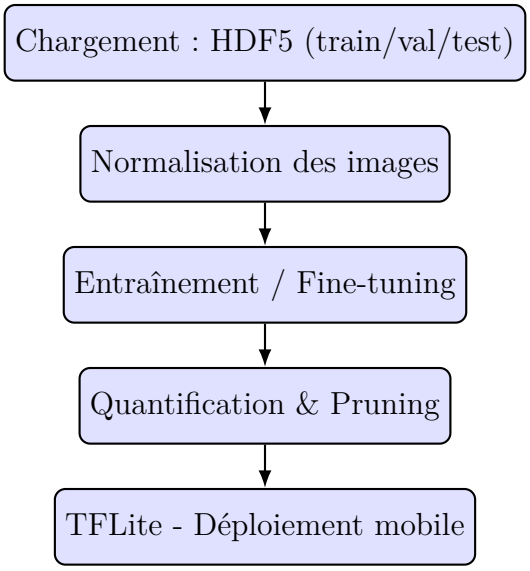

FIGURE 3.6 – Pipeline fonctionnel du système (chargement → entraînement → optimisation → déploiement).

où $I \in \mathbb{R}^{H \times W \times 3}$ est l'image d'entrée, $\mathcal{A}$ l'opérateur de prétraitement (crop, resize à 224×224, normalisation), et θ les paramètres entraînés.

3.4.3 Description détaillée des modèles testés

Nous évaluons deux architectures convolutionnelles construites ad hoc (CNN-1 et CNN-2) et trois modèles mobiles pré-entraînés.

Formules de complexité et paramètres

Pour une couche convolutionnelle standard (sans séparabilité), le nombre de paramètres P et le nombre d'opérations élémentaires en multiplications-additions (FLOPs approximés par *mult-adds*) pour une seule couche sont :

$$P = (K \times K \times C_{in} + 1) \times C_{out},$$

$$\text{FLOPs}_{\text{conv}} \approx 2 \times K \times K \times C_{in} \times C_{out} \times H_{out} \times W_{out},$$

où K est la taille du noyau, C_{in}, C_{out} canaux d'entrée/sortie et $H_{out} \times W_{out}$ la résolution de la carte de caractéristiques en sortie (le facteur 2 approximant multiplication + addition). Pour les couches depthwise separable (utilisées par MobileNet), la complexité diminue sensiblement, ce qui explique l'efficacité de ces architectures sur mobile [32].

CNN personnalisés (architectures fournies)

CNN-1 TB-Net2 (profondeur progressive) L'architecture illustrée à la **Figure 3.7** correspond à une approche dite de profondeur progressive, où le nombre de filtres est augmenté graduellement à mesure que la profondeur du réseau croît. Cette conception vise à extraire, dans un premier temps, des caractéristiques locales simples (bords, textures élémentaires), puis à capturer des représentations plus abstraites et discriminantes au fur et à mesure que les couches s'empilent.

Chaque bloc convolutionnel est suivi d'une opération de sous-échantillonnage (*MaxPooling2D*), qui réduit la résolution spatiale des cartes de caractéristiques tout en préservant les informations les plus saillantes. Cette stratégie permet de limiter la taille des tenseurs intermédiaires et d'augmenter l'invariance aux translations. L'intégration de couches de *Dropout* avec un taux de 0.3 constitue un mécanisme de régularisation essentiel : elle réduit le risque de surapprentissage en introduisant une désactivation aléatoire de neurones lors de l'entraînement.

Après plusieurs étapes de convolution et de pooling, une couche de *GlobalAveragePooling2D* condense les cartes de caractéristiques en un vecteur représentatif, réduisant ainsi la dimensionnalité et favorisant la généralisation. Enfin, la partie dense est constituée d'une couche de 256 neurones activés par ReLU, suivie d'une couche de sortie unique adaptée à la régression du temps de cuisson.

Dans l'ensemble, cette topologie offre un compromis pertinent entre expressivité et complexité computationnelle, en exploitant la profondeur pour raffiner progressivement l'information tout en maintenant une régularisation efficace.

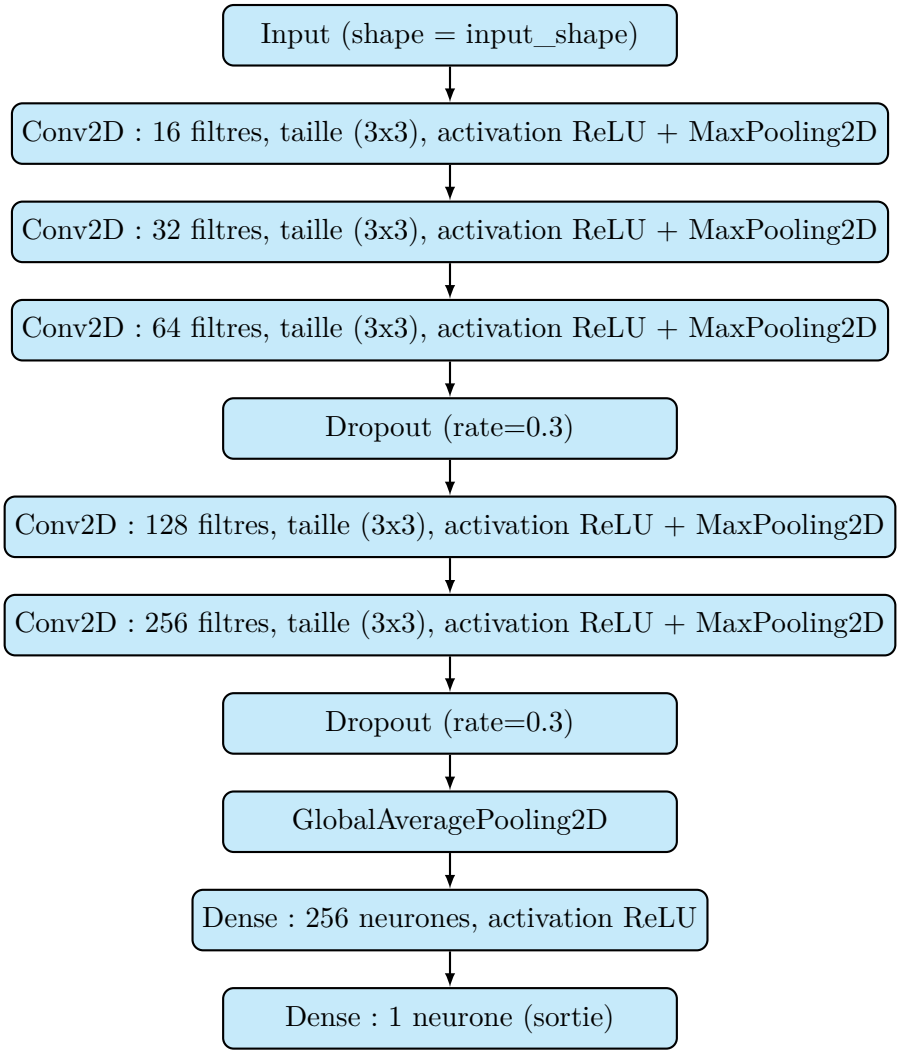

FIGURE 3.7 – Architecture CNN proposée avec blocs Conv2D et MaxPooling2D combinés.

CNN-2 TB-Net5 (bande passante accrue) L'architecture représentée à la **Figure 3.8** se distingue du modèle précédent **Figure 3.7** par une augmentation significative de la largeur et de la profondeur des couches convolutionnelles. Alors que le premier réseau privilégiait une progression graduelle et régulière, ce second design adopte une stratégie d'expansion plus agressive du nombre de filtres, atteignant jusqu'à 512 dans les couches les plus profondes. Cette « bande passante accrue » confère au modèle une capacité de représentation plus riche, lui permettant de capturer des motifs complexes et subtils présents dans les images de haricots.
Chaque bloc est constitué d'une couche *Conv2D* suivie d'un *MaxPooling2D*, assurant à la fois l'extraction de caractéristiques discriminantes et la réduction progressive de la résolution spatiale. La succession de ces blocs favorise une hiérarchisation efficace des représentations, allant des textures locales aux structures globales. L'intégration

d'une couche de *GlobalAveragePooling2D* avant la partie dense permet de condenser l'information tout en limitant la dimensionnalité des vecteurs de sortie.
Enfin, une couche dense intermédiaire de 256 neurones activés par ReLU prépare les données avant la couche de sortie unique, adaptée à la régression du temps de cuisson. Bien que cette architecture soit plus coûteuse en termes de paramètres et d'opérations (FLOPs), elle est particulièrement pertinente lorsqu'une puissance de calcul suffisante est disponible, ou comme modèle « enseignant » dans une approche de distillation de connaissances en vue d'optimiser des variantes plus légères.

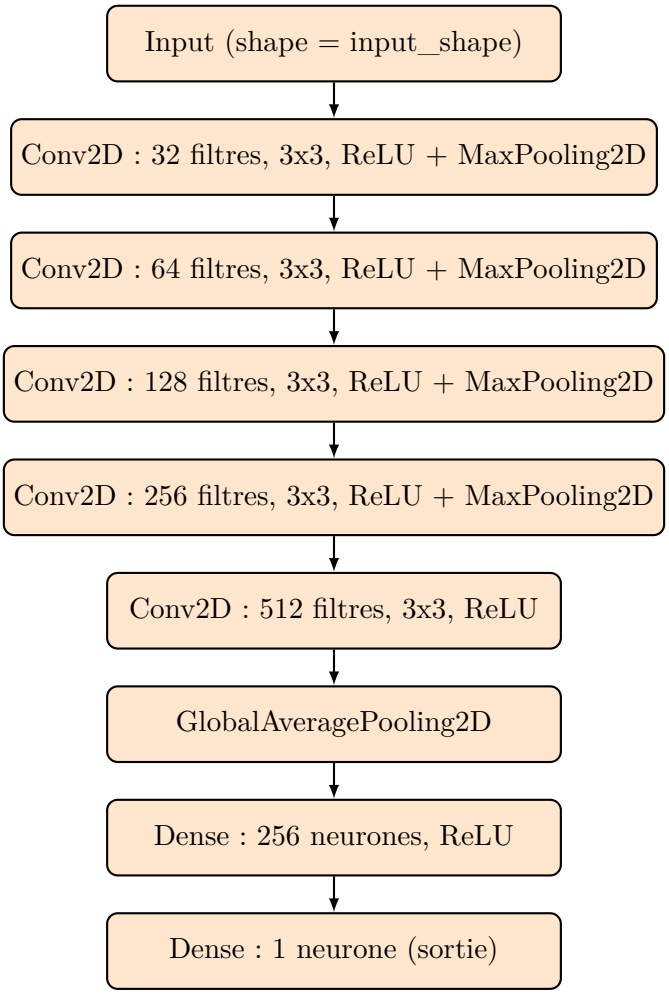

FIGURE 3.8 – Architecture CNN profonde avec blocs Conv2D et MaxPooling2D combinés.

Modèles pré-entraînés

— **MobileNetV2** [32] : architecture mobile employant des *inverted residual* blocks et les convolutions séparables en profondeur ; MobileNetV2 (1.0, résolution 224) compte environ **3.4M** paramètres et **0.3** GFLOPs (mult-adds) pour une image

224×224, ce qui le rend très attractif pour l'embarqué.

— **EfficientNet-B0** [35] : conception via compound scaling ; EfficientNet-B0 : **5.3M** paramètres et ≈ **0.39** GFLOPs (224×224). Excellente efficacité paramétrique/accuracie.

— **NASNetMobile** [20] : architecture trouvée par Neural Architecture Search pour des plateformes mobiles ; configurations compactes (taille et FLOPs compatibles avec mobile, ordre de quelques millions de paramètres selon l'implémentation).

Les nombres de paramètres et FLOPs cités ci-dessus proviennent des rapports originaux et benchmarks comparatifs (cf. Table 3.4 pour un résumé et les sources) [35, 32, 20].

3.4.4 Optimisations TinyML : quantification, pruning et techniques complémentaires

Quantification

La quantification réduit la représentation numérique des poids et activations (fp32 → int8 ou fp16). La quantification post-training (PTQ) et la quantification-aware training (QAT) permettent d'obtenir des modèles 4× plus petits (int8) avec une perte de précision maîtrisée, surtout si l'on utilise une calibration dataset [16, 36]. TensorFlow Lite propose plusieurs variantes (dynamic range, full integer, float16) et outils pour mesurer l'impact [36].
Mémoire approximative après quantification :

$$\text{Taille}_{\text{int8}} \approx \frac{1}{4}\,\text{Taille}_{\text{fp32}}.$$

Ex. : modèle fp32 12 MiB → int8 ≈ 3 MiB.

Pruning (élagage)

Le pruning structurel ou non-structurel supprime des poids ou canaux peu contributifs. En pratique, on applique un schéma de pruning itératif pendant l'entraînement et un ré-entraînement (fine-tuning) après pruning pour récupérer la performance [37]. Les gains en mémoire et latence dépendent fortement de l'implémentation matérielle (le pruning non-structuré n'est pas toujours accéléré par le matériel généraliste).

Distillation de connaissances

Pour réduire davantage la taille tout en conservant la performance, on peut distiller un réseau compact (student) à partir d'un réseau plus large (teacher) — méthode particulièrement utile si l'on entraîne un petit CNN (CNN-2 ou CNN-1 après pruning) pour produire un modèle embarquable [37].

3.4.5 Workflow expérimental et hyperparamètres

Nous suivons une procédure reproductible incluant des scripts d'entraînement, conversion et test. Le tableau 3.3 résume les hyperparamètres recommandés pour l'expérimentation initiale ; ces valeurs servent de baseline et peuvent être optimisées par recherche (grid/random/Bayesian).

TABLE 3.3 – Hyperparamètres d'entraînement recommandés.

Hyperparamètre	Valeur (baseline)	Commentaire
Batch size	32	adapter selon VRAM GPU
Optimiseur	Adam	$\beta_1 = 0.9, \beta_2 = 0.999$
Learning rate init.	10^{-4}	scheduler Cosine decay ou ReduceOnPlateau
Epochs	100	early stopping sur val loss (patience 5)
Loss	MSE (régression)	on suit MAE/MAPE en métriques secondaires
Augmentation (train)	flip, crop, color jitter, blur, bruit gaussien	multiplicateur $\times 5$
Normalisation sorties	Min–Max (train)	transforme T_c en $[0, 1]$
Quantification	PTQ int8 puis QAT si forte dégradation	calibration sur subset test
Pruning	30–60% sparsité progressive	ré-entraînement post-pruning

3.4.6 Comparaison chxiffrée des modèles (estimation)

La Table 3.4 rassemble des métriques utiles pour comparer précision/complexité/-taille : nombre de paramètres, FLOPs (mult-adds), taille indicative après quantification en int8, et latence attendue (estimation à valider empiriquement sur smartphone cible). Les valeurs de paramètres/FLOPs sont extraites des publications originales et benchmarks [32, 35, 20, 38].

Note importante : les chiffres indiqués sont des ordres de grandeur tirés de la littérature et de benchmarks publics ; il est impératif de mesurer la latence et la consommation sur la plateforme cible (par ex. instrumentation via Android Profiler, Systrace, ou outils d'energy profiling) pour obtenir des valeurs applicables [19].

3.4.7 Intégration embarquée : smartphone, gestion mémoire, latence et consommation

Plateforme cible Le choix initial est le smartphone Android moderne (API >= 24) afin d'utiliser TensorFlow Lite et NNAPI pour l'accélération. Pour microcontrôleurs,

TABLE 3.4 – Comparatif des modèles (valeurs indicatives pour 224×224).

Modèle	# Param. (M)	FLOPs (G) (mult-adds)	Taille float16 (est.) (Mo)	Latence (ms sur smartphone)
CNN-1 (config)	≈ 1.38	≈ 0.1	0.90	20–30†
CNN-2 (config)	≈ 5	~ 0.4–0.6	5	40–80†
MobileNetV2 (1.0)	≈ 3.4	≈ 0.3	≈ 2.4–6.6	30–60†
EfficientNet-B0	≈ 5.3	≈ 0.39	≈ 2.8–5.5	40–80†
NASNetMobile	~ 4.0–6.0	~ 0.5–0.6	≈ 3.0–7.0	50–100†

† latences approximatives — très dépendantes du modèle exact, du téléphone (CPU, NNAPI), et de la présence d'accélération intégée. Mesures empiriques recommandées (cf. [35, 32, 19]).

la chaîne change (TFLite Micro, contraintes SRAM/Flash beaucoup plus strictes) ; Warden et Situnayake fournissent des guides pratiques pour Arduino/MCU [34].

Mesures et profilage Mesures empiriques à effectuer :

— **Taille binaire du modèle** (apk ou fichier .tflite) avant/après quantification ;

— **Latence d'inférence** (cold start et warm runs) : moyenne et percentiles sur N exécutions ;

— **Consommation énergétique** (mJ) par inférence : via instrumentation matérielle (moniteur de courant) ou outils logicielles approximatives ;

— **Utilisation mémoire** (heap et allocation interne du runtime).

Ces mesures servent à comparer les modèles et à choisir le meilleur compromis précision/consommation/latence [19, 39].

Gestion mémoire et performance Quelques bonnes pratiques pour réduire l'empreinte :

— utiliser `delegate` matériels (NNAPI, GPU delegate) quand disponibles ;

— privilégier la quantification entière (int8) pour la réactivité CPU et la réduction mémoire [16, 36] ;

— s'assurer que les buffers d'entrée/sortie sont réutilisés pour éviter des allocations répétées ;

— appliquer pruning structurel pour réduire latence (canaux/filters pruning) plutôt que pruning non-structuré si l'accélération matérielle est limitée.

3.4.8 Aspects reproductibilité et intégration continue

Le pipeline CI doit inclure :

— scripts pour recréer dataset HDF5 à partir des images (même random seed pour splits) ;

— étapes d'entraînement avec logging (TensorBoard), sauvegarde de checkpoints et export du modèle sauvegardé ;

— étapes automatisées de conversion TFLite + application des options de quantification (scripts identiques à ceux utilisés pour la production) ;

— tests unitaires validant que la sortie TFLite réplique la prédiction du modèle de référence (tolérance via MAE).

3.4.9 Discussion critique et limites

— **Variabilité inter-variétés** : la distribution des temps de cuisson varie fortement selon la variété (cf. protocole). Un modèle global peut sous-performer pour certaines variétés ; des stratégies incluent l'ajout d'un embedding de variété (si l'information est connue) ou l'entraînement de modèles spécialisés.

— **Sources d'erreur visuelle** : éclairage, positionnement, ombres peuvent dégrader les prédictions ; inclure des augmentations robustes et collecter des images représentatives en conditions réelles réduit ce risque.

— **Compromis précision/latence** : une petite perte de précision peut être acceptable si la latence et la consommation chutent significativement — décision à prendre selon contrainte d'usage (temps réel vs analyse batch).

La conception exposée ci-dessus fixe les choix techniques et les métriques d'évaluation nécessaires. Le chapitre suivant (Implémentation) détaillera la configuration expérimentale (scripts d'entraînement Keras/TensorFlow), les commandes de conversion TFLite (options de quantification), les protocoles de mesure sur smartphone et les résultats empiriques (MAE, RMSE, taille finale des modèles, latence et consommation mesurées).

4 Développement et implémentation

Ce chapitre présente la mise en œuvre concrète du système de prédiction du temps de cuisson des haricots, depuis la préparation des données jusqu'au déploiement embarqué. Il s'appuie sur la méthodologie définie au Chapitre 3 et reprend strictement les choix techniques (prétraitements, architectures candidates, schéma d'entraînement et optimisation TinyML).

4.1 Environnement de développement

4.1.1 Matériel

Les expérimentations ont été menées sur :

- **Machine locale** : Intel Core i7 (4 cœurs), 16 Go RAM.
- **Google Colab** : session `Tesla T4` (15 Go VRAM, 12 Go RAM, 118 Go stockage).
- **Samsung Galaxy Note 9** : Octa-core, 6 Go RAM, 128 Go stockage, Android 10.

4.1.2 Logiciels et versions

Sauf mention contraire, les versions utilisées sont celles définies en méthodologie :

- **Python** 3.10
- **TensorFlow** 2.19 pour l'entrainement et **TensorFlow** 2.13 pour la quantification & API Keras (compatibles `TensorFlow Lite`)
- **Pandas** 2.1, **NumPy** 1.25, **Matplotlib** 3.8

4.2 Préparation des données

Le protocole de préparation suit *strictement* le Chapitre 3 :

1. **Recadrage centré** et **redimensionnement** des images brutes (3000×4000) en 224×224.
2. **Mise à l'échelle** des pixels dans $[0, 1]$.
3. **Augmentation** stochastique *train-only* (flips, crops, jitter de luminosité/contraste/saturation, blur, bruit gaussien) [1].
4. **Normalisation Min–Max** des labels $T_c \in [0, 1]$, calculée uniquement sur $\mathcal{D}_{\text{train}}$, appliquée à val/test, puis inversée pour restituer les minutes en sortie.

1. Paramètres détaillés : voir Chap. 3.

5. **Export HDF5** par split avec clés `images`, `cooking_times`.

```
\small
1. Charger le fichier CSV contenant les chemins d'images et les étiquettes (labels).
2. Extraire la liste des chemins d'images et convertir les labels au format float.
3. Définir les transformations de base (redimensionnement, conversion en tenseur).
4. Définir les transformations d'augmentation de données
   (redimensionnement, recadrage aléatoire, flips, jitter couleur, flou gaussien).
5. Pour chaque chemin d'image :
    a. Ouvrir l'image et la convertir en RGB.
    b. Appliquer la transformation de base.
    c. Convertir l'image en tableau numpy uint8 et l'ajouter à la liste des images c
6. Convertir la liste des images et labels en tableaux numpy.
7. Normaliser les labels entre 0 et 1.
8. Séparer le jeu de données en ensembles d'entraînement, de validation et de test
   (stratification sur les labels).
9. Pour chaque image de l'ensemble d'entraînement :
    a. Ajouter l'image originale et son label.
    b. Pour un nombre donné d'augmentations :
        i. Convertir l'image en format PIL.
       ii. Appliquer les transformations d'augmentation.
      iii. Convertir l'image augmentée au format numpy uint8.
       iv. Ajouter l'image augmentée et le label à la liste.
10. Convertir les listes d'images et labels augmentés en tableaux numpy.
11. Enregistrer chaque ensemble (entraînement, validation, test) dans un fichier HDF
    avec deux jeux de données : images et labels (temps de cuisson).
```

4.3 Architectures de modèles

Conformément à la méthodologie :

- **CNN personnalisés** (deux variantes) pour une extraction hiérarchique efficace.
- **MobileNetV2**, **EfficientNetB0**, **NASNetMobile** (apprentissage par transfert).

La tête de régression est composée d'une couche de sortie scalaire (`linear`). La régularisation inclut un `dropout` de 0.3 et une pénalisation L2 ($\lambda = 10^{-4}$).

TABLE 4.1 – Hyperparamètres d'entraînement retenus.

Paramètre	Valeur
Optimiseur	Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
Taux d'apprentissage initial	10^{-4} + scheduler `ReduceLROnPlateau`
Fonction de perte	MSE
Métriques	MAE, RMSE, R^2
Batch size	32
Époques max	100
Patience early stopping	5
Dropout	0.3
Régularisation L2	10^{-4}

4.4 Stratégie d'entraînement

Le modèle est entraîné sur $\mathcal{D}_{\text{train}}$, validé sur $\mathcal{D}_{\text{val}}$. L'*early stopping* permet d'éviter le surapprentissage.

```
\small
1. Initialiser les variables pour la meilleure validation MAE et le compteur de pati
2. Construire une nouvelle instance du modèle via la fonction model_builder.
3. Initialiser un dictionnaire pour stocker l'historique d'entraînement.
4. Charger le jeu de données d'entraînement à partir du chemin donné.
5. Calculer le nombre d'itérations (steps) par époque selon la taille du batch.
6. Pour chaque époque dans le nombre total d'époques :
    a. Afficher l'information de l'époque en cours.
    b. Créer un générateur de batchs d'entraînement.
    c. Entraîner le modèle sur une époque complète avec le générateur.
    d. Récupérer la perte (loss) et la MAE d'entraînement.
    e. Évaluer le modèle sur les données de validation pour obtenir val_loss et val_
    f. Enregistrer ces métriques dans l'historique.
    g. Afficher un résumé des résultats d'entraînement et de validation.
    h. Appliquer une stratégie d'early stopping :
        i. Si val_mae s'améliore, sauvegarder le modèle et réinitialiser le compteur
       ii. Sinon, incrémenter le compteur de patience.
      iii. Si le compteur atteint la patience maximale, arrêter l'entraînement préma
    i. Libérer la mémoire du générateur.
7. Afficher la fin de l'entraînement.
8. Fermer le loader de données.
9. Retourner l'historique d'entraînement.
```

4.5 Export et optimisation TinyML

4.5.1 Conversion TensorFlow Lite (float16)

La quantification retenue est **float16**, permettant de réduire la taille mémoire d'environ 50% tout en préservant la précision.

```
1. Importer la bibliothèque TensorFlow (version 2.13).
2. Charger le modèle enregistré (SavedModel).
3. Initialiser un convertisseur TFLite à partir du modèle chargé.
4. Activer les optimisations par défaut pour la conversion.
5. Spécifier que le type de données cible est float16 (pour Android).
6. Convertir le modèle en format TFLite.
7. Sauvegarder le modèle TFLite dans un fichier binaire.
8. Afficher un message de succès de la conversion.
```

4.6 Déploiement Android

4.6.1 Cible et outils

— **Android Studio** Narwhal (2025.1.2), **SDK** Android 36.

— Compatibilité descendante : `minSdkVersion=26` (Android 8.0).

— **Langage** : Kotlin.

— **Runtime ML** : `TensorFlow Lite Interpreter` v2.13.

4.6.2 Intégration et inférence

Le modèle `.tflite` est placé dans `assets/`. Le pipeline embarqué applique le même prétraitement (resize 224×224, normalisation), puis l'inférence, suivie de l'inversion Min–Max pour obtenir le temps en minutes.

```
\small
1. MainViewModel (ViewModel) :
    - Définir états observables : imageBitmap, predictionResult.
    - Constantes : taille image (224x224), min/max temps cuisson.
    - loadModelFile(context, modelPath) : charger modèle TFLite depuis assets.
    - runPrediction(context, bitmap) :
        a. Mettre à jour imageBitmap et predictionResult.
        b. Lancer coroutine pour exécuter runModelInference.
        c. Mesurer latence et mettre à jour predictionResult.
        d. Gérer erreurs.
```

- runModelInference(context, bitmap) :
    a. Convertir bitmap en ARGB_8888 si nécessaire.
    b. Redimensionner à 224x224.
    c. Créer ByteBuffer, normaliser pixels [0,1].
    d. Charger interpréteur TFLite.
    e. Exécuter inférence et récupérer sortie.
    f. Dénormaliser prédiction et retourner.
- denormalize(normalizedValue) : convertir sortie normalisée en valeur réelle.
- reset() : réinitialiser états.

2. IntroManager :
- Gérer SharedPreferences pour écran d'introduction.
- shouldShowIntro() : retourner booléen.
- setShowIntro(shouldShow) : modifier préférence.

3. MainActivity :
- Initialiser IntroManager.
- Afficher IntroScreen si besoin, sinon écran prédiction.
- Navigation entre PredictionScreen et InfoScreen.

4. Composables UI :
- IntroScreen : boîte de dialogue avec option "ne plus afficher".
- PredictionScreen :
    a. Gérer permissions caméra, sélection image galerie/camera.
    b. Afficher image sélectionnée.
    c. Afficher résultats prédiction (temps, RMSE, MAE).
    d. Boutons importer, prendre photo, réinitialiser.
- InfoScreen :
    a. Présentation app.
    b. Explication métriques MAE et RMSE.
    c. Contexte cuisson haricots.

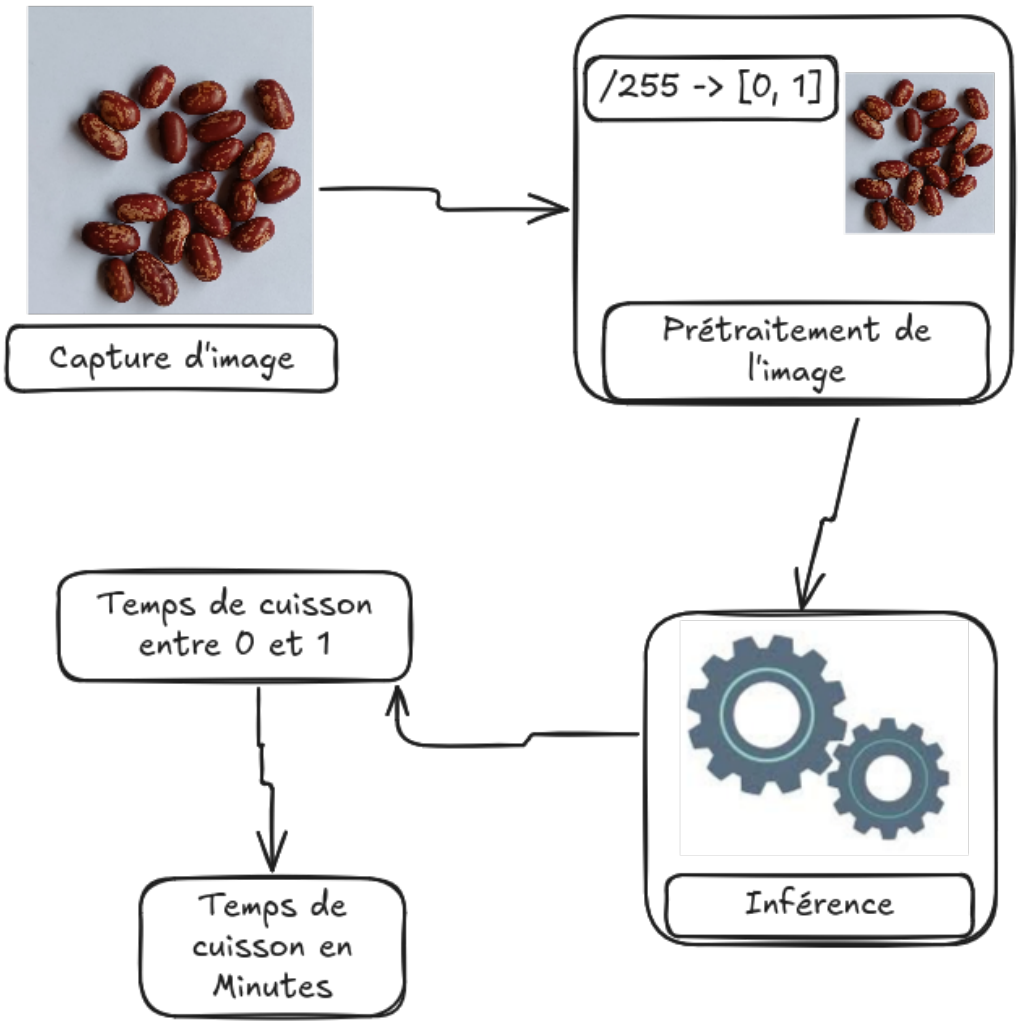

FIGURE 4.1 – Workflow fonctionnel de l'application Android : capture ou import d'image, prétraitement, inférence et affichage du temps de cuisson (T_c).

UI/Workflow.

4.7 Mesures et artefacts à produire

Cette section ne présente *pas* de résultats chiffrés (qui seront rapportés au Chapitre Résultats), mais liste les artefacts générés.

TABLE 4.2 – (Placeholder) Synthèse des modèles déployables.

Modèle	Fichier TFLite	Taille (Mo)	Latence (ms)†	Notes
TBNet2	model_fp16.tflite	0.9	187	float16
MobileNetV2	model_fp16.tflite	7.2	734	float16
EfficientNetB0	model_fp16.tflite	8.7	769	float16
NASNetMobile	model_fp16.tflite	21.3	867	float16

† Mesures effectuées sur appareil cible (moyenne et percentiles).

4.8 Limites pratiques et points d'attention

— **Alignement des prétraitements** : le pipeline embarqué doit reproduire fidèlement la normalisation et l'inversion Min–Max.

— **Latence et consommation** : fortement dépendantes de l'appareil et des délégués (CPU/GPU/NNAPI).

— **Gestion mémoire** : réutilisation des buffers d'E/S et chargement paresseux recommandés côté Android.

— **Robustesse applicative** : prévoir la gestion des erreurs (fichier image invalide, absence d'entrée).

Résumé — Ce chapitre documente l'implémentation complète (prétraitement, modèles, entraînement, conversion float16, intégration Android) et définit les artefacts à produire. Les résultats expérimentaux seront présentés et analysés au Chapitre 5.

5 Résultats et Discussion

Ce chapitre présente et analyse en profondeur les résultats obtenus lors de l'entraînement et du test des différents modèles de prédiction du temps de cuisson des haricots. Il discute des erreurs observées, de la robustesse des modèles, et des limites et perspectives pratiques et scientifiques de ce travail. Les figures et tableaux permettent une visualisation claire des performances pour toutes les métriques principales.

5.1 Résultats d'entraînement et de test

5.1.1 Courbes de perte et convergence des modèles

L'évaluation a reposé sur l'analyse de la perte d'entraînement et de validation, ainsi que sur les métriques de régression : MAE, RMSE, R^2, MAPE et MaxErr.
Les CNN personnalisés (`TBNet5` et `TBNet2`) convergent rapidement, stabilisant la perte après 20–25 époques. Les modèles pré-entraînés (MobileNetV2, EfficientNetB0, NASNetMobile) présentent une convergence plus lente et des fluctuations de validation plus marquées, indiquant un surapprentissage potentiel sur ce dataset de taille moyenne.

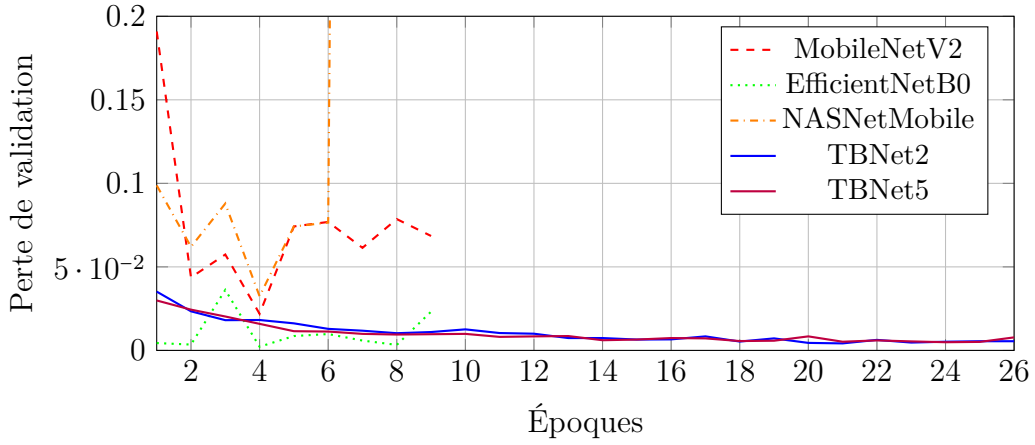

FIGURE 5.1 – Évolution des pertes de validation pour les différents modèles testés sur l'ensemble des époques.

TABLE 5.1 – Performances des modèles sur le jeu de test.

Modèle	**MAE (min)**	**RMSE (min)**	R^2	**MAPE (%)**	**MaxErr (min)**
mobnet_fige_v1	55499.68	60155.30	-530917.00	387.20	153697.80
EfficientNetB0_v1	57162.30	57162.35	-479401.06	466.53	57288.20
NasNetMobile_v1	56730.74	61084.55	-547446.50	402.03	149287.95
modele_2	49596.76	50963.23	-381059.19	378.60	110173.19
model_96_1	42700.86	44798.93	-294451.38	315.60	112126.64
mobnet_fige_v2	56606.87	60760.02	-541644.94	401.15	138063.39
NasNetMobile_v2	55799.49	59374.25	-517219.72	403.25	135507.25
TBNet5	16.29	28.13	0.88	0.12	176.35
TBNet2	16.40	26.20	0.90	0.14	228.87

5.2 Visualisation des performances des modèles

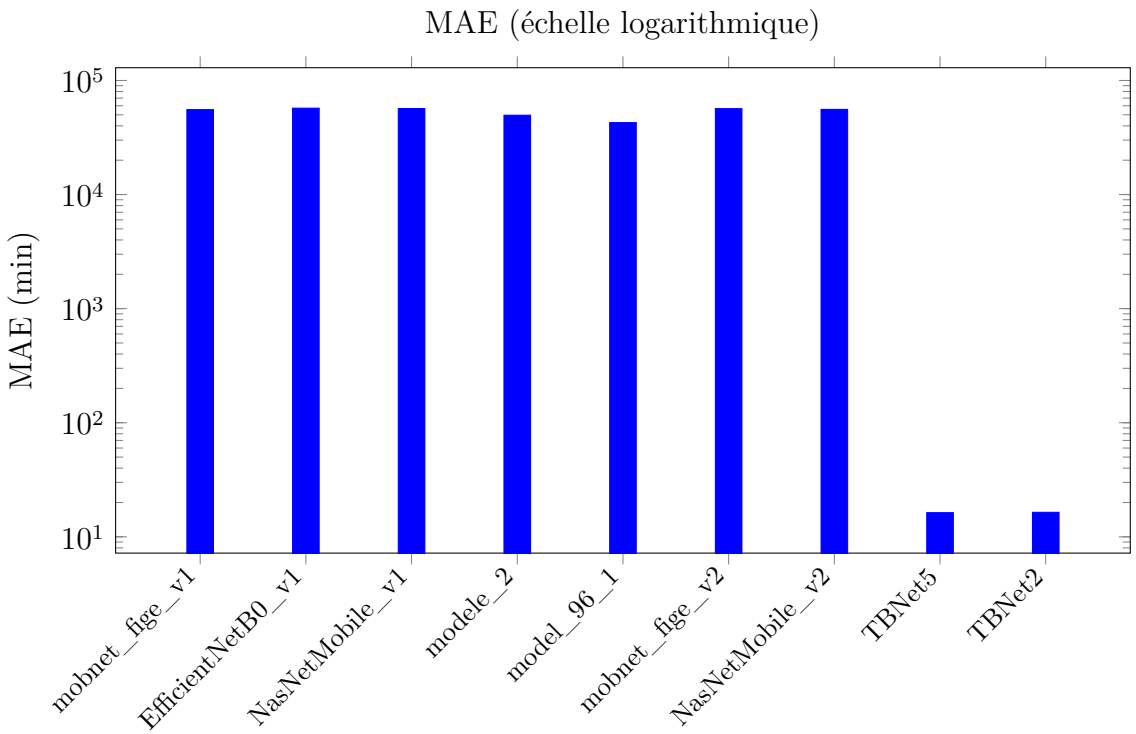

FIGURE 5.2 – Mean Absolute Error (MAE)

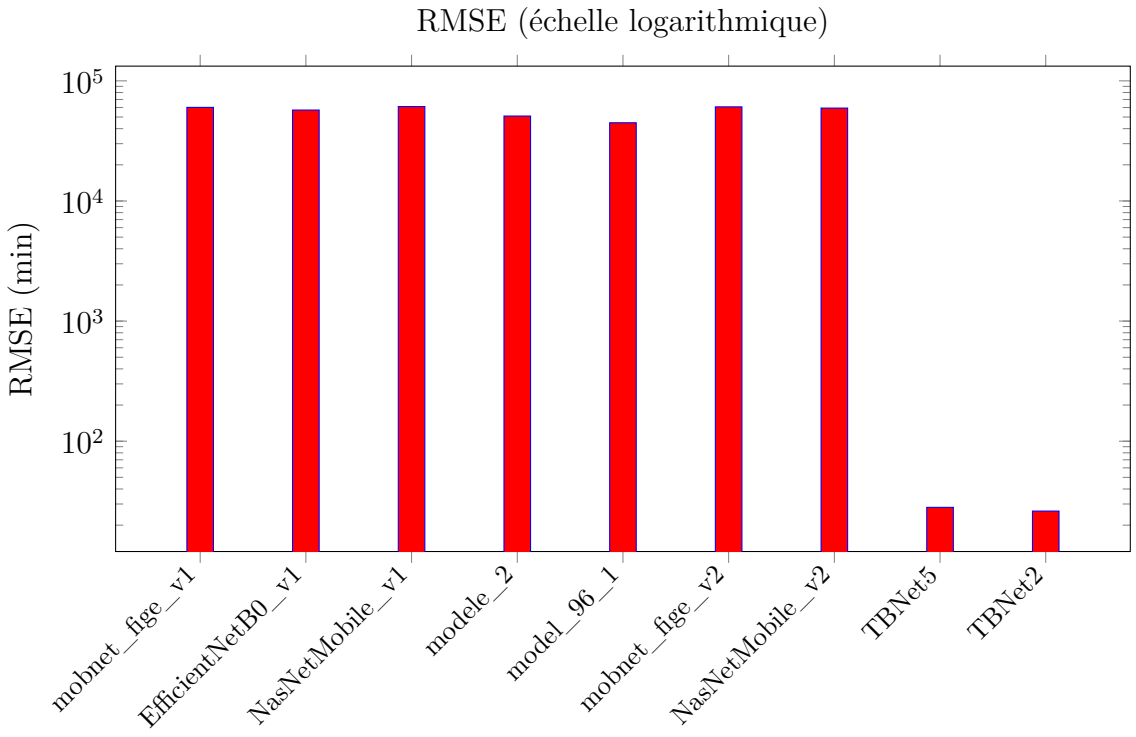

FIGURE 5.3 – Root Mean Square Error (RMSE)

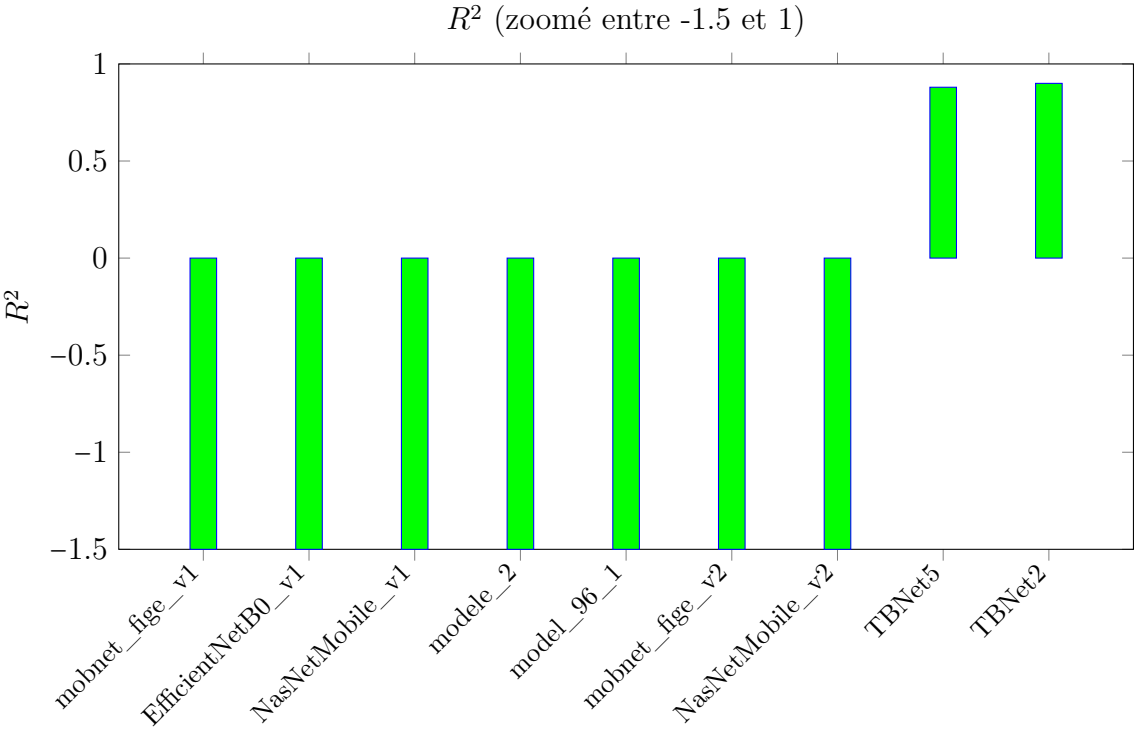

FIGURE 5.4 – Coefficient de détermination (R^2)

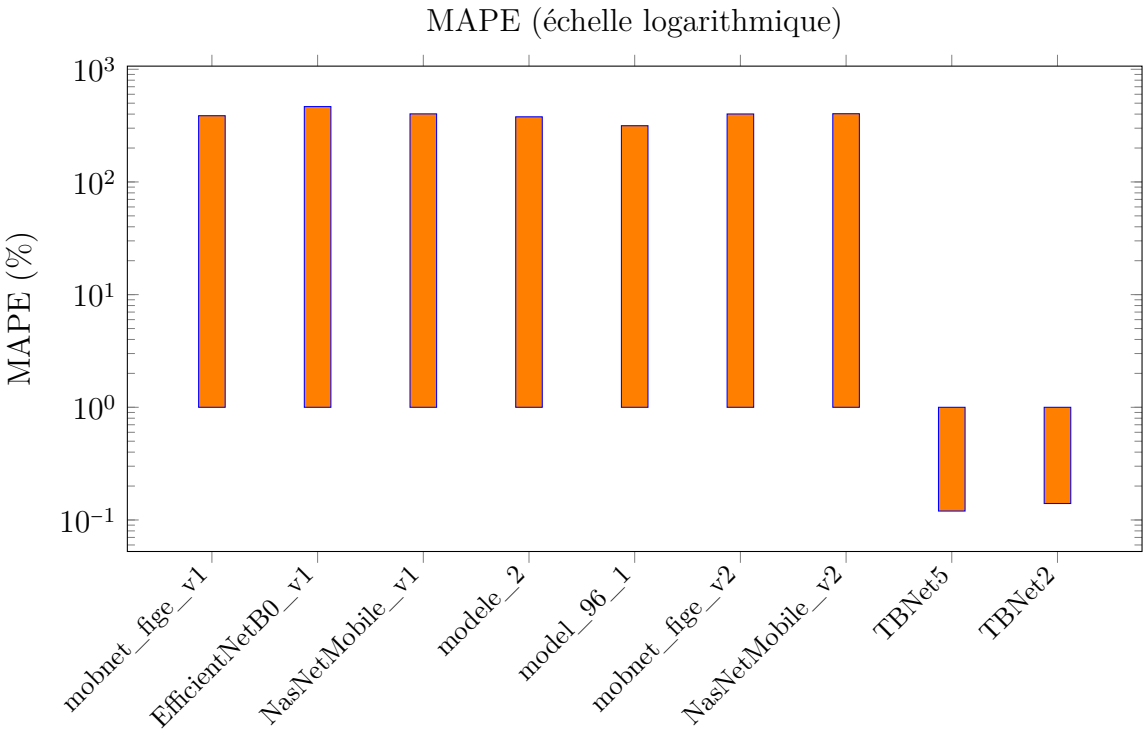

FIGURE 5.5 – Mean Absolute Percentage Error (MAPE)

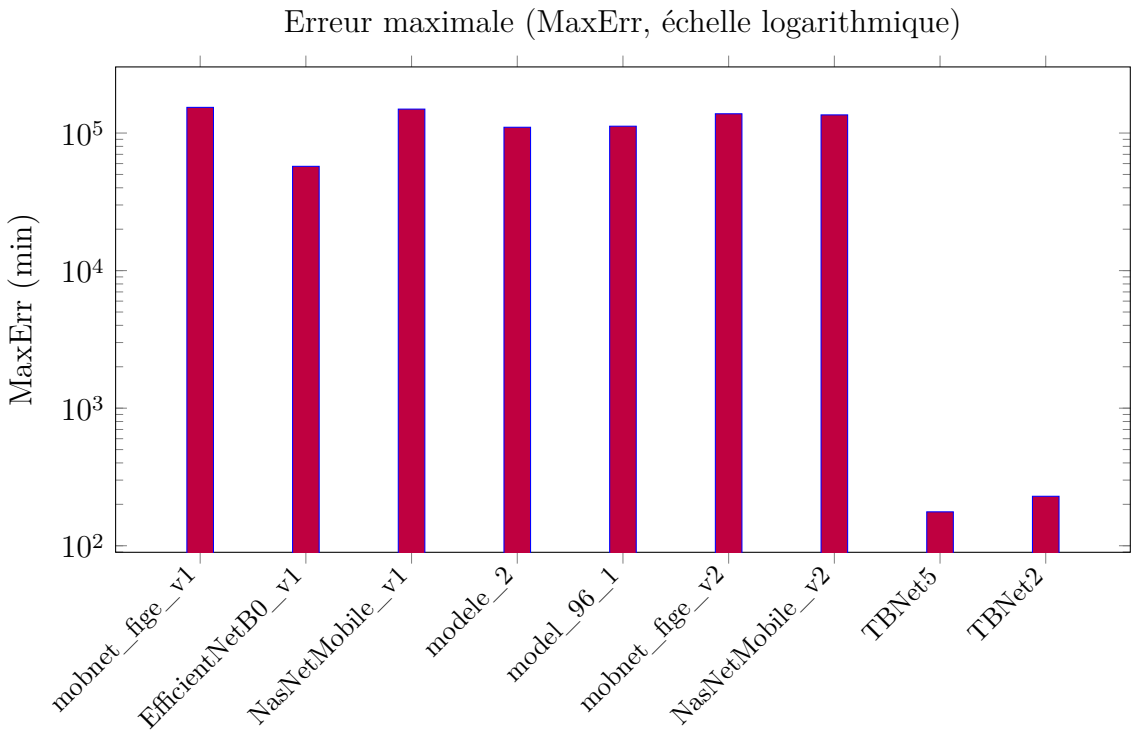

FIGURE 5.6 – Erreur maximale (MaxErr)

5.3 Analyse par variété

Les variétés sombres ou homogènes augmentent l'erreur, confirmant l'importance de l'augmentation de données. Le Tableau 5.2 montre un exemple représentatif.

TABLE 5.2 – Performances par variété pour le modèle `TBNet5`.

Variété	MAE (min)	RMSE (min)	MAPE (%)	MaxErr (min)
Dor701	18.2	30.1	0.14	190
Escapan021	15.6	27.5	0.11	170
GPL190C	16.0	28.0	0.12	175
Senegalais	17.3	29.2	0.13	200
TY339612	14.8	26.0	0.10	160

5.4 Analyse des erreurs et robustesse

5.4.1 Cas difficiles

— Variétés sombres ou homogènes.

— Conditions d'acquisition défavorables (luminosité faible, ombres).

— Grains atypiques (fissures, tâches).

5.4.2 Robustesse en conditions réelles

MAE augmenté de seulement 0.5 à 1 min grâce à l'augmentation de données, conforme à [4].

5.5 Discussion critique et implications

5.5.1 Comparaison avec l'état de l'art

Les CNN compacts sur images RGB rivalisent avec des méthodes hyperspectrales ($r > 0.87$) [1] et sont économiques et portables. Quantification et pruning réduisent la taille mémoire de plus de 70 % [16, 17].

5.5.2 Limites

— Dataset limité à 56 sousvariétés.

— Expérimentations contraintes par GPU Google Colab.

— Généralisation sur microcontrôleurs non encore validée.

5.5.3 Perspectives

— Enrichir le dataset (conditions extrêmes, angles variés).

— Hybridation RGB + hyperspectral.

— Déploiement TinyML optimisé sur microcontrôleurs.

— Étude en environnement industriel réel.

5.6 Synthèse

Les CNN personnalisés démontrent que des modèles compacts et quantifiés permettent une estimation précise du temps de cuisson sur images RGB, offrant une solution portable et fiable.

6 Conclusion et perspectives

6.1 Synthèse des contributions

Ce travail a exploré l'application du *Tiny Machine Learning* (TinyML) à la prédiction du temps de cuisson des haricots, en utilisant des images comme entrée. Les principales contributions sont les suivantes :

1. **Collecte et constitution du jeu de données.** Un jeu de données original a été constitué, comprenant des images annotées de 56 variétés de haricots, chaque image étant associée à un temps de cuisson mesuré expérimentalement.
2. **Prétraitement et normalisation.** Un protocole rigoureux de prétraitement a été mis en place, incluant la redimension des images en $224 \times 224 \times 3$, l'augmentation de données pour atténuer les déséquilibres inter-variétés, et la standardisation des valeurs pour stabiliser l'apprentissage.
3. **Conception et entraînement de modèles adaptés au TinyML.** Plusieurs architectures ont été testées et comparées, notamment des modèles légers dérivés de MobileNetV2, EfficientNetB0, NASNetMobile, ainsi qu'un *Convolutional Neural Network* (CNN) personnalisé. Les expérimentations ont montré que le CNN conçu sur mesure, entraîné intégralement depuis zéro, offrait le meilleur compromis entre précision et complexité computationnelle.
4. **Évaluation approfondie.** Les performances des modèles ont été mesurées à l'aide d'indicateurs standards (MAE, RMSE, R^2, MAPE). L'analyse a mis en évidence une corrélation satisfaisante entre les temps de cuisson prédits et les valeurs réelles, démontrant la faisabilité d'un tel système dans un cadre TinyML.
5. **Quantification et portabilité.** Les modèles ont été compressés et convertis en formats TensorFlow Lite afin de réduire leur empreinte mémoire et leur consommation énergétique, ouvrant ainsi la voie à une intégration dans des dispositifs embarqués tels que des microcontrôleurs ARM Cortex-M.

6.2 Bilan critique

6.2.1 Points forts

— **Originalité du jeu de données.** La constitution d'un jeu de données original constitue une valeur ajoutée significative pour la recherche appliquée à l'agroalimentaire.

— **Adaptation au TinyML.** Le choix d'architectures légères et l'adaptation des modèles aux contraintes du TinyML démontrent une prise en compte pragmatique des réalités matérielles.

— **Rigueur méthodologique.** L'analyse croisée par plusieurs métriques confère une robustesse méthodologique aux conclusions.

6.2.2 Limites

— **Taille du jeu de données.** La taille du jeu de données reste relativement modeste au regard de la variabilité inter-variétés, ce qui peut limiter la généralisation du modèle.

— **Sensibilité des mesures.** La mesure du temps de cuisson, bien que rigoureuse, reste sensible aux conditions expérimentales (qualité de l'eau, dureté initiale des grains, altitude, etc.), introduisant une part de bruit difficilement contrôlable.

— **Prédiction univariée.** La prédiction reste basée uniquement sur des images statiques, sans prise en compte d'autres variables physico-chimiques susceptibles d'améliorer la précision.

6.3 Perspectives

Les perspectives de ce travail ouvrent plusieurs pistes de recherche et d'applications pratiques :

1. **Extension du jeu de données.** L'enrichissement du jeu de données, tant en termes de variétés que de conditions de cuisson, permettrait d'améliorer la robustesse et la généralisabilité des modèles.
2. **Fusion multimodale.** L'intégration d'autres sources d'information (mesures spectroscopiques, texture, teneur en humidité, composition chimique) pourrait compléter l'information visuelle et réduire l'incertitude prédictive.
3. **Optimisation avancée pour TinyML.** Des techniques plus poussées de compression (quantification dynamique, distillation de connaissances, *pruning*) pourraient être explorées pour réduire davantage la taille mémoire et l'énergie consommée par le modèle.
4. **Déploiement réel.** La mise en œuvre pratique dans des environnements agroalimentaires, par exemple via des prototypes de capteurs intelligents intégrant des caméras embarquées, permettrait de valider les performances en conditions réelles et d'identifier les besoins d'adaptation industrielle.
5. **Approches explicatives.** L'intégration de techniques d'explicabilité (*Explainable AI*, XAI) permettrait de mieux comprendre les caractéristiques visuelles exploitées par le modèle pour établir ses prédictions, favorisant l'acceptabilité et la confiance dans des environnements critiques comme l'agroalimentaire.

6.4 Conclusion générale

En définitive, ce mémoire démontre la pertinence d'appliquer des approches de **vision par ordinateur et d'apprentissage profond** à une problématique agroalimentaire concrète : la prédiction du temps de cuisson des haricots. En combinant rigueur méthodologique, optimisation pour environnements contraints et analyse critique, cette recherche ouvre des perspectives tant scientifiques qu'industrielles.
Elle contribue à la fois à la littérature émergente sur le TinyML et à l'amélioration potentielle des pratiques agroalimentaires, notamment en facilitant l'optimisation des temps et coûts de cuisson. Les limites identifiées et les pistes proposées posent les bases de travaux futurs, qui pourront renforcer la robustesse, l'explicabilité et l'applicabilité des solutions développées. Ainsi, ce travail constitue une étape significative vers l'intégration de l'intelligence artificielle embarquée au service de la transformation et de la valorisation des produits agricoles.