2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) | 978-1-6654-9628-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSTW55395.2022.00034

TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks

Mohamed Abdelkarim
Siemens EDA
Cairo, Egypt
mohamed.abdelkarim@siemens.com

Reem ElAdawi
Siemens EDA
Cairo, Egypt
reem.eladawi@siemens.com

***Abstract*—Regression testing is facing a bottleneck due to the growing number of test cases and the wide adoption of continuous integration (CI) in software projects, which increases the frequency of running software builds, making it challenging to run all the regression test cases. Machine learning (ML) techniques can be used to save time and hardware resources without compromising quality. In this work, we introduce a novel end-to-end, self-configurable, and incremental learning deep neural network (DNN) tool for test case prioritization (TCP-Net). TCP-Net is fed with source code-related features, test case metadata, test case coverage information, and test case failure history, to learn a high dimensional correlation between source files and test cases. We experimentally show that TCP-Net can be efficiently used for test case prioritization by evaluating it on three different real-life industrial software packages.**

***Index Terms*—regression testing, test case prioritization, neural networks, deep learning, fusion network, incremental learning, hyperparameter optimization**

I. Introduction

Software testing plays a crucial role in the overall software development process. It detects errors and defects in the software of a system and ensures the software works in accordance with its technical specifications. Introducing new features to software may result in some errors appearing in the existing features. Regression testing is performed to identify these errors. Regression testing is a critical process in software testing, as it allows developers to ensure that a modified application will not introduce defects in existing features [1], [2]. However, due to the fast production cycle of custom software and the daily growing number of test cases, the limited amount of testing time available makes it hard, if not impossible, to run all the regression test cases with any modifications in the software source code [3], [4], [5]. In addition, testing requires at least 50% of the resources allocated to a software project [6], so optimizing the testing process not only optimizes the use of time and resources, but also enhances the overall production cycle.

As software development grows, identifying and fixing regressions becomes one of the most challenging problems facing the software development industry. Many tools and techniques exist to test the changed software within time and resource constraints. There are three categories of regression testing techniques [7]:

1) Retest all: The retest all technique considers all previous test cases in testing the changed system. Therefore, it is a time-consuming technique.
2) Test case selection: The test case selection technique selects some test cases from the test suite based on some criteria.
3) Test case prioritization (TCP): The TCP technique orders test cases based on some calculated priorities. The main aim of this technique is the early execution of test cases that have higher priority, in order to increase the early fault detection rate [8], [9]. Prioritization of test cases has one advantage over test case selection: it allows continuous testing until resources are exhausted or all test cases are executed, while always focusing on the most important test cases [10].

Deep neural networks (DNN) comprise a recent category of machine learning algorithms that uses multiple layers to progressively extract higher-level features from the raw input. DNNs recently made huge advances, achieving impressive results in a wide variety of problems and fields like computer vision, natural language processing (NLP), and much more. Many studies began introducing machine learning (ML) and deep learning (DL) techniques to TCP, and they are showing promising results. Using the full power of these recent advances in techniques and architectures of deep learning, we introduce a novel end-to-end, self-configurable, and incremental learning DNN tool for test case prioritization (TCP-Net), which operates on coverage and history features. TCP-Net is fed by test cases metadata, source files coverage information, and test cases execution history. TCP-Net optimizes all feature extraction process steps internally, learns a high dimension correlation between test cases and source files from the coverage information and the history of test cases failure, and then uses these features to prioritize test cases based on certain changes in source files for a given new build. This work was implemented and tested on regressions of three industrial software products provided by Siemens EDA: Calibre Auto-Waivers [11], Calibre PERC [12], and Calibre nmDRC [13].

The contributions of this paper are:

1) Combination of coverage and history-based features in an efficient, novel DNN architecture that improves industry practice in regression testing and significantly

978-1-6654-9628-5/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSTW55395.2022.00034

reduces time and resources.

2) Introduction of an end-to-end, fully automated, self-configurable, incremental learning tool containing the novel DNN architecture as its engine, which efficiently performs dataset analysis, model architecture design and optimization, hyper-parameters tuning, and periodic incremental learning.
3) Experimentation with the TCP-Net on three industrial datasets, and extensive analysis of model performance, showing the potential of our approach for integration into CI regression testing in practical environments.

The rest of the paper is organized as follows: Section II discusses related work, Section III presents the methodology, providing all needed theoretical background, as well as details of the TCP-Net architecture implementations and training, Section IV provides dataset and experimentation details, Section V presents and analyzes results, and Section VI provides a conclusion and potential future work.

II. RELATED WORK

Lately, many studies have tried to optimize TCP to reduce time and resources, and to highlight the most relevant test cases to run first. Many studies began introducing ML and DL techniques to TCP. The authors in [14] proposed a test case prioritization technique for system testing using supervised machine learning that analyzes metadata and natural language artifacts like test case age, test case description (natural language), number of linked requirements, and more to compute test case priority values, but this work didn't include history information. In [15], a similar approach was used to present an optimization mechanism for ranking test cases by their likelihood of revealing faults in an industrial environment. This ranking model was fed with four features: code coverage, textual similarity, fault history, and test age. These features were then used to train a support vector machine model, resulting in a score function that sorts test cases. In [16], a logistic regression model was trained and fed with similarity-based features, such as similar lines of code. This work used a simple logistic regression model and only simple features, so the accuracy was not very high. The work in [17] proved that prioritization at the commit-level, instead of test-level, would enhance fault detection rates. Another technique was introduced in [18], [19] using semi-supervised learning approaches, like clustering algorithms, and endorsing coverage-based techniques, grouping test cases in clusters from which humans could make selections. Deep learning algorithms, introduced in [20], [21], implemented deeper architectures like deep belief networks to generate more expressive features from the original dataset. In this work an ML classifier was used to get the more-likely-to-fail test cases. There were also some studies using reinforcement learning in TCP-like processes [22], [23], [24]. This method prioritized test cases based on their historical execution results and duration.

More recently in [25], the authors proposed a data-driven deep learning framework called NNE-TCP (neural networks embeddings for test case prioritization) that, based on historical data, can learn which modified files in a given commit led to test case transitions, and from this information, deduce the aforementioned file-test case relationships. The drawback of this work is that the input features fed to the model and represented by the embedding are not rich enough with information that can be used by the model to make a good correlation between test cases and source files. The model only learns from the history of test cases, with no other information or metadata about test cases or source files provided, so it only sees them as black boxes. As a result, the model used was too simple and the final result was not good enough that running a simple hardcoded prioritization algorithm to compare test case history using the assumption that ”tests that suffered transitions recently are more likely to transition again” achieved better results than the trained deep model.

III. METHODOLOGY

A. Dataset

In this section, model input features and the complete dataset structure are discussed and explained. Based on the discussions in [26], the authors argue that most software under test (SUT) in existing studies cannot be considered adequate for the evaluation, as available SUTs tend to have an extremely low number of test cases and failure rates, making them unsuitable for evaluating ML-based test case selection and prioritization (TSP) techniques. This condition motivated us to create a fully automated data collection tool that was used to collect the regression data for three real-life industrial software products. Details of the datasets are provided in Table I. This work was implemented and validated on the aforementioned datasets. TCP-Net operates on two main types of input features: source-file-related features and test-case-related features. During training, the model tries to learn a high dimensional correlation between the two input features to learn the failure pattern of each test case, given certain changes in certain source files during different builds.

TABLE I
OVERVIEW OF INDUSTRIAL DATASETS USED IN THE TRAINING AND EVALUATION OF TCP-NET

Dataset	No. test cases	No. source files	No. builds	Failure %	Final dataset size
Calibre Auto-Waivers	2089	35	945	0.57%	50083
Calibre PERC	11497	222	941	0.83%	335621
Calibre nmDRC	3324	69	904	1.43%	329912

1) source-file-related features: These features include the properties that describe the SUT source files, their directories' tree structure, and the modifications done in the source files throughout different builds across time. Here, the target is to provide the model with the names of all the files that were modified in a certain build. As ML models don't accept categorical data, the file names must be encoded first in numerical form. To do so, we use the bag-of-words (BOW) model, which is a way of extracting features from the text for use in modeling, such as with ML algorithms. Using BOW to construct the source-file-related features dataset for the model training, the following steps are executed:

1) A database is constructed containing all the unique names of all files and directories in the whole dataset (e.g. 1000 builds). Each build results in some modified source files (a source file is considered modified if operated on by a developer, whatever the modification was). Our encoding technique operates on the complete path of each source file from the top directory. This filename, along with all directory names that appear in the path of the source file, is used. This encoding technique makes the model perceive the tree structure and the distribution of source files under different functionality directories of the SUT, and enables the model to operate on new source files never seen before during inference time by knowing only their tree and top directories. Let us assume the total number of found unique names (files/directories) = *n*.
2) A feature vector is constructed for each build that is an array of length *n* (same length of the previous step), and each entry contains a count of the appearance of the corresponding unique word that indicates a certain file or directory, or 0 if it doesn't appear in this build. Now we have a two-dimensional matrix of dimension 1000 (number of builds) x *n* (number of unique file/directory names).
3) The produced feature vector from the previous step is a sparse vector (most entries are 0), and thus can be reduced using the principal component analysis (PCA) technique, which reduces the dimensions of a certain vector to only the most important number of axes that maximally represents the data variance (total length= *n'* where *n'* << *n*).

We now have a two-dimensional matrix of dimension 1000 (number of builds) x *n'* (number of unique file/dir names compressed using PCA), which represents the files modified in each build in the last 1000 builds.

2) Test-case-related features: For test-case-related features, we must construct a database that contains some features/metadata for all the SUT regression test cases. The features used are:

- Test case ID (a unique number given to each test case)
- Age of test case (first execution date)
- Test case run time
- Test case required resources (memory/no. of cores)
- Test case files coverage

Test case coverage information is the most important feature vector. It can be collected using three methods: dynamic analysis, static analysis, or code similarity analysis [26]. Dynamic analysis is the most precise method, so we used this method in our dataset. We generated test case coverage information using Gcov [27], a dynamic analysis tool that runs the test cases and calculates the line coverage (i.e., the number of times each line in a certain .c or .h file was hit), and - from which - the lines coverage percentage can be calculated for each source file for these test cases by dividing the number of hit lines by the original file's total number of lines. After generating a coverage vector for each test case, which is composed of the coverage percentages for the source files covered by the test case, the vector is concatenated with the other test-case-related features above to construct the final test case feature vector. An example of the generated coverage vector is shown in Figure 1.

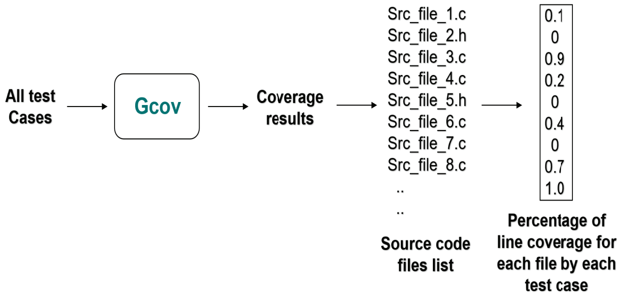

Fig. 1. Generated coverage vector of test-case-related feature vector.

3) Dataset labeling: In supervised learning, the dataset used in training the machine learning model must be labeled. The feature vector consists of the source-file-related features (names of the modified files for each build in the last 1000 builds), and the test-case-related features previously discussed. At this point, a label must be given to each test case each time the regression was run against a certain build. This label is the history of test case execution (pass/fail). These features enable the model to relate each change in the source files with its consequences and effects on the regression test cases that make it pass/fail. After training, it uses this learned complex relation when operating the model on a new change in source files to predict the most probable test cases to fail.

To complete this step, all test cases that were run on each build from the past 1000 builds of the dataset must be labeled. To do so, all the run results must be classified into two classes: pass or fail. The final dataset labeling is *m* regression test cases executed against 1000 builds, so the labeled results number will be 1000**m*. As shown in table I, the failure rates of the three datasets are low, so under-sampling is used to decrease the dataset size and increase failure rates by randomly dropping samples from the majority class (pass class) until reaching a dataset with a failure ratio of 10%, which in return boosts the model training and performance. The final dataset sizes are listed in Table I.

B. TCP-Net model

1) Network architecture: After studying multiple algorithms and techniques that could solve this prioritization problem, we chose end-to-end DNNs because of the great results they achieved in many fields, and their outstanding ability

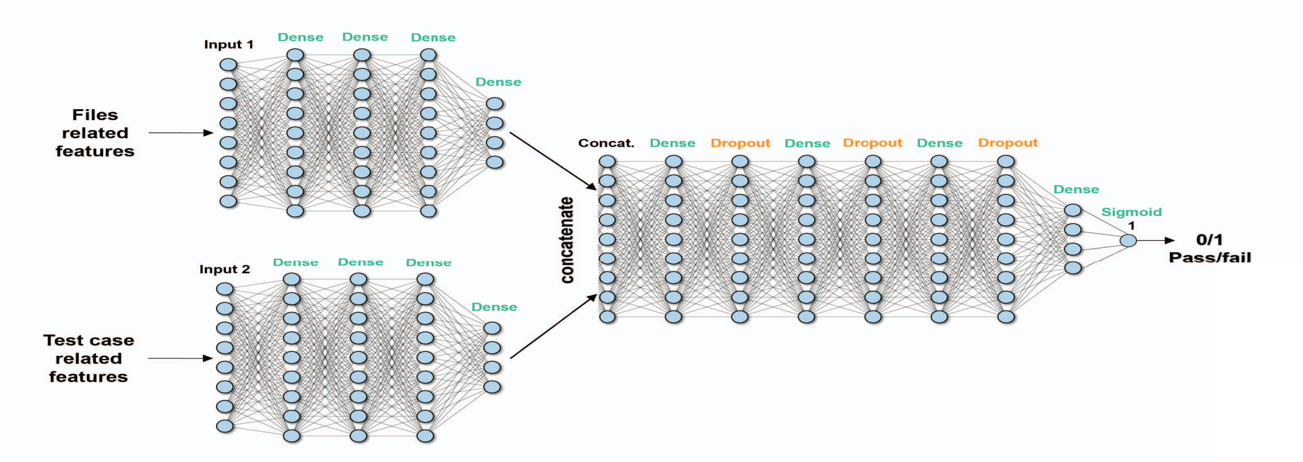

Fig. 2. **TCP-Net Architecture**: source-file-related features and test-case-related features are fed to the two branches of the middle fusion deep fully connected network architecture. Each branch acts as a feature extractor to its input features only, then the extracted features are compressed till reaching the final layer in each branch. The extracted features are then concatenated and fed to the main feature extractor until reaching the final sigmoid classifier layer.

to optimize all process steps internally, automatically extract relevant and most important features, and use them to learn the best possible classification high dimensional boundaries. A ”deep fusion architecture” was used to combine the two input features discussed previously. Fusion techniques include early, middle, and late fusion [28]. In fusion networks, there are multiple inputs. Features are extracted from each input, then fused to make a prediction. The point at which fusion takes place defines the type of fusion. If the fusion occurs directly after extracting features, it is a middle fusion, but if fusion between features occurs after passing the extracted features through more than one fully connected layer, it is called late fusion. Many experiments were done on the project’s data using different types of fusion architectures, and the middle fusions achieved the best results for our problem, so middle fusion architecture was used. After some trials and iterations of architecture optimization, we selected the TCP-Net base model general architecture shown in Figure 2. It is a middle fusion, deep fully-connected network architecture of two branches, one for source-file-related features and one for test-case-related features. Each branch acts as a feature extractor to its input features only, and then the extracted features are compressed until reaching the final layer in each branch. The source-file-related features branch and test-case-related features branch each consist of four fully-connected layers. Extracted features from each branch are first concatenated, then fed to the main feature extractor until reaching the final sigmoid classifier, which learns to correlate the two features to discover the relations between each combination of source files and test cases and the failure pattern of the test cases. The main body of the network has nine alternating fully-connected and dropout layers, to make it learn more generic features and protect it from over-fitting.

2) Imbalanced data handling: The class imbalance problem is a problem that faces most ML/DL classification problems. It occurs when there are one or more classes (majority classes) that occur more frequently than the other classes (minority classes). As a result, the learning becomes biased towards the majority classes. Due to the lack of sufficient examples, the model fails to learn meaningful patterns that could aid it in learning the minority classes. To solve this problem, the class weights [29] technique was used. The class weights technique weighs the loss computed for different samples differently based on whether the sample belongs to the majority or the minority classes. We essentially want to assign a higher weight to the loss encountered by the samples associated with minority classes.

3) Hyper-parameter optimization: When working on an ML project, a series of steps should be followed until reaching the required target, one of which is to execute hyper-parameter optimization on the selected model. Hyper-parameters are different parameter values used to control the learning process. Examples of hyper-parameters in DNNs are the number of layers, number of neurons in each layer, batch size, etc. These parameters are tunable, and can directly affect how well the model trains. Hyper-parameter optimization is the process of finding the right combination of hyper-parameter values to achieve maximum performance on the data in a reasonable amount of time. Furthermore, it makes it possible to automatically configure and optimize the model hyper-parameters on different datasets without human involvement. In this work, we implemented hyper-parameter optimization via the Hyperopt library. Hyperopt [30] is a powerful Python library for hyper-parameter optimization. Hyperopt uses a form of Bayesian optimization for parameter tuning that results in the best parameters for a given model.

The size of each layer (the number of neurons) in the whole network is determined by the Hyperopt library. The Hyperopt library has different functions to specify ranges for input parameters. The parameters of the used function are min,

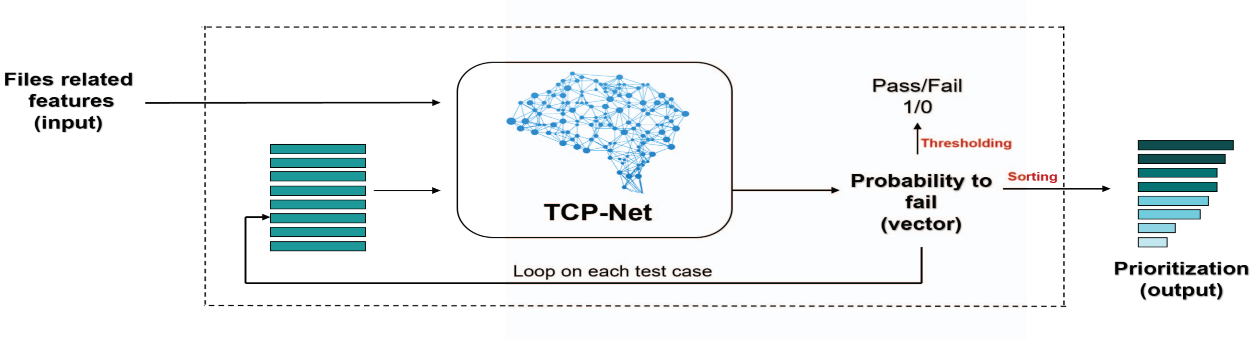

Fig. 3. Model operation at inference time.

max, and q. The min and max hyper-parameters set the upper and lower bounds of the bounded discrete search space, while q is the quantization step. In our model, min is set to 10, which is the smallest possible number of neurons in any layer, while max is set to be equal to the feature length in each of the two branches, and equal to the mean of the the two branches' feature lengths in the main body, which makes the model adapt its total number of parameters according to the requirements of each software under test (number of source files and number of test cases). The q hyper-parameter is set to 10, which makes the objective still somewhat smooth as increasing or decreasing layer size by 10 will not have a remarkable effect on the objective loss function. The settings used are as follows:

- Objective function: minimize(loss)
- Search algorithm: Adaptive tree of Parzen estimators (ATPE)
- Maximum number of evaluations: 200
- One iteration loss calculated after: 10 epochs

C. Evaluation metrics

1) Average percentage of faults detected (APFD): Depending upon the fault criterion considered, the APFD metric [10] was computed to measure the rate of fault detection of coverage-based prioritization techniques. APFD measures the weighted average of the percentage of faults detected over the life of a test suite. APFD values range from 0 to 100; the higher number simply indicates faster fault detection rates as shown in Equation 1:

$$APFD = 1 - \frac{TF1 + TF2 + ... + TFv}{hv} + \frac{1}{2h} \tag{1}$$

where:
T = test suite under evaluation
v = number of faults contained in the program under test
h = total number of test cases
TFi = position of the first test in T that exposes fault i

2) Accuracy, precision, recall, and F1-score: Assuming a classifier that predicts whether an item (e.g. test case) belongs to a certain class or not (returns true for positive prediction or false otherwise). True positive (TP) and True negative (TN) refer to correct classification cases (ground truth and classifications are identical), whereas False positive (FP) and False negative (FN) refer to incorrect classifications. According to the above definitions, the definitions of accuracy, precision, recall, and F1-score metrics are as follows:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

$$precision = \frac{TP}{TP + FP} \tag{3}$$

$$recall = \frac{TP}{TP + FN} \tag{4}$$

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{5}$$

IV. Experimentation

In this section, we discuss TCP-Net training, operation, and results.

A. Model training

As discussed in Section III-A, there are two types of input features saved in two CSV files, one for test-cases features and one for source-files features. To train the model, a custom-made batch loader traverses the two files, randomly picks and loads *p* test-case features and *p* source-files features with their label (which is the pass/fail result of the picked test case run with the picked source-files changes feature), saves this data in memory, and feeds it to the model for one training iteration, where *p* is the batch size. TCP-Net is then trained to classify using each feature pair as a pass or fail, using the following hyper-parameters :

- Epochs: 250
- Batch size: 1024
- Optimizer: Adam (lr=1e-4)
- Loss function: Binary cross entropy
- Dataset splitting: 80% training, 20% testing

TABLE II
PREDICTION RESULTS FOR TCP-NET ON THE THREE INDUSTRIAL DATASETS

Data set	**Train**				**Test**				**Trainable parameters**
	accuracy	**F1**	**recall**	**precision**	**accuracy**	**F1**	**recall**	**precision**	
Calibre Auto-Waivers	0.97	0.86	0.88	0.85	0.97	0.84	0.87	0.82	269,551
Calibre PERC	0.99	0.94	0.97	0.91	0.98	0.92	0.95	0.89	351,871
Calibre nmDRC	0.97	0.84	0.95	0.76	0.96	0.82	0.92	0.73	323,871

B. Model inference

After training the model and reaching the best possible accuracy, the sigmoid output (confidence), which is a number from 0 to 1 that indicates the probability of failure, is used to prioritize the list of regression test cases. After training, there is a saved database that contains the feature vector of each test case in the regression, as shown in Figure 3. When introducing a new build, the names of the modified files are extracted and encoded in the same way discussed in Section III-A, and then these features are given to the model through the source-files features branch. The tool loops on all the saved regression test cases features, inputs them through the second branch of test-cases features, and generates a probability of failure number for each test case. After looping through all the test cases, the tool prioritizes them by their probability to fail. Test cases with a high probability of failure are at the top of the list.

V. RESULTS ANALYSIS

To the best of our knowledge, there is no previous work using the same technique and dataset feature vectors. According to the discussion in [26], there are no evaluation baselines in most recent primary studies, and ML-based TCP techniques are evaluated using a variety of metrics. These metrics may not be computed similarly, or treat all test cases equally regardless of their verdicts, making it difficult to compare the results of different techniques. Accordingly, the results are compared against the currently most used technique in the industry, which is the random shuffling approach.

A. Training results

From the training results in Table II, it is obvious that the classification accuracy of our model through training and testing sets is high. The validation accuracy is slightly less than the training accuracy, so there is no over-fitting. Recall is higher than precision, as recall indicates the number of false positives. In our case, we want the model never to miss a probable failure found by a failing test case, so the model was biased to the recall side at the expense of precision, which is a bit lower (as an indication of a higher rate of false negatives). These false negatives don't affect the results much, other than predicting some passing test cases as failing ones, and running them. We can also see that the Hyperopt optimizer managed to configure the model number of parameters according to the needs of each dataset, to maintain high accuracy and minimum memory utilization. To further study the performance of TCP-Net, we measured the APFD score for each dataset. As shown in Figure 4, the random shuffling APFD score is around 50 for the three datasets. The TCP-Net APFD scored 94.47 for the Calibre Auto-Waivers dataset, 95.12 for the Calibre PERC dataset, and 94.128 for the Calibre nmDRC dataset. These results show the self-configurability of TCP-Net, which enables it to maintain high performance across different subjects with zero human involvement.

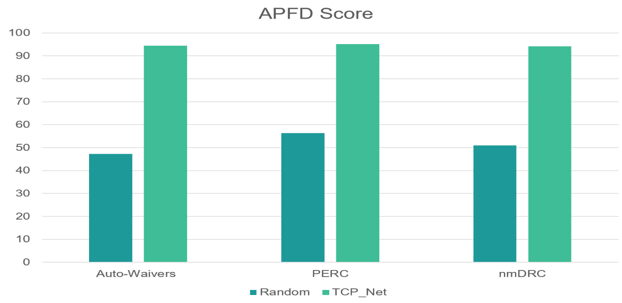

Fig. 4. Comparison of TCP-Net and random shuffling in terms of APFD across three industrial datasets.

B. Pattern-learning analysis

After training a DL model it acts as a black-box function. We only observe the inputs and the outputs without understanding what is going on inside nor can we validate the learning process. Many algorithms and techniques evolved over time to solve this problem. t-Distributed Stochastic Neighbor Embedding (t-SNE) [31] is an unsupervised, non-linear technique used for data exploration and visualizing high-dimensional data. Simply, t-SNE gives a feel or intuition of how the data is arranged in a high-dimensional space. t-SNE was used to compare the high dimensional distribution of the input features before and after passing through the trained TCP-Net model. Figure 5(a) shows that the input data has no pattern to differentiate between passing and failing test cases as they are randomly distributed. Figure 5(b) shows the model successfully managed to learn a high dimensional distribution that can clearly separate passing and failing test cases.

C. Performance analysis

In this section, we further analyze the performance of TCP-Net. Table III lists the hyper-parameter optimization, training, and inference times for each model for the three industrial datasets. The table contains three results columns. The first one displays the time taken by the Hyperopt library to optimize the

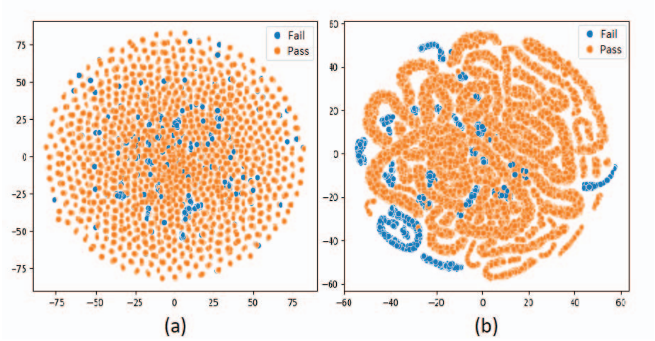

Fig. 5. t-SNE visualization of the input features before and after passing through the trained model using Calibre PERC dataset. Model output features are taken from the last fully-connected layer before the final sigmoid classifier layer.

model architecture. The second column displays the training time for the best model (as selected by Hyperopt). The final column displays the inference time calculated by running all test cases of each software complete regression. It is obvious that the Hyperopt time is far larger than the training and inference times, as the Hyperopt optimizer iterates on many possible points, and in each iteration, the selected model is trained for 10 epochs before measuring its performance. Fortunately, because Hyperopt optimization process is a one-time procedure, it will not affect the continuous integration process efficiency. From the inference time column, the inference time of running the model on the whole regression is negligible compared to the regressions' actual running time (∼3-24 hours), which makes it possible to run the prioritization tool multiple times per day. Finally, the training time is moderate for the three regressions and it's scalable with the number of test cases in each regression. (All the experiments in this paper were conducted on an NVIDIA TESLA P100 GPU).

TABLE III
TIMING ANALYSIS FOR EACH MODEL FOR THE THREE INDUSTRIAL DATASETS **(H:M:S)**

Dataset	**Hyperopt**	**Training**	**Inference**
Calibre Auto-Waivers	00:07:16	00:00:54	00:00:0.2
Calibre PERC	01:48:01	00:13:36	00:00:3.2
Calibre nmDRC	01:36:12	00:12:22	00:00:2.8

D. Incremental learning

The majority of reported works using ML techniques do not consider incremental learning (i.e., continuous integration of new data into already-constructed models), which can be problematic, especially in CI environments [26]. This omission causes practical issues, since modeling performance gradually decays after some number of cycles. Figure 6 shows the performance decay in the models trained on the three datasets across build cycles. The x-axis contains the number of builds (starting from the last build the model was trained on), and the y-axis contains the F1-scores generated by running the model on the new dataset and smoothing the results, using a simple moving average filter to clearly illustrate the decay behavior.

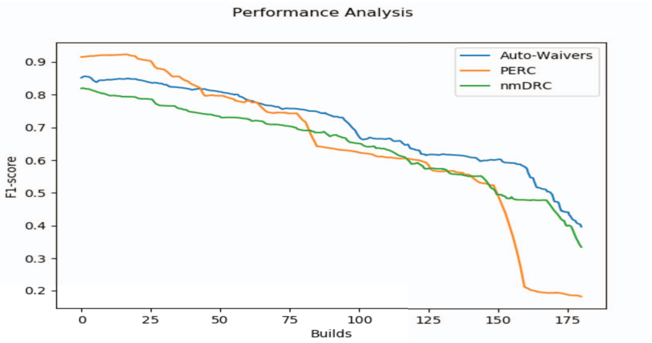

Fig. 6. Visualization of performance decay in the models trained on the three datasets across 180 unseen builds cycles.

To prevent this performance decay, an incremental learning technique is implemented in the TCP-Net tool. As stated in [32], the main challenges of building incremental learning models are adding new classes, runtime, and memory usage. For example, in image classifiers, the datasets are typically extremely large, and it takes quite a long time to train a model (∼ days). Furthermore, the probability of a new class addition during the lifecycle of the model is high. On the contrary, our target problem of TCP is much simpler. There are only two classes (pass/fail), they will never change or be incremented, and the dataset consists of numerical values stored in CSV files, so its total size is relatively small. Lastly, thanks to our TCP-Net optimized model, the training time is relatively small (∼ 1-15 mins). These features enable the TCP-Net tool to utilize the most simple, yet effective way of incremental learning, which is periodic training. Given the decay curves in Figure 6, the TCP-Net tool enables the user to choose a period for the training cycle based on the required accuracy. The model automatically starts a training process on the new data and a subset of the old data, which is determined using a moving window (e.g. if the new dataset builds number is 10, then the oldest 10 builds are dropped and the new builds are added). This approach ensures the model stays updated and quickly learns new patterns in the data stream.

Figure 7 shows the performance analysis after implementing the incremental learning technique. This result was achieved by conducting one training cycle every 18 builds, with a training window size of 180 builds. The first result to notice is the big improvement in the initial F1-scores of the three datasets, which is a result of fine-tuning the model on a smaller window of data. Secondly, it is obvious that the periodic training process improved the model performance across time in the three datasets, and nearly eliminated the performance long-term decay effect. From figure 7, we can see that all products have show an improved decay in F1-score. The Calibre Auto-Waivers curve has shown the most improvement in decay which indicates that the window size used (18 builds) in incremental training is suitable for this dataset. The window

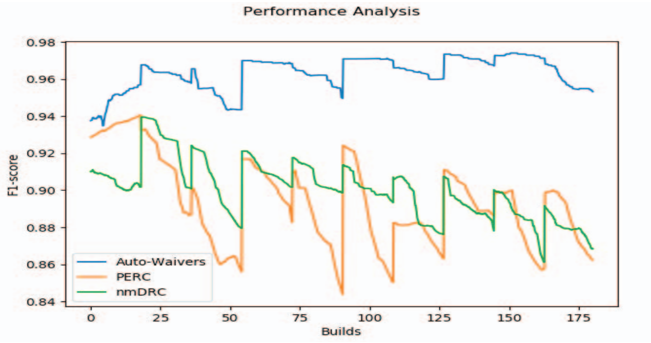

Fig. 7. Visualization of performance analysis after using incremental learning across 180 unseen builds cycles.

size was fixed for all products for comparison purposes but can be adjusted to improve incremental learning to overcome the decay of F1-score with time. Comparing the F1-scores at the end of the 180 build test period of the three datasets before and after the incremental learning, we can see that the Calibre Auto-Waivers dataset F1-score improved by 55%, the Calibre PERC dataset improved by 67%, and the Calibre nmDRC dataset improved by 53%.

VI. Conclusion & future work

In this work, we introduced TCP-Net, a novel end-to-end, self-configurable, and incremental learning DNN tool for TCP. TCP-Net combines coverage and history features in an efficient, novel, DNN architecture that improves industry practice in regression testing. We completed extensive analysis and experiments (performance, configurability, and sustainability) with the TCP-Net tool on three industrial datasets, and performed deep model analysis showing the potential of our approach for integration into CI regression testing in practical environments. For future work, more incremental learning approaches will be explored and tested.

References

[1] D. Suleiman, M. Alian, and A. Hudaib, "A survey on prioritization regression testing test case," in *2017 8th International Conference on Information Technology (ICIT)*. IEEE, 2017, pp. 854–862.

[2] P. K. Chittimalli and M. J. Harrold, "Recomputing coverage information to assist regression testing," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 452–469, 2009.

[3] E. Khanna, "Regression testing based on genetic algorithms," *Int. J. Comput. Appl*, vol. 975, pp. 43–46, 2016.

[4] M. Al-Refai, S. Ghosh, and W. Cazzola, "Model-based regression test selection for validating runtime adaptation of software systems," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 288–298.

[5] J. S. Rajal and S. Sharma, "A review on various techniques for regression testing and test case prioritization," *International Journal of Computer Applications*, vol. 116, no. 16, 2015.

[6] M. J. Harrold, "Testing: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 61–72.

[7] G. Duggal and B. Suri, "Understanding regression testing techniques," in *Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology*, 2008.

[8] S. Varun Kumar and M. Kumar, "Test case prioritization using fault severity," *IJCST*, vol. 1, no. 1, 2010.

[9] S. Yoo and M. Harman, "Tr-09-09: Regression testing minimisation, selection and prioritisation-a survey," 2009.

[10] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[11] "Calibre auto-waivers official web page," Available at https://eda.sw.siemens.com/en-US/ic/calibre-design/physical-verification/auto-waivers/.

[12] "Calibre perc official web page," Available at https://eda.sw.siemens.com/en-US/ic/calibre-design/reliability-verification/perc/.

[13] "Calibre nmdrc official web page," Available at https://eda.sw.siemens.com/en-US/ic/calibre-design/physical-verification/nmdrc/.

[14] R. Lachmann, "Machine learning-driven test case prioritization approaches for black-box software testing," in *The European Test and Telemetry Conference, Nuremberg, Germany*, 2018.

[15] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 975–980.

[16] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, "An improvement to test case failure prediction in the context of test case prioritization," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2018, pp. 80–89.

[17] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: continuous prioritization for continuous integration," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 688–698.

[18] S. Yoo, M. Harman, and S. Ur, "Measuring and improving latency to avoid test suite wear out," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 101–110.

[19] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 1–10.

[20] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[21] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[22] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.

[23] Z. Wu, Y. Yang, Z. Li, and R. Zhao, "A time window based reinforcement learning reward for test case prioritization in continuous integration," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 2019, pp. 1–6.

[24] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, 2021.

[25] J. Lousada and M. Ribeiro, "Neural network embeddings for test case prioritization," *arXiv preprint arXiv:2012.10154*, 2020.

[26] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–43, 2022.

[27] "gcov official web page," Available at https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[28] T. Meng, X. Jing, Z. Yan, and W. Pedrycz, "A survey on machine learning for data fusion," *Information Fusion*, vol. 57, pp. 115–129, 2020.

[29] Z. Xu, C. Dan, J. Khim, and P. Ravikumar, "Class-weighted classification: Trade-offs and robust approaches," in *International Conference on Machine Learning*. PMLR, 2020, pp. 10 544–10 554.

[30] J. Bergstra, "Hyperopt library official web page," Available at https://hyperopt.github.io/hyperopt/.

[31] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[32] Q. Yang, Y. Gu, and D. Wu, "Survey of incremental learning," in *2019 Chinese Control And Decision Conference (CCDC)*. IEEE, 2019, pp. 399–404.