

INF8225 - Rapport de projet

Olivier Desclaux (2097696), Ralph Saber (2047612), Ghassen Cherni (1732397) and Antonin Tranchon (2096752)

Abstract

Dans ce rapport, nous allons présenter notre implémentation de l'algorithme CenterNet sur différentes bases de données. Nous avons utilisé une approche plus simple mais efficace, et avons exploré différentes méthodes afin d'optimiser nos résultats. Notre modèle implémenté a été appliqué principalement sur la détection de caractères cursifs Japonais et a atteint des résultats qualitatifs et quantitatifs prometteurs. Le modèle a été aussi testé sur une autre base de données afin de comparer sa performance à des travaux antérieurs.

1 Introduction

Notre projet consiste à implémenter l'architecture CenterNet afin de faire de la localisation d'objets, en nous appuyant sur l'implémentation présentée dans le papier de [Xingyi *et al.*, 2019]. Dans le cadre d'une compétition Kaggle, nous nous sommes intéressés à la détection des Kuzushijis. Les Kuzushijis sont d'anciens caractères japonais, qui ont été utilisés pendant des siècles, mais qui ont été délaissé au XX^e siècle en faveur de l'écriture japonaise moderne. Aujourd'hui, très peu de personnes savent les lire alors que des millions de livres sont écrits avec; et c'est pourquoi il est intéressant de créer un modèle capable de les reconnaître. Nous allons uniquement nous intéresser à la détection et la localisation des caractères. Des travaux antérieurs sur la détection et la classification des Kuzushijis ont été réalisés. [Clanuwat *et al.*] implémentent un modèle intitulé KuroNet pour cette tâche. Leur modèle a été entrainé sur 29 livres contenant des caractères de Kuzushijis et testé sur 15. En ce qui concerne CenterNet, il a été implémenté et testé pour plusieurs objectifs tel que la détection d'objets et l'estimation de pose (citation de l'article de centernet). Nous allons aussi utiliser CenterNet sur PascalVOC afin de pouvoir comparer notre modèle aux résultats obtenus dans le papier.

2 Dataset

2.1 Kuzushiji

L'écriture Kuzushiji est cursive. Cette spécificité engendre des difficultés pour le modèle de détection d'objets.

Tout d'abord, il peut être difficile de distinguer deux caractères : ils peuvent se retrouver très proches, et il sera alors difficile de les dissocier. Le contexte est donc très important, notamment pour la tâche de classification, car un caractère peut prendre un sens différent selon le symbole qui le précède.

De plus, il existe plus de 4000 Kuzushijis différents (dont des symboles qui se traduisent aujourd'hui par le même caractère en Japonais moderne). Certains sont relativement rares et n'apparaissent très peu sur le jeu de données, et le modèle ne pourra pas beaucoup s'entraîner sur eux.

Il peut aussi y avoir des annotations écrites entre les colonnes des Kuzushiji, qui pourraient elles mêmes être considérées comme des caractères alors qu'ils sont écrits en Japonais moderne. Elles ne sont pas annotées pour le jeu de données, et il faudrait étudier le contexte pour pouvoir les considérer comme des annotations et ne pas s'entraîner dessus.

Le jeu de données présente 3605 images pour l'entraînement et 1730 images de validation.

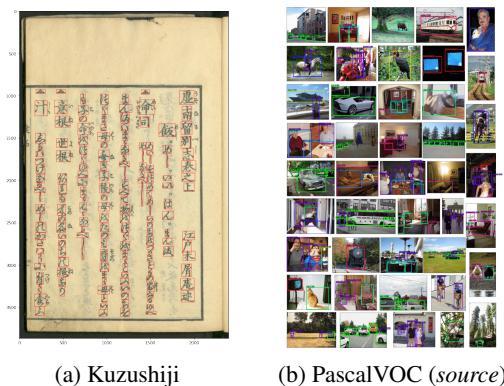


Figure 1: Exemples des différentes bases de données testées

2.2 PascalVOC

Pour valider notre approche, nous avons souhaité retrouver des résultats similaires à [Xingyi *et al.*, 2019] sur une base de données de référence. Nous avons donc travaillé sur la base de données PascalVOC2007, téléchargée depuis [ce lien](#). PascalVOC2007 contient des images de 20 classes différentes

(humains, véhicules, objets de la maison, etc.). Il est séparé en deux sous-dossiers, un pour l'entraînement et la validation (5011 images) et un pour le test (4952 images).

3 Méthodologie

3.1 CenterNet

Le modèle que l'on a utilisé pour la détection des caractères de Kuzushiji dans les images est basé sur CenterNet. Contrairement à d'autres modèles de détection d'objets qui représentent les objets par leurs rectangles englobants, cette méthode représente les objets par leur centre, puis régresse ensuite vers la taille de l'objet ainsi que l'offset qui apparaît à cause du downsampling des images en entrée.

Ainsi le modèle reçoit en entrée des images avec des objets à détecter et produit plusieurs heatmaps:

- Une heatmap principale qui encode les centres des objets
- Deux heatmaps qui encodent la taille de l'objet selon x et selon y
- Deux heatmaps qui encodent l'offset de notre centre, dû au fait que l'on travaille avec des heatmaps de taille (128,128). Ainsi, deux centres peuvent se retrouver au même endroit dans la heatmap.

Il est à noter que CenterNet peut aussi régresser à d'autres points clés tel que la position d'articulations dans les tâches de détection de pose. La **Figure 2** tirée de l'article original présente un exemple d'output de CenterNet avec différents points clés détectés.

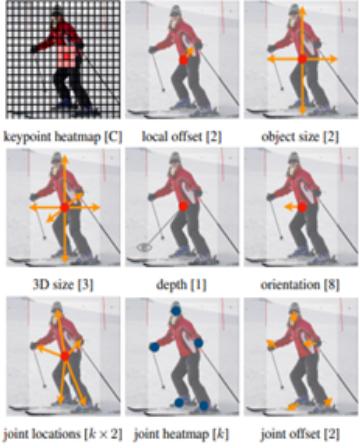


Figure 2: Exemples de prédictions à l'aide de CenterNet

Afin de produire ces heatmaps, dans l'architecture originale de CenterNet, l'image en input passe d'abord dans un backbone afin d'en extraire les caractéristiques, puis par des couches de upsampling, basées sur des convolutions transposées entre connectées avec des convolutions déformables comme on peut le voir dans la figure 3

Afin de minimiser les temps de calcul, nous avons opté pour plusieurs choix. Nous avons tout d'abord choisi d'utiliser Resnet-18 comme backbone, puisqu'il est le plus

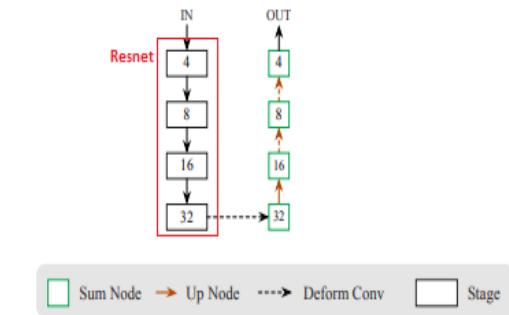


Figure 3: Architecture de CenterNet

petit des backbones testés par les auteurs. De plus, afin de contourner l'aspect énergivore des convolutions déformables, nous avons utilisée un simple upsampling bilinéaire avec un facteur 2.

3.2 Génération des heatmaps

Pour chaque image, nous avons donc 5 heatmaps à générer. Une pour le centre, deux pour les offsets (en abscisse et en ordonnée) et deux pour la taille de l'objet (largeur et hauteur).

Heatmaps des centres Chaque centre est représenté par une distribution gaussienne dont la moyenne n'est autre que le centre de l'objet. Pour la valeur de l'écart type σ de la gaussienne, nous nous sommes appuyés sur la méthode de CornerNet [Law and Deng, 2019]. L'idée est d'avoir un écart type adapté à la taille de l'objet. Dans le cas de PascalVOC, nous avons utilisé leur méthode (lien ici). Dans le cas de Kuzushiji, nous avons utilisé une version simplifiée qui prend comme valeur de σ 10% de la taille de la boîte dans chaque dimension.

Heatmaps de l'offset Dans de cas, la heatmap est initialisée à 0, puis tous les pixels correspondant au centre d'un objet se voient attribuer une valeur d'offset. Comme nous travaillons avec des images (512,512) et que nos heatmaps ont pour dimension (128,128), la valeur de l'offset est bornée par 3 en x et en y.

Heatmap des dimensions de la boîte De la même façon que pour les offsets, ces heatmaps ont été initialisées à 0 puis "remplies" au niveau des centres. Cependant, les valeurs des pixels non-nuls étaient bornées par 512 (cas d'un objet qui ferait toute la taille de l'image). Dans le cas de Kuzushiji, nos boîtes englobantes avaient une taille régulière de quelques pixels (cf **Figure 1**), alors que dans PascalVOC, on avait une très grande variété de tailles de boîtes. Dans certains cas, l'objet pouvait d'ailleurs être aussi gros que toute l'image. De tels pics de valeurs dans une heatmap quasi-nulle altérait la capacité de notre modèle à apprendre. Nous avons donc décider de les réduire d'un coefficient constant afin d'homogénéiser la heatmap finale.

3.3 Métriques utilisées

Fonction de perte Notre fonction de perte consiste en la somme de trois sous-fonction, une pour chaque type de heatmap. Pour les heatmaps d'offset et de taille de l'objet,

nous nous appuyons sur la même métrique, qui est une norme L1 normalisée selon l'équation. Dorénavant, reg indiquera les heatmaps de taille d'objet ou d'offset.

$$L_{reg} = \frac{S(|reg_{pred} - reg_{gt}| * hm_{gt})}{S(hm_{gt})} \quad (1)$$

où reg_{pred} indique notre heatmap prédictive, reg_{gt} notre ground truth, et hm_{gt} indique la groundtruth des centres. La fonction S calcule la somme globale selon les deux dimensions d'une matrice 2D:

$$S(A) = \sum_{i,j} A_{ij} \quad (2)$$

Pour la heatmap des centres, nous utilisons une version modifiée de la focal loss. C'est cette loss qui est utilisée par les auteurs de CenterNet et de CornerNet. Pour chaque pixel p_{ij} dans notre prédiction, on regarde son pixel associé dans la ground-truth y_{ij} . Alors:

$$L_c = -\frac{1}{N} \sum_{i=1}^H \sum_{j=1}^W \begin{cases} (1 - p_{ij})^\alpha \log(p_{ij}) & \text{si } y_{ij} = 1 \\ (1 - y_{ij})^\beta p_{ij}^\alpha \log(1 - p_{ij}) & \text{sinon} \end{cases} \quad (3)$$

où N représente le nombre d'objets dans l'image, α et β sont des hyperparamètres (de valeur 2 et 4 respectivement).

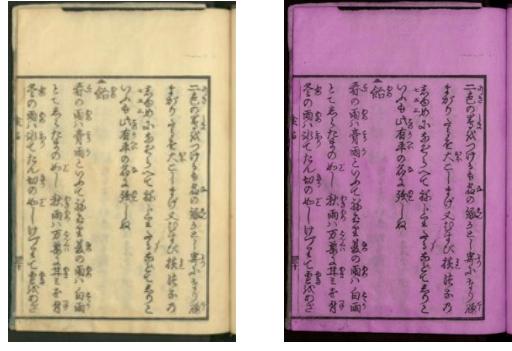
Evaluation de la performance Afin d'évaluer la performance de notre modèle, nous avons utilisé la mAP (mean Average Precision). Nous avons choisi un threshold d'IOU de 50 %, et avons utilisé la version implémentée dans [ce lien](#).

4 Expériences

4.1 Augmentation de données

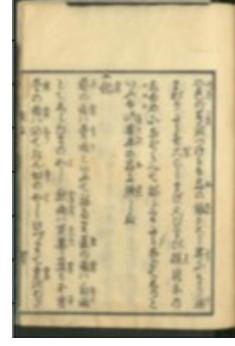
Afin d'augmenter la capacité du modèle implémenté à généraliser, des techniques d'augmentation de données ont été appliquées. La première méthode consiste à altérer la couleur, le contraste, la teinte ainsi que la luminosité avec des degrés aléatoires. Le degré d'altération de ces paramètres a été choisi comme provenant d'une distribution uniforme avec une marge d'altération allant jusqu'à 60% des valeurs des pixels dans le cas de la saturation et 40% dans le cas des autres paramètres.

Pour la deuxième méthode, on a appliqué du flou Gaussien (Gaussian blur) avec un kernel de taille (65 x 65) et un sigma choisi d'une distribution uniforme entre 0.1 et 5. La troisième transformation appliquée est le recadrage (cropping) avec un ratio compris entre 75% et 100%. Pour un ratio de 100%, l'image est passée en sortie sans recadrage. Pour cette transformation, les centres des caractères ainsi que les largeurs et les hauteurs doivent subir une transformation linéaire puisque chaque caractère aura de nouvelles coordonnées dans le repère de l'image recadrée. Pour ce faire, les coordonnées des centres subissent une translation par un offset précalculé et les largeurs et hauteurs des boîtes englobantes subissent un redimensionnement (scaling) égal au rapport des dimensions dans l'image recadrée sur les dimensions originales.

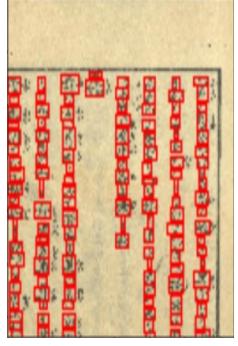


(a) Image originale

(b) Altération des couleurs



(c) Flou Gaussien



(d) Recadrage

Figure 4: Exemples des différentes méthodes de data augmentation utilisées.

Il est à noter que ces techniques d'augmentation des données sont implémentées à la volée au cours de l'entraînement de manière à générer des images avec de nouvelles transformations à chaque batch choisi.

4.2 Adaptive learning rate

Afin d'améliorer notre système, nous avons décidé d'effectuer des expériences pour choisir les meilleurs hyperparamètres. Pour cela, nous nous sommes basés sur le papier de [Smith, 2018] pour trouver le meilleur learning rate, permettant d'obtenir des résultats convainquant sans faire du sur-apprentissage. Pour cela, on va utiliser des méthodes qui vont adapter le learning rate durant l'entraînement, en choisissant des bornes de learning rate maximum et minimum à parcourir.

Pour identifier ces bornes, nous avons mené une première expérience (déttaillée dans [Smith, 2017]) qui consiste à faire modifier la valeur du learning rate à chaque epoch entre deux bornes. On mesure alors les pertes, et cela permet d'avoir une idée du learning rate à utiliser. On trace alors la perte obtenue en fonction du learning rate, et on essaye de trouver une borne minimale à partir de laquelle la perte est bonne, et une borne maximale où elle va se mettre à diverger (ou avoir un comportement plus aléatoire).

En effectuant l'expérience sur Kuzushiji (cf **Figure 5**), on peut voir que la perte converge assez rapidement, et qu'on n'a pas de comportement trop aléatoire lorsque le learning rate augmente. Nous avons donc pris une valeur minimale

du learning rate à 10^{-3} pour le test cosine, et le learning rate maximal à $5 * 10^{-2}$ pour le test One Cycle.

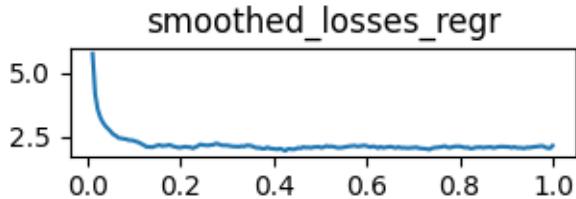


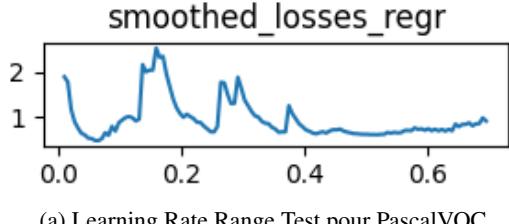
Figure 5: Learning Rate Range Test pour Kuzushiji

Avec les valeurs des bornes obtenues pour le learning rate, nous allons pouvoir effectuer des expériences avec un adaptative learning rate.

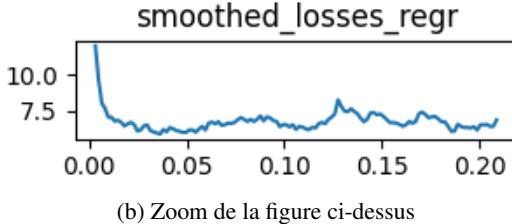
Nous utiliserons en premier la méthode du cosine. Le learning rate va alors être modifié au long de l’entraînement en suivant une fonction sinusoïdale.

La seconde méthode d’adaptative learning rate est le One Cycle. Cette fois ci , le learning rate augmente jusqu’à la valeur maximale puis diminue, de manière linéaire.

Pour PascalVOC, nous avons uniquement fait tourner la première expérience pour trouver la bonne valeur du learning rate, que nous gardons fixe ensuite. En faisant tourner le test sur une grande plage de learning rates, nous avions beaucoup plus de variations cette fois (cf **Figure 6a**). Nous avons mené un premier essai avec $lr = 10^{-4}$ qui a mené à de l’overfitting au bout d’une trentaine d’epochs. Nous avons donc décidé de mettre le learning rate à 10^{-4} pour les 15 premiers epochs, puis à 10^{-5} pour finir l’entraînement.



(a) Learning Rate Range Test pour PascalVOC



(b) Zoom de la figure ci-dessus

Figure 6: Learning Rate Range Test pour PascalVOC

4.3 Liste des expériences et hyperparamètres

Les différentes expériences que nous avons réalisées sont rapportées dans la **Table 1**. Nos expériences KZ, KZ-Cosine et KZ-OneCycle ont pour but de trouver le Learning Rate optimal pour notre expérience. Nous souhaitons ensuite comparé notre modèle avec de la data augmentation (auquel on

rajoutera le OneCycle learning rate) Nous avons aussi réalisé une expérience simple sur PascalVOC afin de valider notre modèle.

Name	Dataset	Learning Rate	Data Aug
PV	PascalVOC	Fixe	Non
KZ	Kuzushiji	Fixe	Non
KZ-Cosine	Kuzushiji	Cosine	Non
KZ-OneCycle	Kuzushiji	One-Cycle	Non
KZ-DA	Kuzushiji	One-Cycle	Oui

Table 1: Liste des expériences réalisées

Le choix des hyperparamètres utilisés par notre modèle est donné dans la **Table 2**

Model Hyperparameters	
Epochs	100 - 150
Batch size	20
Learning Rate	0.001
Learning Rate Decay	0.1 (epoch 15)
Optimizer	Adam

(a)

Slurm Hyperparameters	
Number of Nodes	1
CPUs per task	4
GPU used	NVidia P100
GPU memory	16 Gb

(b)

Table 2: (a) Hyperparamètres des différents modèles. Le learning rate est celui utilisé dans les expériences à learning rate fixe. (b) Ressources utilisées dans le calcul sur Compute Canada.

5 Résultats et discussions

5.1 Résultat des expériences

La première expérience sur Kuzushiji se fait avec un learning rate fixe à 10^{-3} et sans data augmentation, avec 100 epochs. Les différentes courbes de pertes et précisions sont représentées dans la **Figure 7**. On arrive à avoir une convergence des fonctions de perte, et une mAP de 32 % sur l’ensemble de validation.

Nous avons ensuite effectué de nouvelles expériences avec le learning rate adaptatif. Lorsqu’on utilise le learning rate adaptatif cosine, nos résultats, présentés sur la **Figure 8** (le calcul de précision prenant du temps, nous avons décidé de le calculer uniquement toutes les 10 epochs), nous permettent de voir que la convergence est alors plus lente, et que les pertes ont plus de variations. Nous avons du faire tourner ce modèle sur 150 epochs pour voir la convergence.

Nous avons aussi utilisé le learning rate adaptatif One Cycle (cf **figure 9**). Cela nous a permis d’obtenir une convergence des pertes plus rapidement.

En voyant les résultats obtenus sur ces expériences, nous avons donc décidé de faire tourner notre modèle avec data

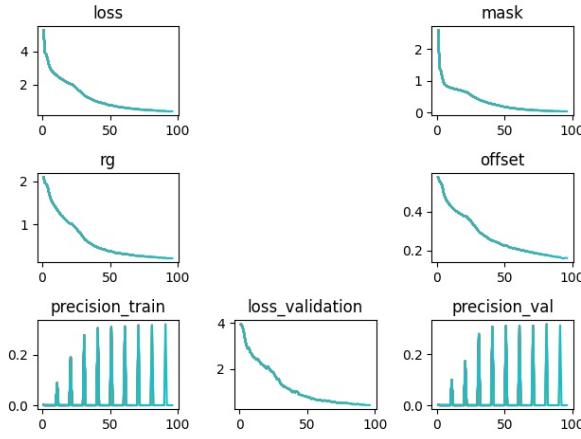


Figure 7: Résultats initiaux obtenus sur Kuzushiji avec un learning rate fixe

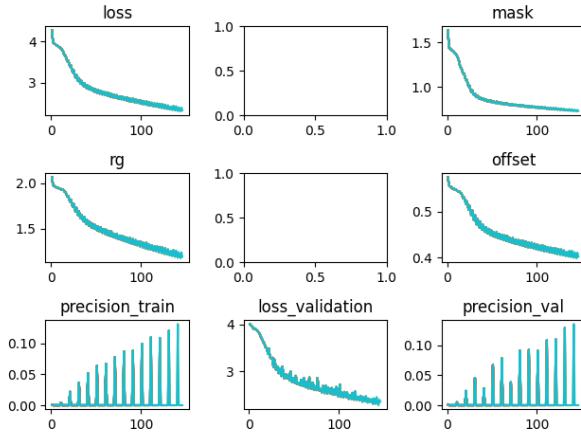


Figure 8: Résultats obtenus avec Cosine Adaptive Learning Rate

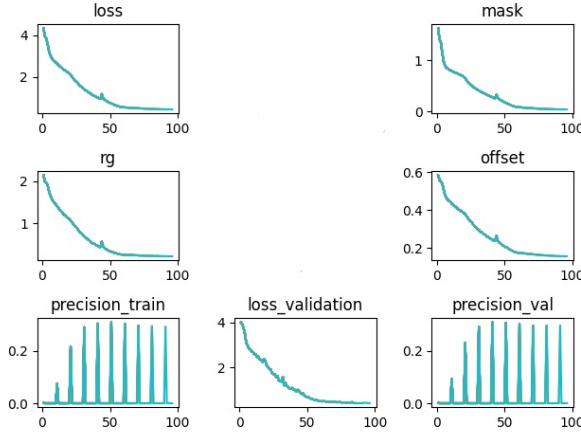


Figure 9: Résultats obtenus avec OneCycle Adaptive Learning Rate

augmentation en rajoutant le OneCycle adaptative learning rate. Nous obtenons les résultats présentés dans la figure 10.

Les résultats ne sont pas forcément bien meilleurs, mais on arrive à converger plus vite que dans les autres expériences.

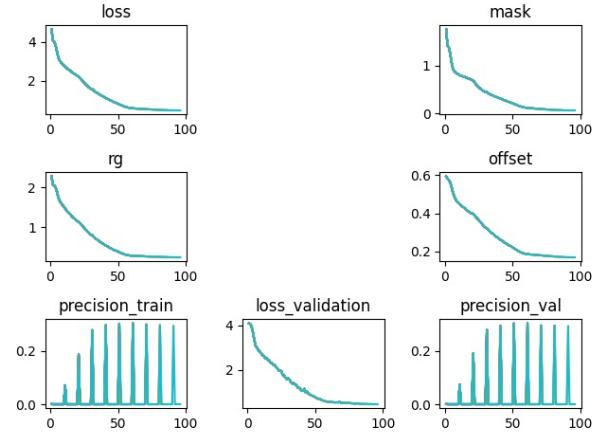


Figure 10: Résultats obtenus avec OneCycle Adaptive Learning Rate et Data Augmentation

Pour PascalVOC, nous avions précédemment remarqué qu'un learning rate trop élevé conduirait à de l'overfitting. Nos premières expériences, avec un learning rate 10^{-4} nous l'ont montré, on trouvait ce problème au bout d'une trentaine d'epochs (voir figure 11).

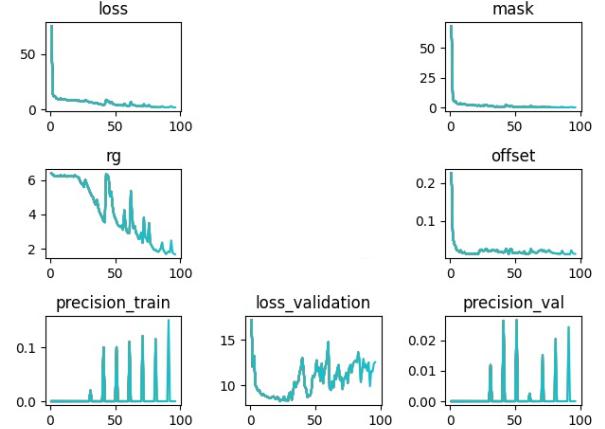


Figure 11: Résultats obtenus sur PascalVOC avec $lr = 10^{-4}$

Nous avons donc décidé de mettre le learning rate à 10^{-4} pour les 15 premiers epochs, puis à 10^{-5} (laisser à 10^{-4} tout du long ne permettait pas d'apprendre suffisamment). Nous obtenons les résultats de la figure

La valeur de l'AP après 100, 130 ou 150 epochs pour nos différents modèles est rapportée dans la **Table 3**.

Nous pouvons voir dans la **Figure 13** un exemple de détection des caractères de Kuzushiji d'une image de l'ensemble de test:

5.2 Discussion

Performance sur Kuzushiji Comme on peut le voir sur la **Figure 13**, les résultats sont très satisfaisants d'un point de

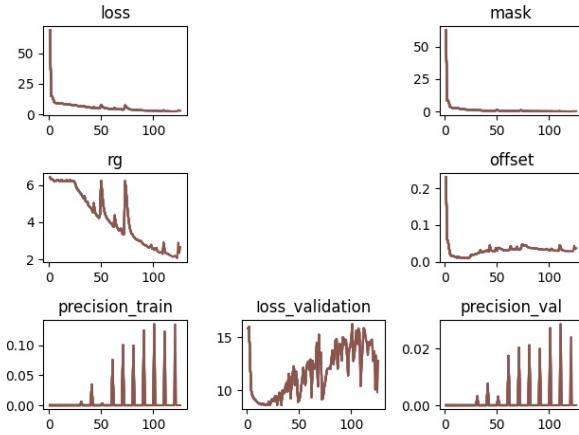


Figure 12: Résultats obtenus sur PascalVOC avec $lr = 10^{-4}$ puis $lr = 10^{-5}$

AP	Train	Validation
PV (100 epochs)	0.134	0.0283
KZ (100 epochs)	0.321	0.32
KZ-Cosine (150 epochs)	132	0.13
KZ-OneCycle (100 epochs)	0.289	0.29
KZ-DA (130 epochs)	0.287	0.29

Table 3: Précision moyenne (AP) de nos différents modèles après 100 epochs



Figure 13: Résultats de la prédiction de Kuzushiji

vue visuel. La quasi totalité des caractères sont détectés, et avec une estimation du centre cohérente. Il est intéressant de noter que le modèle ne détecte pas les caractères du verso de la page, visibles par transparence, ce qui est un comportement attendu.

Performance sur PascalVOC Réussir à obtenir des performances similaires au papier original de CenterNet a été une vraie difficulté. A titre de comparaison, les auteurs affirment obtenir une mAP de 50% sur PascalVOC 2007, et ce en utilisant le même backbone que nous. Nous pourrions

penser que la stratégie d’effectuer un upsampling bilinéaire simple plutôt que d’apprendre les paramètres (issues des convolutions déformables) serait la raison. Pourtant, nous avons utilisé exactement la même architecture sur PascalVOC et Kuzushiji, et les bons résultats sur cette dernière contredisent cette théorie.

En réalité, nous avons constaté que le modèle avait énormément de mal à obtenir la taille des boîtes englobantes ainsi que les offsets. Comme expliqué dans la section 3.2, nous initialisons la heatmap à zéro, et seuls quelques pixels sont non-nuls. Dans le cas de Kuzushiji, la heatmap est assez “remplie”, du fait du grand nombre de caractères à détecter. Dans le cas de PascalVOC cependant, il n’y a parfois qu’un seul objet, et donc qu’une seule valeur non-nulle. Lors de l’entraînement du modèle, nous avons alors constaté qu’il prédisait quasiment exclusivement des heatmaps de regression nulles. Pour contrer cela nous avons généré des “patchs” de taille 5 par 5 afin d’encoder la taille de notre objet, et ainsi réduire le nombre de valeurs non-nulles dans notre heatmap. Ceci a permis d’obtenir les valeurs finales de mAP de 13.4%.

Classification Pour notre projet, nous avons décidé de ne pas faire d’implémentation de la classification, car nous avons préféré améliorer nos résultats sur la localisation et la taille des objets. Cependant, cette tâche aurait été impossible en rajoutant une couche par objet à reconnaître.

Pour Kuzushiji, il aurait alors fallu créer une heatmap par caractère existant, et on aurait obtenu un centre à l’emplacement d’un Kuzushiji si c’était celui de la heatmap.

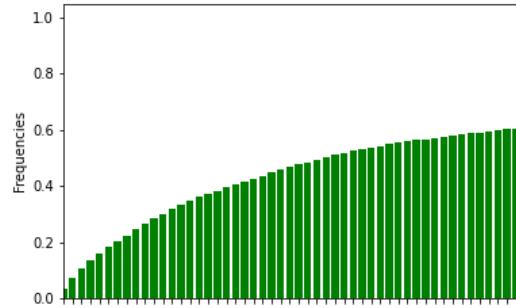


Figure 14: Fréquences cumulatives des 50 Kuzushijis les plus fréquents

Il aurait été tout de fois été très difficile de s’entraîner sur certains caractères : en effet, il existe plus de 4000 Kuzushijis différents, et certains n’apparaissent que quelques fois dans le jeu de données. De plus, cela aurait engendré des temps de calcul et des coûts en mémoire plus importants. En s’intéressant seulement aux cinquantes Kuzushijis les plus fréquents (cf Figure 14), on aurait tout de même pu classifier plus de la moitié des caractères présents dans le jeu de données, pour trouver un milieu entre les difficultés calculatrices et suffisamment de classification.

6 Conclusion

Dans ce travail, on a réalisé une implémentation de CenterNet, un modèle de détection d’objets qui traite chaque objet comme un point. Ce modèle, appliqué sur une base de

données de caractères Japonais Kuzushiji, a été capable de bien détecter les caractères. Des travaux supplémentaires doivent être fait afin de classifier les caractères les plus fréquemment rencontrés.

Vous pouvez retrouver notre implémentation sur notre *GitHub*

References

- [Clanuwat *et al.*,] Tarin Clanuwat, Alex Lamb, and Asanobu Kitamoto. Kuronet: Pre-modern japanese kuzushiji character recognition with deep learning. In *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR*, pages 607–614. IEEE Computer Society.
- [Law and Deng, 2019] Hei Law and Jia Deng. Cornernet: Detecting objects as paired keypoints, 2019.
- [Smith, 2017] Leslie N. Smith. Cyclical learning rates for training neural networks. In *Proceedings - 2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017*, pages 464–472. Institute of Electrical and Electronics Engineers Inc., 2017.
- [Smith, 2018] L. N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay [arxiv]. *arXiv*, page 21, 2018.
- [Xingyi *et al.*, 2019] Zhou Xingyi, Wang Dequan, and P. Kraumlhenbuumlhl. Objects as points [arxiv]. *arXiv*, page 12, 2019.