Documentation Cardent

Chaoui Aziz Youssef

Sommaire

1. Réalisation du jeu de données.

Le jeu de données à été majoritairement basé sur les deux premières banques d'images en annexe(Damaged Cars Dataset (kaggle.com) , CarDD: A New Dataset for Vision-based Car Damage Detection (cardd-ustc.github.io)), les principaux critères de sélection des images étaient la sévérité du dégât ainsi que l'équilibre des classes. En effet, le modèle est destiné à inférer sur des voitures légèrement endommagées. De plus, il faut veiller à conserver un certain équilibre quantitatif des classes afin de garantir des scores de précisions corrects pour l'ensemble des classes. D'autres banques d'images plus minoritaires seront citées en annexe

2. Annotation des images

Dans un premier temps nous avons utilisé l'outil labelimg pour annoter nos images, c'était satisfaisant au début mais en raison de crash récurrents j'ai choisi de passer sur le site roboflow, de plus le site fournit quelques options de data augmentation ce qui servira pour l'apprentissage. Les annotations sont une grande piste d'amélioration potentielle du modèle, en effet en raison d'un manque de rigueur dans ma méthode d'annotation il est possible que la mise en place d'une stratégie plus rigide et stricte permettra d'améliorer l'apprentissage. Pour nommer un exemple pour les fissures sur un impact on fera parfois le choix d'annoter l'impact en entier et parfois on isolera plusieurs petites fissures de la vitre ou du pare-brise. Une autre stratégie intéressante pour le futur peut être de s'intéresser aux zones de la voitures pour s'assurer de conserver une certaine rigueur d'annotation.

3. Entrainement

Pour l'apprentissage de notre modèle il faut passer par certaines étapes(voir le fichier train.py) tout d'abord il faut s'assurer que l'entraînement soit effectué sur le gpu, c'est pour cela qu'on affecte à la variable device la valeur cuda. Ensuite il faut savoir qu'on entraine un modèle qui à déjà appris, on appelle cela le fine tuning. C'est pour cela que pour le modèle chargé on va désactiver la descente de gradients sur tous les paramètres du modèle inhérents au backbone, les poids seront donc "freezés". On sauvegarde chaque modèle produit à chaque epoch pour notre validation custom via le paramètre save_period=1.

4. Augmentation

Plusieurs méthodes d'augmentation différentes ont été testées. Dans un premier temps j'ai simplement utilisé les options d'augmentation proposées par roboflow mais celles-ci sont assez limitées car on n'aura au maximum que 3 fois plus d'images et que le champ des possibles en termes d'augmentation est relativement limité. Dans un second temps je me suis intéressé aux augmentations fournies directement via le modèle yolo qu'on peut observer dans le fichier train.py. En effet, en observant on voit que des paramètres comme hsv_h sont fixés à 0.3 cela correspond à une probabilité d'application de l'augmentation de saturation de l'image. Cela nous pose problème car nous ne pouvons pas savoir avec certitude quand l'augmentation sera appliquée. Il faut noter qu' utiliser ces paramètres d'augmentation implique une augmentation significative de la durée de l'entraînement. Enfin dans un troisième temps je me suis concentré sur une augmentation plus contrôlée et customiser via la librairie torch.transforms (voir le fichier augment.py) dans ce fichier on créera un custom dataset basé sur notre dataset dans lequel on ajoutera les transformations de notre choix. Il est d'ailleurs important de réfléchir à la pertinence de la transformation

appliquée. En effet, lorsque l'on observe une voiture il est peu probable que l'on ait à observer cette voiture renversée à la verticale (vertical flip) ce ne sera donc pas une augmentation pertinente pour notre modèle, l'horizontal flip sera par exemple plus adapté. Le programme augment.py permettra d' appliquer de la data augmentation sur l'ensemble des images présentes dans le répertoire courant. Cependant le nombre d'images généré est conséquent et l'entraînement sera donc fortement impacté.

5. Méthode de validation.

Yolo fournit un ensemble de métriques pour évaluer la capacité de notre modèle à généraliser (c'est la capacité d'un modèle à fonctionner sur de nouvelles données). Nous avons fait appel à ces métriques et notamment à la matrice de confusion, sauf qu'il se trouve que la façon dont cette matrice est construite et notamment la présence du champ background nous pose problème, de plus nous voulons réaliser un système qui se concentrera sur la détection et non pas la localisation et la position de la bounding box. Nous avons donc fait le choix de réaliser notre propre système d'évaluation de notre modèle. (Ce système est disponible dans l'historique des versions il à été adapté pour notre validation custom). Pour cela on va d'abord inférer sur un ensemble d'images dites de test. Pour ces images, on aura les annotations correctes dites de ground truth contenu dans la variable label_path. Les annotations obtenues après inférence seront contenues dans la var label_infered_path. (Ces deux variables sont en argument de la fonction recup_effectif_label qui permet de récupérer la précision en sortie). Une fois que cela est fait on va parcourir le répertoire ground truth, à chaque dès qu'une classe sera observée dans un fichier d'annotation on incrémente cette valeur, mais attention cette valeur sera incrémentée une seule fois par fichier Si on voit deux rayures dans une seule image le champ comptant l'effectif de la classe sera incrémenté une fois. Ensuite on parcourt les champs de classe inférée et si la classe est observée on augmente la valeur de précision. enfin on finit par diviser cette variable de précision par l'effectif total ce qui nous donne une précision pour chaque classe. Dans le cas ou l'on souhaite simplement tester sur un répertoire d'images il suffit de placer le flag predict a true et de mettre le chemin du répertoire voulu dans la var image_path. Les images annotées seront récupérables dans le répertoire runs/detect/predict

Selon moi la raison principale qui explique ces courbes c'est simplement la taille insuffisante du dataset qui fait qu'entrainer le modèle sur un grand nombre d'epoch génère nécessairement de l'overfitting. Il faudra donc à l'avenir travailler à la fois sur la data augmentation mais aussi sur l'agrandissement du dataset courant.

Par la suite on s'est ensuite intéressé aux courbes de loss sur la validation. Cette métrique est très importante car elle permet d'évaluer la capacité de généralisation d'un modèle. On observait dans ces courbes que la loss augmentait à long terme (voir annexe) alors qu'elle est censée baisser. C'est pour cela qu'on à décidé de mettre en place un système de validation custom basé sur les fichiers validation.py et assess.py ainsi que train.py. Dans un premier temps il faudra demander un entraînement sur model.train avec le paramètre save_period=1 pour sauvegarder le modèle produit à chaque epoch. Une fois que l'entraînement est fini et que l'on à notre centaine d'epoch ou plus, on va demander une validation avec les paramètres suivants, plots= False car les courbes fournies par yolo ne nous intéressent pas , save_txt = True pour conserver les fenêtres d'annotation à l'issue de

chaque validation et conf= 0.2 pour s'assurer que nous ne conservons que des images avec un seuil de confiance supérieur ou égal à 0.2 car le cas contraire nous aurions des fenêtres avec 0.002 de confiance dans nos annotations (cette valeur de 0.2 est empirique on peut surement trouver mieux). Une fois que cela est fait cela va nous générer des répertoire val_i avec i allant de 0 au nombre d'epoch d'entraînement. Dans ce répertoire il y'aura les annotations propres à la validation de chaque modèle. On appellera donc assess.py et on affichera une courbe de la variation de la précision moyenne (moyenne de toute les classes) en fonction des epochs (voir annexe).

6. Interface d'annotation binaire

Afin de s'affranchir du problème de stratégie d'annotation on à réaliser une interface basée sur un modèle de détection de zones de la voiture (par exemple détecter un capot une fenêtre un pare brise). Malheureusement, le modèle (en annexe) sur lequel on souhaitait se baser n'est plus utilisable car il est construit sur un environnement qui est obsolète. On à donc décidé de se baser sur une interface qui se basera sur les inférences de notre modèle courant. Cette interface repose sur les fenêtres englobantes générées lors des prédictions de notre modèle. On affiche tout simplement les images du répertoire choisi lors du lancement de l'interface ainsi que les annotations correspondantes (obtenues en mettant le champ save_txt à true lors de la commande model.predict) à l'image, (les annotations choisies proviennent également d'un répertoire choisi par l'utilisateur). Il faut également veiller à ce que le format d'annotation soit du type: class_id, x_center, y_center, w, h. Cela correspond au format yolo c'est le seul type d'annotation accepté en entrée. Une fois les répertoires choisis une fenêtre apparaît (voir annexe). Elle contiendra une image affichée via la méthode show_image, des fenêtres englobantes obtenues via la méthode draw_annotation et qui sont obtenues via une conversion pour s'adapter à l'affichage de l'interface, un bouton précédent, un bouton dégât présent qui passera à 0 ou 1 la variable new_class_id pour indiquer ou non la présence du dégât sur les fenêtres englobantes concernées. Et enfin un bouton bouton suivant qui va permettre de générer les annotations résultantes dans répertoire annotation_result (qui est créé dans le même répertoire que celui contenant les annotations de départ) et dans lequel seront créés les nouvelles annotations avec le class_id remplacé par le new_class_id puis le nouveau fichier d'annotation généré conservera tous ses champs à l'exception du champ class_id remplacé par new_class_id. Pour améliorer cette interface la principale chose à faire c'est d'isoler chacune des fenêtres englobantes pour indiquer si le dégât est présent dans une certaine fenêtre et non pas sur l'ensemble des fenêtres.

7. Organisation du git

Dans le git certains répertoires n'ont pas pu être ajoutés en raison de leur taille conséquente notamment les répertoire ayant permis de tester notre modèle de validation custom car il contenait une centaine de modèles ce qui pose problème. Pour pouvoir tester la validation custom, lancer un entrainement avec l'argument save_period = 10. On peut retrouver une multitude de fichiers:
test/test_ancien, valid/valid_ancien, train/train_ancien: deux versions du dataset séparées en répertoire pour entrainer l'ia. Pour choisir lequel des deux va permettre d'entrainer l'ia

changer le champ dans le fichier data.yaml. Les meilleurs résultats ont pour l'instant étés obtenus via l'ancien dataset.

runs/detect: contient des fichiers avec des répertoires train et predict. Les répertoires predict contiennent des résultats issus d'inférences de nos modèles. Les répertoires train contiennent quant à eux les modèles (best.pt ou last.pt dans le répertoire weights) ainsi que les courbes correspondantes. Le meilleur modèles se trouve dans le répertoire train25.

On peut également observer quelques courbes comme celles données en annexe.

Le modèle sur lequel va se baser notre fine tuning se nomme yolov8s.pt. Pour utiliser notre validation custom il faudra passer par le fichier assess.py ainsi que validation.py.

Les fichiers val.py et validator.py sont eux les fichiers qui structurent la validation proposée par yolo. Ils peuvent être intéressant dans le cas ou l'on souhaite réaliser une autre forme de validation custom.

Enfin on retrouve le fichier augment.py ainsi que deux exemples d'augmentation obtenues via le fichier utilisé

Annexe

Jeu de données:

[CarDD: A New Dataset for Vision-based Car Damage Detection (cardd-ustc.github.io)]

[Car damage detection (kaggle.com)]

[https://www.kaggle.com/code/lplenka/detectron2-car-damage-detection?scriptVersionId=52171508&cellId=34]

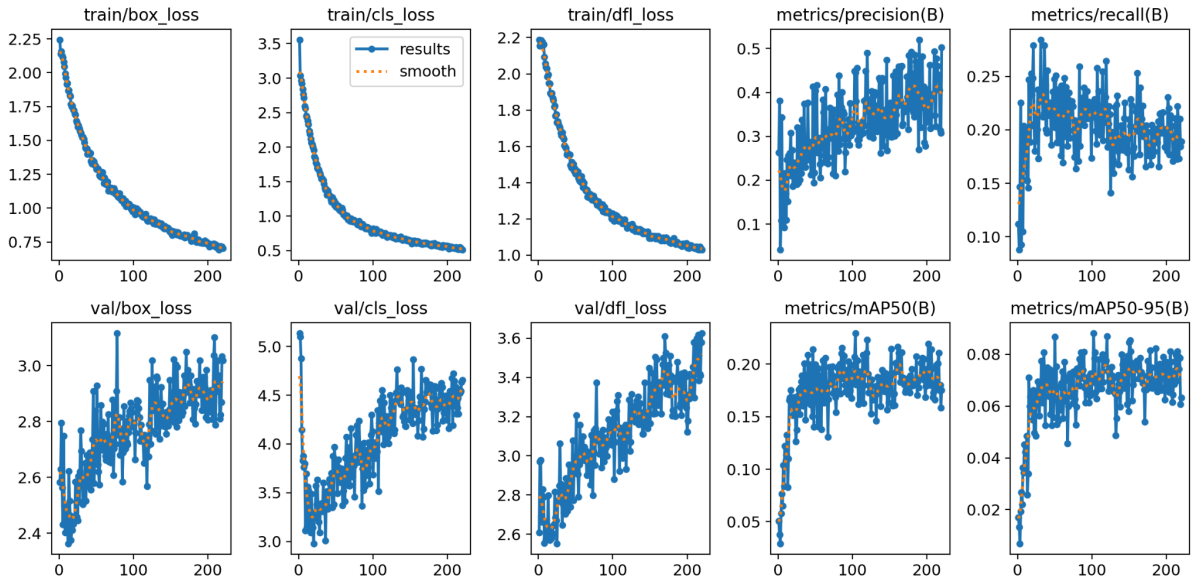
[Damaged Cars Dataset (kaggle.com)]

[car-damage-detector/dataset/test at master · nicolasmetallo/car-damage-detector · GitHub]

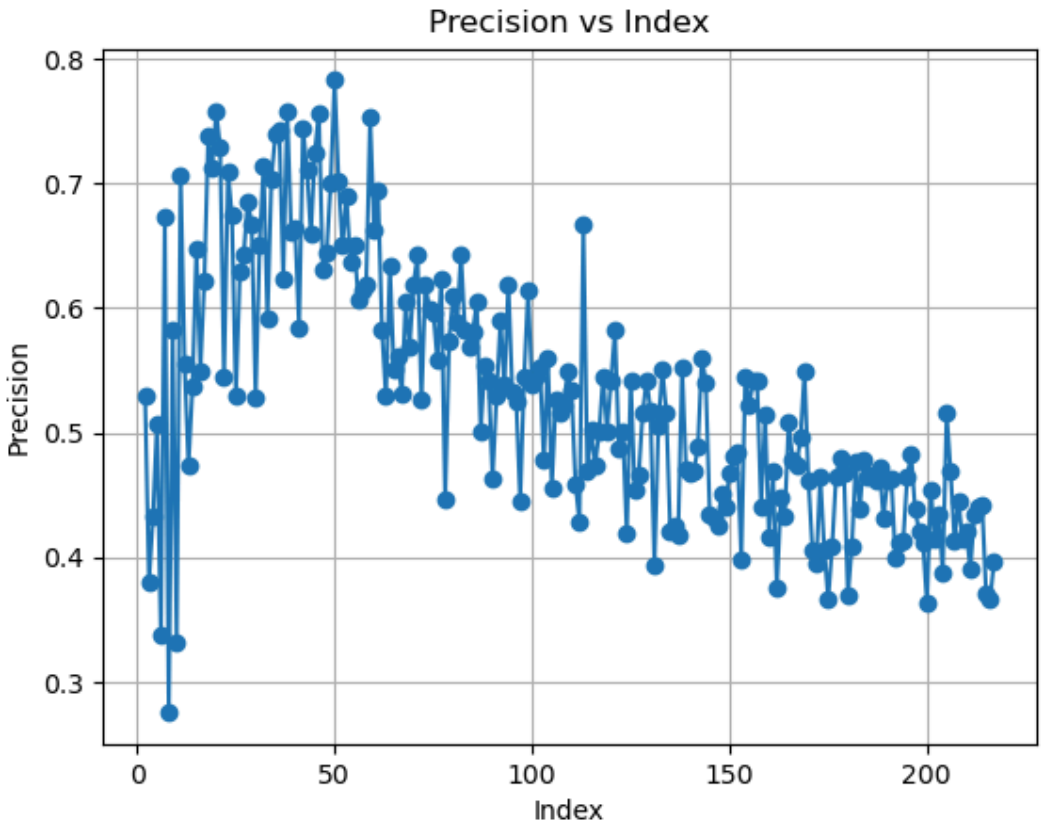
[Damaged Car Datasets (kaggle.com)]

[Car Damage Severity Dataset (kaggle.com)]

Courbes de validation



Courbe validation custom



Modèle de détection des zones d'une voiture:
[bhadreshpsavani/CarPartsDetectionChallenge: Train YOLOv3 for Car Parts Detection (github.com)]

Interface d’annotation binaire.