PatchNet: A Tool for Deep Patch Classification

Thong Hoang[1], Julia Lawall[2], Richard J. Oentaryo[4], Yuan Tian[3], David Lo[1]

[1]Singapore Management University/Singapore, [2]Sorbonne University/Inria/LIP6

[3]Queen's University/Canada, [4]McLaren Applied Technologies/Singapore

vdthoang.2016@smu.edu.sg, Julia.Lawall@lip6.fr, richard.oentaryo@mclaren.com,

yuan.tian@cs.queensu.ca, davidlo@smu.edu.sg

Abstract—This work proposes PatchNet, an automated tool based on hierarchical deep learning for classifying patches by extracting features from commit messages and code changes. PatchNet contains a deep hierarchical structure that mirrors the hierarchical and sequential structure of a code change, differentiating it from the existing deep learning models on source code. PatchNet provides several options allowing users to select parameters for the training process. The tool has been validated in the context of automatic identification of stable-relevant patches in the Linux kernel and is potentially applicable to automate other software engineering tasks that can be formulated as patch classification problems. A video demonstrating PatchNet is available at https://goo.gl/CZjG6X. The PatchNet implementation is available at https://github.com/hvdthong/PatchNetTool.

I. INTRODUCTION AND RELATED WORK

Deep learning techniques have recently been used to automate some software engineering tasks such as code clone detection [1], software traceability link recovery [2], and bug localization [3]. However, no existing work has investigated the problem of learning a semantic representation of code changes, *i.e.*, patches, for classifying them into predefined classes. Patches are composed of a short text describing a change, the lines removed by the change, and the lines added by the change, and all these pieces need to be considered in a holistic way to produce a good semantic representation. Patch classification is an important problem since many automated software engineering tasks, such as just-in-time defect prediction [4], tangled change prediction [5], code reviewer recommendation for a commit [6], etc. can be mapped to it.

Close to our work on patch classification is the prior work by Tian et al. [7] that proposes an automated patch classification approach integrating LPU (Learning from Positive and Unlabeled Examples) [8] and SVM (Support Vector Machine) to build a classification model. To apply LPU+SVM to patches, Tian et al. manually defined a set of features extracted from patches. However, this manual creation process may overlook features that are helpful to classify patches. The chosen features are also specific to a particular patch classification setting (i.e., bug fixing patch identification), and a new set of features is likely needed for other settings. In this work, we replace this manual feature engineering step by leveraging the power of deep learning. In particular, we construct a model that can extract a good semantic representation capturing a patch's hierarchical and structural properties.

This paper presents our tool PatchNet that performs learning on a set of patches. PatchNet performs deep patch classification in two phases. In the training phase, it takes as input a set of labeled patches to learn a deep learning model. This model is then used in the prediction phase on a set of unlabeled patches to produce scores that estimate how likely the given patches fit the class labels. Specifically, PatchNet aims to automatically learn two embedding vectors, representing a commit message and a set of code changes in a given patch, respectively. The two embedding vectors are then used to compute a prediction score from a given patch estimating how likely the patch is relevant to a particular class.

PatchNet is implemented as a command line tool. In the training phase, the user provides a file of labeled patches and the name of a folder in which to put the trained PatchNet model. In the prediction phase, the command used to produce the prediction scores also takes two inputs: a file of unlabeled patches and the name of the folder containing the previously trained model. PatchNet also provides several supplementary options to allow users to select hyperparameters for the training process. PatchNet targets binary classification tasks, however, it can also be used for multi-label classification tasks by reducing the problem of multi-label classification to multiple binary classification problems [9]. PatchNet currently only supports patches on C code.

PatchNet has been applied to the task of stable patch identification in the Linux kernel. Specifically, the Linux kernel follows a two-tiered release model in which a *mainline* version accepts bug fixes and feature enhancements, and a series of stable versions accepts only bug fixes [10]. While the mainline targets users who want to benefit from the latest features, the stable versions target users who value the stability of their kernel. The stable kernel development process requires that all patches be submitted to the mainline, and then propagated by maintainers from there to review for inclusion in stable kernels. The wide variation in rates at which patches are propagated by maintainers to stable kernels across the code base suggest that some bug-fixing patches may be being overlooked. There is thus the potential for an automated, learning-based approach to improve this process. Our evaluation on 82,403 recent Linux patches shows that PatchNet outperforms the state-of-the-art baseline (i.e., LPU+SVM), achieving precision of 0.839 and recall of 0.907. By releasing our tool, we hope to enable others to apply PatchNet to other patch classification tasks.

The rest of this paper is organized as follows. Section II provides a bird's-eye view of the PatchNet architecture. Section III describes how PatchNet preprocesses the patch data

before initiating the learning process. Section IV illustrates how PatchNet can be used in various scenarios. Section V summarizes some experiments with PatchNet. Finally, we conclude in Section VI.

II. OVERALL DESIGN

Figure 1 shows the overall design of PatchNet. PatchNet accepts as input a set of patches, each of which contains both a commit message and a code change. The output of PatchNet associates each patch with a prediction score reflecting how likely the patch satisfies the criteria of the classification. Our framework includes three main modules: the *commit message module*, the *code change module*, and the *classification module*. The commit message module and the code change module transform the commit message and the code change into embedding vectors e_m and e_c, respectively. The two embedding vectors are then passed to the classification module, which computes the prediction score. In the rest of this section, we give an overview of each of these modules.

Commit message module: The architecture of the commit message module is the same as the one proposed by Kim [11] for sentence classification. This module takes a commit message as input and outputs an embedding vector that represents the most salient features of the message. Specifically, we encode the commit message as a two-dimensional matrix by viewing the message as a sequence of vectors where each vector represents a word appearing in the message. We then apply a convolutional layer followed by a max-pooling operation to obtain the message's salient features.

Code change module: Similar to a commit message, a code change can be viewed as a sequence of words. This view, however, overlooks the structure of code changes, as needed to distinguish between changes to different files, changes in different hunks (contiguous sequences of removed and added code), and different kinds of changes (removals or additions). To address this challenge, PatchNet contains a deep hierarchical structure that mirrors the hierarchical and sequential structure of patch code, making it distinctive from the existing deep learning models on source code [12]. For the code changes in a given patch, PatchNet outputs an embedding vector that represents the most salient features of these changes.

The code change module contains a *file module* that automatically builds an embedding vector representing the code changes made to a given file in the patch. Figure 2 shows the architecture of the file module. This module takes as input two matrices (denoted by "–" and "+" in Figure 2) representing the removed code and added code for the affected file in a patch, respectively. These two matrices are passed to the *removed code module* and the *added code module*, respectively, to construct the corresponding embedding vectors by taking into account the structure of the removed or added code.

The input of the *removed code module* is a three dimensional matrix, indicating the removed code in the affected file of the given patch, denoted by $\mathcal{B}_r \in \mathbb{R}^{\mathcal{H} \times \mathcal{N} \times \mathcal{L}}$, where $\mathcal{H}$, $\mathcal{N}$, and $\mathcal{L}$ are the number of hunks, the number of removed code lines

for each hunk, and the number of words of each removed code line in the affected file, respectively. This module constructs an embedding vector (denoted by e_r) representing the removed code in the affected file. The *added code module* also takes as input a three dimensional matrix, indicating the added code in the affected file of the given patch. It follows the same architecture as the removed code module to construct an embedding vector (denoted by e_a) representing the added code in the affected file. These changes in the added and removed code are padded or truncated to have the same number of hunks ($\mathcal{H}$), number of lines for each hunk ($\mathcal{N}$), and number of words in each line ($\mathcal{L}$) for parallelization. Moreover, both modules also share the same vocabulary.

The two embedding vectors are then concatenated to represent the code changes in each affected file, i.e., $\mathbf{e}_f = \mathbf{e}_r \oplus \mathbf{e}_a$. The embedding vectors of the code changes at the file level are then concatenated into a single vector representing all the code changes made by the patch.

Classification module: The classification module takes as input the commit message embedding vector ($\mathbf{e}_m$) and the code change embedding vector ($\mathbf{e}_c$) (see Figure 1). These two vectors are then concatenated to form a single vector representing the patch, i.e., $\mathbf{e} = \mathbf{e}_m \oplus \mathbf{e}_c$. The concatenated vector $\mathbf{e}$ is passed to a fully-connected layer and an output layer, which computes a probability score for the patch. If an additional source of information is available, PatchNet can be easily extended by concatenating an additional information vector (denoted by $\mathbf{e}_i$), collected from the data, to the commit message embedding vector and the code change embedding vector (i.e., $\mathbf{e} = \mathbf{e}_m \oplus \mathbf{e}_c \oplus \mathbf{e}_i$) to form a new vector representing the given patch.

Parameter learning: During the training process, PatchNet uses adaptive moment estimation (Adam) [13] to minimize the regularized loss function [14]. PatchNet learns the following parameters: the word embedding matrices for commit messages and code changes, the filter matrices and bias of the convolutional layers, and the weights and bias of the fully connected layer and the output layer.

III. DATA SELECTION AND PREPROCESSING

The user is responsible for selecting commits for training and annotating them according to the chosen classification scheme. For each commit, preprocessing is then applied to the commit message and the code changes. For the commit message, PatchNet applies standard natural language preprocessing techniques, such as stemming and stop word elimination. For the code changes, PatchNet detects the changed lines using diff, and then expands these changes to include the complete innermost enclosing simple statement, if any, or the header of a conditional or loop, if the change occurs in such code. Lines within the changes are annotated as *error-checking code* (an `if` test that checks for failure of a previous operation), *error-handling code* (code that performs cleanup in case of failure of a previous operation) or *normal* (for everything else), reflecting one aspect of the semantics of the code. PatchNet keeps the names of called functions that are
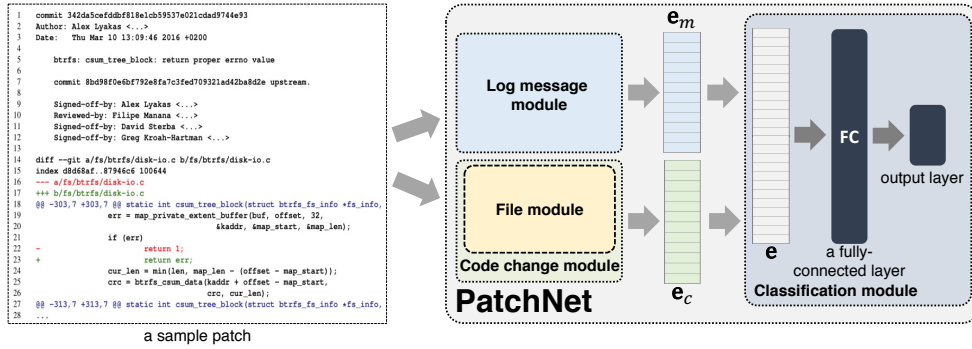
Fig. 1. The overall design of PatchNet to classify stable vs. non-stable patches. A sample patch contains both a textual commit message (lines 5-12) and a set of code changes (lines 14-28) that are applied to an affected file. $\mathbf{e}_m$ and $\mathbf{e}_c$ are embedding vectors collected by the commit message module and code change module, respectively. $\mathbf{e}$ is a single vector formed by concatenating these two embedding vectors.
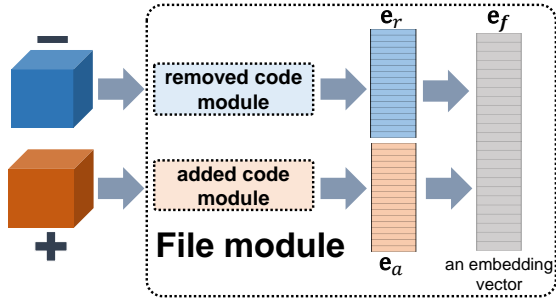

Fig. 2. Architecture of the *file module* for mapping a file in a given patch to an embedding vector. The input of the module is the removed code and added code of the affected file, denoted by "−" and "+", respectively.

not defined in the current file and that occur at least five times in the training dataset. Other identifiers are represented as a generic "identifier" token. Because these preprocessing steps require knowledge of the syntax of the language used by the source code, PatchNet currently only supports C code.

IV. USAGE

In this section, we first describe the use of the preprocessor and then illustrate several usage scenarios for the training and classification process. Moreover, we describe a number of PatchNet's key hyperparameters used during the training process.

A. Preprocessing

The preprocessing step is time consuming for large datasets, and is thus separated from the rest of the training and classification process. The main input to the preprocessing step is a file containing a list of commit identifiers and their labels. The preprocessing step also needs to know the pathname of the git tree containing the commits and a prefix used to construct the names of the various output files. The output is a pair of files, containing the patch data and a dictionary for interpreting the patch data. Only the former needs to be provided to the subsequent training and classification process. A typical command line is:

```
getinfo --commit_list commit_list_file
  --git /path/to/git -o training_data
```

B. Scenario I - Simple Command

PatchNet is implemented in Python 2.7 with Tensorflow 1.4.1,[1] scikit-learn 0.19.1,[2] and numpy1.14.3.[3] PatchNet performs deep patch classification in two phases: the training phase and the prediction phase.

In the training phase, PatchNet takes the command-line arguments `--train` indicating the training phase, `--data` indicating the path of a list of labeled patches, and `--model` indicating the name of the output folder in which to put the model. A sample command that trains a model is as follows:

```
python PatchNet.py --train
  --data training_data.out --model patchnet
```

This command instructs PatchNet to train a model using the data `training_data.out` with the default hyperparameters. In the default setting, the dimension of the embedding vectors, the number of filters, and the number of hidden layers are set to 32, 32, and 10, respectively. The dropout for the training process, the regularization error, and the learning rate are set to 0.5, $1e{-}5$, and $1e{-}4$, respectively. PatchNet automatically creates a new folder with the name `patchnet` (or empties a folder with that name, if it exists) and saves a learned model, consisting of three files, in the folder. For parallelization, the number of changed files, the number of hunks for each file, the number of lines for each hunk, the number of words of each removed or added code are set to 5, 8, 10, and 120, respectively.

In the prediction phase, PatchNet takes the command-line arguments `--predict` indicating the prediction phase, `--data` indicating the path of a list of unlabeled patches, and `--model` indicating the name of the folder containing the model. The following is a sample command to collect a list of prediction scores for a set of unlabeled patches:

```
python PatchNet.py --predict
  --data test_data.out --model patchnet
```

PatchNet has been developed and tested on the Linux platform. However, we believe that PatchNet can be employed

[1] https://www.tensorflow.org/
[2] http://scikit-learn.org/stable/
[3] http://www.numpy.org/

Hyperparameters	Description
`--data_type`	Type of data (commit messages, code change, or both) used to construct a model. Default: both.
`--embedding_dim`	Dimension of embedding vectors. Default: 32.
`--filter_sizes`	Sizes of filters used by the convolutional layers. Default: "1, 2".
`--num_filters`	Number of filters. Default: 32.
`--hidden_layers`	Number of hidden layers. Default: 16.
`--dropout_keep_prob`	Dropout for training PatchNet. Default: 0.5.
`--l2_reg_lambda`	Regularization rate. Default: $1e-5$.
`--learning_rate`	Learning rate. Default: $1e-4$.
`--batch_size`	Batch size. Default: 64.
`--num_epochs`	Number of epochs. Default: 25.

	Accuracy	Precision	Recall	F1	AUC
LPU+SVM	0.731	0.751	0.716	0.733	0.731
PatchNet-C	0.722	0.727	0.748	0.736	0.741
PatchNet-M	0.737	0.732	0.778	0.759	0.753
PatchNet	**0.862**	**0.839**	**0.907**	**0.871**	**0.860**

changes and 14-17% if we ignore the commit message, showing the interest of a model that incorporates both kinds of information. Table II also shows that PatchNet outperforms the state-of-the-art baseline (i.e., LPU+SVM) for the task of stable patch identification.

VI. CONCLUSION

In this work, we present PatchNet, a tool that learns a semantic representation of patches for classification purposes. PatchNet contains a deep hierarchical structure that mirrors the hierarchical and sequential structure of commit code, making it distinctive from the existing deep learning models on source code. We have demonstrated the tool's applicability in identifying stable patches in Linux kernel. We encourage future researchers to benefit from PatchNet by applying it to other tasks that can be mapped to a patch classification problem. PatchNet is open-source and can be run from the command line with simple options.

REFERENCES

[1] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE*, 2016, pp. 87–98.
[2] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *ICSE*, 2017.
[3] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *ICPC*, Buenos Aires, Argentina, 2017, pp. 218–229.
[4] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, 2016.
[5] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *ICPC*. ACM, 2014, pp. 262–265.
[6] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: code reviewer recommendation in github based on cross-project and technology experience," in *ICSE Companion*. IEEE, 2016, pp. 222–231.
[7] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *Proc. of ICSE*. IEEE Press, 2012, pp. 386–396.
[8] F. Letouzey, F. Denis, and R. Gilleron, "Learning from positive and unlabeled examples," in *ALT*. Springer, 2000, pp. 71–85.
[9] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.
[10] G. K. Lee and R. E. Cole, "From a firm-based to a community-based model of knowledge creation: The case of the Linux kernel development," *Organization science*, vol. 14, no. 6, pp. 633–649, 2003.
[11] Y. Kim, "Convolutional neural networks for sentence classification," in *EMNLP*, 2014, pp. 1746–1751.
[12] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *IJCAI*, 2017.
[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.
[14] S. S. Haykin, *Kalman filtering and neural networks*. Wiley Online Library, 2001.

on other platforms such as Windows or MacOS if all the necessary libraries are installed (i.e., Tensorflow, scikit-learn, and numpy). More details about PatchNet's installation instructions can be found at https://github.com/hvdthong/PatchNetTool.

C. Scenario II - Tuning PatchNet's Hyperparameters

Table I highlights the key hyperparameters that control how PatchNet trains its model. The `--data_type` flag allows the user to choose what information to include from the patches (i.e., commit messages, code changes, or both). The user can also change other hyperparameters of the model such as the dimension of embedding vectors (i.e., "`--embedding_dim`"), number of filter sizes (i.e., "`--filter_sizes`"), number of filters (i.e.,"`--num_filters`"), etc. The following example illustrates how to execute PatchNet with custom settings:

```
python PatchNet.py --train
  --data data.out --model patchnet
  --embedding_dim 128 --filter_sizes "1,2"
  --num_filters 64
```

V. STABLE PATCH IDENTIFICATION

We have applied PatchNet to the problem of identifying Linux kernel bug-fixing patches that should be backported to previous stable versions. Based on a set of 42,408 stable patches and 39,995 non-stable patches drawn from Linux kernel versions from v3.0 (July 2011) to v4.12 (July 2017), we trained and tested the PatchNet model using its default hyperparameters following 5-fold cross-validation. Considered patches are limited to 100 lines of changed code, following the Linux kernel stable patch guidelines. Non-stable patches in the dataset are chosen to have the same size properties (number of files and number of changes lines) as the stable ones.

Table II shows the performance (i.e., accuracy, precision, recall, and F1) on this problem of LPU+SVM [7] and three variants of PatchNet: PatchNet-C, PatchNet-M, and Patch-Net. PatchNet-C uses only code change information while PatchNet-M uses only commit message information. PatchNet uses both commit message and code change information. Accuracy, precision, recall, F1, and AUC for the stable patch identification problem drop by 15-20% if we ignore code