2023 IEEE International Conference On Artificial Intelligence Testing (AITest) | 979-8-3503-3629-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/AITest58265.2023.00024

TCP-Net++: Test Case Prioritization Using End-to-End Deep Neural Networks - Deployment Analysis and Enhancements

Mohamed Abdelkarim
Siemens Digital Industries Software
Cairo, Egypt
mohamed.abdelkarim@siemens.com

Reem ElAdawi
Siemens Digital Industries Software
Cairo, Egypt
reem.eladawi@siemens.com

***Abstract*—The increasing number of test cases and frequency of continuous integration in software projects has created a bottleneck in regression testing. To save time and hardware resources, machine learning techniques can be applied without compromising quality. In this work, we present a case study for deployment analysis and results of using our previous work: TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks [1] in a real-life industrial environment, showing roadblocks, challenges, and enhancements done to improve its performance and usability, achieving 90% to 100% failure coverage by running an average of 23% to 39% of the test cases.**

***Index Terms*—software testing, regression testing, test case prioritization, neural networks, deep learning, AI**

I. Introduction

Effective software testing is essential in ensuring the overall quality of the software development process. Testing is crucial in detecting any errors or defects in the software, ensuring that the software works in line with its technical specifications. However, when new features are introduced, it can lead to errors appearing in existing features. To address this issue, regression testing is performed to identify and fix these errors. Regression testing is critical in software testing, as it enables developers to ensure that a modified application will not introduce defects into existing features [2], [3].

Despite the importance of regression testing, it can be challenging to run all the regression test cases with every modifications in the software source code due to limited testing time and the growing number of test cases. Additionally, testing can require a significant amount of resources, with at least 50% of a software project's resources allocated to it [4]. As a result, optimizing the testing process can not only optimize time and resources but also enhance the overall production cycle [5]–[7].

The software development industry faces significant challenges in identifying and fixing regressions. Therefore, various techniques and approaches have been developed to test changed software within time and resource constraints. Recent advances in deep neural networks have shown promising results in solving problems that require data processing, classification, or regression. Many studies have also introduced machine learning (ML) and deep learning (DL) techniques to the software testing process, demonstrating their potential to improve the effectiveness of testing [2], [3].

In this work, we present a case study for deployment analysis and results of our previous work: TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks [1]. TCP-Net operates on coverage and history features and is fed by metadata from test cases, coverage information from source files, and the history of test case failures. It optimizes all feature extraction process steps internally, learns high dimensional correlations between test cases and source files from the coverage information and the history of test case failure, and then uses these features to prioritize test cases based on changes in source files for a given new source code changes. We tested TCP-Net in a real-life industrial environment, showing the roadblocks, challenges, and enhancements we made to improve its performance in real-life scenarios.

This work was deployed and tested focusing on regressions of two industrial software packages provided by Siemens EDA: Calibre PERC [8], and Calibre xRC [9].

The contributions of this paper are:

1) Deployment of a novel deep learning prioritization technique TCP-Net in a real-life industrial environment showing deployment results analysis which examines the success of TCP-Net in improving regression testing time in the industrial environment .
2) Introducing new techniques, metrics and informative graphs to clearly measure the real performance of TCP-Net over time focusing on the time saved, for each build regression testing.
3) Developed an improved core model for TCP-Net using ensemble learning [10] for higher performance, and more stable operation over time.

The paper is structured as follows: Section II examines the related work, Section III describes the dataset and methodology, including the TCP-Net++ architecture and the implementation and training of the improved ensemble learning model. Section IV presents the experimentation details, while Section V presents and analyzes the results. Finally, Section VI offers a conclusion and outlines potential future work.

979-8-3503-3629-0/23/$31.00 ©2023 IEEE
DOI 10.1109/AITest58265.2023.00024

II. RELATED WORK

Recently, there has been a surge in studies aimed at optimizing Test Case Prioritization (TCP) in order to reduce time and resources, and to prioritize relevant test cases. To this end, ML and DL techniques have been introduced in TCP research. Lachmann et al. [11] proposed a supervised machine learning-based test case prioritization technique for system testing, which analyzed metadata and natural language artifacts, such as test case age, description, and number of linked requirements, to compute test case priority values. However, this approach did not take history information into account. Busjäger et al. [12] used a similar approach to optimize test case ranking in an industrial setting, feeding a ranking model with features like code coverage, textual similarity, fault history, and test age, which were used to train a support vector machine model. Palma et al. [13] trained a logistic regression model using similarity-based features, such as similar lines of code. Liang et al. [14] demonstrated that prioritization at the commit-level, instead of the test-level, could enhance fault detection rates. Yoo et al. [15] and Chen et al. [16] used clustering algorithms and coverage-based techniques to group test cases in clusters, from which humans could select test cases.

Deep learning algorithms were implemented in TCP research, such as the work by Yang et al. [17] and Kamei et al. [18], which used deep belief networks to generate more expressive features from the original dataset. These works employed ML classifiers to predict the most likely test cases to fail.

Lousada et al. [19] proposed a data-driven deep learning framework, NNE-TCP, which can learn file-test case relationships based on historical data. However, the input features fed to the model and represented by the embedding were not rich enough, resulting in a relatively simple model with suboptimal performance. As a result, a simple hardcoded prioritization algorithm was proposed, based on the assumption that ”tests that suffered transitions recently are more likely to transition again,” which achieved better results than the trained deep model.

More recently, the work in [20] proposes a test case selection approach for black-box regression testing of embedded systems, which uses previously fixed defects to select a subset of test cases associated with the modified parts of the source code. A case study on three consumer electronics projects demonstrates that the approach can detect a significant percentage of defects by selecting a smaller subset of the test cases. Specifically, from 65% up to 85% of the defects detected by the whole test suite can be detected by selecting from 30% up to 70% of the test cases.

The paper [21] proposes a comprehensive approach to evaluate the effectiveness of ML-based Test Case Prioritization (TCP) techniques in Continuous Integration (CI). The approach includes a data model, a comprehensive set of features, and methods and tools to collect a dataset for 25 open-source software systems. The paper answers four research questions related to the effectiveness of ML-based TCP, feature impact, decay of ML-based TCP models, and the trade-off between data collection time and effectiveness of ML-based TCP techniques. the results indicate that collecting Record Features (REC) features achieves an APFD close to models trained with the full set of features, and not retraining models for more than 11 builds leads to unstable APFD, while coverage-related features are the most expensive features to collect but have the least impact on the effectiveness of ML TCP models, and the authors have made their datasets publicly available to serve as a benchmark for future studies.

The authors in [22] propose an online learn-to-rank model using reinforcement learning techniques for test case prioritization in CI environments. The approach minimizes testing overhead, adapts to changes in the environment, and reports over 95% of test failures with only 40% of available test cases executed. The paper validates the approach on an industrial case study.

This paper [23] compares Test Case Selection (TCS) and Test Case Prioritization (TCP) techniques, and develops a new combination of the two called Fastazi. The study shows that Fastazi can detect the first test failure nearly two times faster than using TCS or TCP alone. The study was performed on 12 Java-based open-source projects, and statistical analysis showed that Fastazi was more effective than Ekstazi and FAST [24] [25]. Fastazi was found to have negligible additional testing time and can improve failure detection even when time for testing is limited.

III. METHODOLOGY

A. Dataset

In this section, we discuss and explain the dataset used to train the model. TCP-Net operates on two main types of input features, as shown in Figure 1: source-file-related features and test-case-related features. During training, the model attempts to learn a high-dimensional correlation between these input features to identify the failure pattern of each test case, given certain changes in certain source files during different builds. More details concerning the input features and the complete dataset structure are discussed and explained in the dataset section of [1].

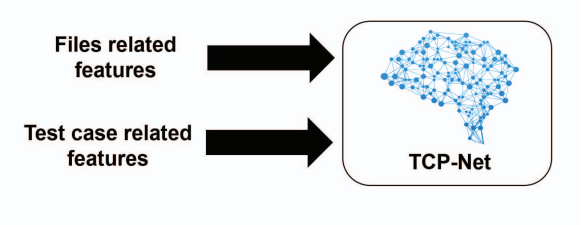

Fig. 1. Model inputs.

The datasets used for this work’s experiments are listed in Table I. This work was deployed and tested using two industrial software packages provided by Siemens EDA: Calibre PERC [8] and Calibre xRC [9]. In the previous work [1], we

collected, sorted, and labeled all test cases that were run on each daily build from the past 1000 days to form the final training dataset. However, experiments conducted after TCP-Net deployment showed that 1000 builds was an extremely large period to fit by the model. Data drift occurs in any time-dependent dataset, and fitting a very large period of about 2.7 years was very difficult for the model to find one specific failure pattern through such a long period of continuously changing code, resulting in less accurate results.

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

Dataset	Test cases	Source files	Builds	Failure %	Final dataset size
Calibre PERC	13615	477	183	1.88%	241098
Calibre xRC	88426	714	181	0.77%	288893

We conducted an experiment to determine the best period for the model to fit well. For this experiment, we used the PERC dataset. We trained a model normally, as discussed in [1], on a certain number of builds, then used the latest month's builds (30 builds) to test the model's performance on an unseen period of time. We used the mean average percentage of fault detected (APFD) [26] over these 30 builds as a metric to measure the prioritization ability of the trained model for each trial.

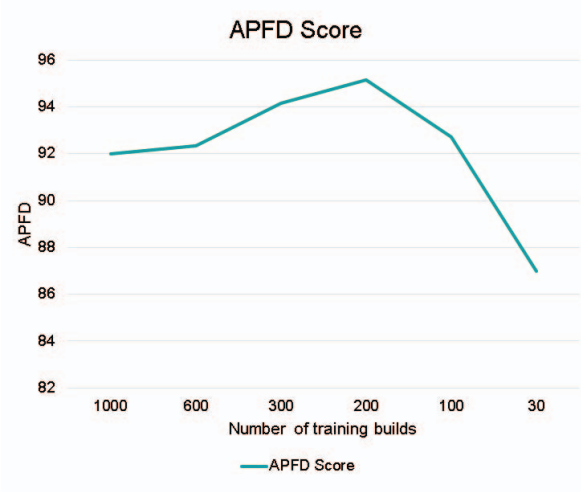

Fig. 2. Mean average percentage of fault detected (APFD) over 30 builds for different numbers of builds used in training.

The results are shown in Figure 2. We observed that as the number of builds decreased, the accuracy of the model improved slowly, indicating a better dataset and a better understanding of internal patterns, until it reached 200. Then it began to decay quickly due to the smaller dataset size and the lack of information for the model to learn from. Therefore, we used 200 as the best number of builds for TCP-Net, besides the small number of builds makes it much faster and easier to collect the required dataset. The dataset sizes used for 200 builds are listed in Table I.

B. TCP-Net model Network architecture

Figure 4 displays the general architecture of the TCP-Net base model. The model is a deep fully-connected network with a middle fusion design, consisting of two branches that serve as feature extractors for source-file-related and test-case-related features separately. Each branch compresses the extracted features until reaching the final layer, and each branch comprises four fully-connected dense layers. The extracted features from both branches are concatenated and fed to the main feature extractor, which includes nine alternating dense and dropout layers. The final Activation function is responsible for learning the correlation between the combined source-file and test-case features and the failure pattern of the test cases. This design helps the network learn more general features and avoid overfitting. More information and details about TCP-Net imbalanced dataset handling, hyperparameters optimization, evaluation metrics, and more can be found in [1].

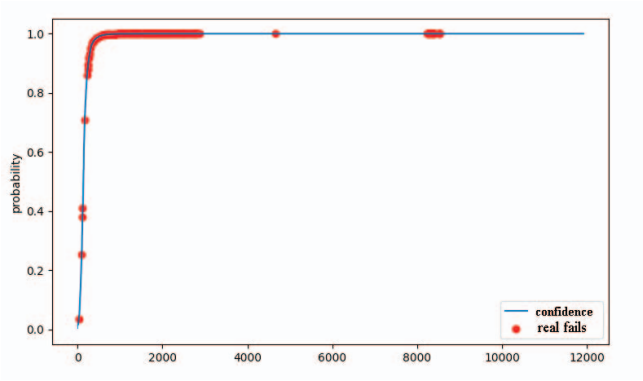

Fig. 3. Confidence curve for each test case of one build of PERC regression sorted from smallest to largest, and the red dots are the real failures positions in that build.

C. Sigmoid saturation problem

After TCP-Net model deployment [1] was done and running, a strange performance decay behavior was detected in multiple builds. It was found that most of the failures are located at the very beginning of the regression after prioritization, but some failures escaped to nearly the end of the regression. Figure 3 shows the confidence curve which is the pass probability generated by the model for each test case of one build of PERC regression sorted from smallest to largest. A smaller value means a higher probability of failing. The red dots are the real failures positions in that build. We can see that most of the failures are located at the beginning, as mentioned before, but some test cases located nearly at 80% of the regression.

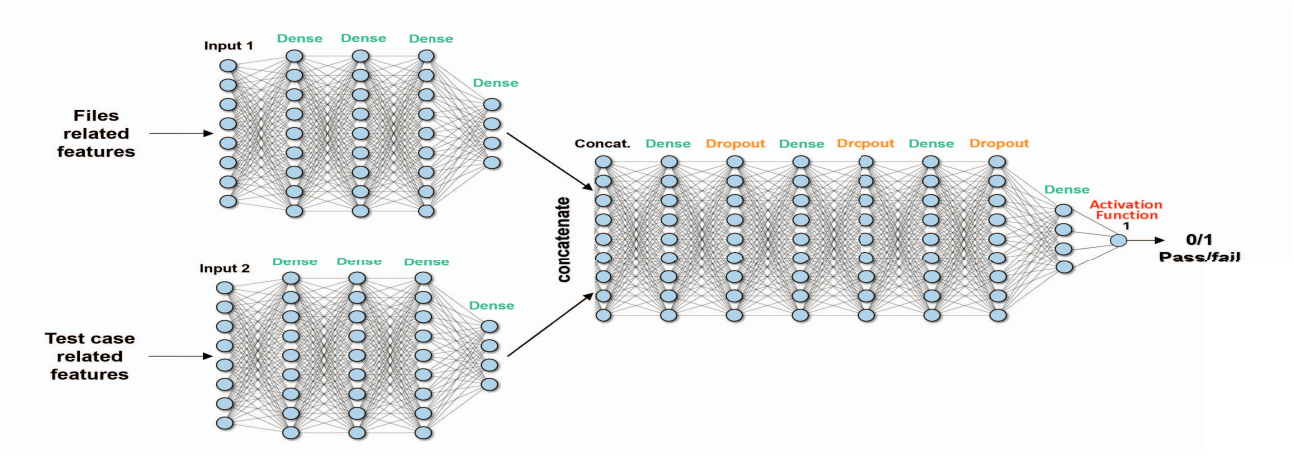

Fig. 4. **TCP-Net Architecture**: source-files-related features and test-case-related features are fed to the two branches of the middle fusion deep fully connected network architecture. Each branch act as a feature extractor to its input features only, then the extracted features are compressed till reaching the final layer in each branch, then the extracted features are concatenated then fed to the main feature extractor till reaching the final sigmoid classifier layer.

After conducting some analysis, it was found that the cause of the observed behavior was sigmoid saturation problem [27]. The sigmoid function is a frequently used activation function in the output layer of neural networks. It maps any input value to a value between 0 and 1, making it useful for binary classification and other problems. However, the sigmoid function suffers from a problem known as sigmoid saturation. This problem occurs when the input to the sigmoid function is very large or very small, causing the function to output values very close to 0 or 1.

In our specific problem, the effect of sigmoid saturation is magnified because a large portion of the test cases pass, with confidence values of one, since the pass class represents more than 98% of the dataset. As a result, when sorting test cases based on their probability of failure, the ones are sorted randomly since they all have the same saturated value. To resolve this issue, various solutions were tested and evaluated. The two best solutions were to use the ReLU (Rectified Linear Unit) activation function instead of the sigmoid in the last layer, removing the space compression between 0 and 1 and making it free to output any value between 0 and infinity. The second solution was to use an idea similar to soft targets. In machine learning, soft targets (also known as soft labels or label smoothing) refer to a technique for training models in which probabilities or distributions over labels are used as training targets instead of hard labels (binary labels with values of 0 or 1). Soft targets can enhance a model's generalization ability by introducing more information during the training process. This is accomplished by smoothing the distribution of labels that the model will learn, which helps the model learn a more robust and generalized decision boundary. For our specific case, we will not train the model on 0's and 1's. Instead, we will train it on 0's for failures and 0.5 instead of 1 for pass cases. The goal is to be located far from the 1 boundaries of the sigmoid function to prevent it from saturating so that test cases can be sorted correctly.

D. Ensemble learning

Ensemble learning [28] is a powerful technique in machine learning that combines the predictions of multiple models to improve the overall accuracy and robustness of predictions. It is based on the principle that a group of diverse models can make more accurate predictions than a single model alone. In ensemble learning, multiple models are trained on the same data using different algorithms or parameters, and their predictions are combined using various methods such as averaging, voting, or stacking. Ensemble learning has been successfully applied to a wide range of scientific and engineering problems, including classification, regression, and anomaly detection. By equipping TCP-Net++ with ensemble learning, the accuracy and stability of the model was improved over time, as discussed in detail in the results section. Ensemble learning was implemented by running the two best models discussed previously and averaging their outputs. To average and combine the output of the two models, their output must have the same mean and variance. The sigmoid model output values are approximately between 0 and 0.5, and the ReLU model outputs values between 0 and infinity, unbounded. To address this, we normalized both models by sorting the test cases with the output of each model, replacing the models' output with the test case index in the regression from 0 to N, and then dividing by N to generate a normalized probability from 0 to 1. Afterward, we averaged the probabilities generated from each model to get the final fail probability for each test case

E. Evaluation metrics and graphs

As a result of replacing the normal sigmoid activation function with an unbounded ensemble learning model, TCP-Net is no longer a normal classification problem. Therefore, using metrics such as accuracy, F1-score, precision, and recall,

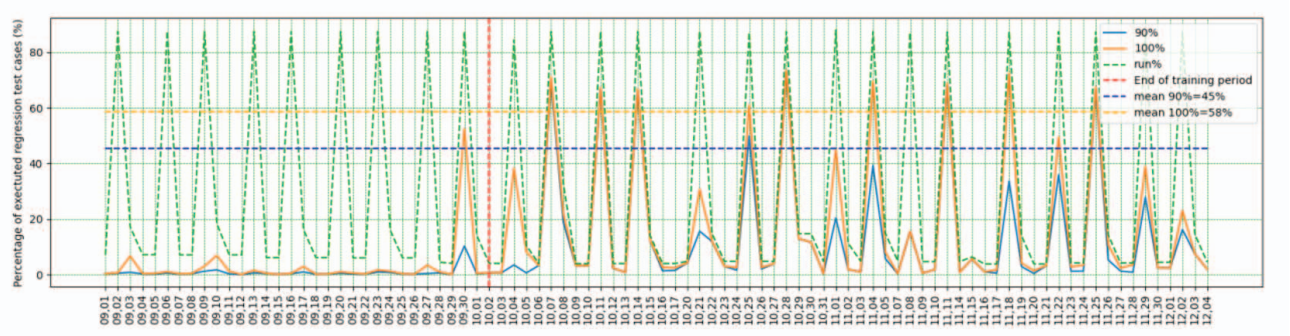

Fig. 5. An example of Coverage curve for three months-period.

as in [1], will not be appropriate and will not generate accurate performance measurements. The only metric that can be used in this case is the Average Percentage of Faults Detected (APFD) metric [26]. The objective of using the APFD metric is to quantify the efficiency of coverage-based prioritization techniques in detecting faults. APFD computes a weighted mean of the proportion of detected faults throughout the entire duration of a test suite, with values varying between 0 and 100. Higher APFD scores signify faster detection of faults, as demonstrated in Equation 1.

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

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

During testing TCP-Net after deployment, the aforementioned metrics were not sufficient to assess the model's performance on real-world data and accurately determine the performance boundaries to decide whether to rely solely on the model for regression testing. So, another performance metric, which is coverage graphs, was introduced. A coverage graph is a graph that represents how quickly TCP-Net model identifies the faults of the given test suite when running it through each build over a certain period of time. Figure 5 shows an example of the coverage curve for a three-month period. The first month is a part of the training set until October 2nd, and the last two months are a totally unseen period to test the model's performance in real-life on unseen data. The x-axis contains the days, and the y-axis contains the percentage of the executed regression test cases from 0 to 100%. The orange curve indicates at which percentage of regression running all the failures in that build were found, and the blue one indicates where 90% of the failures were found. The green dashed curve indicates the percentage of the full regression executed in each build. For example, on November 18nd, in the build, the tool caught 90% of the failures by running about 32% of the regression and caught 100% of the failures by running 70% of the regression on our example. The average of the 90% coverage over the two unseen months is 45%, indicated by the horizontal blue line, and the 100% coverage is at 49%, indicated by the horizontal orange line.

IV. Experimentation

In this section, we discuss TCP-Net training process details.

A. Model training

In the Dataset section, it was previously mentioned that two CSV files contain two distinct types of input features: one file stores test-case features, while the other file stores source-files features. In order to train the model, a custom batch loader randomly selects and loads a set of p test-case features and p source-files features, along with their corresponding pass/fail label (which is determined by the test-case run with the source-files changes feature), and saves this information in memory. This data is then fed to the model for one training iteration, with p representing the batch size. TCP-Net is trained to classify each feature pair as a pass or fail using different targets for each model as discussed before using specific hyper-parameters, as described below:

- Epochs: 250
- Batch size: 1024
- Optimizer: Adam (lr=1e-4)
- Loss function: Binary cross entropy
- Dataset splitting: 80% training, 20% testing
- Early stopping technique is used to prevent overfitting

V. Results analysis

To continue analyzing the discussion started in the methodology section, we have Figures 6 A-C, which represent the confidence curves of three cases for a certain build for the PERC dataset which is used as an example to illustrate the effect of each technique on the accuracy of TCP-Net model. These figures illustrate the output of each one of our two best models and the effect of using ensemble learning techniques utilized in TCP-Net++. Figure 6-A represents the confidence curve of the previously discussed sigmoid model, Figure 6-B represents the confidence curve

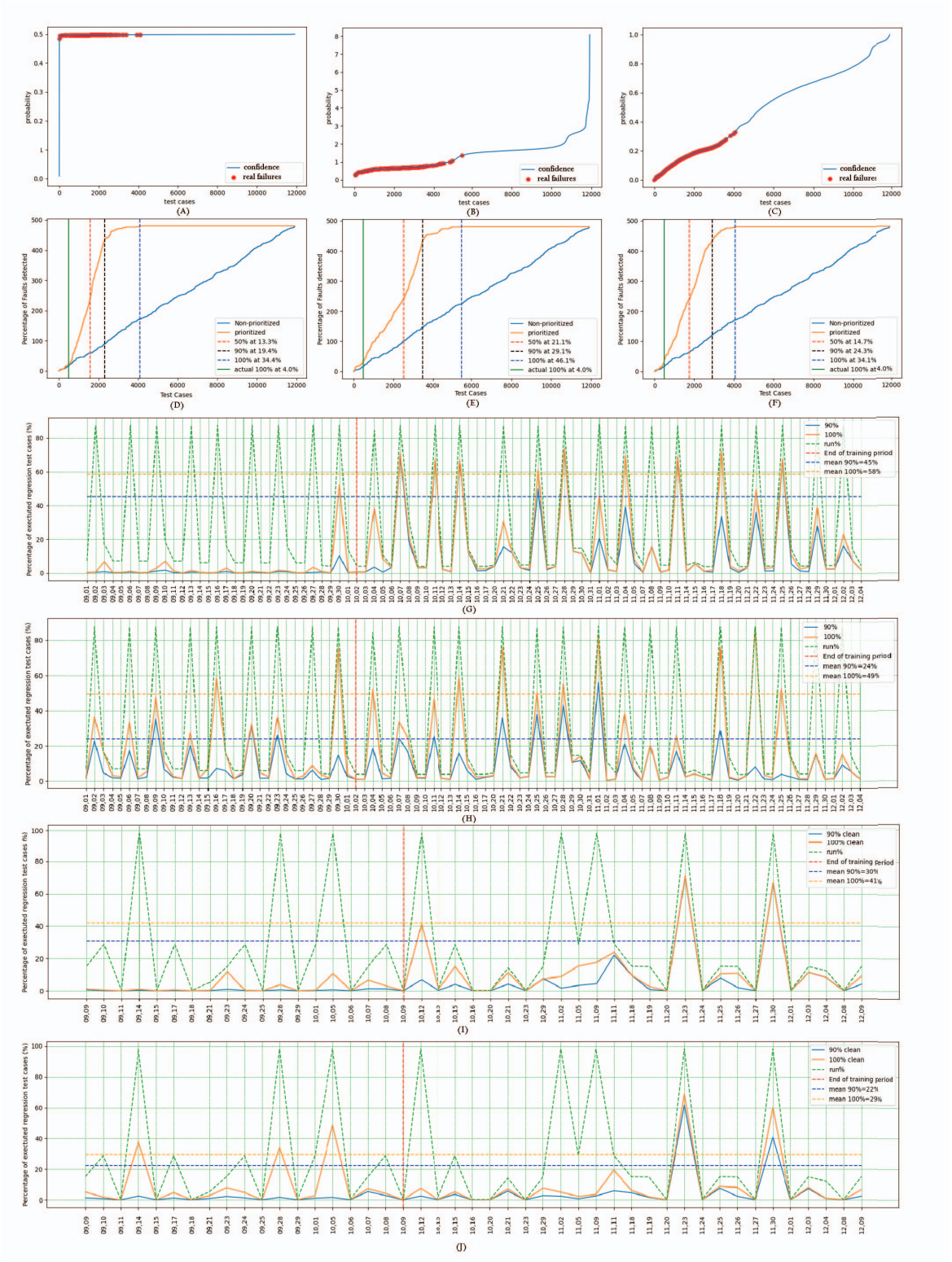

Fig. 6. From the top: the first row Figures A-C represents the confidence curves of the three cases (sigmoid model, ReLU model, and Ensemble learning model) for a specific build for the PERC dataset. The second row, Figures D-F, represents the APFD curves of the three cases. The other four curves, Figures G-J, indicate the TCP-Net/TCP-Net++ failure coverage performance over a three-month period for the Calibre PERC and Calibre xRC datasets. G,H for PERC, I,J for xRC.

of the ReLU model, and the last one, Figure 6-C, represents the confidence curve using our ensemble learning technique.

At first glance, the curve in Figure 6-A looks like the saturated curve of the old TCP-Net model. However, upon closer examination of the confidences of the different test cases, we found that saturation did not occur in this case. This is because we decreased the target values from 1 to 0.5, which moved them away from the saturation boundary of the sigmoid function. The curve reaches its asymptotic limit at 0.5 because the network tries to get the target right, which is 0.5 for pass cases. By using the 0.5 target, we are utilizing only the first half of the sigmoid activation function, but it is not prohibited for the network to output values beyond 0.5. We found that a small number of the latest test cases in the sorted curve exceeded the 0.5 limit by a very small value, nearly in the order of 1/1000. This indicates that the approach successfully solved the sigmoid saturation problem and the test cases are sorted normally.

The other technique used to solve the sigmoid saturation problem was to replace the sigmoid activation function with a ReLU activation function. This way, we are eliminating the upper asymptotic limit of the sigmoid function, making it possible for the model to output values freely from 0 to infinity. Figure 6-B shows the confidence curve of the ReLU model run on the same build. We can observe a different behavior in this case, where the confidences are distributed from 0 to around 8 due to using the boundless activation function. This behavior can be a problem in a normal classification problem, as there is no specific threshold that can determine whether each example is a pass or fail. However, this is not a problem in our case because it is a ranking problem, and we only care about the relative value of each test case confidence level to be compared with the rest and sorted accordingly.

Each one of the two aforementioned techniques successfully solved the saturation problem, furthermore, Combining both techniques, we get a performance that is better than either one alone. Figure 6-C shows the confidence curve of the ensemble learning technique discussed in Section III-D. We can see that the confidence curve is very close to a linear curve, and this behavior is a result of normalizing the confidences of each model and then averaging them.

To better compare the performance of the three techniques, Figures 6 D-F represent the APFD curves of the three cases. Shown in each of the three figures is a comparison between the APFD of the trained model and the APFD of random regression running for the same build. The x-axis represents the percentage of executed test cases, and the y-axis represents the percentage of faults detected by running test cases. The blue line in the figure represents the random case, and the orange one represents the prioritized case. We can see that the blue line is more like a linear relationship, which means that as we run more test cases, more faults will be detected until running 100% of the regression test suite. On the other hand, the orange curve performs a very high jump at the beginning of the test suite, indicating that the first few test cases prioritized by TCP-Net reveal most of the failures in the whole suite. Therefore, running only the first few test cases will reveal most of the faults contained in the whole test suite. For example, in Figure 6-D, 50% of the failures are detected by running only 13.3% of the full regression indicated by the vertical red dotted line, 90% of the failures are detected by running only 19.4% of the full regression indicated by the vertical black dotted line, and 100% of the failures are detected by running only 34.4% of the full regression indicated by the vertical blue dotted line. Comparing the three APFD graphs for the three techniques, we notice that for the 50% failure detection for this build, the sigmoid model 6 D is better than the ReLU one 6 E, and the ensemble learning 6 F performance is better than the average of both of them. The same can be noticed for the 90% and the 100% cases as well.

TABLE II
TCP-NET/TCP-NET++ COVERAGE PERFORMANCE COMPARISON

Data set	**TCP-Net**		**TCP-Net++**	
	90% coverage	100% coverage	90% coverage	100% coverage
Calibre PERC	45%	58%	24%	49%
Calibre xRC	30%	41%	22%	29%

To measure the performance over a multiple-day period, not just using one build, coverage curves are implemented, as discussed before. Figures 6-G and 6-H indicate TCP-Net performance over a three-month period for the baseline TCP-Net model in Figure 6-G, and for the improved ensemble learning model, TCP-Net++, in Figure 6-H. The first month is a part of the training set until October 2nd, and the last two months are a totally unseen period to test the model's performance in real life on unseen data. The Coverage Curve is simply the 90% and 100% labels in the APFD curves measured over multiple builds to study the performance of TCP-Net++ over a relatively large period of time to be as confident as possible. We can see that the performance in Figure 6-H is better than Figure 6-G for the period after training. Generally, the peaks in the improved ensemble model are lower than the ones in the first curve. To better measure it, the means of the 90% and 100% coverage of the period after the last training day are listed in Table II. We can see that TCP-Net++'s performance for PERC is better than TCP-Net for the 90% coverage by nearly 20%, and for the 100% coverage case, it is better by 10%. For the xRC product, it is better than TCP-Net for the 90% coverage by 8%, and for the 100% coverage case, it is better by 12%, which is a very good performance gain for TCP-Net++ over TCP-Net.

VI. Conclusion & future work

In this study, we demonstrated the successful application of TCP-Net++, an end-to-end deep neural network, for test case prioritization in a real-life industrial environment. By using machine learning techniques, we were able to overcome the bottleneck in regression testing caused by the increasing number of test cases and the frequency of continuous integration. The results of the deployment analysis showed that our model could achieve 90% to 100% failure coverage by running an average of 23% to 39% of the test cases, which saves time and hardware resources. However, we encountered some roadblocks and challenges during the deployment, and several enhancements were made to improve its performance and increase its usability. Our future work aims at improving the performance and usability of TCP-Net++. Future work will focus on investigating the impact of different hyperparameters and network architectures on its performance. Additionally, evaluating TCP-Net++ in a wider range of industrial environments and software projects with different characteristics could provide insights into its generalization capabilities and scalability.

References

[1] M. Abdelkarim and R. ElAdawi, "Tcp-net: Test case prioritization using end-to-end deep neural networks," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2022, pp. 122–129.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

[10] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] T. Çıngıl and H. Sözer, "Black-box test case selection by relating code changes with previously fixed defects," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, 2022, pp. 30–39.

[21] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, 2022.

[22] S. Omri and C. Sinz, "Learning to rank for test case prioritization," in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2022, pp. 16–24.

[23] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino, "Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 115–125.

[24] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[25] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 222–232.

[26] 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.

[27] "article about sigmoid activation function saturation," Available at https://kharshit.github.io/blog/2018/04/20/don't-use-sigmoid-neural-nets: :text=Sigmoid%20saturate%20and%20kill%20gradients,gradient%20of%20this%2

[28] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma, "A survey on ensemble learning," *Frontiers of Computer Science*, vol. 14, pp. 241–258, 2020.