Rapport TP Deep Learning : Détection d'objet

Adrien ANDRÉ | Baptiste BRUMENT

Table des matières

1 Présentation

1.1 Le problème

Le but de ce tp est de mettre en place des méthodes de Deep Learning permettant de localiser des objets dans une image. Il s'agit de déterminer une boite englobante de l'objet recherché définies par les 4 valeurs suivantes :

— sa largeur $\mathbf{b_w}$,
— sa hauteur $\mathbf{b_h}$,
— les coordonnées $\mathbf{b_x}$ et $\mathbf{b_y}$ de son centre.

Ces valeurs sont illustrées sur la figure 1. Nous souhaitons également déterminer la classe de l'objet présent dans la boite englobante.

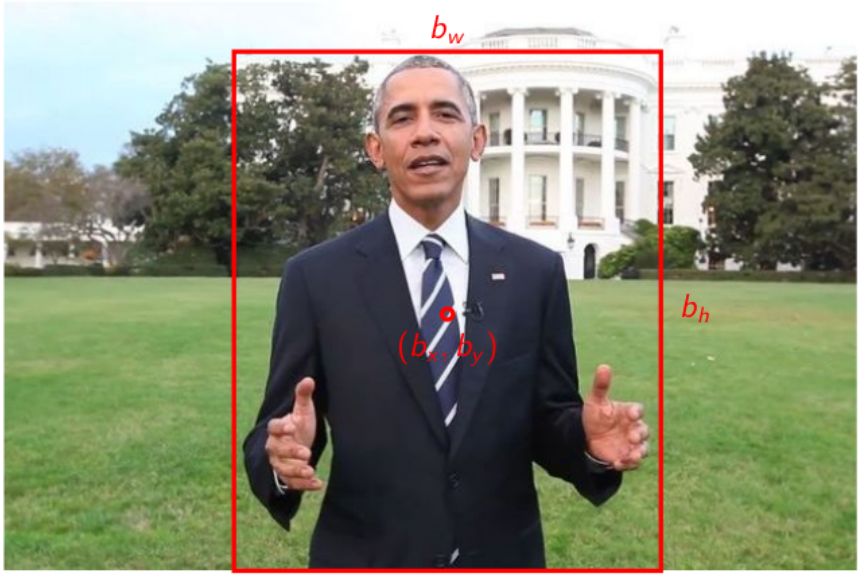


FIGURE 1 – Exemple d'une boite englobante sur Barack Obama

1.2 Dataset utilisé

Le dataset utilisé est issu de kaggle et comporte 376 images pour chacune des 4 classes d'animaux suivants :

— buffle,
— éléphant
— rhinocéros
— zèbre.

Le dataset est donc au total composé de $376 \times 4 = 1504$ images. Il s'agit donc d'un dataset relativement petit. Quelques exemples de ces images sont donnés sur la figure 2.

FIGURE 2 – Exemple d'images du dataset

1.3 Pré-traitement des données

En Deep Learning nous avons besoin de données d'entraînement pour entraîner le modèle, et également de données de validation pour évaluer ses performances et sa capacité de généralisation. Nous avons donc divisé

aléatoirement le dataset de façon à ce que les données d'entraînement représentent 85% du dataset, et celles de validation 15%. Le dataset est ainsi divisé de la façon suivante :

— 1278 images d'entraînement

— 226 images de validation

Les modèles de Deep Learning que nous allons utiliser requièrent une taille d'image fixe. Nous avons choisi une taille de 64×64 qui permet d'entraîner les modèles en des temps raisonnables tout en ayant des résultats intéressants.

En ce qui concerne les labels, chaque image est labellisée par un vecteur $\in \mathbb{R}^9$ définit comme suit :

$$(p \;\; b_x \;\; b_y \;\; b_w \;\; b_h \;\; c_1 \;\; c_2 \;\; c_3 \;\; c_4)^\top \tag{1}$$

où p désigne la probabilité de présence de l'objet (dans le cas de nos labels initiaux on a toujours $p = 1$), b_x, b_y, b_w et b_h désignent les coordonnées de la bounding-box comme expliqué plus haut, et le vecteur $(c_1 \; c_2 \; c_3 \; c_4)$ est un vecteur encodé "one-hot" permettant de déterminer la classe de l'objet. Pour obtenir de meilleurs résultats, les coordonnées des bounding-box ont été centrées-réduites, et les valeurs des couleurs des images ont été normalisées.

2 Problème de localisation

2.1 Hypothèses simplificatrices

Nous avons dans un premier temps simplifié le problème en ne prenant en compte qu'une seule bounding-box par images. Dans le cas où plusieurs animaux sont présents sur l'image, nous gardons celui dont la bounding-box est de plus grande taille. Ainsi, le problème de détection d'objet est transformé en un problème de localisation : on suppose qu'il y a un objet sur l'image et on cherche à le localiser.

2.2 Modèle utilisé

Nous avons appliqué un simple CNN comme décrit sur la figure 3 :

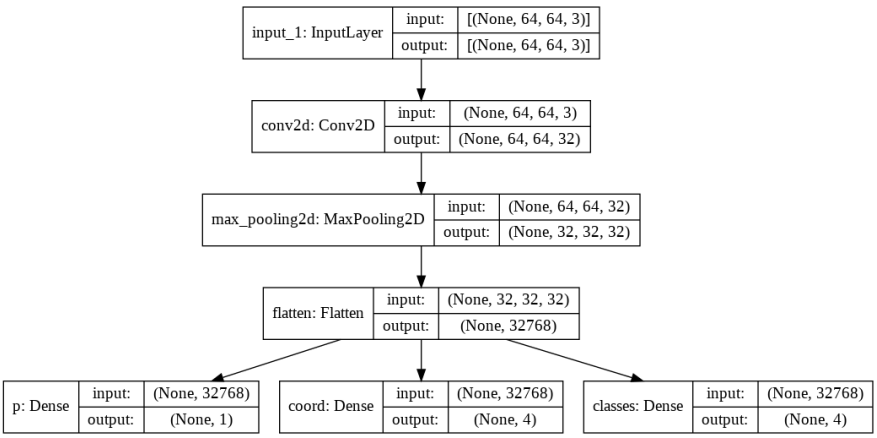


FIGURE 3 – CNN utilisé

Le CNN crée 3 différentes sorties à l'aide de couches `Dense` :

— la première sortie est de taille 1 et correspond au p de l'équation (1)

— la deuxième sortie est de taille 4 et correspond au à la boite englobante $(b_x,\ b_y,\ b_w,\ b_h)$

— la troisième sortie est également de taille 4 et correspond au sous-vecteur $(c_1,\ c_2,\ c_3,\ c_4)$ de l'équation (1).

L'avantage de créer plusieurs sorties est de pouvoir utiliser différentes *loss* pour chacune de ces sorties, ainsi que différents poids sur ces loss. C'est ce que nous allons faire dans la sous-partie suivante.

2.3 Entraînement

2.3.1 Avec des loss usuelles

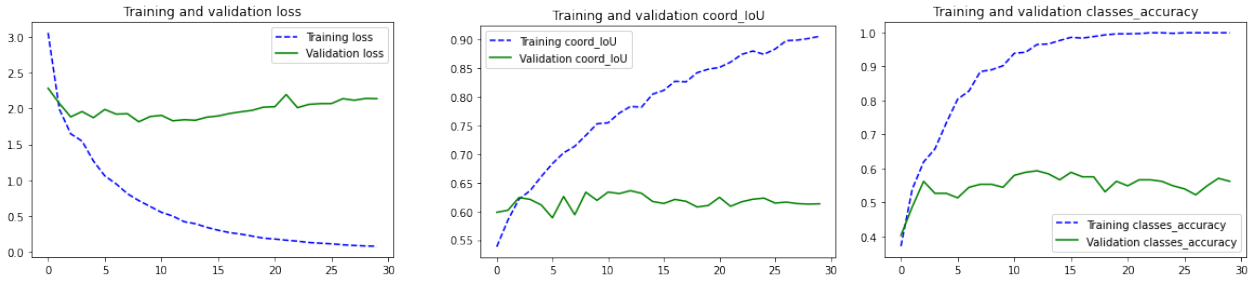
Choix des loss

Notre CNN comprend 3 sorties, nous pouvons subdiviser notre problème de localisation d'objets en 3 sous-problèmes : le premier correspond à un problème de classification binaire, on utilisera donc une `sigmoïde`, le deuxième sous-problème correspond à un problème de régression, une `MSE` semble donc adaptée. Enfin, le dernier sous-problème est un problème de classification multi-classes, ce qui nous incite à utiliser un `softmax` comme loss. Nous avons pondéré ces loss à l'aide de poids.

Évaluation des résultats

Pour évaluer nos résultats nous avons mis en place des metrics pour chacun des trois sous-problèmes décrits précédemment. Pour le sous-problème de détection et pour le sous-problème de classification d'objet (sous-problèmes 1 et 3 précédemment décrits), nous pouvons mettre en place une accuracy, tandis que pour le problème de détermination des coordonnées de la boite englobante, nous pouvons utiliser le coefficient de Jaccard (*"IoU" : Intersection Over Union*). Comme son nom l'indique, ce coefficient se calcule de la façon suivante :

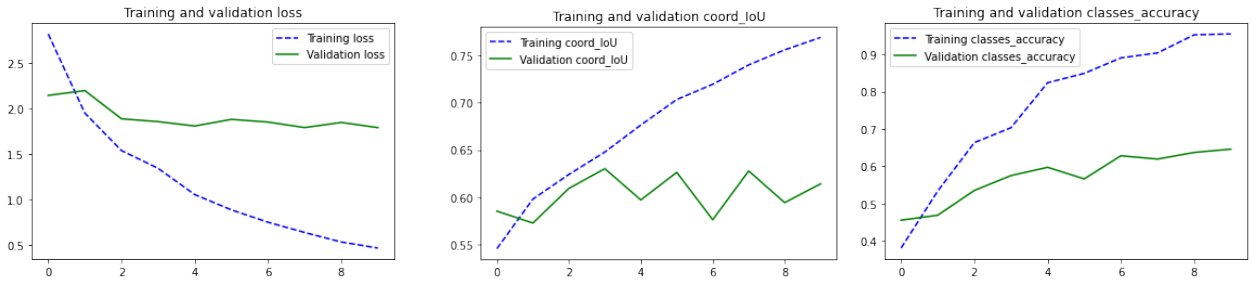
$$\mathbf{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} \tag{2}$$



(a) Loss (b) IoU (c) Classes Accuracy

FIGURE 4 – Courbes d'entraînement du modèle avec 30 epochs

La figure 4 montre différentes courbes nous permettant d'évaluer l'entraînement de notre CNN. On remarque que la loss de validation commence a stagner dès la 5ème epochs puis à remonter à partir de la 10ème epochs, tandis que la loss d'apprentissage continue de baisser. Les courbes des metrics sont quant à elle en stagnation. Tous ces éléments nous indiquent que l'on est en léger sur-apprentissage et que l'on peut fixer le nombre d'epoch à 10. C'est ce que nous avons fait sur la figure 5.

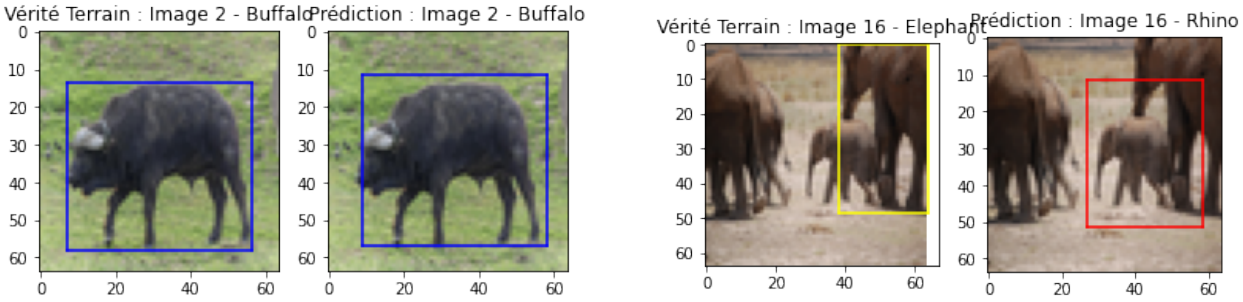


(a) Loss (b) IoU (c) Classes Accuracy

FIGURE 5 – Courbes d'entraînement du modèle avec 10 epochs

Nous pouvons désormais évaluer la précision de nos résultats en les visualisant. Sur la figure 6a nous pouvons voir un exemple de prédiction satisfaisante. La boite englobante a été déterminée de manière très précise ($IoU = 0.88$) et la classe prédite est correcte.
En revanche, la figure 6b montre un exemple de prédiction totalement incorrecte. Cela est dû au fait que plusieurs animaux sont visibles sur l'image, la boite englobante a ainsi été prédite sur le mauvais animal. En effet, lorsque plusieurs animaux sont présent sur l'image, notre modèle a tendance a privilégier l'animal le plus au centre de

l'image et non celui prenant le plus de place sur l'image. De plus, la classe prédite n'est pas bonne, l'éléphanteau a été confondu avec un rhino.



(a) Prédiction correcte

(b) Prédiction incorrecte

FIGURE 6 – Exemple de prédiction

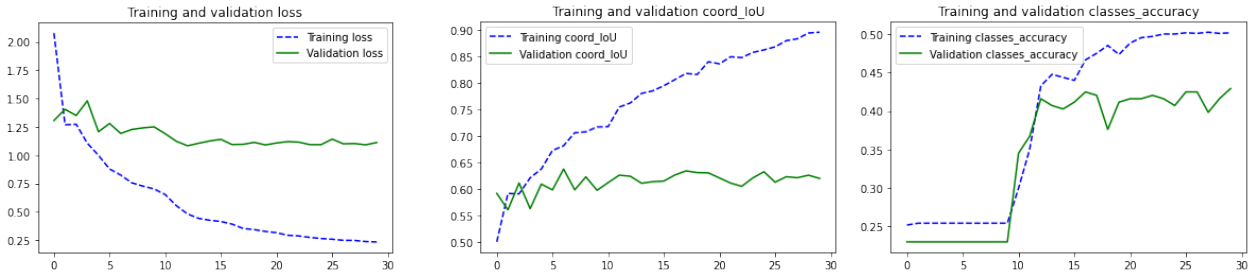
2.3.2 Avec des MSE

Choix des loss

Nous pouvons également tenter d'utiliser des `MSE` sur chacun des 3 sous-problèmes. Les `MSE` étant du même ordre de grandeur pour chacun des sous-problèmes, nous avons fixé les poids de chaque loss à 1.

Entraînement

Comme précédemment, nous pouvons tracer les courbes d'entraînement du modèle. Nous obtenons ainsi les courbes de la figure 7.



(a) Loss

(b) IoU

(c) Classes Accuracy

FIGURE 7 – Courbes d'entraînement du modèle avec des `MSE`

Pour le calcul de l'*IoU*, on utilisait déjà une `MSE` dans la sous partie précédente. Nous obtenons donc une *IoU* de validation similaire à la précédente (≈ 0.65). En revanche l'accuracy sur les classes est moins bonne (0.45 vs 0.6). On remarque d'ailleurs que le modèle sur-apprend énormément. Pour pallier à cela, nous avons crée une séquence permettant de faire de l'augmentation de données. Cette séquence effectue des rotations, des translations selon les axes x et y, ainsi que des "flips" horizontaux. L'utilisation de cette séquence en lieu et place des générateurs d'augmentation de données de `keras` nous a permit d'effectuer les transformations à la fois sur les images et sur leurs labels (les boîtes englobantes). Comme nous l'espérions, l'augmentation de données nous a permit de régler les problèmes de sur-apprentissage. Les courbes d'entraînement de notre modèle sont visibles sur la figure 8.

Nous souhaitons désormais améliorer les performances de localisation de notre réseau. Pour cela, nous pouvons tenter d'utiliser un CNN bien plus complexe et de faire du transfer-learning. On peut par exemple utiliser des gros CNN pré-entrainés sur `ImageNet`. Nous avons choisi d'utiliser `ResNet50` auquel nous avons ajouté une couche `Dense` ainsi que les 3 sorties utilisées précédemment. Notre réseau est visible sur la figure 9. A noter que des réseaux comme `Xception` ou `Inception` auraient également pu être adaptés à notre problème, mais la taille minimale des images demandée par ces réseaux était supérieure à la taille des images que nous utilisons.

Pour l'entraînement, nous avons de nouveau appliqué de l'augmentation de données et nous avons tenté 3 approches différentes :

(a) Loss (b) IoU (c) Classes Accuracy

FIGURE 8 – Courbes d'entraînement avec des `MSE` et de l'augmentation de données

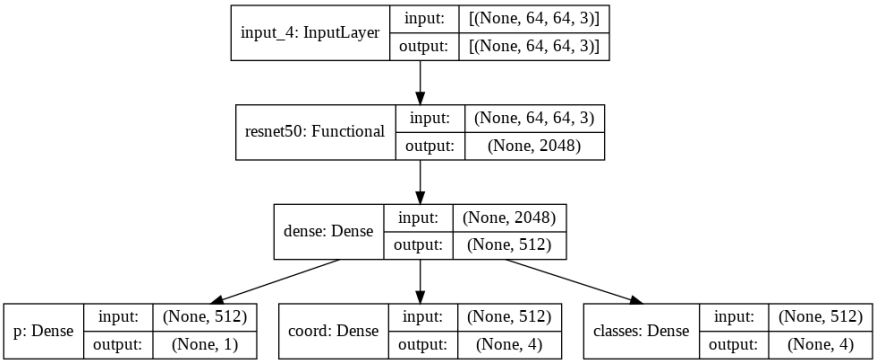
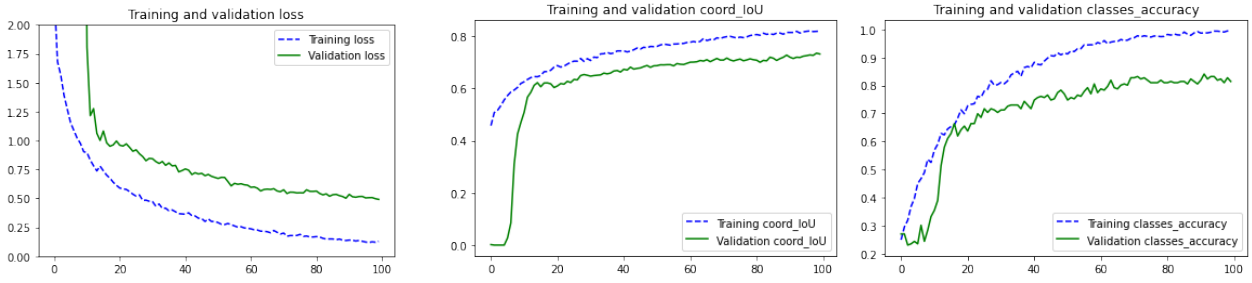


FIGURE 9 – CNN utilisé avec ResNet

— Transfert Learning : utilisation des poids de `ResNet50` pré-entrainé sur ImageNet
— Fine-tuning : entraînement des dernières couches de `ResNet50`
— Entraînement complet de `ResNet50`. On se sert juste de l'initialisation des poids d'ImageNet

Étonnamment, nous avons obtenu les meilleurs résultats en ré-entraînant complètement ResNet50. L'entraînement complet de `ResNet50` ne nous a pas posé de problèmes. En effet, bien que le nombre de paramètres à entraîner soit important (24 millions), comme précisé dans la section 1.3 nous disposons de seulement 1278 images d'entraînement. Le temps d'exécution de chaque epochs reste donc très faible : $4\ sec.$ La figure 10 montre les courbes d'entraînement. Nous ne sommes pas allé au delà de 100 epochs, mais on peut remarquer que le modèle semble toujours apprendre.

L'*IoU* de validation est à 0.74 et est donc bien supérieure à celles que nous avons pu obtenir précédemment. De même, l'accuracy sur les classes est satisfaisante en étant aux alentours de 82%. La figure 11 montre la matrice de confusion des prédictions de classes par notre modèle. On remarque que la principale erreur du modèle concerne la prédiction des éléphants.



(a) Loss (b) IoU (c) Classes Accuracy

FIGURE 10 – Courbes d'entraînement du modèle ResNet50 avec des `MSE` et de l'augmentation de données

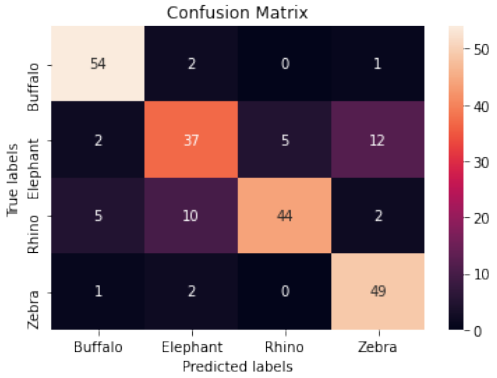


FIGURE 11 – Matrice de confusion obtenue avec ResNet

3 Utilisation de YOLO

3.1 Présentation de YOLO

La méthode précédente nous a permit d'avoir de bons résultats lorsque seulement un animal était présent sur l'image. Nous allons désormais appliquer l'algorithme YOLO développé par Redmon dans le but de pouvoir détecter plusieurs animaux par images. L'algorithme YOLO peut être représenté par le pipeline de la figure 12.

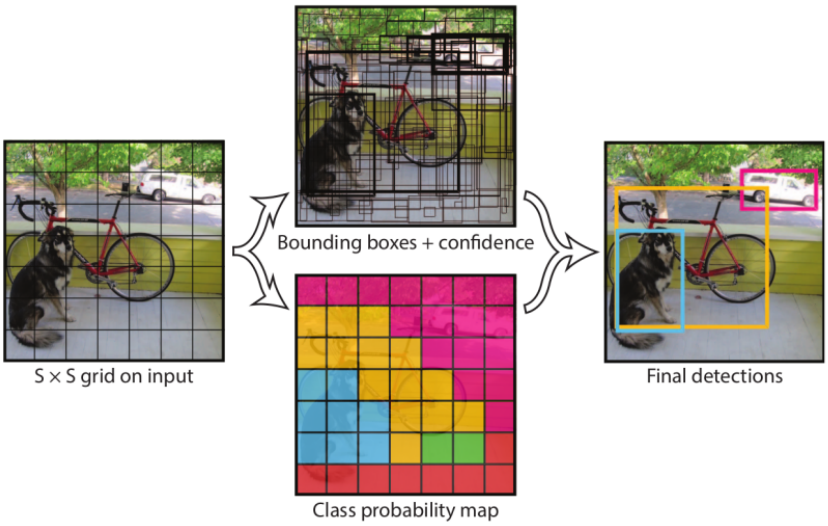


FIGURE 12 – Pipeline de l'algorithme YOLO ([Redmon 2016])

L'idée de YOLO est de découper l'image en une grille de cellules et de réaliser une prédiction de plusieurs boîtes englobantes ainsi qu'une classification par cellule.

La version que nous avons développé est légèrement simplifiée par rapport à la version originale de Redmon. En effet, nous n'utilisons pas le même optimiseur et nous considérons qu'une seule boite englobante peut être détectée par cellules. Nous utilisons toujours des images de tailles 64×64 que nous avons découpée en cellules de tailles 8×8. Chaque images est donc composée de 64 cellules.

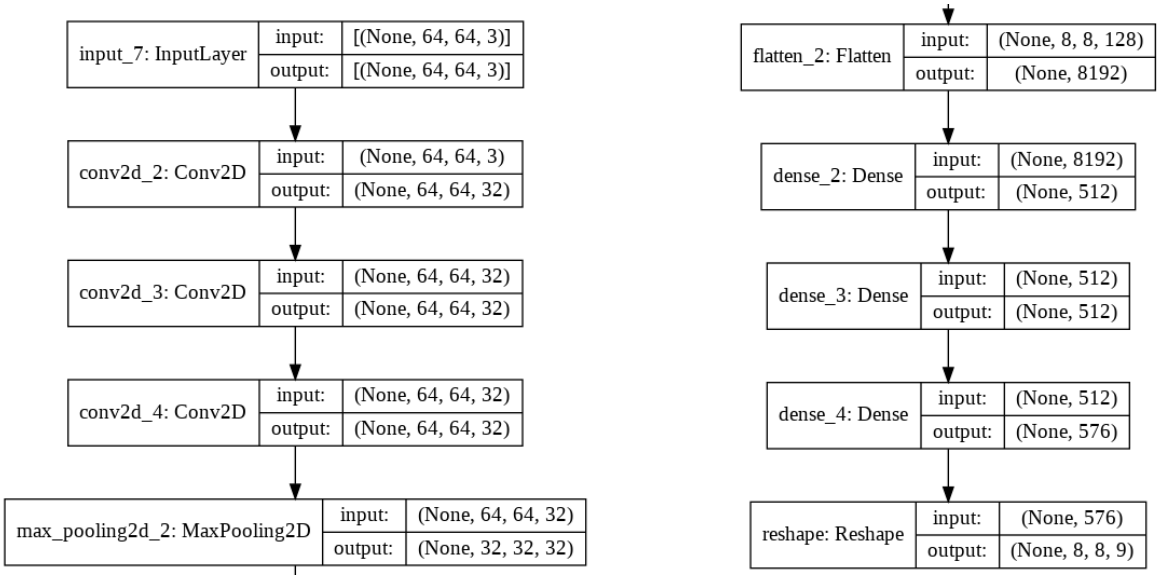
3.2 Création du modèle

Le réseau de neurones de YOLO que nous avons implémenté est composé de 2 parties.
La première partie (voir figure 13a) correspond à 3 blocs composés chacun de 3 couches de *convolutions 2D* ayant un filtre de taille 3 et une profondeur de 32, 64 et 128 (respectivement pour chaque blocs), ainsi qu'une couche de *MaxPooling 2D* dont la "pool size" est de 2. Les fonctions d'activations utilisées sur les couches de convolutions

sont des fonctions `elu`, définies par la formule suivante :

$$elu(x) = \begin{cases} \alpha(e^x - 1) & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \tag{3}$$

La seconde partie du réseau (voir figure 13b) est la partie complètement connectée qui est composée de couches `Dense` ayant elles aussi des fonctions d'activations `elu` (sauf la dernière qui possède une fonction d'activation *linear*). La sortie du réseau est reshape en taille $(8 \times 8 \times 9)$ où le 8×8 correspond aux nombre de cellules dans une image et le 9 correspond aux coordonnées YOLO telles qu'elles ont été définies dans la partie précédente.



(a) Bloc de Convolutions 2D + MaxPooling 2D

(b) Partie complètement connectée

FIGURE 13 – Architecture du réseau de neurone YOLO

3.3 Entraînement

3.3.1 Loss

Afin d'entraîner YOLO, nous avons utilisé la loss définie par Redmon dans le papier YOLOv1. Celle-ci peut s'écrire sous la forme de 3 termes :

$$\text{YOLO\_loss} = \lambda_{coord} \times \text{coord\_loss} + \lambda_{noobj} \times \text{noBox\_loss} + \text{box\_loss} \tag{4}$$

où :

$$\begin{cases} \text{coord\_loss} &= \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\ \text{box\_loss} &= \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \sum_{i=0}^{S^2} \mathbf{1}_{i}^{obj} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \\ \text{noBox\_loss} &= \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \end{cases} \tag{5}$$

Comme l'équation (4) le montre, la loss de YOLO permet de distinguer les données des cellules dites "vides" (i.e ne contenant pas de boite englobante) des "non-vides". Les hyper-paramètres suivants ont été choisis (il s'agit de ceux du papier) : $\lambda_{\text{coord}} = 5$ et $\lambda_{\text{noobj}} = 0.5$

3.3.2 Optimiseur

Pour l'entraînement de YOLO, nous avons utilisé l'optimiseur `Adam` avec un *learning rate* de $3e-5$.

3.3.3 Résultats obtenus

Nous avons entraîné notre modèle sur 100 epochs. L'entraînement est très rapide (1sec/epochs) mais en regardant la courbe d'apprentissage (voir figure 14) on s'aperçoit que le modèle sur-apprend énormément. En effet, à l'issue des 100 epochs, la loss d'entraînement est aux alentours de 15, tandis que la loss de validation reste supérieure à 100.

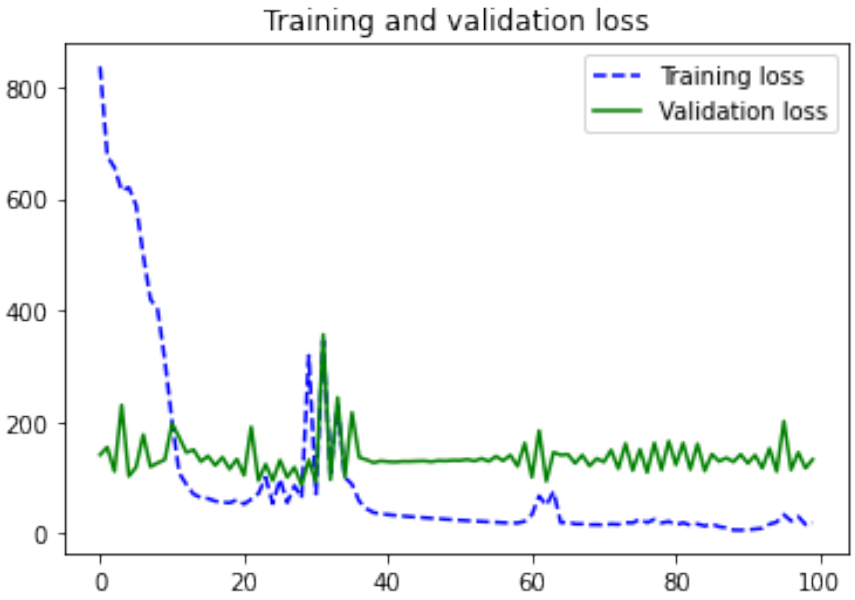


FIGURE 14 – Évolution de la loss du modèle YOLO au cours de l'entraînement

La détection d'objet est donc très performante sur l'ensemble d'apprentissage. On obtient les prédictions de la figure 15. Au niveau de l'*IoU*, on obtient :

— *IoU* d'entraînement : 88%
— *IoU* de validation : 48%

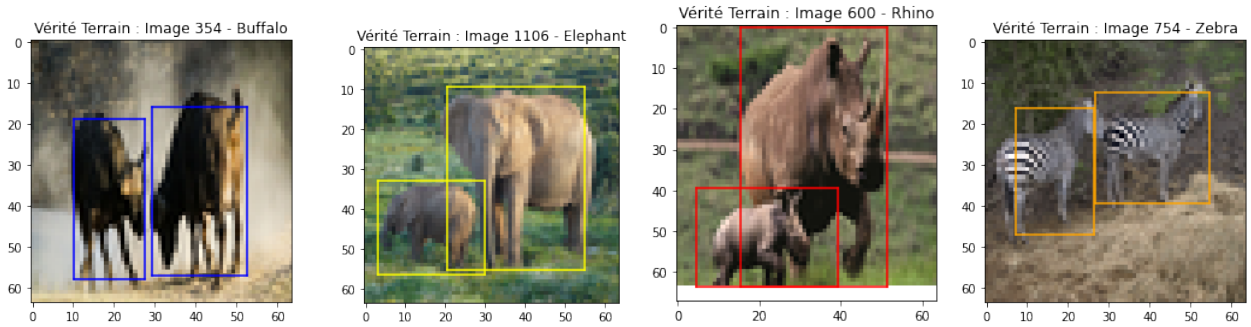


FIGURE 15 – Quelques exemples de prédictions sur l'ensemble d'entraînement

Pour régler le problème du sur-apprentissage, nous avons rajouté de la régularisation L_1 et L_2, ce qui permet de "lisser" la courbe de loss, et ainsi de rendre le modèle plus stable. Néanmoins, cette régularisation n'a pas permit à la loss de validation de baisser. Nous avons donc tenté de faire de l'augmentation de données en utilisant une séquence comme nous l'avions fait dans la partie précédente. Nous avons ainsi appliqué des translations et des rotations sur les images. La principales difficulté a été de changer les labels de cellules lorsque la translation et/ou la rotation les décalaient sur une cellule voisine. L'augmentation de données n'a pas donnée de meilleures résultats, nous en avons donc déduits que celle-ci était mal effectuée.

Notre dernière tentative a été de réduire le learning rate à $6e{-}5$ et de diminuer la batch size. Nous obtenons alors une courbe de loss plus satisfaisante et l'*IoU* montre que le sur-apprentissage à légèrement diminuer :

— *IoU* d'entraînement : 79%
— *IoU* de validation : 57%

4 Conclusion

Pour conclure, ce TP nous a permit d'implémenter des méthodes permettant de résoudre des problèmes de localisation et de détection d'objet dans une image, ce qui nous a ainsi permis de voir que ce dernier était un problème bien plus complexe. Nous avons également pu constater l'importance d'avoir un grand nombre de données annotées dans le but d'avoir de bonnes performances, et nous avons ainsi pu tester différentes méthodes permettant de palier à ce problème.