Fine-tuning de Depth Anything V2 avec LoRA

Prédiction de profondeur à partir d'images RGB et nuages de points (canal Z)
Normalisation inverse + perte L1/Gradient pour améliorer les détails (pneus)

Ayman Zejli Mouad Azennag Abdelali Chikhi Loic Magnan

12 janvier 2026

Contexte. Adapter un modèle fondation de profondeur (*Depth Anything V2*) à un dataset spécifique : pour chaque image RGB, un fichier `.npy` contient (X, Y, Z) par pixel, et on utilise le canal **Z** comme vérité terrain (mm).

Objectif. Obtenir une profondeur cohérente visuellement et plus **fine sur les détails proches** (ex. rainures/contours du pneu).

Choix techniques clés.

— Fine-tuning paramètre-efficiente par **LoRA** (backbone gelé, peu de poids entraînables).

— **Normalisation inverse** de la profondeur pour accentuer l'apprentissage des objets proches.

— **Perte mixte** : **L1 masquée + loss de gradient** (bords) avec pondération des zones de fort gradient.

— **Haute résolution** en entrée (756×1260) et **upsampling bicubique** vers la taille GT.

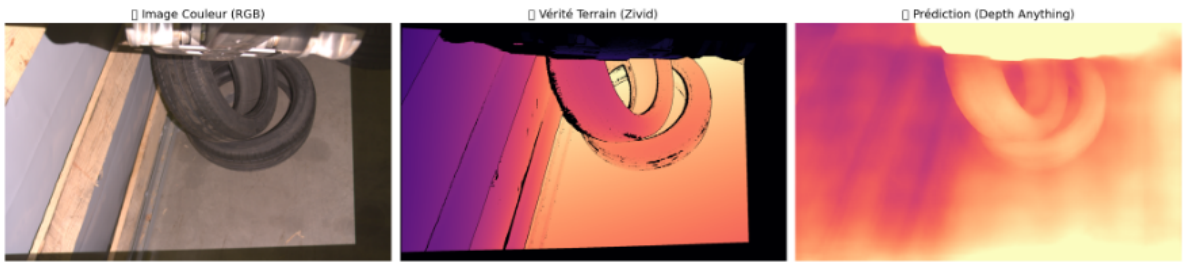

FIGURE 1 – Teaser : comparaison (RGB / profondeur GT / profondeur prédite).

1 Introduction

L'estimation monoculaire de profondeur (*Monocular Depth Estimation, MDE*) consiste à prédire une carte de profondeur dense à partir d'une seule image RGB. Dans ce projet, nous adaptons **Depth Anything V2** à un dataset *RGB + nuage de points* : chaque pixel possède une mesure (X, Y, Z), et la profondeur vérité terrain est Z (en mm).

Un défi majeur est la présence de zones **invalides** (NaN, trous capteurs, valeurs nulles). Nous utilisons donc un **masque de validité** et une **perte masquée** pour ne pas rétropropager sur des pixels sans profondeur fiable.

2 Modèle : Depth Anything V2 et adaptation LoRA

2.1 Modèle pré-entraîné

Nous utilisons :

`depth-anything/Depth-Anything-V2-Small-hf`.

Les images RGB sont prétraitées via `AutoImageProcessor`, puis le modèle produit une carte `predicted_depth`.

2.2 Pourquoi LoRA ?

Un fine-tuning complet met à jour tous les poids du Transformer (coût mémoire/temps, risque de sur-apprentissage avec peu de données). LoRA (*Low-Rank Adaptation*) limite l'apprentissage à une petite mise à jour de rang faible.

2.3 Principe

Pour un poids W, LoRA apprend :

$$W' = W + \Delta W, \qquad \Delta W = BA,$$

où $\text{rang}(BA) = r$ avec $r \ll d$. Le backbone est gelé et seuls les paramètres LoRA (A, B) sont entraînés.

2.4 Configuration LoRA (Query/Key/Value)

LoRA est appliqué aux modules d'attention :

`target_modules = ["query","key","value"]`.

Hyperparamètres :

— rang $r = 16$,

— $\alpha = 32$,

— dropout $= 0.05$,

— `bias = "none"`.

Paramètres entraînables. On obtient typiquement $\approx 1.75\%$ de paramètres entraînables (adaptation paramètre-efficiente).

3 Données et prétraitement

3.1 Format des données

Chaque échantillon contient :

— une image RGB,

— un fichier `.npy` de forme $(H, W, 3)$ représentant (X, Y, Z).

La profondeur GT est extraite par $Z =$ `xyz[:,:,2]`.

3.2 Masque de validité et valeurs manquantes

On construit un masque binaire :

$$M(i) = \begin{cases} 1 & \text{si } Z(i) \text{ est valide (non NaN, } Z > 0, \text{ et borné)} \\ 0 & \text{sinon.} \end{cases}$$

Les valeurs invalides sont mises à 0 (ex. `np.nan_to_num`), puis ignorées via M dans la loss.

3.3 Statistiques globales (mm)

Sur l'ensemble du dataset (58 paires), on calcule des bornes globales pour normaliser :

$$z_{\min} \approx 251.74 \text{ mm}, \qquad z_{\max} \approx 3907.45 \text{ mm}.$$

Ces bornes stabilisent l'entraînement et permettent une dénormalisation cohérente en mm.

3.4 Normalisation inverse (focus objets proches)

Pour mieux apprendre les détails proches (pneus), nous utilisons une normalisation inverse :

$$z_{\text{inv}} = \frac{1}{z + \varepsilon}.$$

On normalise ensuite dans $[0, 1]$ via des bornes inverses :

$$z_{\min}^{inv} = \frac{1}{z_{\max}}, \qquad z_{\max}^{inv} = \frac{1}{z_{\min}},$$

$$z_{\text{norm}} = \text{clip}\left(\frac{z_{\text{inv}} - z_{\min}^{inv}}{z_{\max}^{inv} - z_{\min}^{inv}}, 0, 1 \right).$$

Ainsi, les petites profondeurs (objets proches) occupent une plage plus large après normalisation, ce qui améliore la précision locale.

Dénormalisation (pour affichage en mm).

$$z_{\text{inv}} = z_{\min}^{inv} + z_{\text{norm}}(z_{\max}^{inv} - z_{\min}^{inv}), \qquad z \approx \frac{1}{z_{\text{inv}} + \varepsilon}.$$

3.5 Résolution et traitement à la volée

Les données sont traitées à la volée dans `__getitem__` :

— extraction de Z, construction du masque,

— normalisation inverse,

— prétraitement RGB avec une taille imposée **756×1260** (multiple de 14), adaptée au modèle.

3.6 Split train/validation

Le dataset contient 58 paires :

$$N_{\text{train}} = 46, \quad N_{\text{val}} = 12.$$

4 Entraînement et fonction de perte

4.1 Alignement des dimensions (upsampling bicubique)

Le modèle produit une profondeur à une résolution interne différente de la GT. Pour comparer correctement, on redimensionne la prédiction à la taille de la GT :

$$\hat{d} \leftarrow \text{Interpolate}(\hat{d}, \text{size} = H \times W),$$

avec un **upsampling bicubique** (meilleure préservation des courbes/contours que bilinéaire dans notre cas).

4.2 Perte 1 : L1 masquée (précision globale)

Sur les pixels valides ($M = 1$), on calcule :

$$\mathcal{L}_{L1} = \frac{\sum_i M_i \, |\hat{d}_i - d_i|}{\sum_i M_i + \varepsilon}.$$

4.3 Perte 2 : loss de gradient (netteté des bords)

On approxime les gradients par différences finies :

$$\partial_x d = d_{:,1:} - d_{:,:-1}, \qquad \partial_y d = d_{1:,:} - d_{:-1,:}.$$

On applique aussi un masque voisinage (M doit être valide sur les 2 pixels) :

$$M_x = M_{:,1:} M_{:,:-1}, \qquad M_y = M_{1:,:} M_{:-1,:}.$$

Pour accentuer les **bords**, on pondère les zones où le gradient GT est fort :

$$w_x = 1 + \gamma \mathbb{1}\left(|\partial_x d| > \tau\right), \qquad w_y = 1 + \gamma \mathbb{1}\left(|\partial_y d| > \tau\right),$$

avec typiquement $\gamma = 10$ et $\tau = 0.02$ (sur profondeurs normalisées). La loss gradient devient :

$$\mathcal{L}_{grad} = \frac{\sum |\partial_x \hat{d} - \partial_x d| \cdot w_x \cdot M_x}{\sum M_x + \varepsilon} + \frac{\sum |\partial_y \hat{d} - \partial_y d| \cdot w_y \cdot M_y}{\sum M_y + \varepsilon}.$$

4.4 Perte totale

$$\mathcal{L} = \mathcal{L}_{L1} + \lambda \, \mathcal{L}_{grad},$$

avec $\lambda = 3.0$.

4.5 Hyperparamètres principaux

5 Résultats et analyse

5.1 Suivi de l'apprentissage

Nous suivons :

— la **training loss** (L1+grad) pendant l'entraînement,

— la **eval_loss** sur validation à chaque époque,

— et nous conservons automatiquement le meilleur checkpoint (selon `eval_loss`).

TABLE 1 – Paramètres d'entraînement (configuration finale)

Paramètre	Valeur
Époques	15
Batch size (par device)	1 (haute résolution)
Gradient accumulation	8 (batch effectif ≈ 8)
Learning rate	5×10^{-5}
FP16	activé
Éval/Sauvegarde	à chaque époque
Best model	`load_best_model_at_end=True`

5.2 Comparaisons qualitatives

La figure suivante illustre la comparaisons RGB / GT (mm) / prédiction (mm), après dénormalisation inverse. Elles permettent d'évaluer la cohérence globale (plans, reliefs) et la qualité locale (contours/rainures de pneus).

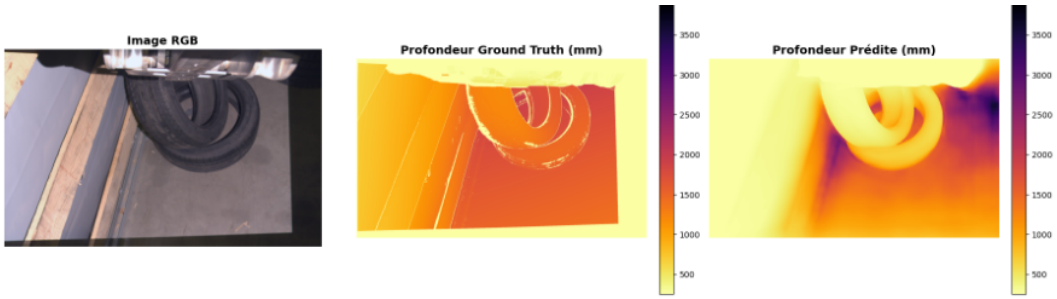

FIGURE 2 – Exemple : comparaison RGB / GT(Z) / prédiction.

5.3 Pourquoi les pneus ne sont pas « parfaits » ? (discussion)

Même avec la loss de gradient, certaines erreurs persistent :

— la GT peut être bruitée / incomplète (zones invalides, bords instables),

— les pneus (noirs/brillants) sont difficiles : ombres et reflets rendent la profondeur ambiguë en monoculaire,

— l'upsampling reconstruit bien les contours, mais ne recrée pas toujours les micro-détails (très hautes fréquences).

6 Conclusion et perspectives

Nous avons adapté **Depth Anything V2 (Small)** à un dataset *RGB + nuages de points* en exploitant Z comme vérité terrain, avec un pipeline robuste (masque de validité) et une stratégie orientée **détails proches** : **normalisation inverse** + **loss L1/Gradient** + **upsampling bicubique**. LoRA permet une adaptation paramètre-efficiente, compatible avec un dataset de taille limitée.