Convolutional Neural Network programs modeling for automatic detection of design smells

Anonymous Author(s)

***Abstract*—TODO: To be completed**

***Keywords*—** Deep Learning, Convolutional Neural Network, Design Smells, Famix, Modeling, Software Engineering

I. Introduction

Deep learning is a subset of machine learning that uses artificial neural networks to learn from unstructured data. Deep learning has become very popular in recent years due to its performance in a number of fields, such as image recognition, speech recognition, machine translation and more. Artificial neural networks are mathematical models inspired by the workings of the human brain. They are composed of several layers of interconnected neurons. Each neuron has an activation function that calculates the neuron's output according to its inputs. There are several types of artificial neural network architecture. These architectures have been developed to address specific problems, such as convolution architecture for image recognition, recurrent neural networks for speech recognition, and so on. In this paper, we focus on convolution architecture, as it can handle a wide range of problems, such as image recognition, text recognition and sequence recognition. Moreover, convolution architecture is part of feed-forward architectures, a class of artificial neural networks composed of several layers of neurons and containing no cycles. Feed-forward neural networks are the most widely used in deep learning.

A. Convolutional Neural Network

In this paper, we will refer to deep learning programs as those containing convolutional neural networks (CNN). Convolutional Neural Network consists of the following three main stages:

Convolution: Convolution is the action of passing filters over an image to extract patterns. A filter is a classical neural network that scans an image part by part to detect a pattern. The filter scans a window (part) of the image and then calculates the output in one pixel. The filter then browses the entire image to calculate the output of each pixel in a new image. The new image is called a feature map. Convolution increases the depth of the input image by generating multiple feature maps while reducing its size. Recall that each depth layer is an image (reduced in size) generated by a filter.

Pooling: Pooling is an operation that reduces image size by selecting the largest pixel in a window. This reduces computation and memory consumption. It also reduces Overfitting on training data, as information on the position of detected patterns is lost. So we're going to use pooling on the feature maps generated by convolution. Convolution is followed by pooling, and this operation is repeated several times. The deeper we go in the convolution, the more precise patterns are detected, likely to facilitate classification.

Flatten: The flatten operation transforms the feature maps generated by convolution and pooling into a single vector. This vector is then used as the input to a classical neural network (*dense layer*). The output of the convolutional neural network will come from this neuron.

B. Design smell

A code smell is a poor design and implementation choice that can have a negative impact on software quality [1]. Just like any other program, deep learning programs can contain code smells. In this paper, we focus on design smells because they are introduced early in the software development cycle and can have a significant negative impact on software performance and quality. These smells are introduced by the developer during the software design phase.
Nikanjam et al. proposed a list of 8 design smells for feed-forward learning programs [**?**]. They found these smells in literature reviews and the open-source platforms Github and StackOverflow. Their empirical study showed that the proposed smells are perceived as relevant by developers. We have therefore used this list of smells to propose an automatic smell detection system for developers.
The 8 proposed smells are classified into two main categories as follows:

1) **Design smells during convolution and pooling**: These smells are introduced when feature maps are formed. They are linked to the size of the feature map or filter and the layer layout. The list of smells is presented in the table I below.
2) **Design smells linked to the use of regularization methods**: These smells are introduced when using regularization. They are linked to the use of regularization methods such as the dropout layer and the layout of these methods. The list of smells is presented in the table II below.

C. Modeling

Thomas Kühne defines a model as an artifact formulated in a modeling language, such as UML, that describes a system using different types of diagrams [2]. This abstract representation of a system is defined by another model called a metamodel. A model allows a system to be analyzed, understood and transformed efficiently. Its operations can be automated [3]. There are several types of models, such as behavioral models (representing the dynamic behavior of the system) or design models (representing the static structure of the system).
In this paper, we are interested in design models because the smells studied are represented in static form and are introduced at the system structure level.

We'll also be using ***FAST*** (*Famix Abstract Syntax Tree*) models to represent the systems studied in this paper. The *FAST* models are syntax trees derived from the Pharo object-oriented programming language [4], [5], [6]. It inherits the ***Famix*** model, which is defined by Tichelaar et al. as a language-independent representation of object-oriented source code. It is an entity-relationship model that models object-oriented source code at the program entity level [7] [8]. The *FAST* models are developed in the Moose retroengineering environment [9].

D. Motivation and research questions

In Nikanjam et al.'s study, developers identified not only the proposed design smells, but also their relevance and hence the value

Table I: *Design smells in convolution and pooling layers*

Design Smell	Description
Non-expanding feature map	Keep the same number of features or reduce it as the architecture becomes deeper.
Losing local correlation	Start with a relatively large window size for spatial filtering and maintain it for all convolutional layers.
Heterogeneous blocks of CNNs	Build a deeper model by stacking only a set of convolution and pooling layers without appropriate configuration.
Too much down-sampling	Pooling right after each convolutional layer, especially for the first layers.
Non-dominating down-sampling	Use of average-pooling.

Table II: *Design smells related to regularization*

Design Smell	Description
Useless Dropout	Using Dropout before pooling layers.
Bias with Batchnorm	Keep the bias values in the layers when using batchnorm. FNN learning layers are biased with different initializations.
Non-representative Statistics Estimation	Use of batchnorm after dropout.

of avoiding them. However, there is no automatic smell detection system in the literature. In this paper, we propose such a system, which will enable smell detection in CNN deep learning programs. Moreover, our system will cover the most widely used open source Frameworks on the market. To this aim, we have defined the following research questions:

RQ1: Is it possible to detect design smells in CNN deep learning programs with modeling?

RQ2: What are the most prevalent design smells in CNN deep learning programs?

RQ3: Are there any correlations between the design smells in the programs?

Our paper is organized as follows: section II presents past work related to our topic. Section III describes the data collection and processing. Section IV presents the results. Section V discusses the results presented above. Section VI presents threats to validity and section VII presents the limitations of our study and future work.

II. Background

Dans cette section nous présenterons les précédents travaux sur la détection des ordeurs de code dans les logiciels.

L'idée de détecter des ordeurs dans les codes sources des logiciels est un sujet beaucoup étudié dans la littérature. En effet, plusieurs travaux proposent des techniques de détection depuis plus d'une décénie. Les odeurs de code sont étudiées selon plusieurs axes, comme par exemple leur présence dans des logiciels développés dans un langage de programmation spécifique [10], la sécurité dans le code source d'Infrastructure as Code [11], dans les microservices [12], ou leur impact sur la performance des applications Android [13]. Fard et al. [14] propsent JSNOSE (precision: 93% and average recall: 98%) une technique de détection des ordeurs de code du langage JavaScript. Une combinaise d'analyses statique et dynamique pour détecter des odeurs de code coté client. Chen et al. [15] proposent quant à eux Pysmell, un outil capable de détecter des odeurs de code du langage Python avec une précision moyen de 97.7%. La détection d'odeurs de code a un impact sur la qualité du logiciel, Van Emden et al. [16] font de l'assurance qualité de code Java via la détection et la visualisation d'odeurs de code. Pour ne citer que ces exemples, Moha et al. [17] proposent une approche de spéccification et de détection d'odeurs de code nommée DECOR avec une precision de 60.5%, et un recall de 100%. On trouve également dans la litterature des approches de détection d'odeurs basées sur la modélisation, comme l'approche proposée par Araujo et al. [18] pour identifier les odeurs organisationnelles dans les système multi-agents via la métamodélisation, ou celle de Khomh et al. [19] qui convertissent les règles de détection dans la litterature en un modèle probabilistique dans le but de détecter les odeurs de code. Ce dernier ouvre la voie à la détection d'odeurs de code avec l'aide de l'intelligence artificielle. On trouve de ce fait, dans les dernières années plusieurs travaux proposant des approches de détection d'odeurs de code basées sur l'intelligence artificielle [20], [21], [22]. À la lumière de ces travaux, nous nous sommes posés les questions suivantes: **Est-il possible de détecter efficacement et de manière statique les odeurs de code dans les logiciels d'apprentissage profond?**
Dans la litterature Nikanjam et al. ont initer la réponse à ces questions en proposant NeuraLint (precision: 100% and recall: 70,5%) [23], une approche basées sur modèle de détection d'odeurs de code. Cette approche utilise la métamodelisation pour atteindre son but. Cependant la porté de NeuraLint ne s'est limité qu'aux architectures feedforward multilayer perceptron (MLP). L'apprentissage profond est un domaine en pleine expansion, et les architectures de réseaux de neurones sont de plus en plus complexes. Il est donc nécessaire de pouvoir détecter des odeurs de code d'autres architectures de réseaux de neurones. L'architecture CNN étant une des plus utilisée dans la pratique, nous nous sommes fixés dans ce papier, l'objectif de permettre la détection d'odeurs de code dans les réseaux de neurones de type CNN.

III. Study Design

In this sub-section, we describe the details of the data collection and processing approach followed to answer our different research questions.

A. Data Collection

Le développement du système de détection d'odeurs de conception s'est fait à partir des exemples de code de programmes d'apprentissage profond. Ces exemples représentent des programmes d'apprentissage profond dans lesquels on y trouve des odeurs de conceptions. Ses exemples ont été collectés par Nikanjam et al. [24] dans le cadre de leur étude empirique sur les odeurs de conception dans les programmes d'apprentissage profond. En effet, dans cette étude, ils ont présenté des exemples de programmes d'apprentissage profond contenant les ordeurs de conceptions énumérées dans l'introduction I-B. Ces exemples de code source ont été collectés à partir des plateformes StackOverflow et Github.

Le système de détection d'odeurs de conception ainsi développé a servi sur un ensemble de programmes d'apprentissage profond collectés à partir de la plateforme Github. Ces programmes d'apprentissage profond ont été collectés selon un processus bien défini. En effet, nous avons utilisé l'API de Github et plus précisement les requêtes de recherche de type *search code*. Cette

requête permet de rechercher des fichiers dans les dépôts Github contenant des mots clés spécifiques définis dans notre système. Étant donné que notre papier se concentre uniquement sur les libraires *Keras*, *Tensorflow* et *Pytorch*, nous avons utilisé les mots clés présentés dans le tableau III. Ces mots clés représentent les modules et les fonctions de ces trois librairies.

Table III: *Liste des mots clés utilisés pour la recherche de programmes d'apprentissage profond dans les dépôts Github.*

Mots clés	**Librairie**
keras.layers	Keras
keras.layers.convolutional	Keras
AveragePooling2D	Keras/Tensorflow
MaxPooling2D	Keras/Tensorflow
tensorflow.keras.layers	Tensorflow
Conv2D	Keras/Tensorflow
Convolution2D	Keras/Tensorflow
BatchNormalization	Keras/Tensorflow
import torch	Pytorch
import torchvision.models	Pytorch
torch.nn.Sequential	Pytorch
torch.nn.Conv2d	Pytorch
torch.nn.BatchNorm2d	Pytorch
torch.nn.MaxPool2d	Pytorch

La requête de recherche de type *search code* retourne un ensemble d'informations sur le repository et le fichier contenant les mots clés recherchés. Cette collecte retourne un ensemble de données brute de 9572 repositories Github diférents. Nous sélectionnons ensuite aléatoirement un sous ensemble de 10% de notre ensemble de données brutes, soit 958 repositories Github afin d'optimiser l'étape de filtrage manuel ci-après.

À partir de ce sous ensemble, nous procédons ensuite à un filtrage des répertoires selon les critères suivants: (1) nombre de commit ($\geq$ 100), (2) nombre d'étoile ($\geq$ 100), (3) dernière date du commit($\leq$ 5 years), (4) nombre de contributeurs ($\geq$ 5). Ce filtrage nous permet de ne garder que les répertoires qui sont les plus populaires et qui sont les plus actifs. Nous avons ensuite procédé à un filtrage manuel des répertoires en éliminant les projets qui ne sont pas des programmes d'apprentissage profond ou ne pas pertinente pour notre étude (n'utilise pas les librairies choisies, ou n'implémente pas de réseau de neurones). De ce fait nous nous sommes retrouvé avec 500 projets sur lesquels nous allons appliquer notre système de détection d'odeurs de conception.
Le schema 1 présente le processus de collecte des programmes d'apprentissage profond.

B. Data Processing

Le processus de collecte des programmes d'apprentissage profond nous a permis d'avoir un ensemble de programmes d'apprentissage profond sur lesquels nous allons appliquer notre système de détection d'odeurs de conception. Cependant, il est important de noter que les programmes d'apprentissage profond sont dans notre cas des programmes uniquement écrits en langage Python. Il est donc essentiel de noter que, le système de détection d'odeurs de conception développé dans ce papier est un système qui est uniquement capable de détecter les odeurs de conception dans les programmes d'apprentissage profond écrits en langage Python.

Afin de répondre à la question de recherche *RQ1*, nous avons développé un système de détection d'odeurs de conception dans les programmes d'apprentissage profond. Ce système se décompose en trois parties. La première partie est la partie relative à la modélisation des programmes d'apprentissage profond. La deuxième partie est celle relative à la détection des odeurs de conception dans les programmes d'apprentissage profond. Et la troisième est la partie relative à l'analyse des résultats de la détection.

1) Modélisation des programmes d'apprentissage profond: La modélisation des programmes d'apprentissage profond est la première étape du processus de détection des odeurs de conception. Cette étape consiste à transformer le code source des programmes d'apprentissage profond collecté en un modèle intermédiaire qui sera utilisé pour la détection des odeurs de conception. Ce modèle intermédiaire est un modèle de type *Famix Abstract Syntax Tree* (*FAST*) 2. Le modèle *FAST* est un modèle qui est utilisé pour représenter les programmes sous forme d'arbre syntaxique. Il hérite de la représentation abstraite de codes sources *Famix* dévéloppé dans le langage de programmation *Pharo*. Avec *Famix*, le code source est représenté sous forme d'objet et selon un métamodèle défini pour symplifier l'analyse et la représentation de programmes complexes. Un modèle *Famix* est une représentation agnostique en terme de langage du code source d'origine et embarque des fonctions qui permettent entre autres sa navigation et sa transformation.

La transformation du code source collecté en modèle *FAST* se fait comme suit. Chaque projet collecté est cloné dans un répertoire local. Son code source est ensuite parsé à l'aide de la librairie *ast.py* de Python. Cette librairie permet de transformer le code source en un arbre syntaxique sous forme *json*. Le *json* obtenu est ensuite transformé en un modèle *FAST* à l'aide d'un importateur développé sous le patron de conception visiteur. Les visiteurs sont des fonctions qui permettent de parcourir un arbre syntaxique pour effectuer sur ses noeuds des opérations spécifiques. Dans notre cas, nous avons développé des visiteurs qui nous permettent de parcourir l'arbre syntaxique (*json*) et de transformer chaque noeud de cet arbre en une entité de *FAST*.

Le schema 3 illustre le processus de modélisation des programmes d'apprentissage profond.

2) Détection des odeurs de conception: Le modèle *FAST* obtenu à partir de la modélisation des programmes d'apprentissage profond est ensuite utilisé pour la détection des odeurs de conception. Cette étape consiste à parcourir le modèle *FAST* et à appliquer les règles de détection des odeurs de conception. Ces règles sont définies dans le langage de programmation *Pharo*.
Une règle ou plusieurs règles peuvent être appliquées sur un noeud de l'arbre syntaxique afin de détecter une odeur de conception.

Le schema 4 illustre le processus de détection des odeurs de conception.

3) Analyse des résultats de la détection des odeurs de conception: Les résultats de la détection des odeurs de conception de chaque repository sont stockés sous forme *csv*. Pour chaque repository nous enregistrons le nom de chaque odeur de conception détectée, le nombre de fois que cette odeur est détectée dans un fichier du repository et le chemin de ce fichier. Ces résultats permettent de répondre aux questions de recherche *RQ2* et *RQ3*.
On calcul ensuite la répartition des odeurs de conception dans chaque repository en divisant le nombre de fois qu'une odeur de conception est détectée par le nombre total d'odeurs de conception détectées dans le repository. Cette répartition permet de répondre à la question de recherche *RQ2*. On calcul également la répartition des odeurs de conception dans l'ensemble des 500 repositories en divisant le nombre de fois qu'une odeur de conception est détectée par le nombre total d'odeurs de conception détectées dans l'ensemble des 500 repositories. Cette répartition permet aussi de répondre à la question de recherche *RQ2*.

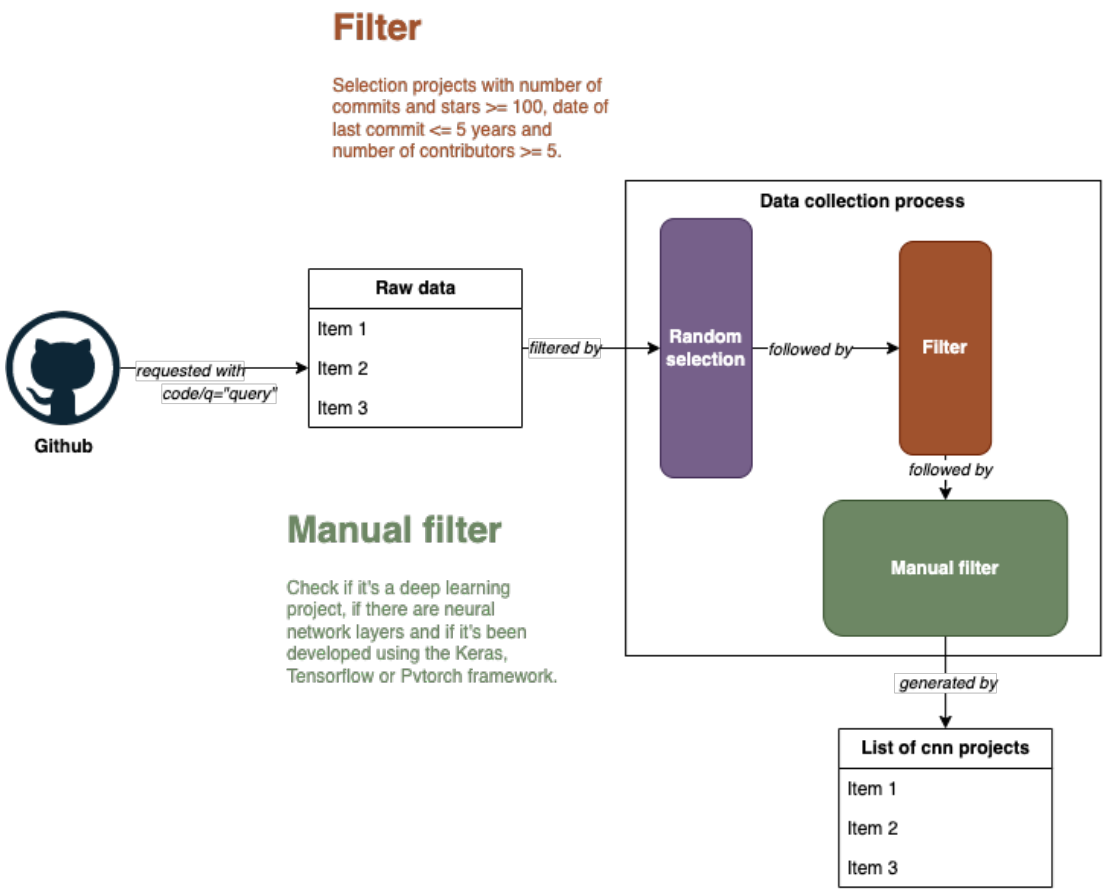

Figure 1: *Processus de collecte des projets d'apprentissage profond dans Github.*

On calcul ensuite la matrice de corrélation entre les odeurs de conception dans l'ensemble des 500 repositories. Cette matrice permet de répondre à la question de recherche *RQ3*. Une matrice de corrélation permet de calculer la corrélation entre deux variables.

Un coeficient de corrélation est calculé à l'aide de la formule suivante: $r = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$ où r est le coeficient de corrélation, $cov(X,Y)$ est la covariance entre les deux variables X et Y, σ_X est l'écart type de la variable X et σ_Y est l'écart type de la variable Y. Ce coeficient de corrélation est compris entre -1 et 1. On parle de correlation positive lorsque le coeficient de corrélation est compris entre 0 et 1. Dans ce cas, plus il est proche de 1, plus les deux variables sont corrélées positivement, et plus il est proche de 0, moins les deux variables sont corrélées. On parle de correlation négatif lorsque le coeficient de corrélation est compris entre -1 et 0. Dans ce cas, plus le coeficient de corrélation est proche de -1, plus les deux variables sont corrélées négativement, et plus il est proche de 0, moins les deux variables sont corrélées.

Dans notre cas, les variables sont les odeurs de conception. La corrélation entre deux odeurs de conception est calculée en divisant le nombre de fois que les deux odeurs de conception sont détectées ensemble par le nombre total de fois que les deux odeurs de conception sont détectées. Cette corrélation est calculée pour chaque paire d'odeurs de conception. Plus elle est proche de 1, plus les deux odeurs de conception sont corrélées, et plus elle est proche de 0, moins les deux odeurs de conception sont corrélées.

Une corrélation positive signifie que les deux odeurs de conception sont souvent détectées ensemble. Et une corrélation négative signifie que les deux odeurs de conception sont rarement détectées ensemble. Enfin une corrélation nulle signifie que les deux odeurs de conception ne sont jamais détectées ensemble.

Le schema 5 illustre le processus de cette dernière partie.

C. Replication Package

Le code source du parse de code source en arbre de Syntax *json* est sur Github à l'adresse https://github.com/aurpur/parserPythonToJson et celui du système de détection d'odeurs est disponible sur Github à l'adresse https://github.com/aurpur/famixPythonImporter. Le système de collection de données est aussi disponible sur Github à l'adresse https://github.com/aurpur/ms-github-data-collection.

IV. Case Study Results

Dans cette section nous présentons les résultats de notre étude. Nous présentons d'abord les résultats relative à la détection des odeurs de conception dans les programmes étudiés. Puis, nous présentons les résultats relative de l'analyse des résultats. **TODO: To be completed**

A. Détection des odeurs de conception

Ici nous présenterons les chiffres sur la détection des odeurs de conception dans l'ensemble des repositories tel que la répartition des

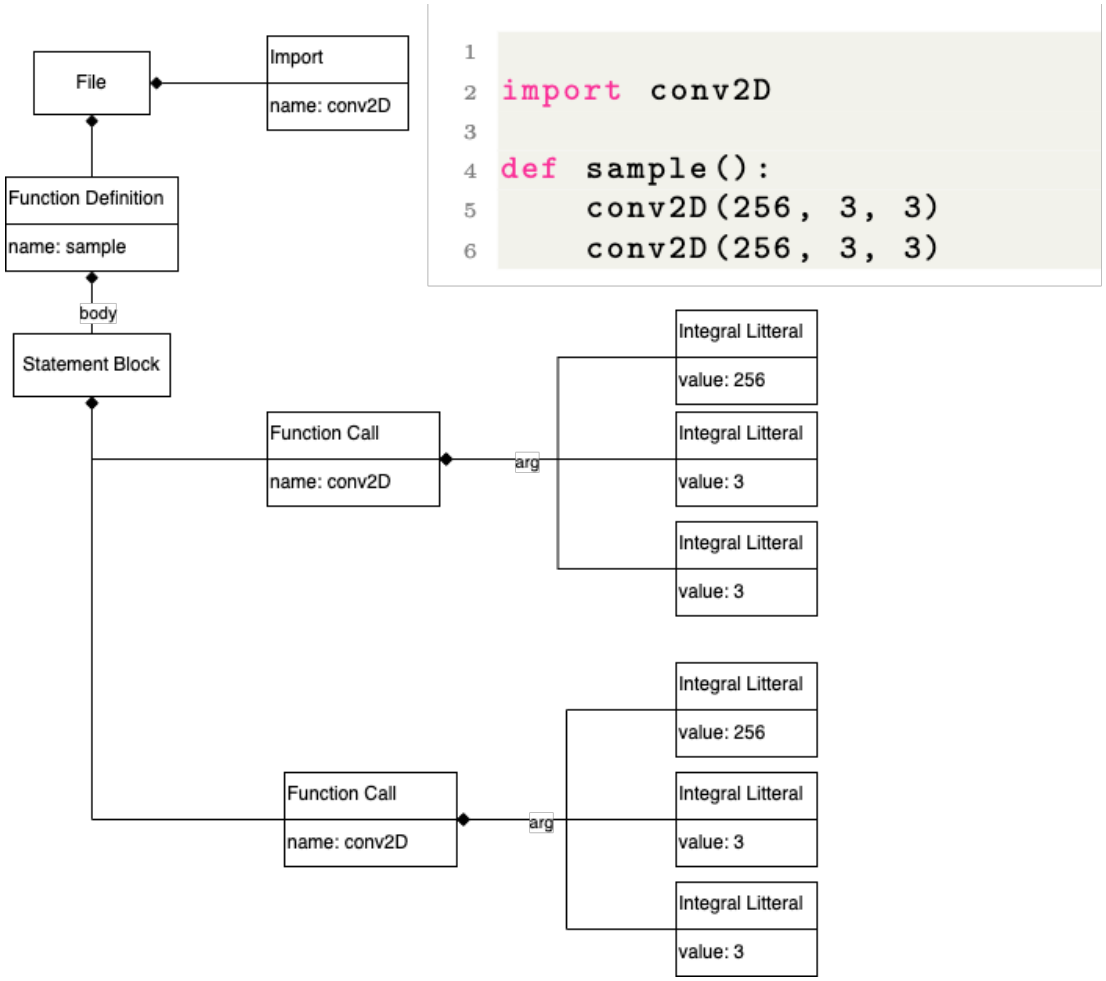

Figure 2: *Exemple d'un modèle* FAST *d'un programme fictif d'apprentissage profond.*

odeurs de conception dans le dataset (Le nombre de repositories par odeurs de conception), ou la profondeur (Nombre d'occurence d'une odeur de conception dans un repository).

Cela soutiendra l'hypothèse selon laquelle les odeurs de conception peuvent être détectées sur les programmes CNN à travers un model *FAST* dans Pharo. Et de ce fait, on peut donc détecter les odeurs de conception à l'aide de la modélisation.

B. Analyse des résultats de la détection

Ici nous présenterons les chiffres sur l'analyse des résultats de la détection des odeurs de conception dans les programmes étudiés. Le but est de présenter le classement des odeurs de conception, et de présenter les chiffres sur les possibles correlation entre elles.

Cela soutiendra l'hypothèse selon laquelle il y a des odeurs de conception qui sont plus répandu que d'autres dans les programmes CNN (dans le référentiel de notre étude) et que certaines odeurs de conception présentent des correlations entre elles.

V. Discussion

Dans cette section, nous discutons les résultats de notre étude et répondons à nos questions de recherche. Nous discutons également des conséquences de nos résultats dans le contexte de la recherche et du développement de logiciels. **TODO: To be completed**

A. RQ1: Is it possible to detect design smells in CNN deep learning programs with modeling?

Ici nous discuterons des résultats de notre première question de recherche. Nous discuterons de la technique de détection des odeurs de conception proposée et de sa performance. Nous calculerons l'exactitude, la précision et le rappel de la technique proposée. On discutera également du temps d'exécution du système proposée.

Nous parlerons de l'objectif derrière la question (Motivation), rappelerons la méthode utilisée et les trouvailles.

B. RQ2: What are the most prevalent design smells in CNN deep learning programs?

Ici nous discuterons des résultats de notre deuxième question de recherche. Nous discuterons de la répartition et la profondeur des odeurs de conception dans l'ensemble de données.

Nous parlerons de l'objectif derrière la question (Motivation), rappelerons la méthode utilisée et les trouvailles.

C. RQ3: Are there any correlations between the design smells in the programs?

Ici nous discuterons des relations entre les odeurs de conception dans les programmes d'apprentissage profond.

Nous parlerons de l'objectif derrière la question (Motivation), rappelerons la méthode utilisée et les trouvailles.

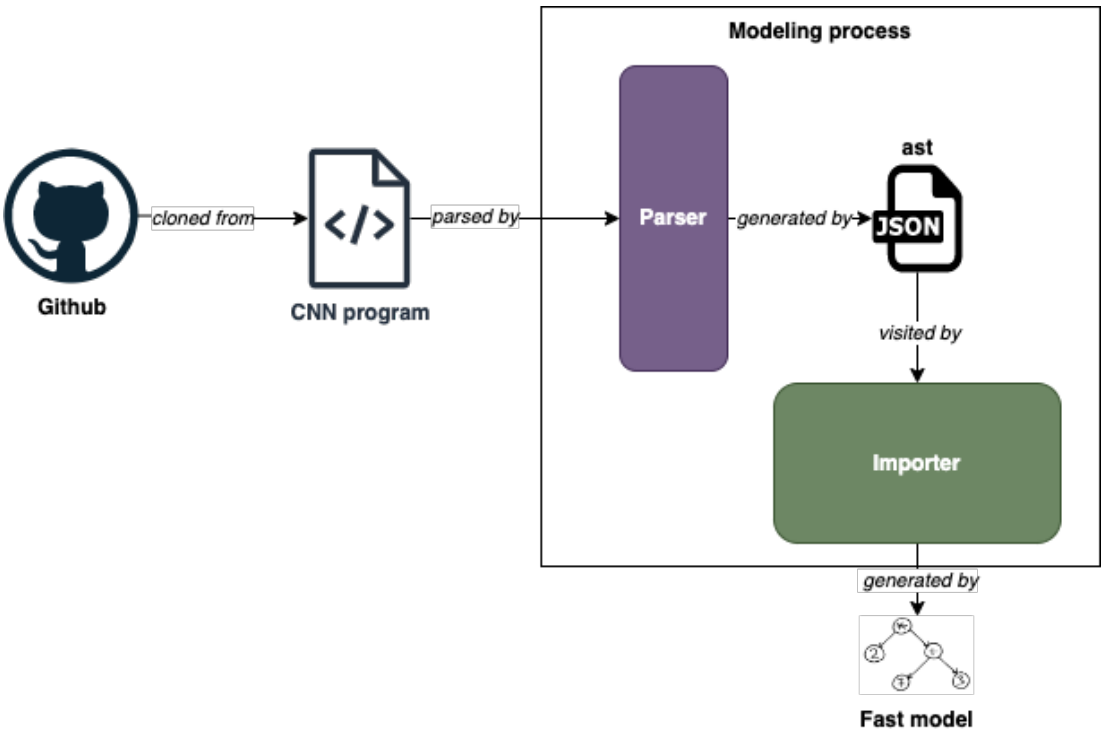

Figure 3: *Processus de modeling des projets d'apprentissage profond.*

D. Implications

Ici nous parlerons de l'impact de notre étude sur la recherche et le développement de logiciels. Nous parlerons aussi de l'impact de nos résultats.

VI. Threats to validity

Notre étude est sujet de plusieurs menaces à la validité. Dans cette section nous présonterons ces menaces et les stratégies que nous avons utilisé pour les mitiger.

Premièrement, notre étude est sujet de la menace de validité de construction. En effet le modèle *FAST* généré par notre approche pourrait ne pas être correct et conforme au métamodèle défini ou encore mal représenter le programme d'apprentissage profond en entrée. Pour mitiger cette menace, nous avons mis en place des tests unitaires pour chaque méthode visiteur. Ces tests unitaires nous permettent de vérifier que les méthodes implémentées dans l'importateur son correctement implémentées et que le modèle *FAST* généré est conforme au métamodèle défini. De plus, nous avons implémenté pour un ensemble de données de test (des exemples synthétiques) des tests fonctionnels (Smock Testing) pour vérifier que le modèle *FAST* généré représente bien le programme d'apprentissage profond en entrée.

Deuxièmement, notre étude est sujet de la menace de validité interne. Il subsiste un risque que les conclusions obtenues ne soient pas causées par le code source en entrée. Pour mitiger cette menace, nous avons en plus de tester notre système sur des exemples synthétiques, nous avons utilisé des projets réels collectés dans Github pour valider le fonctionnement de notre système.

Troisièmement, notre étude est sujet de la menace de validité externe. Il y a un risque que les résultats obtenus ne soient pas généralisables. Pour mitiger cette menace, nous avons collecté aléatoirement des projets dans Github afin de représenter différente application de l'architecture CNN.

Quatrièmement, notre étude est sujet de la menace de validité de fiabilité. Il y a enfin un risque que les résultats obtenus ne soient pas reproductibles. Pour mitiger cette menace, nous avons en plus d'utilisé des projets de différentes applications de l'architecture CNN, utilisé des projets implémentées dans les trois librairies open source les plus utilisées dans l'industrie (Tensorflow, Pytorch et Keras).

VII. Limitations and Future Work

1) Limitations: Notre étude s'est fait dans un contexte limité que nous allons décrire dans cette section. Quand nous parlons de programmes d'apprentissage profond, nous faisons référence aux programmes contenant un réseau de neurones à convolution (CNN). Il est donc important de noter qu'il existe d'autres types de programmes d'apprentissage profond qui ne sont pas couverts par notre étude. De plus, il existe plusieurs types de librairies qui permettent de créer des programmes d'apprentissage profond. Notre étude se limite aux programmes créés à partir des librairies TensorFlow, Keras et PyTorch. Et nos données proviennent exclusivement de référentiel open source Github ou StackOverflow (pour les exemples). Enfin, notre système de détection d'odeurs de conception est limité à la détection des 8 ordeurs de conception décrit plus haut.

2) Future Work: Les travaux futurs peuvent aller dans plusieurs directions. En effet il est possible d'appliquer notre approche à d'autres types de programmes d'apprentissage profond, d'autres librairies, d'autres odeurs de conception, d'autres langages de programmation et d'autres types de référentiels (non open source par exemple). Nous avons fait, une analyse statique des programmes d'apprentissage profond, il est également envisageable de faire une analyse dynamique en simulant l'exécution des programmes à travers de la modélisation. En plus de la détection via la modélisation, d'autres champ de recherche sont la détection via l'apprentissage

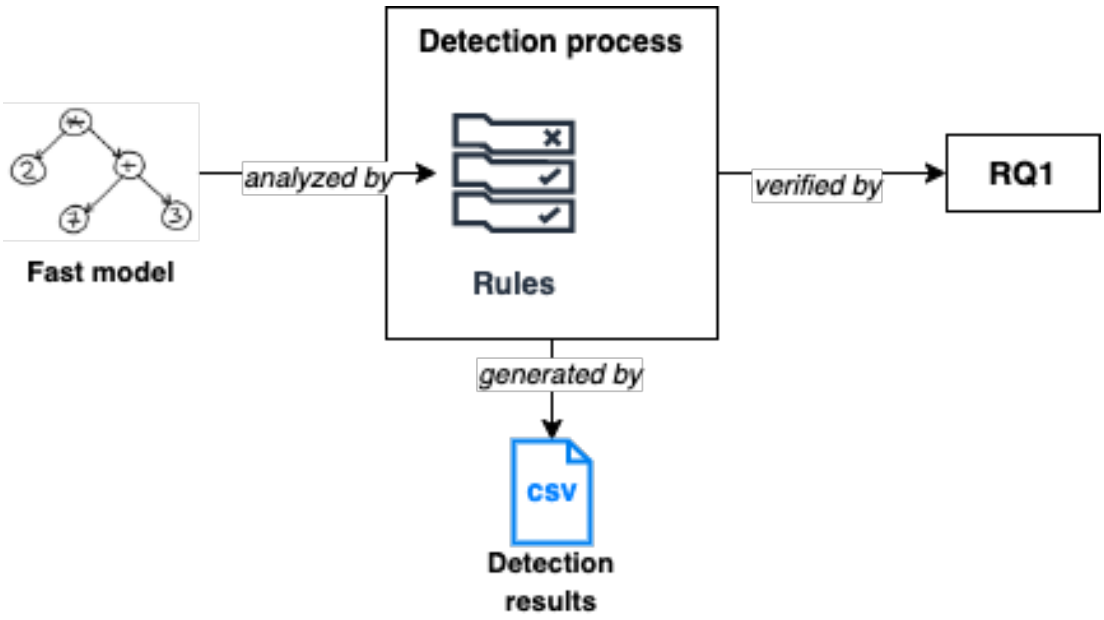

Figure 4: *Processus de détection des odeurs de conception.*

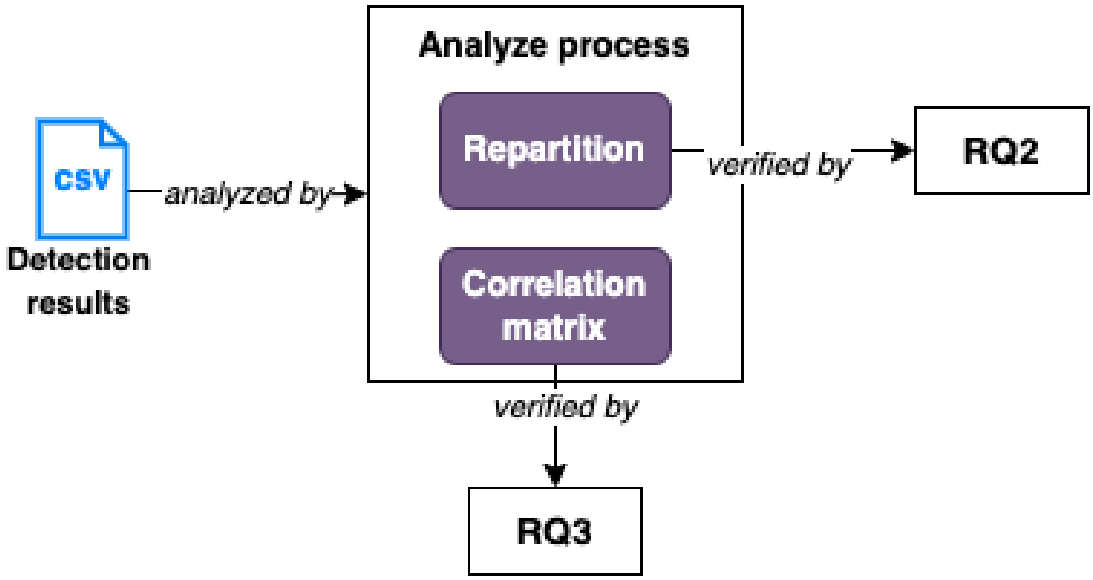

Figure 5: *Processus d'analyse des résultats de la détection des odeurs de conception.*

automatique et la correction automatique des odeurs de conception. Ces derniers sont des sujets de recherche très prométeurs qui pourront avoir un impact très important sur la qualité des programmes d'apprentissage profond futur.

VIII. Conclusion

TODO: To be completed

References

[1] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

[2] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, pp. 369–385, 2006.

[3] C. A. González and J. Cabot, "Formal verification of static software models in mde: A systematic review," *Information and Software Technology*, vol. 56, no. 8, pp. 821–838, 2014.

[4] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by example*. Lulu. com, 2010.

[5] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Lulu. com, 2013.

[6] O. Zaitsev, S. Ducasse, and N. Anquetil, "Characterizing pharo code: A technical report," Ph.D. dissertation, Inria Lille Nord Europe-Laboratoire CRIStAL-Université de Lille; Arolla, 2020.

[7] S. Tichelaar, S. Ducasse, and S. Demeyer, "Famix and xmi," in *Proceedings Seventh Working Conference on Reverse Engineering*, 2000, pp. 296–298.

[8] S. Demeyer, S. Ducasse, and S. Tichelaar, "Why famix and not uml," in *Proceedings of UML'99*, vol. 1723, 1999.

[9] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," in *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, vol. 4, 2000.

[10] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 294–305.

[11] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–31, 2021.

[12] T. Cerny, A. Al Maruf, A. Janes, and D. Taibi, "Microservice anti-patterns and bad smells. how to classify, and how to detect them. a tertiary study," *How to Classify, and How to Detect Them. A Tertiary Study*.

[13] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[14] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013, pp. 116–125.

[15] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2016, pp. 18–23.

[16] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002, pp. 97–106.

[17] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[18] P. Araujo, S. Rodríguez, and V. Hilaire, "A metamodeling approach for the identification of organizational smells in multi-agent systems: Application to aspecs," *Artificial Intelligence Review*, vol. 49, pp. 183–210, 2018.

[19] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

[20] S. Alawadi, K. Alkharabsheh, F. Alkhabbas, V. Kebande, F. M. Awaysheh, and F. Palomba, "Fedcsd: A federated learning based approach for code-smell detection," *arXiv preprint arXiv:2306.00038*, 2023.

[21] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," *PeerJ Computer Science*, vol. 9, p. e1370, 2023.

[22] A. Alazba, H. Aljamaan, and M. Alshayeb, "Deep learning approaches for bad smell detection: a systematic literature review," *Empirical Software Engineering*, vol. 28, no. 3, p. 77, 2023.

[23] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, "Automatic fault detection for deep learning programs using graph transformations," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[24] A. Nikanjam and F. Khomh, "Design smells in deep learning programs: an empirical study," in *2021 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2021, pp. 332–342.