Machine Learning-based Run-Time Anomaly Detection for Software Systems

Alexander Heckmann
City, University of London
alexander.heckmann@city.ac.uk
London, United Kingdom

Abstract

Software systems are ubiquitous in our everyday life. Deploying and managing software systems has become an increasingly hard job, especially in making sure that these systems are reliable and secure. This research paper explores the problem of multivariate time series classification for operational data from a real-world software system. The study compares the performance of two models, RNN with LSTM and MLP, using limited windows of training data and multiple hyperparameter tuning. Holdout validation is employed to test the accuracy of the models. The results demonstrate that RNN with LSTM significantly outperforms MLP in terms of validation accuracy and, depending on the data sequencing approach for MLP, even training time. Consequently, the study concludes that RNN with LSTM is the superior model choice for this multivariate time series classification problem.

1. Introduction

Software systems are the backbone of modern society, providing the infrastructure for businesses, organizations, and individuals to operate efficiently. However, the likelihood of encountering anomalies increases as these systems become more complex. These anomalies can have serious consequences, such as system downtime, data corruption, and security breaches. Detecting and mitigating these anomalies is therefore critical for ensuring the reliability, security, and performance of software systems and an enjoyable user experience. Traditional approaches include a combination of fixed rules like thresholds, probing requests, and system-specific health checks. However, this manual approach is insufficient and time-consuming for DevOps environments [1], [2].

In recent years, machine learning techniques have shown promising results in detecting anomalies in software systems [2].

Problem. Insufficient methods for run-time anomaly detection in software systems lead to service failure and risk business continuity. How can deep learning be leveraged in order to prevent any negative business impact due to anomalies?

Solution. We compare the performance of existing solutions such as recurrent neural networks (RNN) with long short-term memory (LSTM) against MLP for run-time anomaly detection. Our hypothesis is that the RNN with LSTM is the superior model choice for multivariate time series classification.

Contribution. We demonstrate our methodology on a complex real-world enterprise application, and discuss the results of our comparison of LSTM and MLP and their applicability as well as provide guidance on which model is most suitable for detecting anomalies in software systems. For training and evaluation, we use a dataset containing operational data collected from a real-world enterprise system [2]; the data was provided by Huch *et al.* [3].

Overall, the findings of this study will be significant for software engineers and developers who seek to improve the reliability and performance of their software systems. By identifying the most effective machine learning models for detecting run-time anomalies, this research can help to mitigate the risks associated with anomalies, ultimately contributing to the development of more reliable and secure software systems.

Section 2 gives a brief overview of the dataset used for conducting the experiments. Section 3 compares the approaches and methods used during building the models. Section 4 critically evaluates the results and compares the models. Finally, Section 5 concludes the paper.

1.1. LSTM

RNNs are designed to process sequential data and maintain a hidden state that updates at each time step based on the input and previous hidden state. However, traditional RNNs suffer from the vanishing gradient problem when processing long sequences.

LSTM cells were introduced to address this problem. LSTMs have three gating mechanisms (input, forget, and output gates) that control the flow of information into and out of the cell. The gates are controlled by sigmoid activation functions that take in the current input and the previous hidden state, allowing the network to selectively remember or forget information over long sequences. Outputs are activated using the tanh function.

RNNs with LSTM are therefore better suited for capturing temporal dependencies since they maintain hidden state and can handle arbitrary time series data. Downsides are their limited interpretability and the training costs due to their additional complexity.

1.2. Multi-Layer Perceptrons

MLPs are a type of feedforward neural network that consist of multiple layers of perceptrons (nodes) that process input data in a forward direction. To train MLPs for sequential data, one needs to apply windowing on the dataset [4]. Windowing divides the input data into windows of given size, which then network is trained on to predict the output for each window based on the input data within the window. Sliding windows refer to a technique where the input data is divided into overlapping windows of fixed size; tumbling windows are a special case of sliding windows where the step size equals the window size and therefore creates non-overlapping windows. Increasing windows, on the other hand, don't have a fixed size, but instead, grow over time as more data becomes available. The use of windows allows the network to capture local patterns in the input data that are relevant for predicting the output, as well as to maintain some temporal context and its auto-correlation across the sequence. Each window is treated as a separate input sample for the network, and the network parameters are learned through backpropagation.

MLPs shine due to their simplicity compared to RNN with LSTM. This makes them computationally cheaper to train, with fewer hyperparameters to search and their clear input-output mapping. On the other hand, they have limited ability to capture temporal relations in sequential data.

2. Data

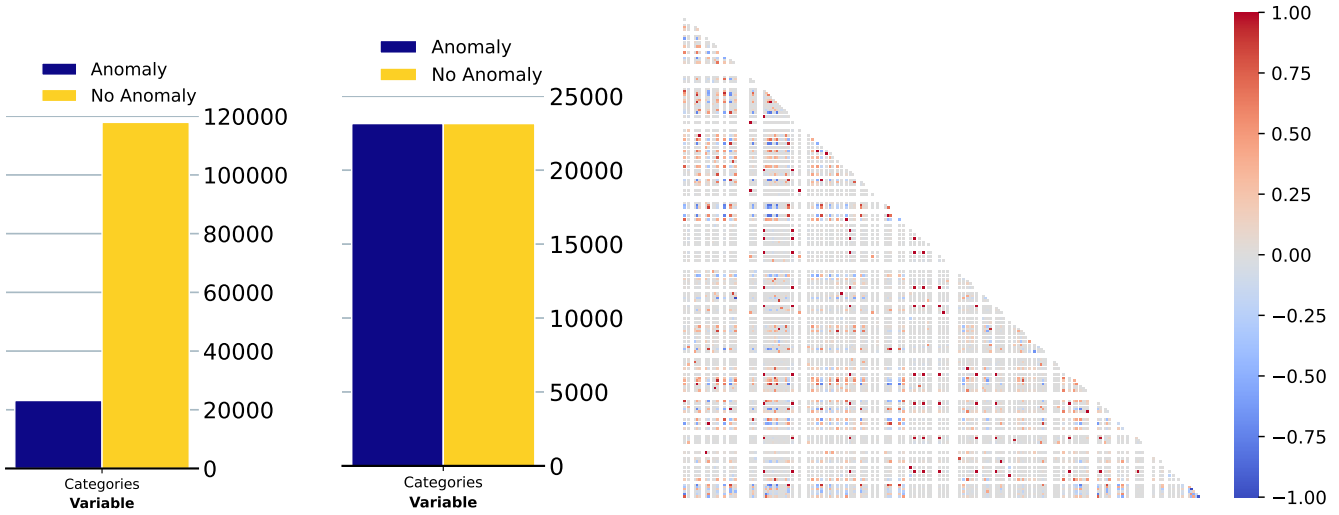
In the following, we describe the characteristics of the dataset and our data preparation steps.

To build a dynamic anomaly detection system using deep learning, we use real-world operational data of an enterprise software system, preprocessed and provided by Huch *et al.* [3]. The dataset used in this study includes various operational metrics which can be summarized as memory usage, CPU usage, swap usage, database connection and transaction details, and garbage collection details [3]. For this study, only about 5% or 188068 rows of the original dataset were used. Overall, the dataset contains 235 columns, including the target variable. The target variable was created by Huch *et al.* based on system activity preceding a node restart [2].

The following steps were taken to preprocess the data for our experiments.

1) *Missing values.* Although the used dataset does not contain missing values, it is important to note that the original, unprocessed dataset did in fact have missing values. Huch *et al.* replaced missing values using linear interpolation to retain the information and allow machine learning algorithms to work with the data [2]. The *No data* column resembles this situation, therefore we remove this artificial column.
2) *Label encoding.* To enable the models to work with the data, we label encoded the target variable.
3) *Encoding categorical variables.* We transformed the categorical variables, i.e. the *host* and the *process* columns by using one-hot encoding for an integer representation.
4) *Handling time.* To work with the timestamps, we converted the datetime into UNIX timestamps that can be represented by integers. We then sorted the data by time.

5) *Holdout validation split.* Holdout validation was used for our experiments. We used an 80-10-10 ratio for splitting the data into a training, validation, and test set.
6) *Highly correlated values.* The dataset contains quite a few correlated features (see fig. 1c). We removed features with high correlation and high anti-correlation to further increase model performance. For our experiments, we took 0.95 as the removal threshold.
7) *Constant values.* We identified columns that contain just constant values, i.e. columns that have one as the number of unique values and removed them to increase model performance.
8) *Rebalancing the dataset.* In our case, only 16.4% of the target variables belong to the minority class (see figure 1a).



(a) Before oversampling (b) After oversampling (c) Correlation heatmap

Figure 1: Initial data analysis

Imbalance of the target variable is an inherent problem in anomaly detection, due to the low likelihood of occurrences of the minority class. Rebalancing the training set has shown to result in better performance of classifiers [5]. For our studies, we used random undersampling [6] to rebalance the training set.

	Memory space usage ...	Active connections : ...	Heap usage activity ...	Connection delay : ...	Stuck threads ...	DB connection started ...
count	188068.000000	188068.0	1.880680e+05	188068.000000	188068.000000	188068.000000
mean	0.701453	0.0	1.380974e+05	76.764394	0.181716	0.148941
std	0.333852	0.0	6.684241e+08	155.551443	0.433309	0.964170
min	0.053711	0.0	-9.546473e+09	0.000000	0.000000	0.000000
max	0.980469	0.0	4.953991e+09	509.000000	3.000000	32.000000

Figure 2: Difference in scale of numerical features (excerpt)

9) *Scaling.* Numerical features in the dataset differ in their value range (see fig. 2). As scaling the dataset tends to increase model performance [7], we normalized all numerical features f_i so that $f_i = \frac{f_i - \mu}{\sigma}$ where μ is the mean and σ is the standard deviation. We scaled the training set, validation set, and test set separately. Further processing was then applied for model specifications.

3. Methods

In the following section, we describe our methodology for conducting the experiments, along with a description of the architecture and the selection of hyperparameters used to construct both the RNN and MLP models.

For both RNN and MLP we kept the number of epochs at 10 throughout all experiments. The volume of training data makes training computationally expensive, therefore we aim for model parameters that result in high convergence.

The hyperparameter selection was conducted using a stepwise gridsearch, meaning that small experiments were conducted and the best output was used for the next experiments. This drastically reduces this otherwise huge combinatorial problem into a smaller area of the searched space.

As mentioned in sec. 2, we split the data into an 80% training set, a 10% validation set for testing hyperparameter tuning performance, and a 10% test set for final comparison of the best models. Both models use the same data, although in different forms (see the following sections). This allows a fair comparison of both algorithms. The holdout validation allows testing the data on unseen data.

3.1. RNN model choices

The data was further processed after the steps mentioned in sec. 2. We transformed the data into sequences of 30, meaning that the model gets 30-minute intervals of operational data. We then further batched the data to work with LSTM into batches of 128.

The model consists of 1, in later experiments 2, layers of LSTM cells connected to a linear output layer. We chose stochastic gradient descent with RMSProp as the optimizer like Huch *et al.* did, and cross-entropy as the loss function.

For tuning our RNN model, we decided to experiment with the following hyperparameters for LSTM: *learning rate, weight decay, hidden layer size*, and *number of LSTM cell layers*. We didn't experiment with momentum since we used gradient stochatic descent with RMSProp, which already has an adaptive learning rate; therefore, momentum is not used.

3.2. MLP model choices

For our experiments with MLP, we differentiated between tumbling, sliding, and increasing windows to see the effect data availability has on model performance. For this, we first processed the data into windows of 30 elements. The step size for the sliding window approach and the increasing window approach is 15, meaning the boundary shifts by 15 elements after each iteration.

The model follows the following structure: the input layer, tanh activation, another layer, tanh function, the output layer and softmax activation. We chose stochastic gradient descent as the optimizer, and cross-entropy as the loss function.

For tuning our MLP model, we decided to experiment with the following hyperparameters for MLP: *learning rate, weight decay, momentum*, and *hidden layer sizes*.

4. Findings

The results of tuning RNN with LSTM revealed several key findings. Firstly, reducing the number of layers from 2 to 1 led to a significant reduction in training time, but also caused a slight decrease in accuracy. Secondly, incorporating a dropout rate resulted in an improvement in model performance. Thirdly, introducing weight decay caused a drastic decrease in model performance. Finally, the best RNN model was achieved with 2 layers, a hidden size of 100, a learning rate of 0.0025, no weight decay or L2 regularization, no momentum, and a dropout rate of 0.4. These findings suggest that careful tuning of hyperparameters can lead to significant improvements in RNN model performance for multivariate time series classification tasks. Validation accuracy was between 0.65 and 0.9 for most of the experiments, showing promising results, while the training cost was 15 seconds to 40 seconds.

The results of the MLP experiment revealed that the performance of the MLP models varied significantly depending on the windowing strategy used. Sliding windows consistently yielded the worst results, while

tumbling windows were marginally better. Although increasing windows produced the most promising results, they required significantly greater computational efforts, making them more expensive to use. Despite this, even the best-performing MLP model with increasing windows had significantly lower accuracy than the RNN/LSTM models. The optimal MLP model had a hidden size of (50, 100, 100), a learning rate of 0.003, no weight decay/L2 regularization, and no momentum. Results were inconsistent though, since MLP weights and biases are randomly initialized, making them prone to fluctuation in performance and local minima. MLP validation accuracy consistently was between 0.60 and 0.75, showing its limited ability to capture temporal relationships. Training time using GPU, on the other hand, was between 15 seconds and 120 seconds, depending on the windowing strategy.

There are several factors that may explain the results observed in this study.

Firstly, it is well-established that RNN with LSTM is particularly well-suited for handling sequential data, such as time series data. This is due to the memory cells in the LSTM layer that can selectively ”remember” important information from previous time steps, allowing the model to capture temporal dependencies in the data. In contrast, MLPs are not inherently designed to handle sequential data and may struggle to capture the complex patterns in multivariate time series data.

Regarding the MLP results, the fact that the performance varied significantly depending on the windowing strategy used highlights the importance of careful preprocessing and feature engineering in machine learning tasks. It is possible that the sliding window strategy may have introduced noise or irrelevant data into the model, leading to poorer performance. It is also worth noting that MLPs are typically less well-suited to handle large amounts of data than RNNs, which may explain why the best-performing MLP model had lower accuracy than the LSTM models. Finally, the fact that MLP weights and biases are randomly initialized may explain the inconsistent results observed in this study, as the model may have converged to different local minima depending on the initialization.

5. Conclusion, Reflections & Further Work

In our experiments, we assess the performance of two trained models, a Multilayer Perceptron and a Recurrent Neural Network, in accurately classifying whether an unseen measurement of operational data is an anomaly that needs to be inspected or not. The conclusion is that RNN greatly outperforms MLP in terms of validation accuracy, and depending on the data sequencing approach for MLP even the training time. Therefore, RNN with LSTM showed to be the better model choice for this multivariate time series classification problem.

During the experiments, we learned about the importance of optimizing algorithms and hardware usage. To handle the computational challenges of working with large datasets, we used a generator function to load data in chunks instead of loading the entire dataset into memory at once. This reduced the memory requirement and allowed us to work with a large dataset. We also optimized program runtime by running the model on the GPU using PyTorch’s CUDA support [8]. Overall, this project was very challenging given the time constraints; reducing the problem would’ve resulted in much higher degree of research quality.

Additionally, the sliding window and the increasing window approach for MLP drastically increases its algorithmic complexity due to the increase of possible combinations within windows, diminishing its usual advantage in terms of training costs compared to RNN. On the other hand, especially the increasing window approach resulted in higher model performance. Therefore, it needs to be carefully considered which approach to use; for large datasets it’s definitely a trade-off between model performance vs training costs.

Further research can explore various options for reducing the dataset, including dimensionality reduction techniques such as principal component analysis, and improved feature selection methods such as ranking feature importance of predictors. Tuning the dataset instead of the models could be an important step for increased model performance and decreased training costs. To further increase LSTM performance for multivariate time series classification, it could be researched how ensembling LSTM with other methods such as autoencoders performs, ultimately resulting in highly accurate, dynamic anomaly detection for reliable and secure software systems.

References

[1] A. H. Fawzy, K. Wassif and H. Moussa, "Framework for automatic detection of anomalies in devops", *Journal of King Saud University-Computer and Information Sciences*, vol. 35, no. 3, pp. 8–19, 2023.

[2] F. Huch, M. Golagha, A. Petrovska and A. Krauss, "Machine learning-based run-time anomaly detection in software systems: An industrial evaluation", in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, 2018, pp. 13–18. DOI: 10.1109/MALTESQUE.2018.8368453.

[3] F. Huch, M. Golagha, A. Petrovska and A. Krauss. "Software operational data, processed and labeled". (19 Jan. 2018), [Online]. Available: https://www.kaggle.com/datasets/anomalydetectionml/features (visited on 20/03/2023).

[4] H. Tawfeig, V. S. Asirvadam and N. Saad, "Sliding-window learning using mlp networks with data store management", in *2011 National Postgraduate Conference*, 2011, pp. 1–5. DOI: 10.1109/NatPC.2011.6136391.

[5] G. M. Weiss and F. Provost, "The effect of class distribution on classifier learning: An empirical study", Rutgers University, Tech. Rep., 2001.

[6] R. Zuech, J. Hancock and T. M. Khoshgoftaar, "Detecting web attacks using random undersampling and ensemble learners", *Journal of Big Data*, vol. 8, no. 1, pp. 1–20, 2021.

[7] W. Wang, X. Zhang, S. Gombault and S. J. Knapskog, "Attribute normalization in network intrusion detection", in *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, IEEE, 2009, pp. 448–453.

[8] T. P. Foundation. "Torch.cuda". (), [Online]. Available: https://pytorch.org/docs/stable/cuda.html (visited on 06/04/2023).