Université de Technologie de Compiègne

TX

Semestre de Printemps 2018

One-Class SVM pour la détection de défaut de surface

Étudiants :
Alexandre MILESI
Sylvain MARCHIENNE

Superviseurs :
Jonathan DEKHTIAR
Alexandre DURUPT

3 juillet 2018

Table des matières

Table des figures

1 Introduction

Dans le domaine de l'apprentissage, l'*anomaly detection*[1] ou encore détection de la nouveauté, consiste à reconnaître les données qui diffèrent d'une manière ou d'une autre de celles que l'on voit habituellement. C'est une technique utile dans les cas où une classe stratégique de données est sous-représentée dans l'ensemble d'apprentissage. En conséquence, les performances du modèle seront médiocres pour cette classe. Dans certaines circonstances, comme par exemple pour la détection de défauts, c'est souvent la classe des données ayant un défaut qui est sous-représentée et à la fois celle que le modèle doit détecter. En pratique, il y a souvent aucune ou trop peu de données à disposition appartenant à cette classe. Les modèles prévus pour ces problèmes d'*anomaly detection* ne sont appris que sur des jeux de données où cette classe n'est pas présente et détecte ensuite les nouvelles données qui ne se situent pas dans la région de « normalité » obtenue à l'issue de l'entraînement du modèle.

L'industrie est typiquement confrontée à ce problème. Dans notre étude nous considérons comme objectif final de construire un modèle capable de détecter (à partir d'images) des défauts sur des pièces usinées. Nous utiliserons le jeu de données DAGM[2] pour cela. Pour l'apprentissage, nous ne disposons que de pièces dites normales, c'est à dire sans défaut. Ces images seront appelées par la suite **données positives**. Des images présentant un défaut sont disponibles mais seulement pour la phase d'évaluation des performances, pas pour l'apprentissage du modèle. Ces images avec défaut seront appellées **données négatives**.

Si l'*anomaly detection* est un problème largement connu et qu'il existe de nombreuses implémentations de méthodes dans des bibliothèques ou *frameworks* classiques de *machine learning* (*Scikit-Learn*, *R*), les *frameworks* de *deep learning* sont moins fournis dans cette catégorie. Pour les problèmes où les données sont des images, des *frameworks* comme *TensorFlow* sont très utilisés. Une solution à notre problème de reconnaissance de défauts serait d'utiliser *TensorFlow* pour extraire des images une représentation réduite sous forme de vecteur et d'utiliser ensuite avec *Scikit-Learn* un algorithme d'*anomaly detection*. Le problème est que ce transfert d'un *framework* à l'autre pose des problèmes pour la production comme pour l'apprentissage. Multiplier les *frameworks* n'est pas efficace puisque chacun possède sa propre structure de données et le passage de l'un à l'autre peut être lent voir dangereux en termes de perte de précision numérique. Maintenir un programme dans un seul *framework* est aussi moins complexe. Nous nous concentrerons donc sur les techniques d'*anomaly detection* basées sur les SVM et nous utiliserons uniquement *TensorFlow* pour les implémenter et montrer qu'il est possible de résoudre le problème de détection de défaut en restant dans le même *framework*.

Dans une première partie nous définirons mathématiquement les deux modèles SVM à implémenter. Nous expliquerons ensuite la manière dont nous l'avons fait dans *TensorFlow* avec les avantages et inconvénients que nous avons rencontrés, mais aussi les ajustements de modèle que nous avons dû faire. Enfin dans une troisième partie nous appliquerons notre solution sur le jeu de données DAGM.

2 One-class classification avec des SVMs

Les *Support Vector Machines* ont prouvé leur utilité par le passé, et sont toujours très efficaces dans la classification binaire ou multiclasse par apprentissage supervisé. Elles consistent à apprendre une frontière séparant de manière optimale les données. Cette frontière est obtenue suite à la maximisation d'une marge qui est strictement définie par certains des points d'apprentissages, appelés *support vectors*, sur lesquels repose la marge. À l'inférence, les points situés d'un côté de la frontière sont classés dans la première catégorie, et les autres sont classés dans la seconde. En général, les données de la première

classe ont le *label* 1, et les autres ont le *label* -1.

La frontière apprise est linéaire : c'est une droite, ou plus généralement un hyperplan. Pour traiter des données non linéairement séparables, on utilise le *kernel trick*, qui revient à *mapper* implicitement les points de leur espace d'origine vers un espace de dimension supérieure, pour qu'ils soient séparables par un hyperplan dans ce nouvel espace. La fonction de noyau est alors K, avec

$$K(\mathrm{x}, \mathrm{x}') = \Phi(\mathrm{x}) \cdot \Phi(\mathrm{x}'), \tag{1}$$

où x et x′ sont deux points quelconques du jeu de données. Pendant l'optimisation et l'inférence, les produits scalaires $\mathrm{x} \cdot \mathrm{x}'$ du problème sont remplacés par $K(\mathrm{x}, \mathrm{x}')$: la fonction de transformation Φ n'apparaît nulle part, elle est donc implicite et il n'est pas nécéssaire qu'elle soit calculable, d'où l'intérêt de cette astuce.

Certains modèles se sont basés sur les travaux des SVMs pour proposer une solution dans le cas où une seule classe est disponible pour l'apprentissage, et où la distribution de cette classe doit être apprise pour différencier les points positifs (dans la zone principale) des points négatifs (les *outliers* en dehors de la zone). Ces solutions sont ainsi adaptées à la détection d'anomalies, où on sépare les points normaux des points anormaux minoritaires. Nous nous sommes penchés sur les deux solutions les plus populaires : celle de Schölkopf et celle de Tax & Duin.

2.1 OCSVM de Schölkopf

Un des précurseurs a été Schölkopf et son OCSVM[3] [4] en 1999 (ou ν-SVC) qui cherche un hyperplan séparant l'origine des données, en maximisant la marge du côté de l'origine. L'espace est séparé en deux : les points du côté de l'origine sont considérés comme négatifs, et les autres positifs.

Si on considère N vecteurs $\mathrm{x_i}$ de $\mathbb{R}^d$, l'objectif à minimiser est alors l'équation

$$\min_{\mathrm{w},\rho,\xi} \quad \frac{1}{2}||\mathrm{w}||^2 - \rho + \frac{1}{\nu N}\sum_i \xi, \tag{2}$$

sous les contraintes

$$\forall i \quad \mathrm{w} \cdot \mathrm{x_i} \geq \rho - \xi_i, \quad \xi_i \geq 0. \tag{3}$$

ρ représente la largeur de la marge, et w est un vecteur orthogonal à l'hyperplan. Ici, les ξ_i permettent une *soft margin*, c'est à dire que pendant l'entraînement, des points sont autorisés à être du mauvais côté du plan, mais ils ajoutent alors une pénalité au coût. Si $\mathrm{x_i}$ est du bon côté du plan, alors ξ_i est nul. Si on retire ces termes de (2) et de (3), on obtient des contraintes plus strictes, ou aucun point n'est autorisé à être du mauvais côté, c'est ce qu'on appelle une *hard margin*. L'hyperparamètre ν permet de contrôler la contribution de ces points mal placés. Plus ν est petit, plus on approche une *hard margin*.

À l'inférence, on évalue l'expression

$$\mathrm{sgn}(\mathrm{w} \cdot \mathrm{x} - \rho), \tag{4}$$

où sgn est la fonction de signe. Le résultat de (4) est -1 sur le point est anormal, du côté de l'origine, et 1 si le point est normal, de l'autre côté.

Cela n'est valable que pour un kernel linéaire. Le *kernel trick* fonctionne ici en appliquant une fonction K, qui peut être polynomiale ou RBF par exemple, à la place des produits scalaires $\mathrm{x_i} \cdot \mathrm{x_j}$ qui interviennent dans le développement du problème.

2.2 SVDD de Tax & Duin

La solution de Schölkopf est celle qui est implémentée dans les bibliothèques comme *scikit-learn*, mais il existe une autre solution qui est plus intuitive, celle de Tax & Duin[5][6]. Elle utilise une

hypersphère qui englobe le *cluster* de points positifs, et considère à l'inférence comme négatifs les points en dehors de cette région. Lors de l'entraînement, tous les points sont positifs et ceux en dehors de la sphère pénalisent la fonction de coût.

$$\min_{R,a,\xi} \quad R^2 + C\sum_i \xi_i \tag{5}$$

sous les contraintes

$$\forall i \quad ||\mathrm{x_i} - a||^2 \leq R^2 + \xi_i, \quad \xi_i \geq 0. \tag{6}$$

La sphère de centre a et de rayon R cherche donc à être la plus petite, tout en englobant le plus de points possible. Ici, plus C est grand, plus on s'approche d'une *hard-margin*.

À l'inférence, le point est considéré comme positif si il est à l'intérieur de l'hypersphère, on évalue donc

$$\mathrm{sgn}(R^2 - ||\mathrm{x} - a||). \tag{7}$$

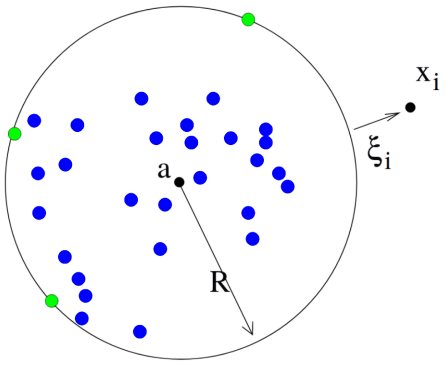

FIGURE 1 – SVDD (*soft margin*) avec les vecteurs de support en vert

Dans sa thèse[6], Tax affirme que le SVDD et le ν-SVC sont équivalent lorsque les données sont de norme 1, et notamment quand le noyau gaussien est utilisé, car celui-ci normalise implicitement les données. Le paramètre C est ainsi comparable au paramètre $\frac{1}{\nu N}$ de Schölkopf.

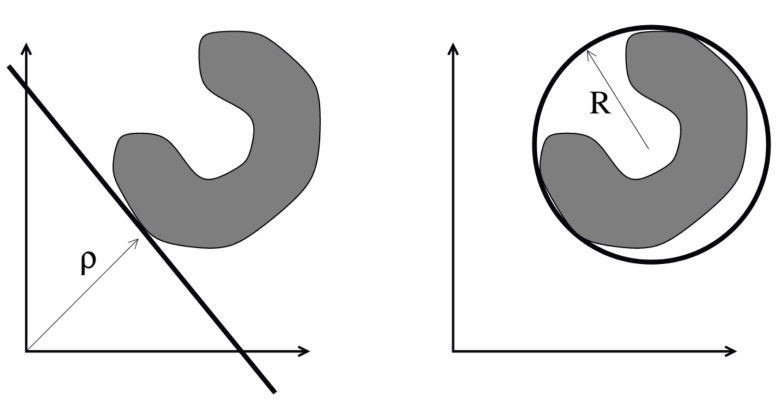

FIGURE 2 – Comparaison des solutions de Schölkopf (gauche) et de Tax (droite)

3 Implémentation dans Tensorflow

3.1 Fonction de coût et contraintes

Dans la majorité des cas, par exemple dans *LibSVM*[7], les problèmes de SVM sont résolus grâce à un optimiseur sous contraintes, ou en dérivant une forme *dual* de la forme *primal* (2 ou 5) grâce aux multiplicateurs de Lagrange. Nous avons choisi d'utiliser les optimiseurs par descente de gradient, natifs à *TensorFlow*. Pour cela, il a fallu combiner les contraintes et l'objectif dans une fonction de coût dérivable.

En prenant l'exemple du SVDD, on remarque, en observant (6), qu'on a

$$\xi_i = \begin{cases} 0 & \text{si } ||\mathrm{x_i} - a||^2 - R^2 \leq 0 \\ ||\mathrm{x_i} - a||^2 - R^2 & \text{sinon} \end{cases} \tag{8}$$

En effet, si $\mathrm{x_i}$ est dans la sphère, alors il ne pénalise pas l'objectif, sinon, il le pénalise proportionnellement sa distance au bord de la sphère. Plus le point est loin, plus il compte dans le coût. On a donc, d'après (8) :

$$\xi_i = \max(||\mathrm{x_i} - a||^2 - R^2, 0),$$

ou de manière équivalente

$$\xi_i = -\min(R^2 - ||\mathrm{x_i} - a||^2, 0). \tag{9}$$

La fonction de coût peut ainsi être écrite en une seule équation :

$$R^2 - C\sum_i \min(R^2 - ||\mathrm{x_i} - a||^2, 0). \tag{10}$$

Or, comme nous entraînons le modèle par *batch*, l'ordre de grandeur de la somme de (10) peut varier en fonction de la taille des batch. Pour résoudre cela, nous divisons cette somme par le nombre N d'éléments du *batch*.

$$R^2 - \frac{C}{N}\sum_i \min(R^2 - ||\mathrm{x_i} - a||^2, 0). \tag{11}$$

Ainsi, peut importe la taille des *batchs* choisie, le paramètre C jouera le même rôle. La fonction de coût (11) est directement implémentée dans *TensorFlow*, avec R une variable réelle, et a une variable de la même dimension que les données. L'optimisateur *Adam* est utilisé pour minimiser cette fonction par rapport à R et a.

Le même raisonnement est suivi pour le OCSVM de Schölkopf, en partant de (2) et de (3). On obtient alors l'objectif

$$\frac{1}{2}||\mathrm{w}||^2 - \rho - \frac{C}{N}\sum_i \min(\mathrm{w}\cdot\mathrm{x_i} - \rho, 0) \tag{12}$$

3.2 Les kernels

Comme nous avons intégré directement la forme *primal* de l'équation, il nous est impossible d'appliquer le *kernel trick*. Pour utiliser un noyau, on doit appliquer aux vecteurs une fonction de transformation Φ explicite.

Nous avons choisi d'implémenter l'équivalent du noyau gaussien. Le problème avec ce noyau, c'est que le *map* implicite associé n'est pas calculable, car de dimension infinie [8]. Or, il existe un noyau explicite qui approche le RBF, c'est le *Random Fourier Features Map*, ou RFFM [9]. Il existe dans *TensorFlow* une classe qui permet de calculer cette transformation, elle prend comme paramètres l'écart type (la largeur de RBF) et le nombre de dimensions de sortie de la transformation.

3.3 Estimator

TensorFlow propose une classe plus haut niveau pour l'apprentissage et l'inférence de modèle : l'*Estimator*. Nous avons choisi de nous baser sur cette classe pour encapsuler nos modèles.

Pour construire un `Estimator` personnalisé [10], il faut écrire une fonction `model_fn` qui décrit le graphe. C'est cette fonction qui va construire le graphe à chaque appel en mode `train` (apprentissage du modèle), `predict` (inférence) ou `evaluate` (calcul de performances). C'est dans cette fonction que sont définies les opérations d'entrainement et les `summaries` des variables, de la `loss` et des gradients. Les paramètres du modèle sont passés à la fonction dans un dictionnaire. Pour faciliter la création de nos modèles, nous avons créé une classe qui hérite de `tf.estimator.Estimator`, et qui prend un certain nombre de paramètres spécifiques à notre architecture.

Intérêt `Estimator` ont été récemment valorisé au cours des dernières conférences *TensorFlow*, notamment en 2017 et 2018. La communauté semble vouloir développer des classes plus haut niveau qui encapsulent la complexité de certains modèles, dans le but de pouvoir partager plus facilement des solutions utilisables directement et munies d'une interface commune (`train`, `predict`, `evaluate`). Notre projet s'inscrit parfaitement dans cette démarche : nous souhaitons mettre à disposition un algorithme d'*anomaly detection* implémenté dans *Tensorflow* et utilisable directement sans avoir à changer de *framework*.

3.4 Dataset

3.4.1 tf.data.Dataset

TensorFlow propose une catégorie de classes et fonctions, `tf.data.Dataset` pour construire des étapes de traitement des données. Un objet `Dataset` peut être utilisé pour représenter une *pipeline* comme étant une suite d'opérations logiques de transformations avec des outils issus de la logique fonctionnelle (`map`, `reduce`). En début de chaîne, les données sont lues par un `generator` python par *batch*, elles sont ensuite transformées et mises à disposition pour les algorithmes d'apprentissage qui suivent. D'autres opérations utiles peuvent être appliquées, comme un mélange (`shuffle`) ce qui est utile pour l'apprentissage notamment.

3.4.2 Pipeline

L'utilisation de l'`Estimator` nécessite d'avoir les données sous forme de vecteurs. Les données d'origine étant des images, il faut donc mettre en place une étape de *preprocessing*. Nous avons besoin de charger les données en mémoire successivement (via des *batchs*), les traiter (régler la taille, mise en format RGB), les mélanger lors de la phase d'apprentissage (pour éviter un phénomène d'apprentissage de l'ordre), les passer en inférence dans un CNN pour en extraire des *features* et donc obtenir un vecteur pour chaque image. Cette suite d'opérations est notre *pipeline* d'entrée des données.

3.4.3 Mode direct ou en cache

Passer les images en inférence dans un CNN coûte cher en calcul et en temps. De manière générale, nous serons dans deux cas de figure.

Le premier cas, le **mode direct**, utilise la chaîne en entier, du chargement de l'image d'origine à son utilisation dans l'`Estimator`. La *pipeline* est la suivante : une image est chargée depuis le disque dur, les transformations sont appliquées puis elle passe dans le CNN et le vecteur résultant est utilisé

dans le modèle. Ce processus se passe entièrement dans une même session *TensorFlow* (`sess.run()`). Du début à la fin, les opérations de la *pipeline* sont réalisées avec des opérateurs TF.

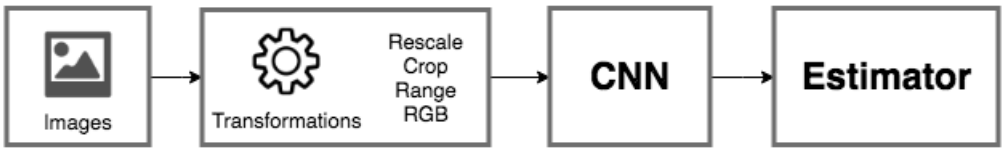

FIGURE 3 – Mode direct : *pipeline* unique et complète

Le deuxième cas, le **mode en cache**, est réalisé en deux temps. Le premier temps est celle de la mise en cache des sorties du CNN issues des images. Le processus est le même que lors du mode direct, mais les vecteurs de sortie du CNN sont stockés sur le disque dur dans des fichiers au format `TFRecord` (ce qui sera détaillé à la section suivante). Le but est de pré calculer ces sorties à l'avance pour économiser du temps de calcul à l'étape du CNN.

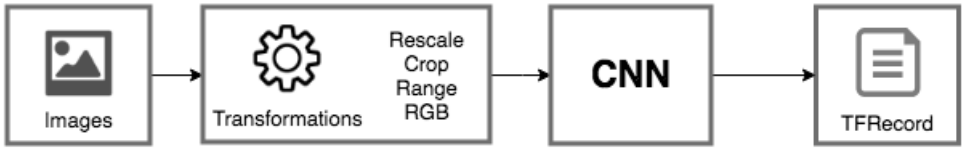

FIGURE 4 – Mode en cache : première *pipeline* pour la mise en cache des sorties du CNN dans des fichiers au format TFRecord

Le second temps est celui de l'utilisation de cette mise en cache. Une seconde *pipeline* lit les fichiers TFRecord puis les utilise dans le modèle `Estimator`.

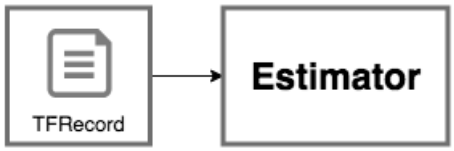

FIGURE 5 – Mode en cache : deuxième *pipeline* pour l'utilisation de l'`Estimator`

Ces deux modes offrent une certaine liberté d'utilisation. Le mode en cache trouve son intérêt pour la phase d'apprentissage. Pendant cette phase, le seul modèle qui est entraîné est celui de l'`Estimator` pour la détection d'anomalie. Le CNN, lui, sert à transformer les images en vecteurs de *features*. Ses poids se sont pas modifiés, ce n'est pas le but. Puisque les images doivent être utilisées plusieurs fois à l'apprentissage, le mode direct est très pénalisant car les images seraient passées dans le CNN plusieurs fois : il y aurait une forte redondance des calculs, ce qui est inutile.

Cependant, le second cas d'utilisation est celui de la production où de nouvelles images arrivent et il faut décider si elle comporte un défaut ou non. Dans ce cas, le mode direct trouve tout son sens.

L'image est lue, pré-traitée de la même manière qu'à l'entraînement puis passée dans le CNN et sa sortie est utilisée par l'`Estimator` pour fournir une prédiction.

En résumé, le mode direct est utilisé en production et le mode en cache pour l'apprentissage puisqu'il permet de s'affranchir de la redondance des calculs à travers le CNN.

3.5 TFRecord

Pour sauvegarder les vecteurs résultants du CNN, nous avions d'abord choisi le format `.npz`. Cependant, pour avoir une chaîne de chargement et de traitement totalement intégré dans *TensorFlow*, nous avons décidé de migrer vers les TFRecords. Lors de la sauvegarde, les vecteurs sont encodés en tant que listes d'octets dans un fichier *Protocol Buffer* en `.tfrecord`. Un autre fichier est utilisé pour stocker les statistiques sur les données : moyenne, variance, minimum et maximum, afin de faciliter une standardisation future. Pour chaque classe, 3 fichiers sont donc créés : **train.tfrecord**, **test.tfrecord** et **train_stats.tfrecord**. Concernant le chargement, les *Datasets* de *TensorFlow* proposent un outil pour charger directement les exemples, c'est-à-dire les vecteurs, depuis un TFRecord.

4 Benchmark

4.1 Indicateurs

Dans notre problème on désigne par **donnée positive** une image sans défaut. Une image avec défaut sera donc désignée par **donnée négative**. Nous présentons ici les métriques que nous utiliserons pour comparer les résultats.

La **matrice de confusion** permet de représenter visuellement les prédictions sur les données de test.

Vrais négatifs	Faux positifs
Faux négatifs	Vrais positifs

— **Vrai positif** : image sans défaut prédite sans défaut
— **Faux positif** : image avec défaut prédite sans défaut
— **Vrai négatif** : image avec défaut prédite avec défaut
— **Faux négatif** : image sans défaut prédite avec défaut

$$\text{Accuracy} = \frac{\text{Nombre de préditions correctes}}{\text{Nombre de prédictions totales}} \tag{13}$$

Une forte **accuracy** signifie que les prédictions faites sont correctes. Dans le cas de *datasets* déséquilibrés (ce qui est le cas pour les problèmes d'*anomaly detection*), l'**accuracy** seule n'est pas suffisante. En effet un modèle qui prédit tout le temps la classe majoritaire aura une forte **accuracy**, mais cela en dit trop peu sur la capacité du modèle à prédire la classe minoritaire.

$$\text{Precision} = \frac{\text{Vrai positifs}}{\text{Vrai positifs} + \text{Faux positifs}} \tag{14}$$

Une forte **precision** signifie que les données prédites positives sont effectivement des positives.

$$\text{Recall} = \frac{\text{Vrai positifs}}{\text{Vrai positifs} + \text{Faux négatifs}} \tag{15}$$

Un fort **recall** signifie que les données qui étaient labelisées positives ont bien été prédites positives.

$$\text{F1-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{16}$$

Le **F1-score** est la moyenne harmonique de la **precision** et du **recall**. Un fort **F1-score** prouve la capacité du modèle à prédire les classes tout en prenant en compte le déséquilibre du dataset, à la différence de l'**accuracy**.

4.2 Résultats

Chaque classe dispose de 150 images négatives en données de test. Pour calculer les performances, nous prendrons donc 300 images en tout, dont 150 positives et 150 négatives pour rester équilibré. Nous avons décidé de choisir le meilleur hyperparamètre par rapport à une classe donnée (ici la 6), et d'entraîner et de tester le modèle avec ce paramètre pour toutes les autres classes. Le CNN utilisé est VGG16[11].

	Classe 6	**Classe 5**	**Classe 4**	**Classe 3**	**Classe 2**	**Classe 1**
Accuracy	65.7	56.7	78.7	55.3	62.3	56.3
Precision	65.5	55.9	87.1	54.2	67.0	55.4
Recall	66.0	63.3	67.3	68.0	48.7	65.3
F1-score	65.8	59.3	75.9	60.3	56.4	60.0
Matrice de confusion	98 | 52 51 | 99	75 | 75 55 | 95	135 | 15 49 | 101	64 | 86 48 | 102	114 | 36 77 | 73	71 | 79 52 | 98

FIGURE 6 – Résultats du SVDD en choisissant le paramètre optimal de la classe 6 ($C = 3$, kernel linéaire)

On remarque que dans tous les cas, on obtient une **accuracy** de plus de 50%, ce qui est mieux que le hasard.

Pour certaines classes, comme la 2 et la 4, on classifie souvent les négatifs en tant que négatifs et ils ne sont manqués que rarement. En revanche, pour les classes 5, 3 et 1, les *outliers* ne sont pas bien reconnus, contrairement aux points positifs, ce qui n'est pas voulu dans l'industrie.

Même si les scores ne sont pas importants, on peut imaginer sur une chaîne de production un premier système de filtrage des pièces défectueuses avec ce SVDD, puis un second, manuel ou avec un système plus précis. Pour cela il nous faudrait un fort taux de vrais négatifs et un faible taux de faux positifs, afin de ne pas laisser passer des éléments négatifs. Cela est par exemple le cas avec notre classe 4.

On peut en conclure que le modèle SVDD est capable de généraliser un minimum à d'autres données, en gardant les mêmes hyperparamètres (ici C).

Remarque sur OCSVM Le modèle de Schölkopf ne nous donne pas de bons résultats et c'est en effet lié à la nature de ce modèle. Le but de ce modèle est de séparer les données par rapport à l'origine. Dans le cas où le noyau est linéaire, il n'y a pas de transformation des données dans un nouvel espace. Le modèle est alors adapté dans le cas où les données sont effectivement linéairement séparables et qu'elles sont d'un côté seulement par rapport à l'origine. Étant donné notre problème, il n'y a aucune raison de valider ces hypothèses, en particulier leur position par rapport à l'origine. Dans le cas où le noyau est gaussien, ces hypothèses difficiles ne sont plus à considérer. Cependant, comme nous l'avons expliqué précédemment, l'implémentation du noyau gaussien est compliquée et il n'est possible que de faire une approximation finie des dimensions. Que ce soit pour le SVDD ou l'OCSVM, le noyau

gaussien ne donne pas de résultats satisfaisants pour notre problème. Pour ces raisons, nous n'avons reporté les résultats sur DAGM qu'avec le SVDD.

	Classe 6	**Classe 5**	**Classe 4**	**Classe 3**	**Classe 2**	**Classe 1**
Accuracy	65.0	53.6	55.0	53.0	46.7	51.0
Precision	69.2	53.4	56.1	53.3	46.4	51.1
Recall	54.0	58.0	46.0	48.7	43.3	48.0
F1-score	60.6	55.6	50.5	40.8	44.8	49.5
Matrice de confusion	114 | 36 69 | 81	74 | 76 63 | 87	96 | 54 81 | 69	86 | 64 77 | 73	75 | 75 85 | 65	81 | 69 78 | 72

FIGURE 7 – Résultats du `OneClassSVM` de *Scikit-Learn* en choisissant le paramètre optimal de la classe 6 ($\nu = 0.5$, kernel linéaire)

Nous nous sommes intéressés aux performances obtenues en entraînant avec le modèle `OneClassSVM` de *Scikit-Learn* sur les mêmes données. Le but est de comparer l'efficacité de notre implémentation. On remarque en lisant le tableau que les performances avec *Scikit-Learn* sont du même ordre de grandeur que nos résultats avec notre `Estimator` SVDD dans *TensorFlow*. Pour la majorité des classes, notre SVDD est plus performant en terme de F1-score et arrive mieux à généraliser que le OCSVM de Schölkopf implémenté dans *Scikit-Learn*.

On remarque que le OCSVM de *Scikit-Learn* converge, contrairement à notre implémentation, ce que nous pouvons expliquer par plusieurs raisons :

— Dans *Scikit-Learn*, le modèle OCSVM est le problème d'optimisation sous contrainte résolu avec un solveur quadratique, ce qui rend le respect des contraintes plus stable. En conséquence, la fonction objective devient plus stable et on peut penser que la convergence vers la solution est elle aussi plus stable.

— Puisque le problème est résolu classiquement par un solveur sous contrainte, toutes les données doivent être chargées en mémoire, il n'y a pas d'entraînement par *batch* à la différence de notre implémentation dans *TensorFlow*. La convergence est donc plus stable.

— Enfin, *Scikit-Learn* se base sur la bibliothèque *LibSVM* qui implémente les différents modèles SVM dont celui de l'OCSVM. Cette bibliothèque est considérée comme étant l'état de l'art en termes de SVM. On peut penser que leur implémentation est plus poussée mathématiquement et optimisée numériquement.

5 Conclusion

5.1 À propos des noyaux

Dans nos tests, nous nous sommes limités au noyau linéaire, par manque de temps. Le papier de Tax suggère cependant que le modèle SVDD est le plus efficace avec un kernel RBF, car la sphère peut ainsi épouser des formes plus complexes, ou même plusieurs *clusters*.

De manière identique, la solution de Schölkopf n'est pas la meilleure avec un kernel linéaire, car cela revient à couper l'espace vectoriel en 2 par un hyperplan (et les problèmes ne sont pas toujours linéairement séparables).

5.2 Implémentation dans *TensorLayer*

Avant de nous pencher vers l'`Estimator`, nous avions essayé d'implémenter notre modèle de SVDD en tant que `layer` avec la bilbiothèque *TensorLayer*. Cela aurait permis d'empiler facilement un CNN

avec notre solution. Nous nous sommes retrouvés face à un problème : notre modèle, contenant deux variables, est dépendant de la fonction de coût utilisée. Nous n'avons trouvé aucun moyen d'intégrer cette `loss` dans un `layer` classique, qui est fait pour être empilable à d'autres couches suivantes, à moins de créer une classe héritant de `Layer` avec une fonction `train`, et qui est condamnée à être la dernière couche. Nous avons trouvé la solution des `Estimators` plus propre et plus adaptée à notre problème.

5.3 Preprocessing

Les vecteurs de sortie de VGG16 sont de dimension 25088. La majorité de ces composants est nulle, ou influe très peu la classification. Une PCA aurait peut-être été adaptée ici, pour réduire le nombre de dimensions et obtenir des composantes qui expliquent mieux la variance. La dimension des vecteurs à l'entrée du modèle aurait donc été beaucoup plus faible et l'utilisation du noyau RBF auraît sûrement amélioré les scores.

Il aurait aussi été judicieux de standardiser les données (moyenne nulle, écart-type à 1). Cela aurait aidé le modèle à généraliser sur d'autres données, notamment sur les autres classes de notre jeu de données.

Pour finir, dans sa thèse, Tax aborde le fait de normaliser les vecteurs, après avoir ajouté une dimension aux vecteurs pour ne pas perdre d'information. Avec cette technique, le SVDD et l'OCSVM sont équivalents [6].

$$x' = \frac{(x, 1)}{||(x, 1)||}$$

5.4 Choix du CNN

Nous avons effectué tous nos tests et nos optimisations sur VGG16, mais InceptionV3 et InceptionV4 étaient à notre disposition. Il aurait fallu tester différents CNN, qui génèrent des vecteurs *feature* de différentes tailles et natures.

5.5 Optimisation sous contraintes

Nous avons intégré les contraintes dans une `loss`, avant de l'optimiser par descente de gradient, mais nous aurions pu implémenter d'autres techniques, comme la méthode du gradient projeté, ou alors utiliser le nouvel optimiseur sous contraintes de *TensorFlow* 1.9 [12].

Également, nous avons traité le problème *primal* pour des questions de compréhension des paramètres. La version *dual* de ces modèles pourrait être testée pour vérifier si elle offre des avatages en termes numériques ou en termes de performances.

5.6 Bilan

Ce projet nous a permis de comprendre la logique du *framework* de *deep earning TensorFlow* et les outils associés (`Dataset`, `Estimator`, ...). Nous avons appris qu'il est parfois préférable de sacrifier la précision du modèle pour optimiser la vitesse des calculs en production et rester dans le même *framework*.

Nos connaissances théoriques sur les CNN, la classification à une classe et les réseaux de neurones ont été profondément renforcées. Nous nous sommes rendu compte de l'importance de s'approprier les mathématiques pour mieux comprendre les travaux de recherche, les implémenter, voir les améliorer.

Références

[1] Stephen Marsland. "Novelty Detection in Learning Systems". In : 3 (nov. 2001).

[2] *DAGM Dataset*. http://resources.mpi-inf.mpg.de/conferences/dagm/2007/prizes.html. Accessed : 2018-06-30.

[3] Bernhard Schölkopf et al. "Support Vector Method for Novelty Detection". In : 12 (jan. 1999), p. 582–588.

[4] Bernhard Schölkopf et al. *Estimating the Support of a High-Dimensional Distribution*. Rapp. tech. Nov. 1999, p. 30. URL : https://www.microsoft.com/en-us/research/publication/estimating-the-support-of-a-high-dimensional-distribution/.

[5] David M.J. Tax et Robert P.W. Duin. "Support Vector Data Description". In : *Machine Learning* 54.1 (jan. 2004), p. 45–66.

[6] David Tax. "One-Class Classification ; Concept-Learning In The Absence Of Counter-Examples". In : (jan. 2001).

[7] *LibSVM - A Library for Support Vector Machines*. https://www.csie.ntu.edu.tw/~cjlin/libsvm/. Accessed : 2018-06-30.

[8] *Why does the RBF kernel map into infinite dimensional space*. https://www.quora.com/Why-does-the-RBF-radial-basis-function-kernel-map-into-infinite-dimensional-space-mentioned-many-times-in-machine-learning-lectures?share=1. Accessed : 2018-06-30.

[9] *Improving Linear Models Using Explicit Kernel Methods*. https://www.tensorflow.org/versions/master/tutorials/kernel_methods. Accessed : 2018-06-30.

[10] *Creating Custom Estimators*. https://www.tensorflow.org/get_started/custom_estimators. Accessed : 2018-06-30.

[11] Karen Simonyan et Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In : *CoRR* abs/1409.1556 (2014). arXiv : 1409.1556. URL : http://arxiv.org/abs/1409.1556.

[12] *ConstrainedOptimization (TFCO)*. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/constrained_optimization. Accessed : 2018-06-30.