**Abstract**

Le but de notre projet est d’implémenter les différentes versions du modèle YOLO appliqué à la détection d’images afin de mesurer la performance et la précision de chacune pour ensuite les comparer entre elles. Le modèle a été implémenté sur Pytorch et entraîné sur la base de données PascalVOC.

Les trois versions sont respectivement fondées sur les articles scientifiques suivants :

* You Only Look Once: Unified, Real-Time Object Detection [1]
* YOLO9000: Better, Faster, Stronger [2]
* YOLOv3: An Incremental Improvement [3]

Pour chacun, une description brève du fonctionnement ou des améliorations qui sont décrits dans ce dernier est présentée. De plus, chacune de celles-ci est complémentée d’une implémentation détaillée du modèle et d’une section d’expérimentation qui tente d’obtenir les meilleurs résultats en les comparant à ceux obtenus avec les modèles précédents.

La référence de chacun ainsi que de toute autre source externe ayant influencé ou aidé nos démarches est fournie à la fin du document.

**1 Introduction**

Antérieurement, les modèles R-CNN étaient le “state of the art” quant à la reconnaissance d’objets. Essentiellement, les propositions de boîtes englobantes de chaque objet d’une image étaient envoyées à un réseau convolutif afin d’en extraire une liste d’attributs pour finalement générer les valeurs des classes. YOLO présente une nouvelle approche: Au lieu de réutiliser des classificateurs en les appliquant au problème de détection, on traite celui-ci comme un problème de régression en subdivisant chaque image en une grille de cellules, chacune ayant la responsabilité de prédire à la fois la présence d’un objet ainsi que sa classe. Les sections qui suivent présentent les nombreuses avancées qui ont été faites depuis le papier qui a originalement introduit cette architecture.

**2 Approche Théorique**

**2.1 YOLOv1**

L’article “You Only Look Once: Unified, Real-Time Object Detection” met en œuvre l’algorithme YOLO pour la détection d’objets. Ce modèle possède sa propre architecture, fonction de perte, jeu de données et utilise plusieurs fonctions externes afin de bien fonctionner. Cette section détaillera toutes les parties cruciales à l’algorithme YOLO.

**2.1.1 Architecture du modèle**

La force principale de l'algorithme Yolo est que celui-ci passe seulement une seule fois sur l'image, alors que les algorithmes précédents devaient passer au travers de l'image plusieurs fois. Le principe fondamental de l'algorithme YOLO est de diviser l'image en S\*S zones de détections. L'image ci-dessous démontre un exemple avec une valeur S de 7.

Figure 1 : Exemple de zone de détections

Par la suite, chaque zone de détection est responsable de créer un certain nombre B de *bounding boxes* et d'accorder une probabilité qu'il s'agisse d'un objet à ces boîtes. Chaque boîte est définie par sa position (x,y) relative à la zone de détection ainsi que sa largeur et hauteur, aussi relatives à la zone de détection. De ce fait, la zone de détection qui est responsable de la détection d'un objet est celle dont le centre de l'objet (centre de la bounding box) est dans la zone de détection. Nous pouvons voir un exemple ci-dessous avec toutes les bounding box générées par les zones de détections.

Figure 2 : Exemple de génération d’encadrements

Finalement, nous avons recours à l’algorithme non-max suppression dans le but de choisir et garder seulement les meilleurs bounding boxes. Le but de cet algorithme est de supprimer les boîtes englobantes qui ont un taux de confiance plus bas qu’un certain seuil et de supprimer les encadrements redondants autour d’un même objet détecté.. En utilisant l'exemple précédent, nous obtenons maintenant une bonne détection d'objets.

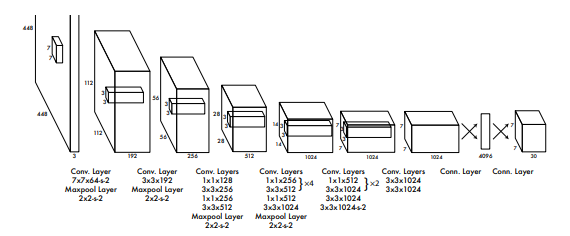
Figure 3 : Exemple de non-max suppression



L'algorithme YOLOv1 contient cependant quelques limitations. En effet, seulement un objet peut être identifié par chaque zone de détection, ce qui implique que si deux objets se trouvent dans la même zone de détection, seul un des deux sera identifié. Cela pose un problème lors de la détection de petits objets qui sont très rapprochés ou d'objets superposés. Ce problème est d'ailleurs réglé par les prochaines versions de YOLO qui permettront au zones de détecter plusieurs objets différents.

Pour l'architecture du modèle, nous allons reproduire l'architecture qui est proposée dans l'article de recherche et que nous pouvons voir dans la figure 4. Nous allons donc avoir 24 couches convolutionnelles suivies de deux couches complètement connectées. Dans l'article, une couche convolutionnelle contient donc simplement une convolution en deux dimensions suivie d'une fonction d'activation, nous avons décidé d'ajouter de la batch normalization avant l'activation, car sans cela notre modèle avait énormément de problèmes lors de l'entraînement. Pour la fonction d'activation, nous allons utiliser la même que dans l'article, soit une leaky ReLU avec paramètre 0.1.

Figure 4 : Architecture YOLO



La sortie du modèle est donc un tensor avec les dimensions suivante : (S, S, C+B\*5), où les deux premières dimensions représentent les S\*S zones de détections, C le nombre de classes et B le nombre de boîtes générées. Pour chaque zone de détection, nous avons donc un vecteur de la forme où :

: 20 classes possibles du Pascal VOC dataset

: probabilité (taux de confiance) que la bounding box i soit correcte

: position en x du centre de la bounding box relative à la zone de détection (entre 0.0 et 1.0)

: position en y du centre de la bounding box relative à la zone de détection (entre 0.0 et 1.0)

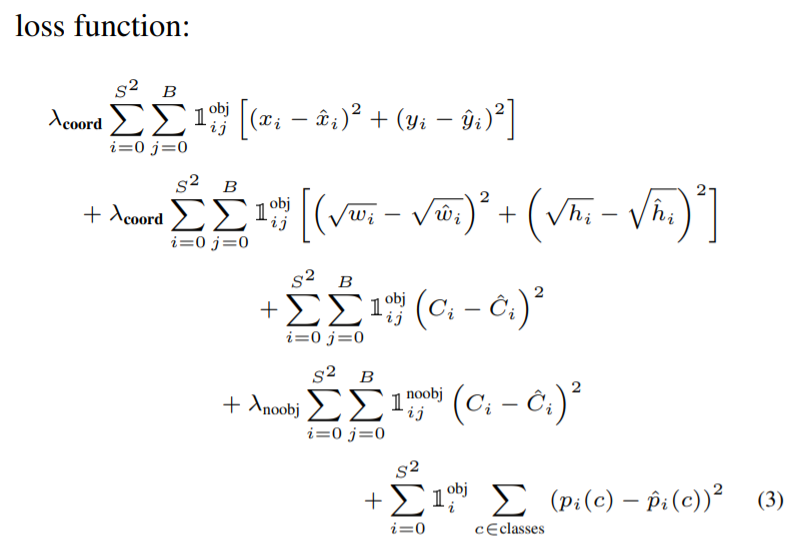
: largeur de la bounding box relative à la zone de détection, donc une valeur de 1.0 est égale à la largeur de la zone de détection.

: hauteur de la bounding box relative à la zone de détection, donc une valeur de 1.0 est égale à la largeur de la zone de détection.

**2.1.2 Fonction de Perte**

La fonction de perte utilisée est la somme des erreurs au carré. Dans le cadre de YOLO, l'article utilise une fonction de perte très complexe que nous pouvons voir ici :

Figure 5 : Fonction de perte complète YOLO

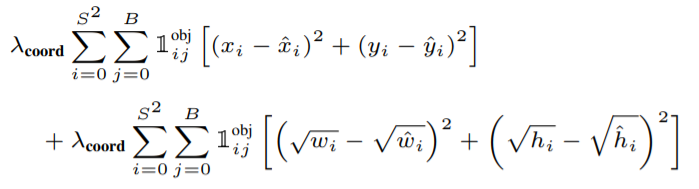


Cette fonction contient deux hyperparamètres, et qui seront initialisés par défaut à 5 et 0.5 respectivement, puisque ce sont les valeurs utilisées dans l'article. Ces hyperparamètres sont utilisés pour augmenter ou diminuer l'impact de certains membres de la fonction de perte.

Tout d'abord, la fonction de perte prend une seule boîte de prédiction alors que nous en générons B=2 lors de nos prédictions. Pour choisir quelle boîte nous allons utiliser comme prédiction, nous allons choisir celle qui contient la plus grande valeur d'*intersection over union* (IoU) avec la boîte cible.

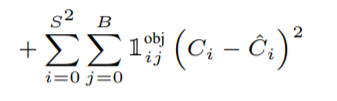
Nous pouvons ensuite séparer la fonction de perte en quatre parties, représentant quatre fonctions de perte distinctes. La partie un est une fonction de perte sur la localisation de la boîte, les parties deux et trois sur la confiance de la boîte, et la partie quatre sur la classification.

Figure 6 : Fonction de perte, partie 1



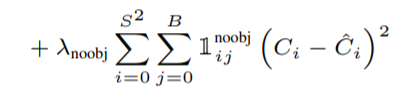
La première partie contient deux termes et calcule la perte par rapport à la position et les dimensions des boîtes. Le premier terme est fonction de la position (x,y) du centre de la boîte, tandis que que le second terme est fonction des dimensions de hauteur et largeur de la boîte. La somme des erreurs carrées est calculée pour chacune des S\*S zones de détection, et multipliée par 1 s'il y a un objet dans cette zone, ou 0 autrement. Le but est de ne pas pénaliser une zone lorsqu'il n'y a pas d'objet à détecter. Ce sera plutôt la troisième partie de la fonction de perte qui s'occupera de cela par la suite. Finalement, on multiplie cette partie de la fonction de perte par l'hyperparamètre afin d’augmenter son impact par rapport au reste de la fonction de perte.

Figure 7 : Fonction de perte, partie 2



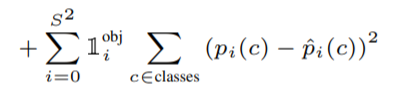
La deuxième partie de la fonction de perte est une fonction de perte pour la confiance vis-à-vis du fait qu'il y a bien un objet dans la zone de détection, elle est donc en fonction du paramètre C. Il s'agit donc encore d'une somme des erreurs au carré, mais cette fois-ci en fonction du taux de confiance. Tout comme la partie précédente, la somme des erreurs carrées est calculée pour chacune des S\*S zones de détection et multipliée par 1 s'il y a un objet dans cette zone ou 0 autrement. Puisque nous regardons seulement les zones avec un objet, la valeur cible de C est donc de 1.

Figure 8 : Fonction de perte, partie 3



La troisième partie est très similaire à la deuxième, mais cette fois pour les zones de détection qui ne contiennent pas d'objet. La valeur cible de C est donc 0 plutôt que 1. Cette partie de la fonction de perte sert à pénaliser le modèle lorsqu'il prédit un objet lorsqu'il n'y en a pas. Comme mentionné précédemment, tout cela est multiplié par l'hyper paramètre pour diminuer un peu son impact par rapport au reste de la fonction de perte.

Figure 9 : Fonction de perte, partie 4



La dernière partie de la fonction de perte est une somme des erreurs carrées sur une classification. Il s'agit donc sensiblement de la même fonction que nous avons utilisée lors des deux premiers laboratoires, mais calculée par rapport à chaque zone de détection qui contient un objet.

**2.1.3 Jeu de donnée**

Le modèle YOLO traite une image comme une grille de cellules. Un objet va donc être détecté par une cellule spécifique lorsque son point milieu se trouve au sein de celle-ci. Le but de ce module est donc de convertir les données de PascalVOC en un format facilement utilisable par YOLO.

L'accès à tout élément de la base de données choisie nous donne une paire (image, label). L'image devra être transformée en tenseur de pixels 448 x 448, tel qu'indiqué par le papier. Le modèle va "*downscale*" cet objet par un facteur de 32 afin de générer un "*feature map*" 14 x 14.

Un label décrit un objet détecté dans une image et est représenté par un dictionnaire dans le code qui suit. On y trouve notamment la taille de l'image et la liste d'objets détectés, chacun indiquant la classe et les dimensions de l'objet en question.

Il est important de noter que les valeurs numériques d'un label sont toutes relatives à l'image au complet. YOLO nécessite, pour chaque cellule qui contient un objet, que les valeurs numériques de cet objet soient relatives à la cellule plutôt qu'à l'image complète.

Soit S la variable qui définit la taille de la grille. Nous voulons générer une matrice S x S x 25 afin de stocker ces valeurs. la troisième dimension correspond au label d'une cellule:

label [0, 19] : Chaque index correspond à une classe.

label [20] : Probabilité que la cellule contienne un objet.

label [21, 24] : Variables x, y, w et h à l'échelle de la cellule.

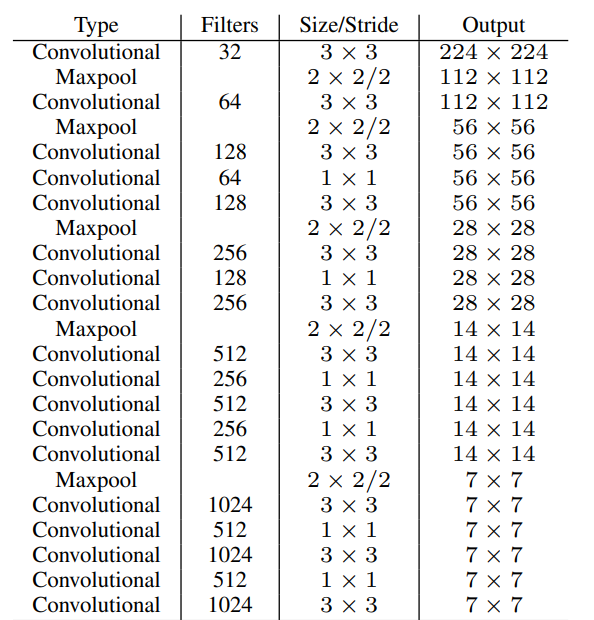
**2.2 YOLOv2**

L’article “YOLO9000: Better, Faster, Stronger” introduit le concept de boîtes d'ancrage et la détection de plusieurs objets par une seule zone de détection. Cette section abordera les principales différences entre les deux premières version de YOLO ainsi que les détails sur son implémentation

**2.2.1 Architecture du modèle**

La plus grande différence entre YoloV1 et YoloV2 est la présence des Anchor Boxes. Cela dit, cela n'affecte pas vraiment l'architecture du modèle. La différence principale concernant le modèle est qu'il n'y a plus présence de couches complètement connectées, il y a seulement utilisation de couches convolutionnelles. L'architecture principale du modèle est donc la suivante :

Figure 10 :Architecture YOLOv2

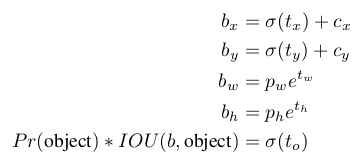


Cette partie de l'architecture est connue Darknet-19. Il est important de mentionner que la section output démontre les dimensions des outputs pour l'utilisation d'une image 224x224, mais que nous allons plutôt utiliser des images de dimensions 416 x 416. Cependant, cette partie de l'architecture ne fournit pas les outputs de dimension désirée. Dans le but d'utiliser le modèle pour la classification d'images, il faut rajouter une convolution 1x1 suivie d'un average pooling et d'une couche de softmax. Dans le cas de la détection d'objets, nous voulons plutôt ajouter une convolution 3x3 avec 1024 filtres suivie d'une convolution 1x1 avec 125 filtres, ce qui nous donne une sortie de 13x13x125. Les deux premières dimensions (13x13) représentent les zones de détection alors que la dernière détection représente la détection de cette zone. La valeur de 125 provient du fait qu'une prédiction pour une boîte englobante est de 25 paramètres, soit 20 pour la classification et 5 pour la confiance, x, y, w et h.

**2.2.2 Fonction de perte**

La fonction de perte pour YoloV2 est très similaire à celle de YoloV1. En effet, elle calcule quatre valeurs de perte différentes : une pour la classification, une pour la présence d'objet, une pour l'absence d'objet et une dernière dédiée aux dimensions des boîtes englobantes. Cependant, la forme des prédictions est légèrement différente et doit être prise en compte. Avec l'utilisation des anchor boxes, nous devons utiliser les formules ci-dessous :

Figure 11 : Formule des boîtes ancrages



Ces formules servent à obtenir la position et les dimensions de la boîte englobante en fonction des sorties du modèle et des boîtes d’ancrage. Les variables t sont les sorties du modèles, les variables c et p sont les données des boîtes d’ancrage et les variables b les dimensions et positions de la boîte englobante résultante. Il est intéressant de remarquer que la fonction sigmoïde s'assure que le point milieu (x, y) des boîtes est bel et bien au sein de la cellule correspondante en délimitant leur valeur à l'échelle [0, 1]. De plus, le paramètre p permet de calculer les dimensions (w, h) en fonction des dimensions d'ancrage précédentes, affectant ainsi les données en considérant les informations précédemment encodées dans le modèle.

**2.2.3 Jeu de donnée**

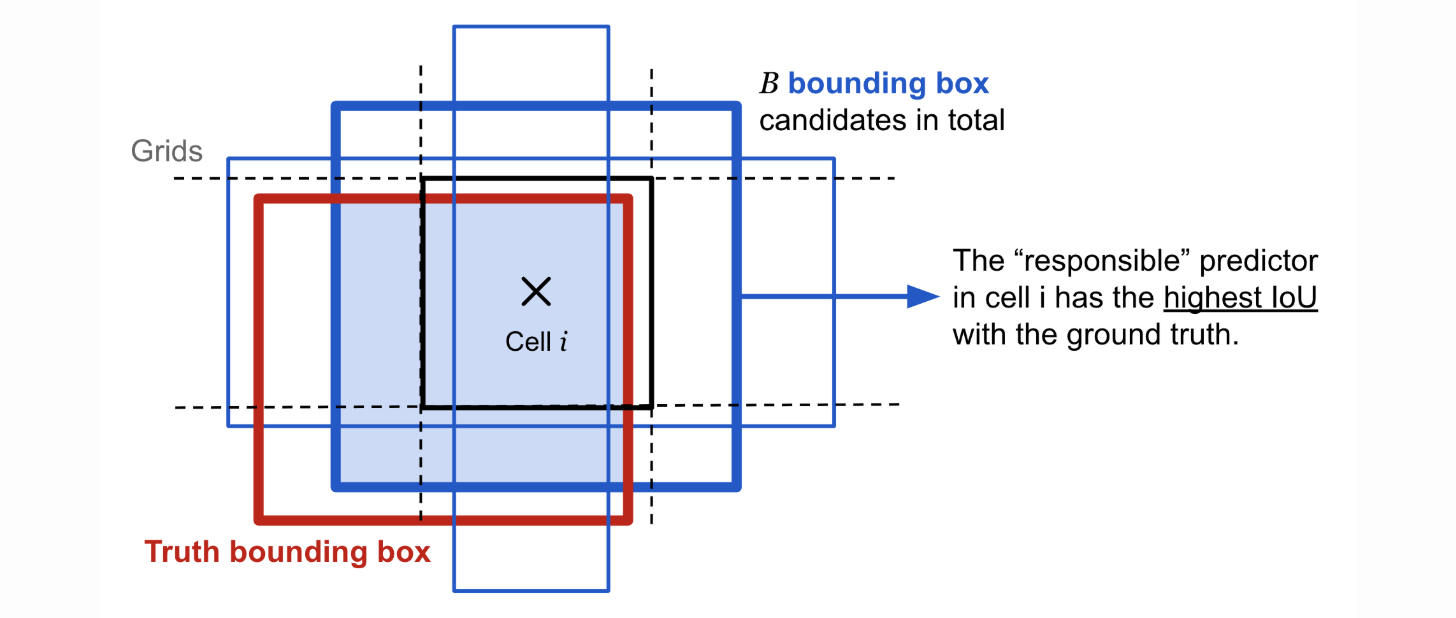
YOLOv2 introduit le concept des Anchor Boxes dans l'algorithme. En assignant 5 boîtes d'ancrage pour chaque cellule de la grille S x S, on génère désormais une prédiction sur la probabilité de classe et d'objet pour chaque boîte au lieu de chaque cellule, ce qui permet deux choses :

-On peut maintenant détecter plus d'un objet par cellule (maximum de 5 dans le papier).

-Chaque boîte d'ancrage a une dimension prédéfinie. Celle-ci modifie les sorties du modèle à chaque fois que la boîte correspondante est identifiée comme responsable du bounding box d'un objet détecté. Ceci permet au modèle de reconnaître plus rapidement un certain objet puisque ses anchor boxes détiennent des connaissances antérieures qui aident à prédire les formes les plus fréquentes qui sont identifiées.

Ce faisant, on modifie le jeu de donné pour trouver la meilleure boîte d’ancrage pour chaque bounding box d'une cellule si celle-ci contient un objet. Pour ce faire, on calcule le IoU de chaque anchor box avec chaque bounding box et on prend les paires avec le meilleur score.

Figure 12 : Définition des boîtes ancrages

De plus, on utilise une résolution de 416 x 416 afin d'obtenir une image 13 x 13 lorsque les CNN de YOLO sous-échantillonnent celle-ci par un facteur de 32. Les auteurs du papier créditent ce changement pour avoir amélioré la performance de l'algorithme puisque le fait d'avoir une cellule en plein milieu de l'image permet de mieux reconnaître les gros objets qui ont tendance à occuper le centre de l'image.

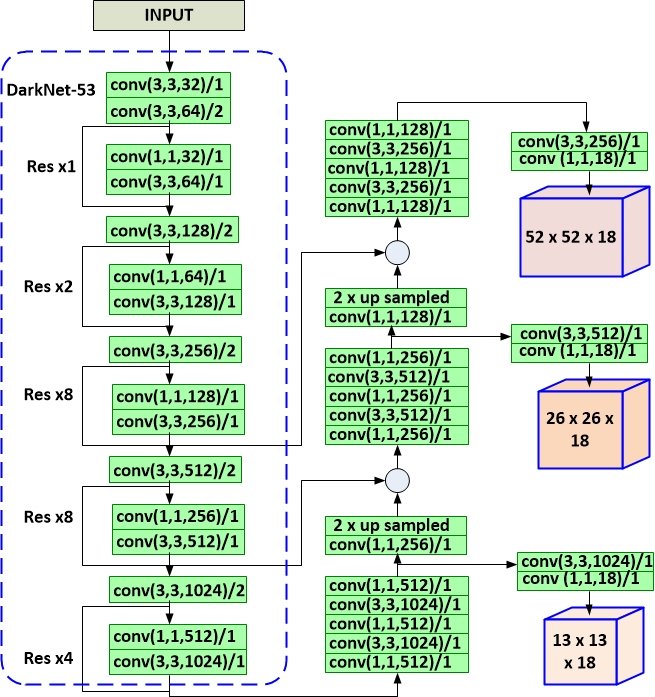
**2.3 YOLOv3**

L’article “YOLOv3 : an Incremental Improvement” introduit la dernière version de YOLO que nous avons étudiée. Cette section présentera donc les détails sur son implémentation, ainsi que les quelques différences qu’il existe entre YOLOv3 et ses prédécesseurs.

**2.3.1 Architecture du modèle**

De la même manière que pour YOLOv2, YOLOv3 prédit des boîtes d’encadrement sous la forme de boîtes d’ancrage, et n’apporte donc aucune nouveauté à ce niveau-là. La plus grande différence se situe au moment de la prédiction de classe. En effet, les modèles précédents effectuaient une évaluation de classe à la fin du réseau. YOLOv3, lui, en effectue trois, et ce à trois échelles différentes. Après la première prédiction, l’idée est de prendre la *feature map* obtenue deux couches plus tôt dans le réseau, l’*upsampler* par deux fois, et la combiner par concaténation avec une *feature map* d’encore plus tôt dans le réseau. Ensuite, il faut encore ajouter quelques couches convolutionnelles pour traiter cette combinaison, et finalement faire une prédiction similaire à la première, qui sera donc deux fois plus grande. Ce design est répété une deuxième fois, et chaque répétition bénéficiera de tous les calculs précédents dans le réseau. Toutes ces opérations sont visibles en détail sur la figure 13, représentant l’architecture du réseau utilisé pour YOLOv3.

Figure 13 : Architecture de YOLOv3

****

On a vu que YOLOv2 reposait sur 19 couches convolutionnelles dans son *feature extractor* (Darknet-19, sur la figure 10), et YOLOv3 quant à lui en utilise 53, d’où le nom de Darknet-53, représenté à gauche sur la figure 13. Comme on peut le voir, YOLOv3 repose sur un réseau globalement plus grand et plus complexe que les versions précédentes, mais jouit alors d’une meilleure précision, et d’une plus grande rapidité d’exécution.

Au-delà d’une plus grande rapidité d’exécution, YOLOv3 est également théoriquement capable de prédire des classification multiples : sur certains Datasets pour l’entraînement -Open Image Dataset en est un exemple-, il est possible pour des objets d’avoir plusieurs labels (un objet peut être identifié comme une personne et une femme en même temps). En utilisant la *non max suppression*, on suppose qu’il n’y aura qu’un seul label par objet, et pour cette raison, YOLOv3 utilise plutôt des classificateurs logistiques indépendants pour chaque classe. Dans notre cas, nous avons simplement utilisé le dataset PascalVOC, dans lequel chacun des objets possède un seul et unique label. Il n’était dès lors pas nécessaire d’effectuer ce changement, et nous avons gardé la fonction de *non max suppression* pour nos expérimentations.

**2.3.2 Fonction de perte**

La fonction de perte utilisée par YOLOv3 est exactement la même que celle employée par YOLOv2, puisqu’il n’y a pas de changement au niveau des boîtes d’ancrage : l’objectif est toujours d’obtenir la position et les dimensions de la boîte englobante en fonction des sorties du modèle et des boîtes d’ancrages.

**2.3.3 Jeu de donnée**

Pour ce qui est du jeu de données, de la même manière que pour les versions précédentes, on récupère une matrice qui contient les informations nécessaires au bon fonctionnement de la fonction de perte. Dans le cas de YOLOv3, cela correspond à associer chaque boîte encadrante à la cellule qui contient son centre, puis déterminer quelle boîte d’ancrage est la plus proche, en réalisant une *intersection over union* entre les deux.

**3 Expérience**

Pour réaliser des expériences sur les différents modèles, nous avons choisi d'utiliser deux datasets différents, Pascal VOC 2007 et Pascal VOC 2012. Nous avons également réalisé différentes expériences avec différents taux d’apprentissage, et réalisé au moins une expérience avec un taux d’apprentissage de 1e-4 pour pouvoir comparer plus facilement les modèles.

**3.1 YOLOv1**

Nous avons réalisé deux expériences sur 150 epochs avec un batch size de 16 et respectivement un taux d’apprentissage de 1e-4 et de 1e-5.

**3.1.1 Pascal VOC 2007**

Figure 14 :Courbe d’apprentissage YOLOv1 sur Pascal VOC 2007

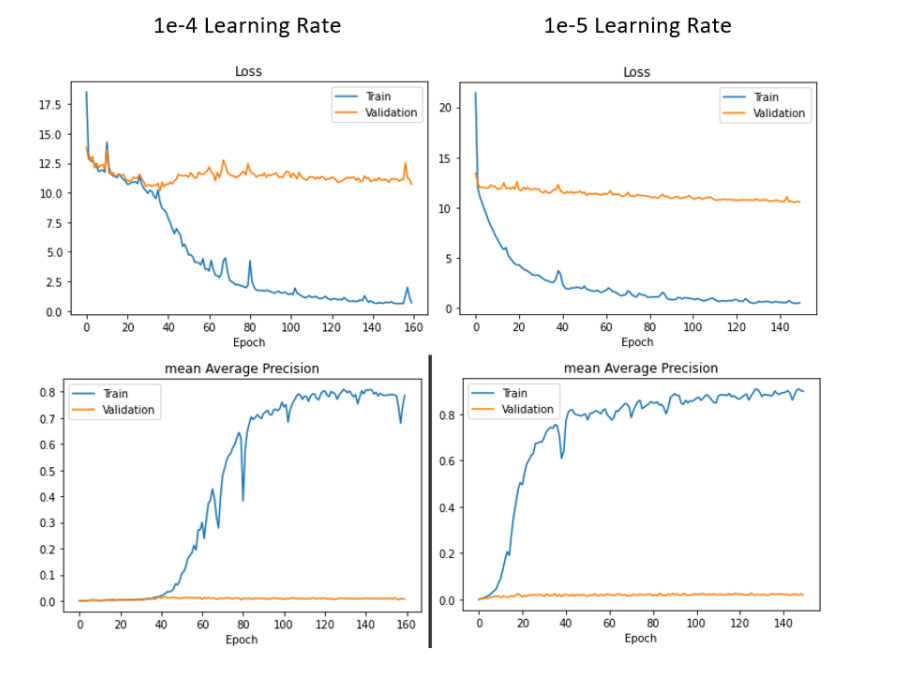
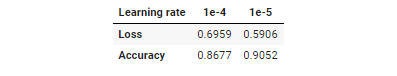


Figure 15: Meilleur résultats YOLOv1 sur Pascal VOC 2007



Une précision de 90% est très bonne, cependant cela est sur l'ensemble d'entraînement. Lorsque nous regardons les graphiques, nous pouvons rapidement voir que le modèle ne performe pas aussi bien sur l'ensemble de validation. Le modèle donne une légère amélioration. Notre première hypothèse était qu'il n'y avait peut-être pas assez de données dans l'ensemble d'entraînement et que le modèle overfit rapidement sur les données d'entraînement.

**3.1.2 Pascal VOC 2012**

Figure 16 : Courbe d’apprentissage YOLOv1 sur Pascal VOC 2012

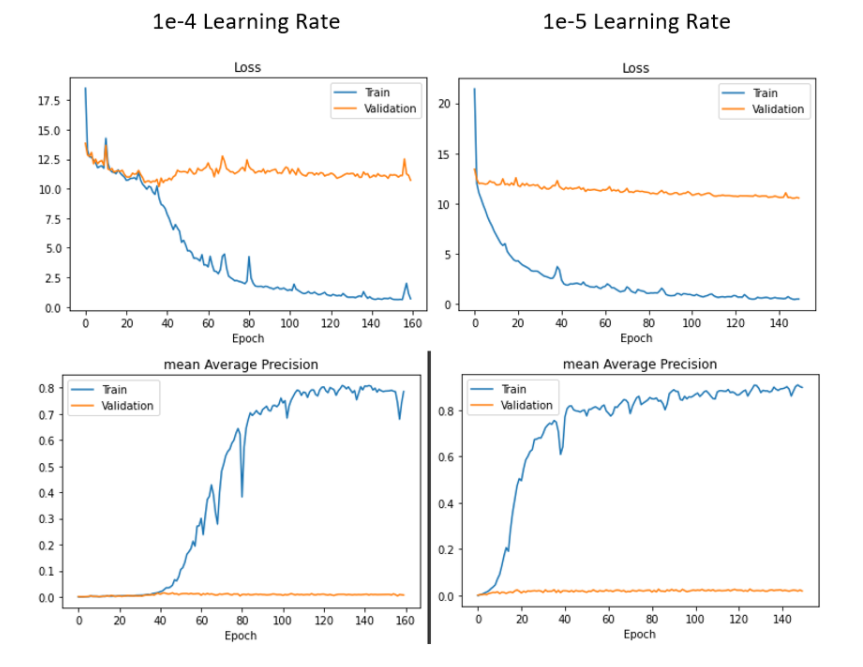
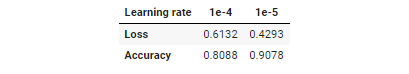
****

Figure 17: Meilleurs résultats YOLOv1 sur Pascal VOC 2012

****

Sur le dataset PascalVOC 2012 on obtient des résultats similaires à PascalVOC 2007.

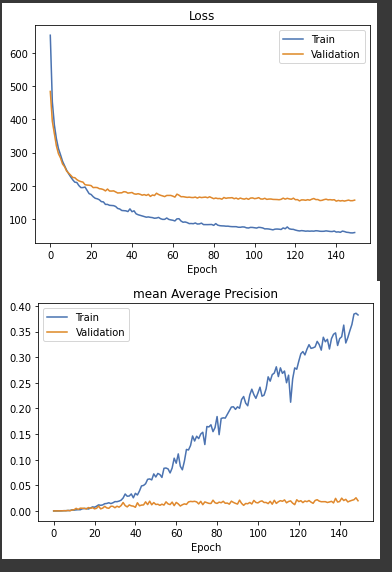
Encore une fois, notre modèle devient vraiment bon pour effectuer des prédictions sur des images qu'il a déjà vues, mais ne performe pas du tout sur de nouvelles images. Nous sommes donc retournés étudier l'article de YOLO pour identifier les étapes prises pour contrer le problème d'overfitting :

"To avoid overfitting we use dropout and extensive data augmentation. A dropout layer with rate = .5 after the first connected layer prevents co-adaptation between layers [18]. For data augmentation we introduce random scaling and translations of up to 20% of the original image size. We also randomly adjust the exposure and saturation of the image by up to a factor of 1.5 in the HSV color space."

Nous avons donc tenté une dernière expérimentation en ajoutant cette couche de dropout juste avant la dernière couche complètement connectée. Nous avons aussi utilisé un taux d'apprentissage de 1e-5 puisque c'est celui qui semblait le mieux fonctionner lors des expériences précédentes

.

Figure 18:Courbe d’apprentissage YOLOv1 avec dropout



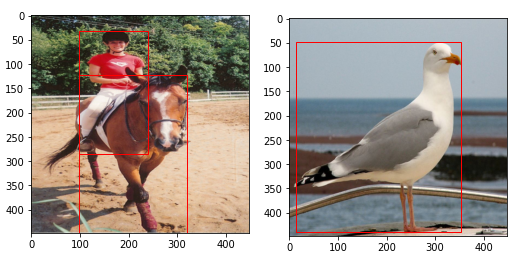
Malheureusement, l'ajout de dropout semble seulement nuire à l'apprentissage du modèle, car en utilisant le même nombre d'epochs que précédemment, nous atteignons des résultats significativement inférieurs.

Nous pensons pouvoir voir quelques causes à ce problème d'overfitting que nous ne sommes pas capable de résoudre. Premièrement le modèle YOLO présenté dans l'article utilise de l'augmentation de données, ce qui aide à réduire le risque d'overfitting. Deuxièmement, dans l'article, le modèle est pré-entraîné pour la classification sur le jeu de données ImageNet. Cette étape de pré-entraînement semble cruciale à l'entraînement du modèle Yolo et l'absence de ce pré-entraînement est peut-être la raison pour laquelle notre modèle ne performe pas aussi bien sur le modèle de validation.

**3.1.3 Détections obtenues**

Pour approfondir et avoir une meilleure idée des résultats avec l'algorithme YOLOv1, nous avons quelques exemples d'images et de détections d'objets qui mettront en valeur les succès et les échecs de notre implémentation. Commençons avec les succès, nous pouvons voir sur la figure ci-après la détection parfaite de l'humain et du cheval dans la première photo et un oiseau dans la deuxième.

Figure 19: Image détection YOLOv1



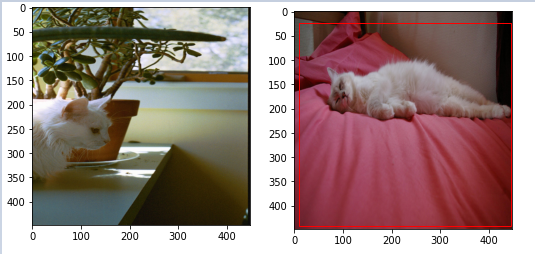
L'exemple ci-dessous est très impressionnant, car nous avons mentionné que la première version de l'algorithme Yolo peut avoir de la difficulté à détecter des objets rapprochés ou de petite taille. L'image ci-dessous présente des humains dans un chariot, l'algorithme réussit à détecter tous les humains ainsi que le cheval, bien que les humains dans le chariot soient rapprochés et que leurs corps soient partiellement cachés.

Figure 20: Image détection concluante YOLOv1



Nous avons aussi deux exemples d'erreurs potentielles de notre modèle. En premier lieu, il n'y a tout simplement aucune détection dans la photo, alors que le chat est très évident. En deuxième lieu, bien que le chat soit correctement identifié, la boîte englobante est de mauvaise taille.

Figure 21: Image détection non concluante YOLOv1



**3.1.4 Temps d'entraînement et temps d’inférence**

L'entraînement sur PascalVOC 2007 a pris 53.5 secondes par epoch, donc 2h 13min 45sec pour 150 epochs L'entraînement sur PascalVOC 2012 a pris 2m 5s par epoch, donc 5h 12min 20s pour 150 epochs. Le temps d'inférence est de 4ms.

**3.2 YOLOv2**

Pour entraîner YOLOv2, nous utilisons une approche similaire à YOLOv1. En effet, nous allons entraîner notre modèle sur les deux jeux de données PascalVOC 2007 et 2012. Pour chacun de ces jeux de données, nous allons utiliser deux taux d'apprentissage. Le premier sera de 1e-4, un taux similaire à celui utilisé avec YOLOv1. Cela nous permettra de faire une comparaison entre les deux modèles. Par la suite, le deuxième taux d'apprentissage sera celui utilisé dans l'article, soit de 10e-3 pour les 60 premières epochs, 10e-4 pour les 40 epochs suivantes et 10e-5 jusqu'à la fin de l'entraînement. Les modèles ont été entraînés sur un total de 160 epochs avec un batch size de 16.

Additionnellement, nous avons ajouté une nouvelle mesure pour la comparaison des modèles. En plus de regarder le Mean Average Precision, nous pouvons maintenant aussi voir les trois mesures suivantes :

* **Class accuracy** : Métrique pour la classification des objets
* **Object accuracy** : pourcentage des objets détectés. Autrement dit, 100% si tous les objets sont détectés. Si un objet est détecté alors qu'il n'y en a pas, cela n'affectera pas cette métrique.
* **No object accuracy** : pourcentage d'absence d'objets détectés. Il s'agit donc de la métrique qui pénalise lorsqu'il est détecté alors qu'il n'y a pas d'objet.

**3.2.1 Pascal VOC 2007**

Figure 22 :Courbe d’apprentissage YOLOv2 PascalVOC 2007

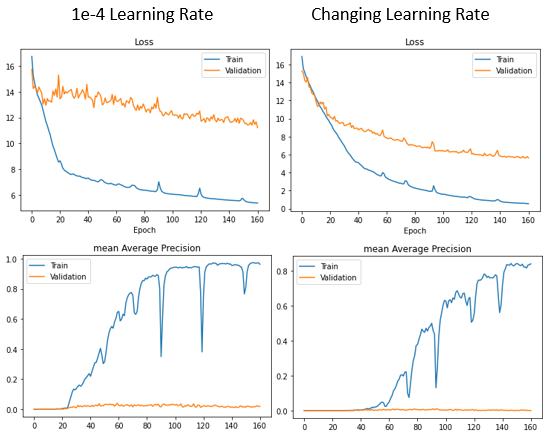
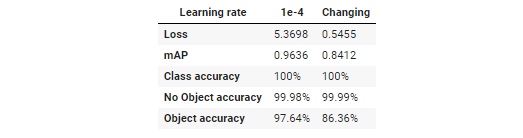
****

Figure 23: Meilleurs résultats YOLOv2 Pascal VOC 2007

****

Nous pouvons donc rapidement remarquer la différence principale entre les deux taux d’accroissement. L'utilisation d'un très petit learning rate nous permet d'obtenir une très bonne valeur mAP et Objet accuracy avec environ 97% dans les deux cas. Au cas contraire, le learning rate de l'article nous permet d'obtenir une valeur de la loss 10 fois plus petite, mais ne performe pas aussi bien que le premier modèle. Cette différence est probablement due au fait que notre loss function priorise grandement le no object accuracy dans le but de ne pas détecter des objets qui n'existent pas. En utilisant un taux d'apprentissage de 10e-3 ou 1e-2 pour les premiers 60 epochs, le modèle minimise extrêmement sa loss function en accordant beaucoup plus d'importance à la no object accuracy plutôt qu'à l'objet accuracy. Selon nous, il serait donc peut-être possible d'obtenir des meilleurs résultats avec ce type de taux d'apprentissage si nous modifions l'importance accordée à chaque partie de la loss function ou si nous entraînons plus longtemps, car l'accuracy continuait à augmenter jusqu'à la tout dernière dernière epoch.

En dehors de cela, nous pouvons remarquer que le Class accuracy est de 100% et le No Object accuracy est de 99.9%. Ce sont d'excellentes valeurs, et nous pouvons donc dire que si un objet est détecté, nous sommes certains à 99.9% qu'il s'agit bien d'un objet et certain à 100% de la classe associée à cet objet. Cependant, cela est seulement vrai pour l'ensemble d'entraînement, nous arrivons encore dans le même problème qu'avec YOLOv1, notre modèle ne performe pas bien sur l'ensemble de validation.

**3.2.2 Pascal VOC 2012**

Figure 24 :Courbe d’apprentissage YOLOv2 sur Pascal VOC 2012

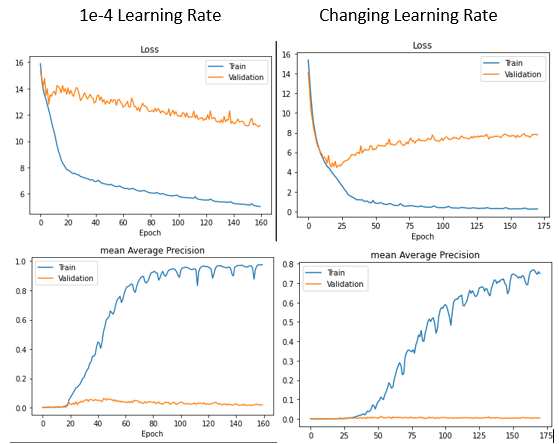
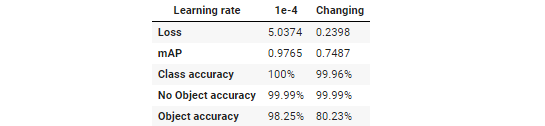
****

Figure 25: Meilleurs résultats YOLOv2 Pascal VOC 2012

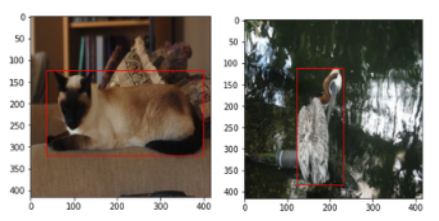
****

L'utilisation d'un taux d'apprentissage de 1e-4 sur le jeu de données de 2012 nous permet donc d'obtenir les meilleures valeurs mAP et object accuracy pour le moment, soit de 97.65% et 98.25% respectivement. Et cela toujours avec des valeurs de no object accuracy et class accuracy presque parfaites. Nous pouvons aussi remarquer que ces modèles ont les mêmes différences que les modèles précédents avec une très grande différence dans la valeur de la loss observé et de très mauvais résultats sur l'ensemble de validation.

**3.2.3 Détections obtenues**

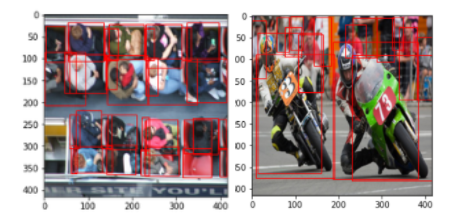
Nous avons à nouveau de la facilité à détecter certains objets correctement lorsque la définition entre celui-ci et son environnement est claire.

Figure 26: Image détection concluante YOLOv2



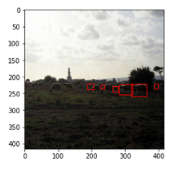
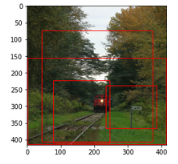
De plus, nous remarquons une aptitude notable de l’algorithme pour les petits objets rapprochés, chose qui valide nos attentes face aux améliorations de YOLOv2.

Figure 27: Image détection concluante YOLOv2



Quelques imperfections sont tout de même notables dans ce contexte spécifique puisque nous observons parfois des boîtes imprécises ou même manquantes, chose qui était présente dans la première version de l’algorithme mais moins fréquente ici.

Figure 28: Image détection non concluante YOLOv2



**3.1.4 Temps d'entraînement et temps d’inférence**

L'entraînement sur PascalVOC 2007 à pris 1m 26s par epoch, donc 3h 49min 20s pour 160 epochs.

L'entraînement sur PascalVOC 2012 à pris 3m 16s par epoch, donc 8h 42min 40s pour 160 epochs. Le temps d'inférence est de 3 ms.

**3.3 YOLOv3**

Dans cette section expérimentation, nous avons décidé d'utiliser une approche similaire à ce que nous avons fait avec YOLOV1 et YOLOV2. L'entraînement du modèle va se faire sur les deux jeux de données Pascal VOC 2007 et 2012. Nous ne présenterons pas de courbe d’apprentissage pour le mAP car le temps d'évaluation et de calcul du mAp est trop important et trop demandant en ressource: il faut compter 43 minutes pour le train set de Pascal VOC 2007.

**3.3.1 Pascal VOC 2007**

Comme précédemment, nous utilisons un batch size de 16, un taux d'apprentissage de 1e-4 pour les 60 premières epochs, puis de 60 à 100 epoch un taux de 1e-5 pour affiner les résultats.

Figure 29 :Courbe d’apprentissage YOLOv3 sur Pascal VOC 2007

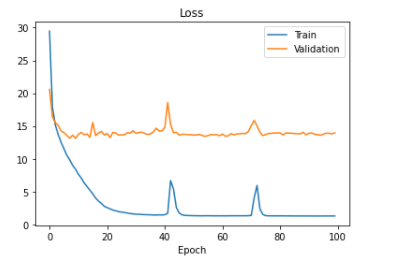
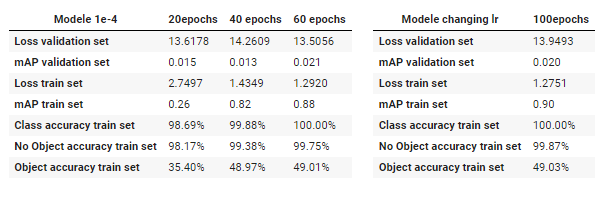


Figure 30: Meilleurs résultats YOLOv3 Pascal VOC 2007



L'utilisation d'un taux d'apprentissage de 1e-4 sur le jeu de données de 2007 nous permet donc d'obtenir une valeur de mAP de 0,88%, avec des valeurs de no object accuracy à 40% et de class accuracy et object accuracy presque parfaites. Comme précédemment, les résultats sur le validation set sont vraiment bas, et on arrive à un palier au bout de 20 epoch seulement, lorsque le map est de 0,015 et la loss de 13.6.

Au bout de 60 epochs on n’observe plus d'amélioration du modèle. Pour essayer d'obtenir de meilleurs résultats, on a poursuivi le training après 60 epochs avec un learning rate de 1e-5.

Cela a permis d'atteindre les 0,90 de map sur le train, de meilleurs résultats pourraient être obtenus en baissant le learning rate et en entraînant un peu plus longtemps.

**3.3.2 Pascal VOC 2012**

Les résultats obtenus sont très similaires à ceux de Pascal VOC 2007, on obtient un mAP de 0,89 pour le set de train. Les résultats sur le set de validation ne sont également pas meilleurs.

Figure 31 :Courbe d’apprentissage YOLOv3 sur Pascal VOC 2012

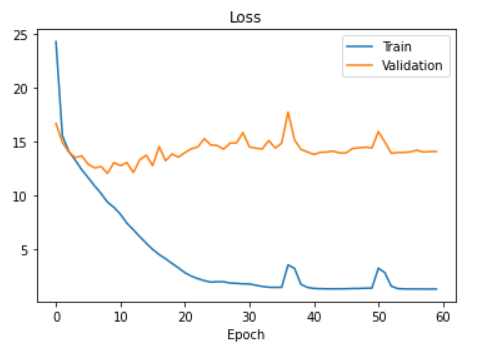
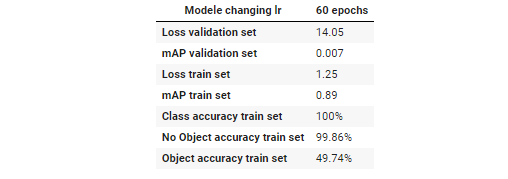


Figure 32: Meilleurs résultats YOLOv3 Pascal VOC 2012



**3.2.3 Détections obtenues**

YOLO V3 semble très à l'aise avec la détection des petits objets rapprochés, comme on peut le voir sur la figure 33 ci-dessous.

Figure 33: Image détection concluante YOLOv3

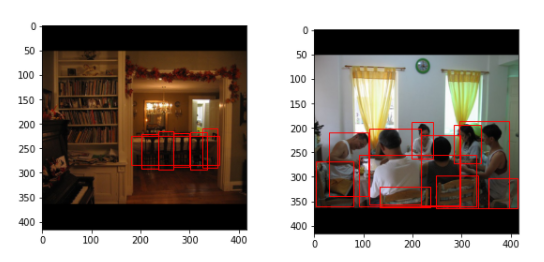
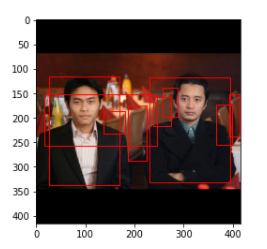


Figure 34: Image détection concluante YOLOv3

On peut également voir que YOLOv3 obtient des résultats concluants pour détecter différents objets qui se trouvent sur des plans différents. Cela est sûrement dû à la détection en trois temps et à trois échelles différentes qui est effectuée par le réseau.

Cependant, quelques erreurs sont encore commises. On peut voir sur l’image de gauche de la figure 35 la multitude d'animaux non détectés, et sur l'image de droite les petites bouteilles du frigo et du rebord de fenêtre ne sont également pas détectées.

Figure 35: Image détection non concluante YOLOv3

****

**3.1.4 Temps d'entraînement et temps d’inférence**

L'entraînement sur PascalVOC 2007 a pris 2m 10s 126ms par epoch, donc 3h 36 min 52s pour 100 epochs. Si on avait réalisé le calcul du mAP cela nous aurait pris 2 jours 23 heures 40 minutes de plus. Pour ce qui est de PascalVOC 2012, l'entraînement a pris 2min 42s par epoch, donc 2 heures 42 minutes pour 60 epochs. Si on avait réalisé le calcul du mAP cela nous aurait pris 5 jours 20 heures 45 minutes de plus. Le temps d'inférence est de 840ms.

**3.4 Comparaison des résultats et critique de l’approche**

Les trois modèles ont le même problème de généralisation et n'obtiennent pas de très bon résultat pour la validation set. YOLOv2 permet d’obtenir les meilleurs résultats de mAP sur le train set, et YOLOv3 à quand même théoriquement -même si nous ne l’avons pas implémenté- l’avantage de pouvoir gérer la classification multiple pour un unique objet. Il permet également une meilleure détection des petits ou très grands objets, ce qui n’est peut-être pas nécessairement mis en valeur par le dataset de Pascal VOC. On aurait pu utiliser d'autres datasets plus spécifiques à la détection d’objets de différente taille par exemple. Cela étant dit, cette efficacité à tout de même un coup, et le réseau qui constitue YOLOv3 est bien plus lourd, et prend donc plus de place mémoire et plus de temps à s'entraîner que YOLOv2. On peut également remarquer que le temps d’inférence est plus rapide pour YOLOv2 que pour YOLOv1, cela est dû à l’utilisation dans v2 de la fonction nms de pytorch.

Il est par ailleurs important de noter que l’outil *Collab pro* a été utilisé pour réaliser les mesure de YOLOV2 et YOLOv1, mais pas pour celles de YOLOv3. Collab Pro serait 1,5 fois plus rapide. Il est intéressant de prendre cette information en compte dans les résultats.

**References**

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. arXiv preprint arXiv:1506.02640, 2015.

[2] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on, pages 6517–6525. IEEE, 2017.

[3] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[4] Aladdin Persson, *YOLOv1 from Scratch*. Oct. 13, 2020. Accessed on: Mai, 28, 2021. [Streaming video]. Available: Youtube.com

[5] Aladdin Persson, *YOLOv3 from Scratch*.Oct. 13, 2020. Accessed on:Mai, 28, 2021. [Streaming video]. Available: Youtube.com

[6]Kamal, A., 2021. YOLO, YOLOv2 and YOLOv3: All You want to know. [Blog] *Amro Kama*, Available at: <https://amrokamal-47691.medium.com/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899> [Accessed 29 April 2021].