Generative Pseudo Labels Refinement for Unsupervised Domain Adaptation

Références

[1] GENERATIVE PSEUDO-LABEL REFINEMENT FOR UNSUPERVISED DOMAIN ADAPTATION, PIETRO MORERIO, RICCARDO VOLPI, RUGGERO RAGONESI, VITTORIO MURINO.

[2] CYCADA: CYCLE-CONSISTENT ADVERSARIAL DOMAIN ADAPTATION, JUDY HOFFMAN, ERIC TZENG, TAESUNG PARK, JUN-YAN ZHU, PHILLIP ISOLA, KATE SAENKO, ALEXEI A. EFROS, TREVOR DARRELL

[3] LIGHT-WEIGHT CALIBRATOR: A SEPARABLE COMPONENT FOR UNSUPERVISED DOMAIN ADAPTATION, SHAOKAI YE, KAILU WU, MU ZHOU, YUNFEI YANG, SIA HUAT TAN,KAIDI XU, JIEBO SONG, CHENGLONG BAO, KAISHENG MA

Contexte

Dans le cadre du cours IA 716, perception pour les systèmes autonomes, j'ai choisi d'implémenter une première approche d'adaptation de domaine non-supervisée (UDA). L'objectif de ce type d'approche est de développer un classifieur d'image performant pour un dataset ne disposant pas de labels.

L'adaptation de domaine consiste à mettre à profit un dataset source qui dispose pour sa part de données labellisées permettant l'apprentissage d'un classifieur. Ce modèle utilisé en prédiction sur les données cibles verra sa performance s'effondrer en comparaison de celle du domaine source.

Les approches d'adaptation de domaine visent donc à régler cette problématique et à développer un classifieur performant en prédiction sur un domaine cible ne disposant pas de données labellisées.

L'approche que j'ai choisie d'implémenter est tirée du papier [1], Générative Pseudo-Label Refinement. Les performances de l'approche et la possibilité de travailler et de progresser sur des architectures cGAN et CNN ont guidées mon choix.

Description

L'idée de ce papier est d'utiliser un classifieur pré-entrainé sur un domaine source pour inférer des pseudo-labels dans le domaine cible. Comme évoqué précédemment, un nombre important de données seront mal classées par le classifieur. Les Gan conditionnels « cGan » ont des propriétés de robustesse sur des données dont les labels sont bruités. Cette propriété des cGAn est exploitée au travers d'une procédure itérative dans laquelle le cGAN et le classifieur apprenne de manière jointe.

Mon objectif a été d'implémenter cette approche en prenant comme domaine source le dataset « SVHN » et comme domaine cible le domaine « MNIST ». L'adaptation de domaine dans ce sens permet de réduire les temps de calcul des réseaux profonds (moins de couches) et d'obtenir des performances de classification que dans le sens inverse.

Il y a deux grandes phases successives à réaliser pour implémenter cette méthode

1. Une phase de pré-entrainement d'un classifieur C0 réalisé sur le dataset source (SVHN), ainsi que d'un cGAN GO et DO. Pour ce faire, le dataset SVHN est converti en Grayscale. Le cGAN est quant à lui entrainé sur un dataset cible MNIST bruité, c'est à dire dont les pseudo-labels sont inférés par le classifieur C0. Les dimensions du dataset MNIST sont transformées pour correspondre à celles du dataset SVHN (32x32). Aussi les données sont normalisées dans la fonction « transform ».
2. Une phase d'apprentissage itérative qui alterne l'apprentissage du classifieur C sur un dataset généré par le générateur G et l'apprentissage du cGAN G et D su un dataset bruité par des pseudos-labels inférés pas le classifieur C. Aussi les données sont normalisées dans la fonction « transform ».

Phase 1 : pré-entrainement des modèles C0, G0 et D0

Pré-entrainement, performance et choix du modèle C0

Afin de choisir un réseau de neurones convolutif pertinent pour cette étude, j'ai comparé la performance en classification de plusieurs modèles sur le dataset SVHN préalablement converti en Grayscale.

- *Le détail des réseaux CNN envisagé sont codés dans le script « CNN.py »,*
- *L'apprentissage des modèles se fait via le script « Train-CNN_SVHN.py »*
- *La prédiction des modèles est documentée dans le script « Predict.py »*

Le modèle dont la performance a été la meilleure est celui du ResNet, c'est donc celui-ci qui a été retenu comme classifieur C0. Le modèle pré-entrainé est sauvegardé dans le dossier « Outputs sous le nom « Model_Trained_SVHN_C1.pth ».

Les performances du classifieur C0 sont de :

- **95%** sur le dataset SVHN converti en grayscale,
- **67%** sur le dataset MNIST transformé en dimension 32x32.

Ces résultats démontrent bien la perte de performance en prédiction du classifieur sur le domaine cible.

Pré-entrainement, performance et choix du modèle cGAN G0 et D0

De la même manière il a fallu construire et développer un cGan sur le dataset MNIST transformé en 32 x 32 pixels.

Lors du développement de ce cGan j'ai rencontré plusieurs problématiques notamment des problèmes de divergence des « loss » très marquées et de « collapse ». Afin de résoudre ces problématiques j'ai dû :

- Implémenter une approche de « label smoothing »,
- Introduire un bruit gaussien dans les images réelles et fausses en entrée du discriminateur D avec un decay,
- Implémenter un "flip" de 5% des labels positifs et négatifs en entrée du discriminateur,
- Implémenter un cDCGAN et complexifier les modèles G et D en ajoutant des convolutions et des « Batch Normalisation »et des « Spectral normalisation ». L'idée était de développer un générateur suffisamment fin pour et avec un contraste suffisamment qualitatif pour générer des images de qualités suffisantes.

Ces approches ont permis de stabiliser l'apprentissage du cGan et d'éviter l'effondrement de la perte du discriminant D et la dérive de la perte du générateur G. j'ai constaté de manière empirique, c'est-à-dire de manière visuelle, que la performance de du générateur était acceptable entre 15 et 25 epochs, tout en restant pas trop long à implémenter.
A noter que pour le cGAN j'ai encodé le label de la même manière pour le générateur et le discriminant en ajoutant par concaténation un canal « label » aux channels existants.

- *Le détail des implémentations du « label smoothing », de l'introduction du bruit gaussien et du flip des labels est disponible via le script « Train_cGAN_MNIST.py »*
- *La structure du cGAn retenu est disponible via le script « cGan.py »*
- *L'apprentissage des pré-modèles a eu lieu via le script "PseudoLabelPretrain.py"*
- *Les images générées par le modèle G0 sont disponibles dans le répertoire /outputs/Résultats G0*
- *Les modèles G0 et D0 pré-entrainés sont sauvegardés dans le dossier « Outputs sous le nom « G0_MNIST.pth » et « D0_MNIST.pth ».*

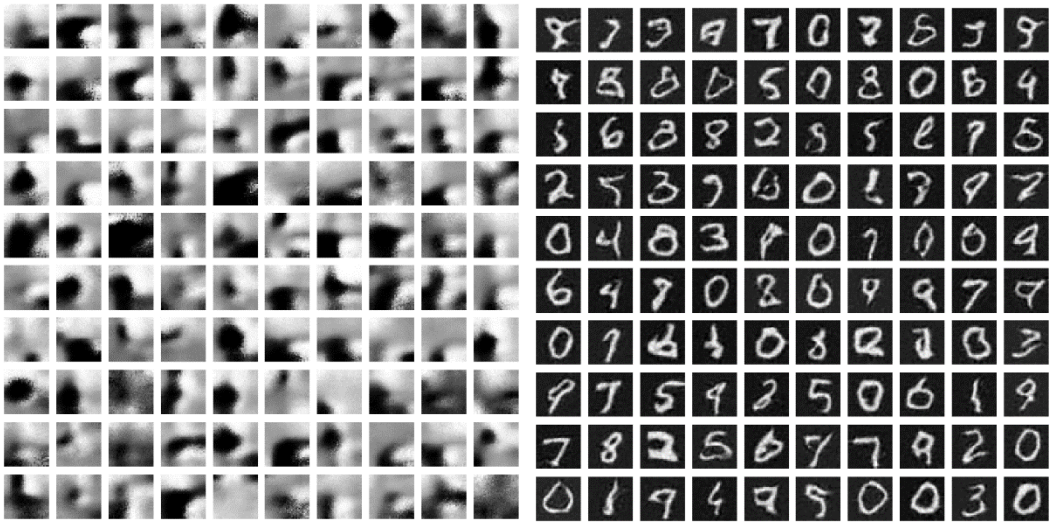

Figure 1: Images générées par G0 après 1 epoch et 25 epochs avec un bruit gaussien décroissant

Phase 2 : Apprentissage itératif de C, G et D

Durant cette phase, les modèles C, G et D font l'objet d'étapes d'apprentissage itératives. Lors de la première itération les modèles C, G et D correspondent aux modèles C0, G0 et D0.

- L'étape 2.1 consiste à générer un dataset MNIST « cible » via le générateur conditionnel "G". Les labels des images sont tirés suivant une loi uniforme parmi les 10 classes disponibles. Les images conditionnelles et ces labels constitue un dataset « cible » qui permet d'affiner l'apprentissage du classifieur C.
- L'étape 2.2 consiste à affiner les modèles cGan G et D. Le dataset utilisé pour l'apprentissage est un échantillon aléatoire d'images du dataset MNIST dont les labels sont inférés par le classifieur C. J'ai décidé de conserver 25 epochs pour l'apprentissage du cGAN dans cette phase. Mon code me permet de générer des images à chaque epoch et ainsi de vérifier visuellement que les images générées sont acceptables.

En alternant ses deux étapes, l'asymétrie du bruit initial diminue et la performance du modèle C sur le dataset MNIST, et ce, alors que l'apprentissage se déroule de manière non supervisée.

J'ai mis en œuvre cette approche sur 10 itérations. A chaque étape, 7680 échantillons sont tirés de manière aléatoire pour les étapes d'apprentissage du classifieur C (2.1) et du conditionnel Gan G (2.2).

- *Le code utilise pour implémenter ces itérations est retranscrit dans le script « PseudoLabelmain.py ».*

Résultats des modèles cGAN G et D

Ci-dessous se trouvent les images générées par le générateur G à la dixième itération.

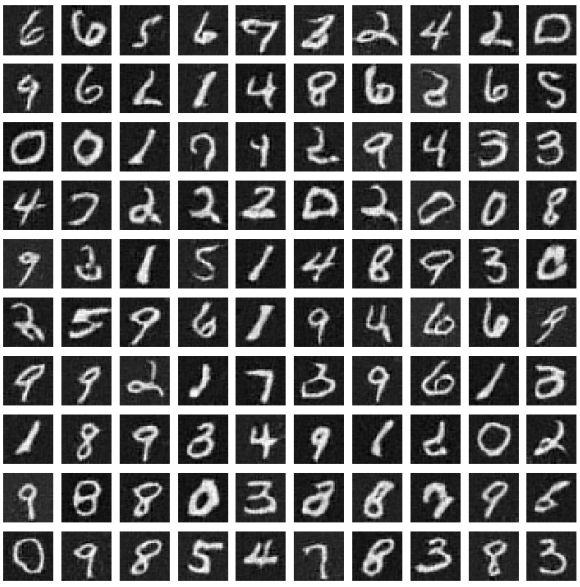

Figure 1Images générées par le cGAN a l'issue des 10 itérations

Performance finale du classifier C sur le dataset MNIST

A l'issue des 10 itérations, la performance en termes de précision du classifieur « C » est de **93,5%** sur le dataset MNIST. On constate donc que la performance du modèle a évolué de 67% à 93,5% avec l'approche « Pseudo-Label Refinement ». On peut donc considérer cette approche comme performante.

Néanmoins, comme précisé dans le papier [3], Light - weight – calibrator, cette approche souffre d'une perte de performance du modèle sur le dataset source d'origine. En effet, j'ai recalculé la performance en précision du modèle final sur le dataset source « SVHN » et j'ai obtenu une performance de 25%, bien en deca des 95% d'origine.

La performance finale de mon modèle sur le dataset MNIST, 93,5%, bien que très bonne est en deca de la performance annoncée dans le papier [1], 97%. Il est possible que mon générateur et mon discriminateur ne soit pas assez bon. En effet, la performance du classifieur « Resnet » n'étant plus à démontrer il est peu probable que cet écart vienne de cette partie de l'algorithme.

A la recherche d'un classifieur capable d'être performant en prédiction sur un dataset SVHN et sur le dataset MNIST, l'approche proposée par le papier [3] pourrait être plus pertinente et adaptée. L'idée est de développer un générateur G, qui transpose une image cible vers une image source (image-to-image Translation) afin que le classifieur source soit autant performant sur une image du domaine source « Xs » que sur une image du domaine cible transposée sur le domaine source « G(Xt) ».

Le modèle final obtenu est sauvegardé dans le répertoire « Outputs » sous le nom de « C_MNIST_CNN.pth ». De plus l'ensemble de ce rapport est aussi disponible dans un notebook « PLR.ipynb »