Tree-based Convolutional Neural Network for software pattern detection

Alfonso Domínguez De La Iglesia
European Industrial Center Capgemini
Machine Learning Scientist (B.A. Mathematics)
Valencia, Spain
alfonso.dominguez-de-la-iglesia-external@capgemini.com

David Minet Casado
European Industrial Center Capgemini
Machine Learning Scientist (B.A. Physicist)
Murcia, Spain
david.minet-casado-external@capgemini.com

Esther Pla Ibáñez
European Industrial Center Capgemini
Machine Learning Scientist (MSc. on Mathematics)
Valencia, Spain
esther.pla-ibanez-external@capgemini.com

Ignacio Gallego Sagastume
European Industrial Center Capgemini
Project Coordinator (Senior Software Engineer)
Valencia, Spain
ignacio.gallego-sagastume@capgemini.com

ABSTRACT

Nowadays, there are powerful and convenient tools to help software engineers to carry out their development work, for example IDEs with numerous functionalities and quality plugins, but there is still a need in decision-making support. In addition, software integration and maintenance can sometimes be very expensive [19]. These reasons mean that productivity is not as high as it could be, increasing costs and time on a development team. In this situation, a software that could be very useful in the future and could offer support to developers, based on deep learning techniques, is presented.

A new software, called Discern, capable of analyzing source code and making a diagnosis of some software design patterns present in a full project or Github repository was developed.[1] The Discern tool is based on a recent deep learning model to perform computer language processing jobs called Tree-Based Convolutional Neural Network.

Finally, the results obtained that demonstrate the good performance of the Discern tool, and the ease of use and extensibility are presented.

CCS CONCEPTS

• **Computing methodologies → Machine learning algorithms**.

1 INTRODUCTION

Software engineering is still a young discipline, where much of the development work is done by hand. Furthermore, the nature of this work entails the interaction of areas with different levels of abstraction. This complicates the implementation of good development practices, as there is sometimes a lack of good documentation that results in a waste of resources. Today's tools lack a certain intelligence, which when incorporated can increase the efficiency of this professional sector.

With this motivation in mind, a Discern tool capable of performing source code diagnostics that works under deep learning techniques was developed [2]. In this way, it is a programming language processing tool.

Software design patterns are solutions to programming problems that developers often appeal to solve a specific problem. These design patterns follow well-defined structures and clear component relationships, although they can be difficult to read for a person who has not written the code. Having the Discern tool could help developers with different profiles and experiences. A junior developer could get used to these expert practices more quickly, greatly improving their learning curve, being able to include complex design patterns in their work thanks to see examples in expert code. In addition, the implementation of software design patterns is associated with greater long-term code maintainability, as well as high reusability. These patterns are classified into three types, depending on the purpose or problem to solve: creational patterns, structural patterns and behavioral patterns [1].

The implementation of the Discern tool can be very useful if it was incorporated in IDEs as a plugin. For example, a developer could have easily a diagnosis of the software design patterns present in a given project. Therefore, the focus on the detection of software design patterns in source code could propitiate time savings in the future.

This software takes advantage of the fact that design patterns have very clear structures and relationships between components, since this should be reflected in the abstract syntax tree (AST) [1] associated with the source code, which will be used as raw material to feed the Discern tool.

The approach has been mainly to the diagnosis of python source

[1]Please open an issue or contact tbcnn.datadons@gmail.com with any questions.

[2]Software available at: https://github.com/a-domingu/tbcnn2

code for the detection of generator and wrapper patterns, but taking into account the results that are appreciated, this idea could be easily extended for other programming languages and other more complex patterns. These patterns were chosen for their simplicity, since they do not usually have relationships that involve too many classes, since the first goal was to demonstrate the existence of a good machine learning based on a programming language.

This paper is divided into several sections: in section 2, the tree based convolutional neural network, a deep learning model implemented in the Discern tool will be presented. In section 3, the software architecture and its extensibility will be explained. In section 4, the results obtained according to the existing design patterns will be exposed. In section 5, future improvements for the Discern tool will be proposed. Finally, in section 6, the conclusions will be addressed.

In addition, the Discern tool has great scalability and can be easily extended. The Discern tool present in this paper can be used by other developers who want to continue developing the idea of this software, since it is open source software. For this part, the ideal would be that developers from around the world continue to contribute with this proposal. This leaves a door open to explore source code diagnostics, bug fixes, etc.

2 THEORETICAL BACKGROUND

As mentioned before, the model behind it is the one called Tree-Based Convolutional Neural Network. As the name suggests, this type of Machine Learning algorithm has the expected basic structure of a Convolutional Neural Network [3], but making some changes to be able to use it over an input that has a tree structure, rather than a matrix.

2.1 Review of Convolutional Neural Networks

Traditional Neural Networks typically receive a matrix as input, and these matrix are sent across a series of layers (convolutional layers, pooling layers, dense layers, etc) and functions (such as activation functions) to finally receive an output that accurately represent the probability of what is trying to be predicted.

The basic idea of the convolutional layer is that a set of small matrices (usually called windows or filters), which will move across our input matrix, and multiply each of the values using a scalar product. Each of these filters is supposed to be trained to identify certain features in the input.

Then, the output of the convolutional layer is passed to a pooling layer. This layer is supposed to reduce the dimension of our matrix in order to only retain significant values. After the convolutional layer, high values are supposed to represent high probability of a given feature, while low numbers means that those features were not found in that part of the data. Therefore, the interest is in removing the elements with low values, and that is why a Pooling layer is needed.

This process can be repeated multiple times, across multiple convolutional and pooling layers. Finally, a *Dense* layer (also called *Hidden* layer) is used, which will simply take our input, set the values in a vector, and multiply it with another vector scalarly and give us a real number. Then, it is needed to pass this number to an activation function. Activation functions are supposed to define our output. The most commonly used are *Sigmoid function, Softmax, ReLU, leaky ReLU* or *tanh* [3] [5].

2.2 The Tree-based Convolutional Neural Network

As mentioned in the previous section, traditional CNNs receive a matrix as input. However, because of the nature of the problem, the data come from AST on Python source code. In simpler words, the input has a tree structure, not a matrix. So the problem now is, to represent a tree as a matrix, in such a way that it conserves the structure? The matrix needs to capture not only the characteristics of each node, such as the type of node we have *(Yield, For, Name, Assign...)*, but it also needs to capture the overall relations between the nodes (parent nodes, sibling nodes...). Also, trees with similar structure should naturally have a similar matrix representation.

This is why a special kind of CNN is applied to trees, idea originally taken from [3].

Some particular approaches have also been adopted, different from the ones outlined in [3], which will be mentioned across this section.

The first challenge is to take each of the nodes and transform it into a vector. The way to go is to take multiple random walks over our tree, with fixed length, and use this random walks to *form sentences with the node types*. These sentences should be a good way to get an overall idea of the structure of our tree. Then, the sentences are passed to a word embedding algorithm which is capable of transforming the words (node types) into vectors based on the random walks relating those tokens that appear most often. The book outlines multiple possibilities, but the chosen algorithm to convert these words into vectors is *Word2Vec* [4].

Once this algorithm is applied, there is a vector for every node type. However, this will not be the final vector representation used for every node. Using one single vector for every node of the same type is not very useful, as two nodes of the same type may have different relations to the surrounding nodes, e.g., one of them might not have children, the other might have multiple children, etc. That is the reason why this representation is used as our *initial vector representation* for a neural network that will better capture the structure of each node in our tree. This part of the TBCNN is called *vector representation*.

[3]To know more about Convolutional Neural Networks you can get additional information in [2]

2.3 Vector Representation

Let p be a non-leaf node with children $c_i,\ i = 1, ..., n$. Also, let $vec(\cdot)$ be the final vector representation of any given node. The following has to be searched:

$$vec(p) \approx ReLU(\sum_{i=1}^{n} l_i W_{code,i} vec(c_i) + b_{code}) \tag{1}$$

Notice that $ReLU$ is being used instead of $tanh$ as this favored in the vanishing gradient problem [5].

As it is explained in the book, $l_i = \frac{\text{\# of children under } c_i}{\text{\# of children under} p}$

Also, $W_{code,i}$ is a matrix that depends on the node c_i but that works as a linear combination of two constant matrices, W^l and W^r, which are the two matrices that, together with the vector b presented in equation 1, are the ones that we will train to get a reliable vector representation.

The coefficients will be dependent on the child node c_i, and were taken from [6]
Therefore, if $n > 1$:

$$W_i = \frac{n-i}{n-1} W^l + \frac{i-1}{n-1} W_r \tag{2}$$

If $n = 1$, then

$$W_i = \frac{1}{2} W^l + \frac{1}{2} W^r$$

.

Since 1 is searched to happen, the loss function is defined as:

$$d = ||vec(p) - ReLU(\sum_{i=1}^{n} l_i W_{code,i} vec(c_i) + b_{code})|| \tag{3}$$

However, this could be a problem because the algorithm could learn the trivial case where $vec(x) = 0\ \forall x$, where x is a node. Since this yields a $loss = 0$, it could be the "perfect" vector representation, when this is not the case.

In order to solve this problem, the loss function is redefined as $loss = d - d' + \Delta$, where d' is the same value as d but changing one of the vectors with another random vector, and Δ is a constant value, usually set to 1.

The mentioned paper does not deal with the particularity of having a leaf node, but in this paper's problem these nodes are not trained and instead use the original vector representation given by *Word2Vec* and as can be seen in result's section, this still produces a reliable outcome.

2.4 CNN for pattern detection

Once there is a vector representation for each of the nodes, a Convolutional Neural Network is applied over these vectors, with the particularity that this is not a "traditional" neural network that works with matrices, but instead there will be work with each the nodes (and their vectors) independently, together with their descendants.

This part mentioned in [3] starts with a *Coding layer*, which was implemented, but without noticed significant differences when it is not present, so the final decision was to leave it out.

Next it continues with a modified *Convolutional layer*.

Unlike traditional convolutional layers, this one uses a triangle-shaped filter, which takes each node and its descendants to whatever depth wanted (normally $depth = 1$ will suffice, meaning the node and its children).

Therefore, if $x_i\ i = 1, ..., n$ are the nodes under the filter, the output is:

$$y = ReLU(\sum_{i=1}^{n} W_{conv,i} x_i + b_{conv}) \tag{4}$$

Again, the ReLU activation function is used instead of $tanh$.

Again, as happened with 1, it is neccesary to compute $W_{conv,i}$ (abbreviated now as W_i for simplicity), since it is dependant on every node.

Well, this matrix will be a linear combination of three fixed matrices needed to train, together with coefficients that depend on each node. These matrices are $W_{left},\ W_{right},\ W_{top}$. So, as it is to be expected, depending on the position of each node in the filter, the higher the respective coefficient will be (e.g, if a node is in the bottom left corner, it should be expected that the coefficient of W_{left} should be higher than those of the other two matrices).

So the next question should be: how do we compute this coefficients so that the result described in the previous paragraph is obtained?

Let c_i be a node under a filter with depth d, and d_i be the depth of the node in the given window. Also, let p_i be the position (i.e, how left or right) of the node, and n be the number of siblings of the node **including the node itself**.

Therefore, the coefficients are the following:

- $\eta_i^t = \frac{d_i - 1}{d-1}$
- $\eta_i^r = (1 - \eta_i^t)\frac{p_1 - 1}{n-1}$
- $\eta_i^l = (1 - \eta_i^t)(1 - \eta_i^r)$

Which are respectively the coefficients for W_{top}, W_{right} and W_{left} for the node c_i.

After the convolutional layer, the book recommends using a "dynamic pooling" [3] (p. 52), but instead is used standart maximum pooling and it offers reliable results. For every vector y, it is stacked them together into a matrix. This matrix will have as many rows as nodes, and as many rows as the number of features from the convolutional layer. Then, for every row the maximum number is taken, and a new vector x is obtained with as many elements as the number of features.

Finally, a traditional hidden layer is applied. If $x \in \mathbb{R}^n$, then let $w \in \mathbb{R}^n,\ b \in \mathbb{R}$, so:

$$output = sigmoid(w * x + b) \tag{5}$$

This will yield the desired output, with a number between 0 and 1 representing the likelihood that the script contains a given pattern. Then, stochastic gradient descent is applied and the parameteres are adjusted accordingly.

To sum up, the matrices and vectors to train in this last step are $W_{top}, W_{right}, W_{top}, w_{hidden}, b_{conv}, b_{hidden}$

3 SOFTWARE IMPLEMENTATION

As mentioned above, the Discern Framework is a project that provides a tool developed with Deep Learning techniques and Machine Learning algorithms. The framework offers to the user the detection of software design patterns in arbitrary source code. To do that, the program has to receive different Python files and train a Convolutional Neural Network (CNN) to detect patterns. The TBCNN is a specific type of CNN that works on Abstract Syntax Trees (ASTs) of source code and, to be able to train this CNN, it is needed to previously transform each AST node into a vector.

3.1 Framework structure

The framework is divided into five connected components: initial vector representation, vector representation, pattern training, pattern test, and pattern detector.

3.1.1 *Initial Vector Representation.*

The first neural network does the vector representation for all files. Because this program reads ASTs and not vectors, to use a CNN, it is necessary to convert each node into a vector. To do that, two independent networks are used: **initial vector representation** and **vector representation**. The first one takes the idea behind ***Word2Vec***, i.e, creates the vectors based on the overall structure of the tree. In figure 1 the class diagram of the network can be seen.

The *InitialVectorRepresentation* class receives several ASTs as input, and in figure 1 it can be seen that it uses the *Embedding* class. *Word2Vec* is applied in the *Embedding* class to assign a vector to each type of AST node. Moreover, the *InitialVectorRepresentation* class uses many *NodeObjectCreator* classes (one for each AST). The *NodeObjectCreator* class has a *fileParser* function that instantiates Node classes. In particular, one for each AST node. The class Node calculates and saves all the required attributes for each node. Finally, the output is a file called *initialVectorRepresentation.csv* with one vector for each type of AST node.

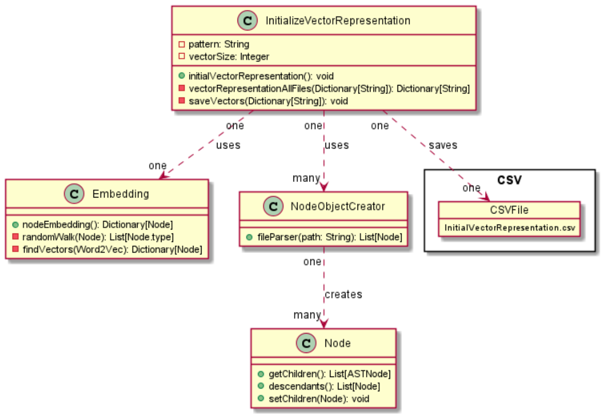

InitializeVectorRepresentation
pattern: String
vectorSize: Integer
initialVectorRepresentation(): void
vectorRepresentationAllFiles(Dictionary[String]): Dictionary[String]
saveVectors(Dictionary[String]): void
one
uses
one
one
uses
many
one
saves
one
Embedding
nodeEmbedding(): Dictionary[Node]
randomWalk(Node): List[Node.type]
findVectors(Word2Vec): Dictionary[Node]
NodeObjectCreator
fileParser(path: String): List[Node]
CSV
CSVFile
InitialVectorRepresentation.csv
one
creates
many
Node
getChildren(): List[ASTNode]
descendants(): List[Node]
setChildren(Node): void

Figure 1: Class diagram of the initial vector representation

3.1.2 *Vector Representation.*

Once the *InitialVectorRepresentation* is done with *Word2Vec*, *VectorRepresentation* class can be called. This class uses the initial vectors to get a better vector representation by learning the context of each node within the tree. The class diagram of this neural network can be seen in figure 2.

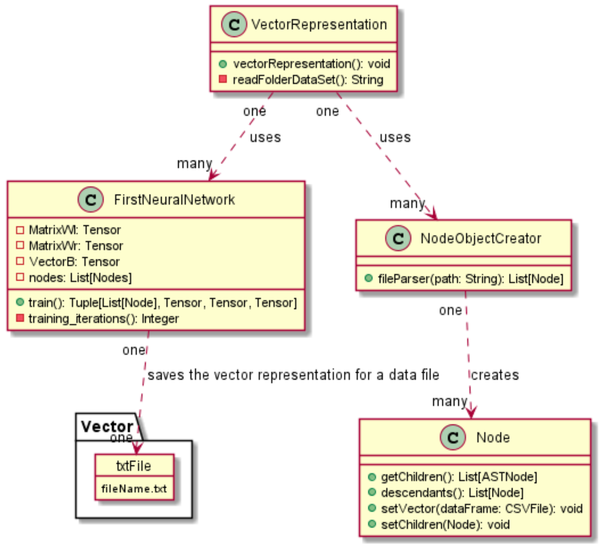
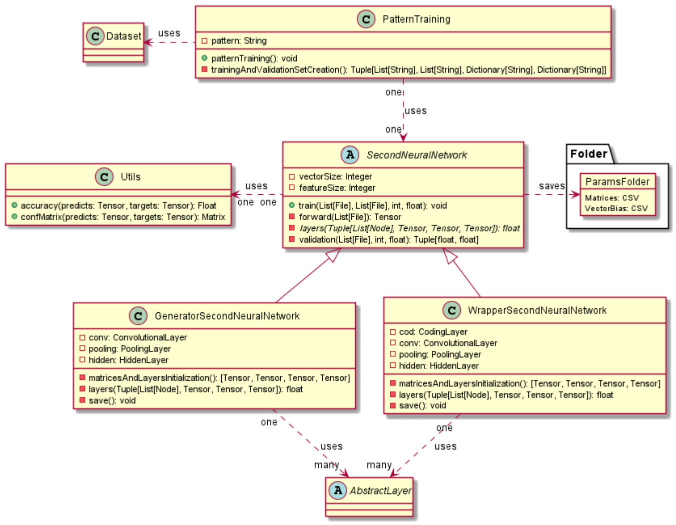

VectorRepresentation
vectorRepresentation(): void
readFolderDataSet(): String
one
uses
many
one
uses
many
FirstNeuralNetwork
MatrixWl: Tensor
MatrixWr: Tensor
VectorB: Tensor
nodes: List[Nodes]
train(): Tuple[List[Node], Tensor, Tensor, Tensor]
training_iterations(): Integer
NodeObjectCreator
fileParser(path: String): List[Node]
one
saves the vector representation for a data file
one
one
creates
many
Vector
txtFile
fileName.txt
Node
getChildren(): List[ASTNode]
descendants(): List[Node]
setVector(dataFrame: CSVFile): void
setChildren(Node): void

Figure 2: Class diagram of the vector representation

The *VectorRepresentation* class receives several ASTs as input, and as it can be seen in figure 2, this class uses many *FirstNeuralNetwork* classes (one for each AST). Each *FirstNeuralNetwork* class is used to train a neural network that returns as output the following information: one vector for each node (the features and the context of each AST node are represented in a vector), two weighted matrices, and one vector bias. Finally, all these parameters are saved into a file using pickle module.

3.1.3 *Pattern Training.*

Once the vector representation is done for each file, the *PatternTraining* class can be called. This class receives the output of the first neural network as input, and thanks to the *Dataset* class, splits the data set into two sets: a training set and a validation set. The training set should have 70% of the files and is used to train the CNN, while the validation set has the other 30% and is used to test the accuracy of the *SecondNeuralNetwork*.



Dataset
uses
PatternTraining
pattern: String
patternTraining(): void
trainingAndValidationSetCreation(): Tuple[List[String], List[String], Dictionary[String], Dictionary[String]]
one
uses
one
SecondNeuralNetwork
vectorSize: Integer
featureSize: Integer
train(List[File], List[File], int, float): void
forward(List[File]): Tensor
layers(Tuple[List[Node], Tensor, Tensor, Tensor]): float
validation(List[File], int, float): Tuple[float, float]
Utils
accuracy(predicts: Tensor, targets: Tensor): Float
confMatrix(predicts: Tensor, targets: Tensor): Matrix
uses
one one
saves
Folder
ParamsFolder
Matrices: CSV
VectorBias: CSV
GeneratorSecondNeuralNetwork
conv: ConvolutionalLayer
pooling: PoolingLayer
hidden: HiddenLayer
matricesAndLayersInitialization(): [Tensor, Tensor, Tensor, Tensor]
layers(Tuple[List[Node], Tensor, Tensor, Tensor]): float
save(): void
WrapperSecondNeuralNetwork
cod: CodingLayer
conv: ConvolutionalLayer
pooling: PoolingLayer
hidden: HiddenLayer
matricesAndLayersInitialization(): [Tensor, Tensor, Tensor, Tensor]
layers(Tuple[List[Node], Tensor, Tensor, Tensor]): float
save(): void
one
uses
many
one
uses
many
AbstractLayer

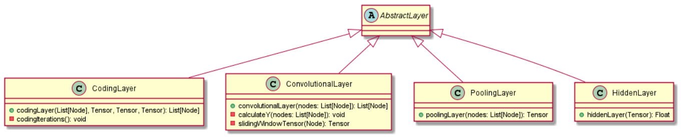
Figure 3: Class diagram of the pattern training

In figure 3, it can be seen how the *PatternTraining* class uses an Abstract class called *SecondNeuralNetwork*. This abstract class allows us to have an extensible framework because to train the CNN to detect new patterns, only a new subclass implementation is required. Currently, this framework can only detect generators and wrapper patterns. However, a new software pattern like a decorator can be added easily, implementing a new subclass.

The *SecondNeuralNetwork* receives the vector representation of each AST node as input and applies several transformations to train the CNN using the *AbstractLayer* class. This abstract class has multiple subclasses (Figure 4) like *CodingLayer*, *ConvolutionalLayer*, *PoolingLayer* and *HiddenLayer*. All these layers are explained in more detail in section 3. After each iteration, the neural network uses the *Utils* class to test the accuracy of its parameters and to record its confusion matrix associated. The accuracy is the proportion of correctly predicted files, while the confusion matrix shows which files are correctly predicted.
Once the neural network is trained, a file is generated as an output. This file has the matrices and vectors that are necessary to detect patterns in the source code.

3.1.4 *Pattern Test.*

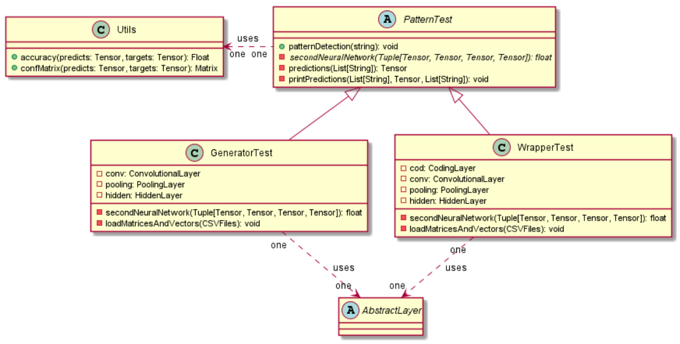
Once the Convolutional Neural Network is trained, the *PatternTest* class will test the accuracy of its trained matrices and vectors necessary to detect patterns in the source code. These matrices and vectors were the output of the *PatternTraining*



AbstractLayer
CodingLayer
codingLayer(List[Node], Tensor, Tensor, Tensor): List[Node]
codingIterations(): void
ConvolutionalLayer
convolutionalLayer(nodes: List[Node]): List[Node]
calculateY(nodes: List[Node]): void
slidingWindowTensor(Node): Tensor
PoolingLayer
poolingLayer(nodes: List[Node]): Tensor
HiddenLayer
hiddenLayer(Tensor): Float

Figure 4: Class diagram of the *AbstractLayer*

class. Moreover, the *PatternTest* class received several ASTs as input that came from the test set.



Utils
accuracy(predicts: Tensor, targets: Tensor): Float
confMatrix(predicts: Tensor, targets: Tensor): Matrix
uses
one one
PatternTest
patternDetection(string): void
secondNeuralNetwork(Tuple[Tensor, Tensor, Tensor, Tensor]): float
predictions(List[String]): Tensor
printPredictions(List[String], Tensor, List[String]): void
GeneratorTest
conv: ConvolutionalLayer
pooling: PoolingLayer
hidden: HiddenLayer
secondNeuralNetwork(Tuple[Tensor, Tensor, Tensor, Tensor]): float
loadMatricesAndVectors(CSVFiles): void
WrapperTest
cod: CodingLayer
conv: ConvolutionalLayer
pooling: PoolingLayer
hidden: HiddenLayer
secondNeuralNetwork(Tuple[Tensor, Tensor, Tensor, Tensor]): float
loadMatricesAndVectors(CSVFiles): void
one
uses
one
one
uses
one
AbstractLayer

Figure 5: Class diagram of the pattern test

As it can be seen in figure 5, the *PatternTest* class is an abstract class that allows choosing which pattern accuracy will be tested (for example, generator, wrapper, decorator, etc.). To be able to test the accuracy of a pattern, the *PatternTest* class uses the layers mentioned above, as well as the trained matrices and vectors. Finally, this class uses the *Utils* class to return the following elements: the accuracy (proportion of correctly predicted files) based on the test set and the confusion matrix that shows which files are correctly predicted.

3.1.5 *Pattern Detector.*

Once the Convolutional Neural Network is trained and it has good accuracy for the test set (up to 90 %), the neural network can be used to detect patterns in the source code. The *PatternDetector* class receives complete software projects as input. These projects can be in a folder or in a GitHub repository. Therefore, the user should indicate the path to the local folder or a URL corresponding to the GitHub repository.

As it can be seen in figure 6, the *PatternDetector* class is an abstract class that allows to detect which files have the required pattern. Each subclass uses the layers mentioned in section 3, as well as the trained matrices and vectors obtained in the pattern training class. Once the program has finished, a diagnosis is returned as output of the files that have the required pattern.

4 RESULTS

Once the software is ready and working properly, classification tasks can be performed. The classification will be carried out using

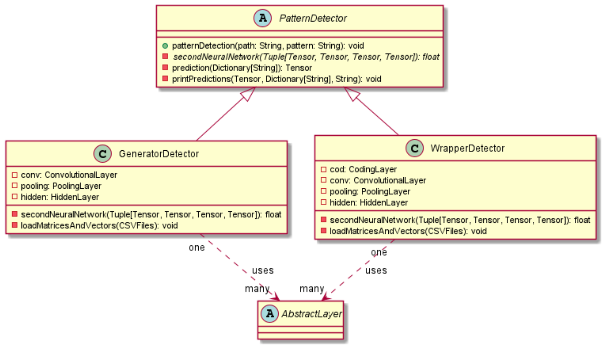

A PatternDetector
patternDetection(path: String, pattern: String): void
secondNeuralNetwork(Tuple[Tensor, Tensor, Tensor, Tensor]): float
prediction(Dictionary[String]): Tensor
printPredictions(Tensor, Dictionary[String], String): void
C GeneratorDetector
conv: ConvolutionalLayer
pooling: PoolingLayer
hidden: HiddenLayer
secondNeuralNetwork(Tuple[Tensor, Tensor, Tensor, Tensor]): float
loadMatricesAndVectors(CSVFiles): void
C WrapperDetector
cod: CodingLayer
conv: ConvolutionalLayer
pooling: PoolingLayer
hidden: HiddenLayer
secondNeuralNetwork(Tuple[Tensor, Tensor, Tensor, Tensor]): float
loadMatricesAndVectors(CSVFiles): void
one
uses
many
one
uses
many
A AbstractLayer

Figure 6: Class diagram of the pattern detector

backpropagation algorithms in conventional neural networks. The procedure will be to receive python code files, perform the vector representation with the tree based convolutional neural network and use that representation to feed the classification neural network.

The operation of the classification neural network follows the mini-batch gradient descent algorithm. In addition, the followed loss criterion has been the one offered by the `BCEWithLogitLoss` Pytorch class. This loss combines a sigmoid layer and the binary cross entropy in one single class. This version is more numerically stable than using a plain sigmoid followed by a binary cross entropy loss as, by combining the operations into one layer, taking advantage of the log-sum-exp trick for numerical stability [9].

As more than one design pattern can coexist in a software project, it is recommended to adapt the neural network trainings for each different pattern. In this way, a classification will be made independently for each pattern to check if it exists in the code, to finally return the complete diagnosis of the design patterns present in a project.

4.1 Data set

Once the software is developed, it is necessary to feed it with a data set. The procedure to build the data set was to collect real and extended projects, ensuring that they were varied in terms of length and complexity of the code, so that the result was as extensible as possible.

The software is prepared so that the label is built automatically. It is only necessary to put the files in a folder according to the label they have. From this set of data separated into folders, the program will randomly choose 70% of the files to be part of the training set and the remaining 30% will be for validation.

Based on experience in this project, where not very complex design patterns have been dealt with, 400 files per design pattern have been sufficient in order to obtain good results (200 files with the pattern and 200 without the pattern). Although in [3] it is recommended for a similar experiment to use at least 500 files per label. Probably, for more complex patterns a larger sample would be required. Also, a class associated with the data set has been designed in the software that facilitates handling with the data set.

4.2 Generator training.

Bearing in mind that this pattern in Python is simple, since only with the presence of a node of type `Yield` it can be detected, the training can be adapted to this pattern. That is, not many epochs in the tree based convolutional neural network are necessary, since the context of each node is not very relevant compared to other design patterns.

With that argument, only 1 epoch is applied in the tree based convolutional neural network. That is, the vector representation of each node will be the initial one provided by the modified *Word2vec* algorithm. In addition, after training with different parameters, the result was finally chosen with a size vector of 30, a learning rate for the classification neural network of 0.001, a feature size of 100, no momentum and no penalty.

In figure 7, a graph that shows the learning of the engine as the epochs progress for the training of the generator pattern classification can be seen.

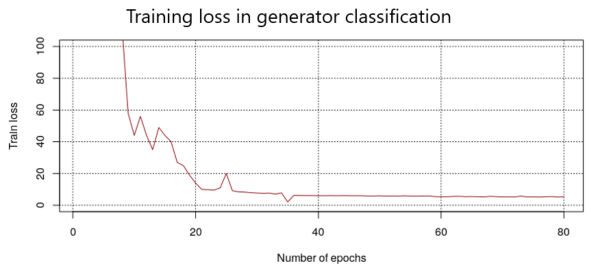


Figure 7: Training loss versus number of epochs in the training of the generator design pattern.

The training is what was expected at the beginning. In the first ten epochs, the loss is too high to take into account. Subsequently, it is progressively reduced until reaching a stable minimum approximately at epoch 35.

At the same time, while the machine is being trained for the training data set, the prediction is being made for the validation set. In figure 8, how the accuracy for the validation set evolves as the epochs progress can be seen.

With this graph of the evolution of the accuracy throughout the training, a good complement for the loss in the training set exists. Obviously, for the first epochs the accuracy is insufficient, but as it

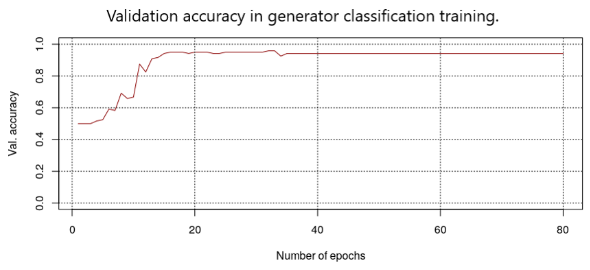


Figure 8: Training loss versus number of epochs in the training of the generator design pattern.

progresses, it remains stable at 94.17% accuracy in the validation set. Also, the software has been developed in such a way that the results with the best accuracy value obtained are recorded. In other words, the classification engine that is saved is the one from the epoch with the best results on accuracy. This has the intention to avoid overfitting.

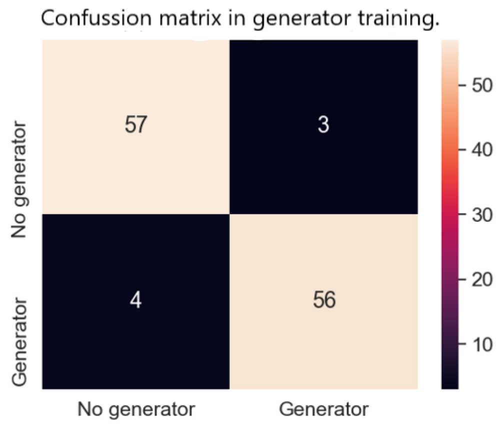


Figure 9: Confusion matrix in generator training with a final accuracy of 94.17%.

These results for the accuracy in the validation set allow to affirm that this model is able with certain security to classify simple patterns in source code.

After obtaining this result in training, a test with a totally external set has been made, reducing the accuracy to 89.00%.

4.3 Wrapper/other patterns.

The potential of the Discern tool is that the training of each pattern is carried out independently, therefore it is possible to adapt the training.

For example, if training is required for a design pattern that involves different classes and needs the context, it can be achieved by increasing the number of epochs in the tree based convolutional neural network. On the other hand, it is also possible to adapt the *Word2vec* dictionary, including certain recurring words to help with the weights (adding as a word `listener` for the observer pattern or `wrapper` for the wrapper pattern).

Consequently, the optimal parameters for training can be altered, so it is most recommended to vary them until a better combination of them is found.

5 FUTURE WORK

Deep learning is making breakthroughs in the field of software pattern detection. However, as can be seen in the literature, machine learning for source code is a young discipline, and hence, few references work with deep learning on Abstract Syntax Tree (AST), [3]. For this reason, some possible improvements for our framework can be addressed, as well as its applications.

5.1 Patterns implementation

The Discern Framework is an extensible and open source software that provides to the user the detection of software design patterns in arbitrary source code. Currently, the framework can detect simple design patterns such as generators, but the intention is to extend it to more complex patterns in the future too. The framework was built as an extensible software with the aim of detecting new patterns in a simple way. Some improvement ideas will be introduced below.

5.1.1 Adding some node attributes. Currently, only the AST node type is taken into account (*FunctionDef*, *Yield*, *Import*,...). However, in principle, by adding other node attributes, like the function's name, the pattern detection accuracy can be improved. For example, in the wrapper pattern detection a new node type called *Wrap* can be included thanks to the *Node* class (mentioned in section 3.1.1). This new node type would include all the *FunctionDef* nodes whose names include the words *wrap* or *decorator*. For other patterns, it would be analogous.

5.1.2 Work with whole AST projects. The framework is implemented to work with ASTs of a python source code, i.e., the program reads and analyzes the ASTs file by file. The program can only detect structural patterns that are contained in an individual Python file. However, another upgrade could be to detect structural patterns like decorators or adapters, which can be implemented in different Python files. To do that, a global AST that relates all the AST files between them is required. Also, the *Import* or *ImportFrom* nodes can be useful to relate different files, as well as the *Node* class. In fact, each time that one class or function is called in another file, this *Call* node can be assigned as a child of the *Function* or *Class* node thanks to the *Node* class. Moreover, the data set should contain complete projects instead of individual Python files.

6 CONCLUSIONS.

A software for detecting design patterns from python source code has been developed. It is a software based on deep learning techniques, more specifically on a model called Tree Based Convolutional Neural Network.

This paper demonstrates the operation of this software and the variations with respect to the original model. In addition, an application for the detection of design patterns is shown, as well as the motivation for it.

Both the training of this machine learning engine and the validations and tests have been carried out with real projects, so that they have been chosen to have a varied sample in length and complexity of the code, for greater extensibility. Without doubt, after seeing the results of project classification according to the existence of a pattern, the proper functioning of the application is consolidated. Although it has been entered for simple and easily traceable design patterns, the possibility of adapting the training to a specific pattern has also been shown, in a way that allows this software to be extended in a not very complicated way to detect other complex patterns.

The potential of this software is mainly to demonstrate that through deep learning techniques a very reliable way can be found to carry out diagnoses on the presence of design patterns in complete projects, and that it is an easily extensible and open source software. In this way, it is also intended to encourage research using deep learning techniques in the area of programming language processing.

7 LITERATURE REVIEW.

Throughout the development of this work, as well as to put in context, it is worth highlighting certain references that have been useful. Since although there are articles that are not directly referenced in this paper, they have served the authors of this research to put in context and better understand the problem they were facing.

7.0.1 Tree-based Convolutional Neural Networks Principles and Applications[3]. This book, as well as all the papers of its authors, are of great help both to understand the problem of programming language processing and to understand the model on which this paper has been based.

7.0.2 Deep Learning With Customized Abstract Syntax Tree for Bug Localization [11]. This paper deals with the same model, tree based convolutional neural network, but for a different application: bug localization. It serves as a complement to the book previously shown as it allows a better understanding of the model, and also allows to see a practical application.

7.0.3 Efficient Representation Learning Using Random Walks for Dynamic Graphs [12]. The fundamentals of algorithms such as *Word2Vec* and in general language processing often resort to random walk. This is why understanding these algorithms can be essential to develop work in deep learning and graphs.

7.0.4 Embedding Directed Graphs using Random Walks [13]. As in previous literature, this document is useful in understanding embedding process.

Another useful literature can be found at [14], [15], [16], [17] and [18].

8 ACKNOWLEDGMENTS.

For some of the members of the team in charge of this research, it has been the first time that they have been involved in this type of work outside the academic field. This is why inexperience can sometimes cause a difficulty in the investigation. We would like to thank the support for months to different people and entities.

First of all, we would like to sincerely thank Capgemini for offering us the opportunity to become part of this family as members. In particular, we would like to single out the Capgemini family of the European Industrial Center and all the people who work there. Their selfless support, as well as how well they have treated us during the months of research, has been an incentive to do our best.

On the other hand, we would like to thank Iwan Van Der Kleijn for having trusted us from the beginning. Your continued support, your wisdom, your ideas and the way you have treated us have been very important to make this project possible. Our professional and personal progress is due to him.

Finally, we would like to thank Ignacio Gallego Sagastume for everything we have learned with him during these months. He has been our project leader but also has been a personal support. Your daily help and input have made this research successful. Without doubt, the lessons that you have taught us during these months have been essential. Without them, this project would not have been possible. They have been fundamental and have made us grow as professionals.

BIBLIOGRAPHY

[1] Gamma E., Helm R., Johnson R. Vlissides J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. 1st Edition. *Addison-Wesley Professional.* ISBN-13: 978-0201633610

[2] Howard, Jeremy & Gugger, Sylvain. *Deep Learning for Coders with fastai & Pytorch* O'Reilly Media, 2020

[3] Mou, Lili & Zhi Jin . *Tree-based Convolutional Neural Networks. Principles and Applications* Springer, 2018

[4] Word2vec documentation https://radimrehurek.com/gensim/models/word2vec.html

[5] Tomasz Szandala. *Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks.* 2010. Wroclaw University of Science and Technology. https://arxiv.org/ftp/arxiv/papers/2010/2010.09458.pdf

[6] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu & Lu Zhang, *Building Program Vector Representations for Deep Learning*, Software Institue School of EECS, Beijing, China.

[7] Lili Mou, Ge Li, Zhi Jin, Lu Zhang & Tao Wang, *Tree-Based Convolutional Neural Network for Programming Language Processing*, CoRR, vol.abs/1409.5718, 2014 http://arxiv.org/abs/1409.5718

[8] Lili Mou, Hao Peng, Ge Li†, Yan Xu, Lu Zhang & Zhi Jin, *Discriminative Neural Sentence Modeling by Tree-Based Convolution*, Software Institute, Peking University, 100871, P. R. China.

[9] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam. (2017). *Automatic differentiation in PyTorch.*. NIPS-W https://pytorch.org/docs/stable/_modules/torch/nn/modules/loss.html#BCEWithLogitsLoss

[10] Hooman Peiro Sajjad, Andrew Docherty & Yuriy Tyshetskiy, *Efficient Representation Learning Using Random Walks for Dynamic Graphs*, journal: CoRR, vol. abs/1901.01346, 2019. https://arxiv.org/pdf/1901.01346.pdf)

[11] Hongliang Liang, Lu Sun, Meilin Wang & Yuxing Yang, *Deep Learning With Customized Abstract Syntax Tree for Bug Localization*, IEEE Access, vol.7, August 2019

[12] Hooman Peiro Sajjad, Andrew Docherty & Yuriy Tyshetskiy. *Efficient Representation Learning Using Random Walks for Dynamic Graphs*. 2009. https://arxiv.org/pdf/1901.01346.pdf

[13] Kilian Ollivier, *Embedding Directed Graphs using Random Walks (Master Thesis)*, 2017 https://www.researchgate.net/publication/341685944_Embedding_Directed_Graphs_using_Random_Walks_Master_Thesis

[14] Rodrigo Aldecoa, *Detección de comunidades en redes complejas*, TFM Master UPV https://riunet.upv.es/bitstream/handle/10251/15337/TFM_RodrigoAldecoa.pdf

[15] Aditya Grover & Jure Leskovec, *node2vec: Scalable Feature Learning for Networks*, journal: CoRR, vol. abs/1607.00653, 2016 https://arxiv.org/pdf/1607.00653.pdf

[16] Bijaya Adhikari, Yao Zhang, Naren Ramakrishnan & B. Aditya Prakash, *Sub2Vec: Feature Learning for Subgraphs*, Department of Computer Science, Virginia Tech, 2018 https://people.cs.vt.edu/~bijaya/papers/sub2vecPAKDD2018.pdf

[17] GnanaJothi Raja Baskararaja & MeenaRani Sundaramoorthy Manickavasagam, *Subgraph Matching Using Graph Neural Network*, Journal of Intelligent Learning Systems and Applications, Vol.4 No.4, November 2012 https://www.scirp.org/journal/paperinformation.aspx?paperid=24827

[18] Emily Alsentzer, Samuel G. Finlayson, Michelle M. Li & Marinka Zitnik, *Subgraph Neural Networks*, CoRR, vol. abs/2006.10538, 2020 https://arxiv.org/pdf/2006.10538.pdf

[19] Torres, Victoria and Gil, Miriam and Pelechano, Vicente, *DECODER - DEveloper COmpanion for Documented and annotatEd code Reference*, Product-Focused Software Process Improvement, 20th International Conference, Proceedings (pp.596-601), 2019.