CHALMERS
UNIVERSITY OF TECHNOLOGY

Online encoder-decoder anomaly detection using encoder-decoder architecture with novel self-configuring neural networks & pure linear

Online Encoder-decoder Anomaly Detection using Encoder-decoder Architecture with Novel Self-configuring Neural Networks & Pure Linear Genetic Programming for Embedded Systems

Gabrielė Kasparavičiūtė[1,2], Malin Thelin[3], Peter Nordin[1,3], Per Söderstam[3], Christian Magnusson[3] and Mattias Almljung[3]

[1]*Chalmers University of Technology, Chalmersplatsen 4, Gothenburg, Sweden*

[2]*University of Gothenburg, Rännvägen 6B, Gothenburg, Sweden*

[3]*Semcon AB, Lindholmsallén 2, Gothenburg, Sweden*

Keywords: Encoder-decoder, Anomaly Detection, Linear Genetic Programming, Evolutionary Algorithm, Genetic Algorithm, Embedded, Self-configuring, Neural Network.

Abstract: Recent anomaly detection techniques focus on the use of neural networks and an encoder-decoder architecture. However, these techniques lead to trade offs if implemented in an embedded environment such as high heat management, power consumption and hardware costs. This paper presents two related new methods for anomaly detection within data sets gathered from an autonomous mini-vehicle with a CAN bus. The first method which to the best of our knowledge is the first use of encoder-decoder architecture for anomaly detection using linear genetic programming (LGP). Second method uses self-configuring neural network that is created using evolutionary algorithm paradigm learning both architecture and weights suitable for embedded systems. Both approaches have the following advantages: it is inexpensive regarding resource use, can be run on almost any embedded board due to linear register machine advantages in computation. The proposed methods are also faster by at least one order of magnitude, and it includes both inference and complete training.

1 INTRODUCTION

The complexity of modern embedded systems such as vehicle systems have increased over the past decade, in part due to the increasing amount of sensors (Hegde et al., 2011). As a consequence, computational effort of the control network of vehicle systems has become a problem, with implications on battery management, weight, price etc. Furthermore it has become difficult to be able to keep track of the total number of system errors or anomalies, which tend to accumulate with additional complexity. Thus, there is a risk that an embedded system such as a vehicle deviates from its expected values without alarms being raised. Furthermore, current methods of anomaly detection are both time and power consuming processes and/or restricted in the anomalies they can detect.

Unexpected behaviours are often interchangeably called errors, anomalies, outliers, or exceptions. Anomaly detection is the act of detecting a deviation from the anticipated value; many times referred to as the data mean behaviour (Chandola et al., 2009). It is used in different fields (and with slightly different meanings) in, e.g., network intrusion detection (Taylor et al., 2015), motor abnormalities in unmanned vehicles (Lu et al., 2018), fraud detection (Ahmed et al., 2016), medical anomaly detection (Pachauri and Sharma, 2015), and the controller area network bus interference (Hangal and Lam, 2002). It is important to clarify that none of these examples use an encoder-decoder architecture.

There is a wide variety of techniques to discover anomalies. Some of these techniques have been developed for certain domains. Lately a technique with a wide applicability has become popular: deep neural networks (Sabokrou et al., 2017). More specifically an encoder-decoder anomaly detection system (EncDecAD) using deep neural networks. However, there are some major issues with the use of this technique. For example, within the automotive industry,

Kasparavičiūtė, G., Thelin, M., Nordin, P., Söderstam, P., Magnusson, C. and Almljung, M.
Online Encoder-decoder Anomaly Detection using Encoder-decoder Architecture with Novel Self-configuring Neural Networks Pure Linear Genetic Programming for Embedded Systems.
DOI: 10.5220/0008064401630171
In *Proceedings of the 11th International Joint Conference on Computational Intelligence (IJCCI 2019)*, pages 163-171
ISBN: 978-989-758-384-1

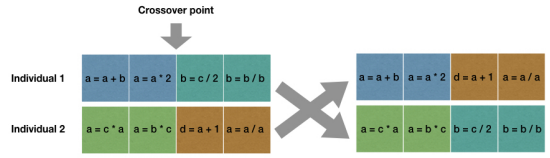

Figure 1: Crossover example in linear genetic programming.

to be able to have a capable deep neural network architecture in a vehicle requires high power consumption and thus, creates a heating, price and efficiency issue.

As a result, this paper introduces the first encoder-decoder anomaly detection system (EncDecAD) using linear genetic programming (LGP) where the trained model is run on an embedded device. Additionally, this paper showcases an embryo of evolutionary technique in self-configuring neural networks and simultaneous training as a variant of the first linear genetic programming method. Vehicle data for both experiments was obtained from an autonomous mini-vehicle that was run on different tracks; the first time the mini-vehicle had no abnormalities, the second time the suspension was loosened on the vehicle to create an anomaly in the data; the third time the vehicle had a resistor soldered in series with the battery of the vehicle. The results show promising potential for the presented techniques.

2 RELATED WORK

2.1 Linear Genetic Programming

Linear genetic programming (LGP) is an artificial intelligence technique that uses a sequence of simple instructions to solve a problem (Banzhaf et al., 1998). The problem with this solution is finding the best function that represents the given data. The ability to find the correct function to solve the problem depends on the expressiveness of the instructions and the evolutionary methods. LGP has a wide application domain, e.g., it has been used in prediction of hydrological processes (Mehr et al., 2013), software optimization (Langdon and Harman, 2015), feature learning for image classification (Shao et al., 2014), and miniature robot behaviours(Nordin and Banzhaf, 1995).

LGP is built on Darwin's principle of natural selection and evolution (Banzhaf et al., 1998). A population is created with random individuals that compete against each other by applying a chosen fitness function. It has been an established practice to use a standard fitness function. This means that the closer the fitness value is to zero, the better an individual scores among the population. The evolutionary concepts in LGP include selection, mutation, and crossover, and are defined as follows:

Representation: LGP utilizes an imperative programming concept which is composed of a set of registers and basic instructions. Execution of the individual respects the order of instructions. The basic example of such an instruction is $a = b + c$, where a, b, and c are all variables. In this case, a is the register that is being written to, b and c are operands that are separated by an operator +. In genetic algorithm terms, one instruction is called a gene. This example shows a 3-register (a, b, c) instruction, where it operates on two arbitrary variables (b and c). Operands can be swapped with constants, e.g., $a = 5 + c$. These constants belong to what is known as the terminal set. The collection of operators that can be selected is known as the function set. Chromosome or individual is a number of instructions followed one after another, e.g., $(a = b + c, b = a - c, a = 2 * a)$.

Initialization: The number of initialized individuals is chosen by the population size. Each individual has a set length. Individuals (usually) consist of ineffective instructions that have no impact on the end fitness value, e.g., $a = a + 0$. Such instructions are called introns.

Selection Operators: One of the most popular selection operator is called the steady-state tournament selection. In this selection usually four tournament members are chosen randomly from the population. They compete among each other and the two best individuals are retained and taken to the crossover and mutation methods. The children received after the previously discussed methods overwrite the losers in the tournament and take their respective positions in the population. The advantage of the steady-state tournament selection is that it does not require individual comparison among all individuals in a population. The experiment ran for EncDecAD utilizes this tournament selection.

Variation Operators: There are a few variation operators. In crossover, two individuals (sometimes called parents) are chosen and portions of each individual are exchanged. Figure 1 shows an example of a one point crossover. The idea behind mutation is that any component of the instruction (destination register, operands, and operator) can be changed for another alternative. If the operator is changed into one of the possible alternatives, then these substitutes are called function sets. The most basic function sets include multiplication, summation, division, and subtraction. For example, in the following instruction $a = a + b$, the operator can be randomly assigned to division,

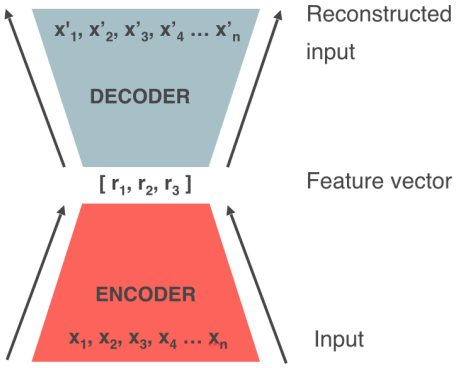

Figure 2: An example of encoder-decoder architecture based on (Cho et al., 2014).

thus the instruction becomes $a = a/b$. The probabilities of both, crossover and mutation, are distinct parameters often called *pCross* and *pMut* respectively.

2.2 Encoder-decoder Anomaly Detection

The proposed anomaly detection techniques are based on an autoencoder (see fig. 2). It has two parts: encoder and decoder, which are used to learn and reconstruct the behaviour of the chosen system and later use the reconstructed error to expose an anomaly (Malhotra et al., 2016). In some literature this is also referred to as a deviation based anomaly detection (An and Cho, 2015). The input data of an autoencoder is also the target data. Therefore, at first an autoencoder is run on normal data sequences that have no anomalies. Later on, when the model has learned the normal behaviour of the system, the same model is run on data that has anomalous sequences. Thus, an autoencoder is trained by unsupervised learning.

There are different techniques to perform anomaly detection, one being neural networks (Xu et al., 2015; An and Cho, 2015; Tsai et al., 2009).

2.3 Neural Network Architecture Evolution

Neural network architecture is defined by the design of how many neurons (circles) are connected together by synapses (lines) (see fig. 3). The function f is an activation function. For example, *ReLu* is an activation function, where $y = max(0,x)$. This means that any value below zero will be zero after running activation function and any number above zero is returned. Weights are described as a unit that determine the strength of each connected synapse. Bias is an additional unit that modifies the output. Different type of neural network usage brought strong results in various

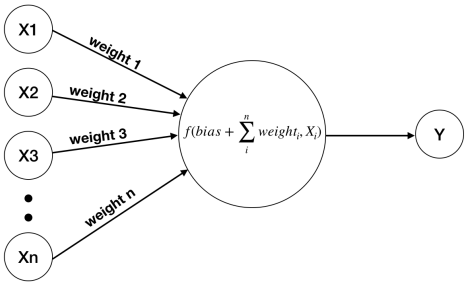

Figure 3: An example of a traditional neural network, based on (Hsu et al., 1995).

disciplines such as diagnosis of myocardial infarction (Baxt, 1991), speech recognition (Mikolov et al., 2010), engine fault detection (Ahmed et al., 2015). Other researchers have evolved the architecture of neural networks with promising outcome (Rawal and Miikkulainen, 2018; Assunção et al., 2018; Miikkulainen et al., 2019; Dabhi and Chaudhary, 2015). A portion of these papers show recent work on training the weights and biases with evolutionary systems (Tsai et al., 2006; Zhang and Suganthan, 2016). Another research shows examples of evolving neural network architecture (Such et al., 2017). Finally, some authors have achieved successful showcases of using evolutionary approaches to find both architectures and weights (Schaffer et al., 1992; Koza and Rice, 1991).

3 EXPERIMENTAL VALIDATION

This paper studies two EncDecAD methods that utilize evolutionary techniques. The first method uses LGP and was executed both offline and online while the second method uses self-configuring neural network that is created using evolutionary paradigms. The offline training means that the training has been conducted on a laptop while the validation has been run on the embedded device. Online training means that both the training and the validation have been performed on the embedded device.

The suggested method for anomaly detection is much faster than a conventional tuned neural network alternative. The neural network with autoencoder architecture was chosen to solve the same issue. It had the following dimensionality of hidden layers: [19, 15, 11, 7, 3, 7, 11, 15, 19]. The neural network was created using Tensorflow. Its parameters are as follows: ReLu activation function, loss function RMSE, learning rate 0.01, with enabled dropout. This neural network's architecture requires 1187 calculations to evaluate a new input, while our proposed method requires maximum 50 (after the introns were removed). Thus our proposed anomaly detection method is at

least 20 times faster.

Furthermore, one of the motivations to use the combination of encoder-decoder architecture and linear genetic programming is that it does not require a high power supply, costly computer, labeling of the data, nor it produces heating problems due to it's speed.

Authors of this paper propose a method to evolve weights and the architecture using a self-configuring neural network. As the results later in this paper show, the proposed technique is efficient to execute due to the use of simple instructions compared to running advanced structures like lists in computer systems that take long time. The parameters for the experiments have been chosen after a comparison within a few runs.

The online (i.e., on-board) experiments were run on a single board computer called STM32-F407. This development board has the following features; STM32F407ZGT6 Cortex-M4 210 DMIPS, 1MB Flash, and 196KB RAM (Olimex, 2018).

3.1 Collection of Data

The data used for training and testing of the proposed anomaly detection model was gathered from a self-driving mini-vehicle. The used mini-vehicle was fully equipped with sensors and a CAN bus (a vehicle bus standard) just like its real-size counterpart (Di Natale, 2000). The vehicle completed a total of 12 runs, each time with a randomized route where the vehicle exhibited both different speeds and various sharp and wide turns.

Twelve runs of approximately 3 min duration each were done with an induced anomaly. The anomaly data consists of two data sets: one where the designed error was produced by loosened suspension on one of the front wheels, and another with a resistor soldered in series with the battery of the vehicle.

The collected data included the following variables: temperature, battery current, motor current, battery voltage, speed, roll, pitch, yaw, three accelerometer values, three gyroscope values, three magnetometer values, tachometer, and commanded steering angle.

3.2 Validation through Offline Training

This subsection describes the settings used in the initial experiment validation, where the training has been applied offline. This step has been done as a proof of concept.

The first step to designing an autoencoder with LGP is to choose the common parameters for the model. Table 1 lists the parameters for all the runs. The fitness function chosen for the autoencoder can be followed in equations 1, 2, and 3. The individual length size is divided into two parts: encoder and decoder. Hereby the first half of the individual, i.e., 1500 instructions, is used to find the proper encoder values while the other half of the individual utilizes the previously found encoder values with the goal to reconstruct the original input-values.

$$ES_x = \alpha \sum_{i=1}^{p} EncoderValue_i \tag{1}$$

$$RE_x = \sqrt{\frac{1}{k} \sum_{i=1}^{k} (X_i - X_i')^2} + ES_x \tag{2}$$

$$Fitness = \frac{1}{n} \sum_{i=1}^{n} RE_i \tag{3}$$

where:

- ES is the encoder-sum.
- p is the number of encoder registers.
- α is the chosen number which provides a weight to the feature vector.
- $EncoderValue_i$ is the element in the feature vector (see fig.2).
- RE is the row error, where row is the input array. k is the number of decoder registers.
- X_i is the decoder register i.
- X_i' is the groundtruth value of the register i.
- n is the number of input rows/arrays.

In other words, when an individual receives the first row of the input array, it only executes half of instructions of one individual. Since the chosen encoder register number in the presented example is 3, the encoder-sum adds the first 3 values of the output which is equal to the first equation. Then these first three values are saved and the other register values are set to zero (in the presented example this becomes the decoder register's array). This allows the second half of the individual to fill the rest of the values itself. When this is done, the second equation is used where it calculates the root mean squared error of the calculated decoder register's value and the input value, i.e., groundtruth value. The addition of the encoder-sum in the second equation plays a role. By providing the encoder-sum to this equation, it indicates to the fitness function that the encoder values need to be significant. If this was not be added, the autoencoder would not work, since the feature vector would be random numbers that do not provide anything to the fitness function. Finally, when all row errors have been calculated the total fitness for the individual is the average of these errors, see equation 3. This implies that the closer the output values are to the input

Table 1: LGP parameters used in two scenarios: the validation through offline training and validation through an online (on-board) embedded training.

Parameter	Offline training settings	Online training settings
Fitness function	Standardized, RMSE	Standardized, RMSE
Terminal set	[-5, 5]	[-1, 1]
Function set	+, -, *, /	+, -, *
Safe division enabled?	Yes	No
Number of encoder registers	3	3
Number of decoder registers	19	19
Population size	50	50
Crossover probability	0.6	0.6
Mutation probability	0.05	0.05
Selection	Steady-state tournament	Steady-state tournament
Number of tournament members	4	4
Termination criteria	None	None
Individual length	3000	50
Undefined constant	1000000	None
Alpha	0.01	0.01

values the closer the autoencoder fitness value will be to zero.

The function set included a safe division (Brameier and Banzhaf, 2007). This means that if an instruction includes a division by zero, it sets the source register to an undefined constant. In this experiment's case that constant is 1000000.

Termination criteria is set to none as the algorithm was allowed to run the set number of generations instead of stopping it when it reached some value of fitness function. This enabled the algorithm to reach its best fitness function without making any assumptions.

The training part of the initial anomaly detection system was executed on a computer with the following specifications: Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz, 16.0 GB installed memory (RAM), 64-bit Operating System, Windows 10 OS. The training model was written in Python 3.6. It was trained on half of the gathered data. The other half of the data was left for validation.

3.3 Validation through On-board Embedded Training

After the initial experiment was demonstrated to uphold the concept behind EncDecAD using evolutionary programming, the next step was to enable not only the testing part, but also the *training part* on the embedded device.

The fitness function is the same as used in the validation through the offline training experiment. However, there are some substantial differences in the chosen LGP parameters for this experiment (see table 1).

To start with, the individual was trained on a smaller set of data, i.e., 2000 input rows of data, since the on-board flash memory has a total storage space of 1MB, and the space that was allocated for training data was 384KB. Secondly, the function set does not include division. The reason for this approach is that division requires 2-12 cycles on a Cortex-M processor, while the other mathematical operations only require 1 cycle. Lastly, the individual length has been reduced to hold between 10 to 50 instructions.

The algorithm was developed in the C++ programming language. The algorithm was trained on the 70% of non-faulty data and validated on the other 30% of unseen non-faulty data. The data arrays were chosen randomly from a large pool of non-faulty data.

3.4 Validation through On-board Embedded Training using Evolutionary Approach to Neural Networks

This paper proposes a novel solution to anomaly detection. The idea behind it is to use an evolutionary approach to neural networks on the embedded device. The concept follows the familiar encoder-decoder anomaly detection architecture but this last experiment employs *virtual neural network* components in it. The evolutionary neural network is constructed as a restrained register machine and shares the advantages with the unconstrained LGP method but in addition generates a pure deep neural network which may be more easily integrated in some data science environments. What we do is that we restrain the functions set and grammar of the LGP in

such a way that everything that is produced is isomorphic to a standard neural network. Similarly any neural network can be expressed using this "register machine syntax". The motivation is twofold; first, many data-scientists are more comfortable using what can be shown to be equivalent to a neural network, and second, there may be advantages in the learning time of this constrained model as compared to the general linear genetic programming approach (Brameier and Banzhaf, 2001). The advantage of the neural model is pure speculation based on information-theoretic reasoning and some early runs but are by no means proven in this paper. The neural network approach is more of a teaser in need of further investigation.

The differences of this neural network approach as compared to the previous LGP approaches in this paper are as follows: to start with, instead of using the linear genetic programming instructions that are generic on the form $a = b - c$, the proposed method employs a different set of more constrained instructions. The instances are shown in the example instructions below.

$$S_x = S_y + S_z \tag{4}$$

$$S_y = ReLu(S_x) * w_z + b_q \tag{5}$$

$$w_z = w_x + w_y \tag{6}$$

$$w_z = w_x * w_y \tag{7}$$

$$b_q = b_x + b_y \tag{8}$$

$$b_z = b_x * b_y \tag{9}$$

where:
- S is a weighted sum.
- w is a weight.
- b is a bias.
- $ReLu$ is an activation function, where $y = max(0, x)$.

The weighted sum simply takes a register and adds it to another, see instruction 4. The instruction in 5 employs the activation function called rectifier, multiplies it by a randomly chosen weight and adds a bias. The last two instruction examples (see 8 and 9) show the manipulation of weights, where the function set consists of only multiplication and summation and one of the weights has the possibility to change into a terminal set (a constant that belongs to the chosen range, e.g., see "Terminal set" in table 1).

Second difference compared to the online training settings experiment is that the individual size is increased from 50 to 200 where the first half manipulates the encoder-related data and the other half handles the decoder data.

Thirdly, self-configuring neural network approach initializes another vector with weights and biases. The usual number for weights is 3 and the values are chosen between $[-1, 1]$. Biases are initialized in the same way. Lastly, the new approach was tested while running for 100 000 generations.

The neural network starts with the required vectors (weight and bias) initialization. The next step is to run an activation function on all of the starting registers in order to progress from the input layer to the hidden layer. After that any instruction takes place as described before. If an instruction employs the activation function, this is a very similar step to creating a new hidden layer in the neural networks (see fig. 3). When 100 instructions have been run, only the first three registers are kept and others are set to zero. This way the neural network keeps the same EncDecAD architecture. The last step in the decoder part is to run an activation function again just as it is done on a traditional neural network. An example of this approach is provided in the figure 4.

Given the provided weights and biases in the figure 4, the proposed evolutionary neural network follows the instructions on the right side of the figure. The numbers next to the instructions show their execution order in the figure. Let's assume the input layer consists of variables 3, 20, and -5. Due to simplification all weights are set to value 1 and bias is set to a value 0.1 (see the top blue box on the right of figure 4). The first three instructions ((1) -(3)) in the example are for running the activation function which resembles adding another neural network layer in the

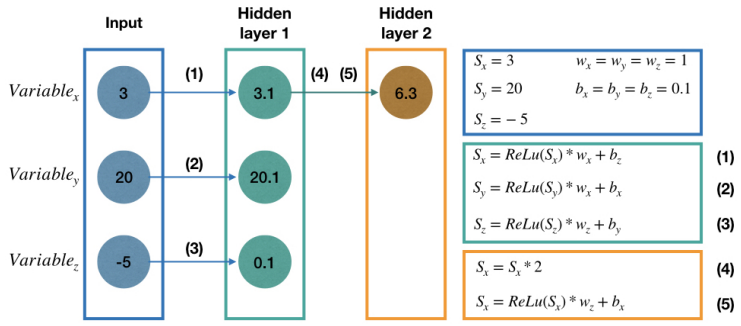

Figure 4: An example of a proposed evolutionary approach to neural network.

Table 2: Confidence intervals.

Experiment	Data set	Sample mean	Standard deviation	Sample size	Population mean with 95% confidence
Online training	Validation data	2031	713	1000	[1987, 2076]
	Resistance testing data	2406	715	1000	[2362, 2450]
	Suspension testing data	2469	575	1000	[2433, 2504]
Online training with self-configuring neural network	Validation data	1153.72	486.84	1000	[120, 1180]
	Resistance testing data	1513.12	397.81	1000	[1490, 1540]
	Suspension testing data	1638.24	1180.20	1000	[1570, 1710]

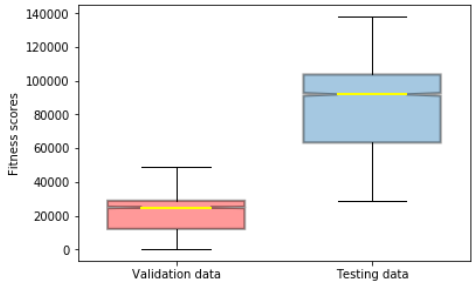

Figure 5: Fitness scores for loosen suspension data in offline training.

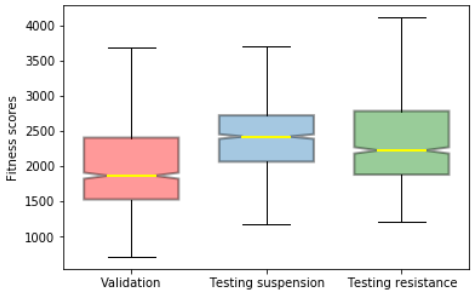

Figure 6: Fitness scores in online training.

traditional scenario. Thus the input layer shown in blue color progresses to the first hidden layer 1 in the green color. For example, if the input variable is 3, then after executing the first instruction (1), the result is 3.1 ($ReLu(3)$ retrieves value 3, which multiplied by 1 and added to 0.1, gives the result 3.1). Instruction (5) executes a multiplication using a neuron S_x from the hidden layer 1. The only way to create another neural network layer is by running the activation function which is achieved by the last instruction (6).

4 RESULTS AND DISCUSSION

The data was processed as follows: when the best individual for the anomaly detection has been found, it has been stripped off the introns. This resulted in the individual that has been reduced in its size by over 120 times (from 3000 to 50). The next step was to do a validation test and run the individual on unseen non-faulty data. In the following step, the best individual is tested on the faulty unseen data (right side of the figures 5, 6, 7). This allows us to determine if the algorithm managed to find the anomaly. This step is executed on the embedded system (see section 3).

The initial step to check the validity of the results in offline training included displaying boxplots of both results: normal (validation) data and the faulty data (testing) of the loosened suspension data set. The non-overlapping notches indicate that the samples come from populations with different medians ($\mu Validation = 23382$, $\mu Testing = 85367$). With a 95% confidence the validation population mean is between 23000 and 23800 while with the same 95% confidence the testing population mean is between 84600 and 86100 (see table 2).

The anomaly detection method presented in the online training made an assumption that the two data sets (faulty and normal) were adequately different from each other to be able to detect the anomaly. Furthermore, the distributions of all acquired data in this step followed a normal distribution. Therefore, Welch's t-test for independent samples following the normal distribution data was applied.

The hypothesis is that fitness scores obtained in the validation and testing differ significantly. The mean of suspension fault (testing) differed significantly to the mean of the normal validation data according to a Welch's t-test, t(1911.20) = -15.069, p $<$ .001, n = 1000. The mean between validation ($\mu = 2031.7$) and suspension fault data ($\mu = 2468.77$) is significant (see table 2).

The identical hypothesis was tested on the validation and resistance faulty data. The mean resistance fault differed significantly to the mean of normal validation data according to a Welch's t-test, t(1997.99) = -11.72, p $<$ .001, n = 1000. The mean between normal validation ($\mu = 2031.7$) and suspension fault data ($\mu = 2406.44$) is significant.

Confidence intervals are also shown in figure 6. Since notches in all boxplots do not overlap, this concludes that the data in all three sets belong to different distributions.

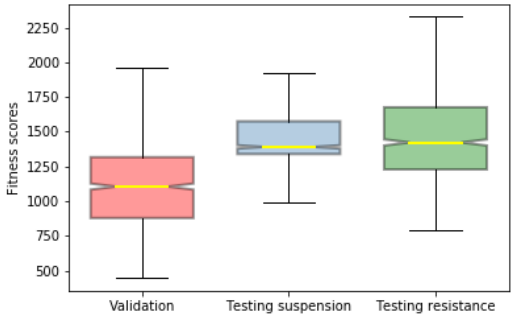

Figure 7: Fitness scores in online training using self-configuring neural network.

The proposed evolving neural network architecture uses the same assumptions as above. The individual was trained for 100 000 generations. The results are as follows: the Welch's t-test between validation and suspension showed that t(1329) = 12.0014, $p < .001$, n = 1000. The mean between normal validation ($\mu = 1153.72$) and suspension fault data ($\mu = 1638.24$) is significant. The results between validation and resistance show similar outcome: t(1921) = 18.0772, $p < .001$, n = 1000. The mean of resistance data is $\mu = 1513.12$ (see table 2). By conventional criteria, this difference is considered to be extremely statistically significant.

Therefore the proposed EncDecAD using linear genetic programming verifies these anomalies.

5 CONCLUSION AND FUTURE WORK

This paper presents two approaches to detect anomalies using encoder-decoder architecture while utilizing linear genetic programming. This is an improvement compared to the conventional neural network approach: firstly, the proposed method is faster by at least one order of magnitude, and secondly, it can be easily run on an embedded device. The first approach includes encoder-decoder anomaly detection using linear genetic programming on an embedded device. The second "teaser" approach presented suggests an approach using a self-evolving neural network employing genetic algorithm to simultaneously evolve the architecture, weights and bias.

The suggested approaches have been evaluated by applying data from an autonomous multisensor mini-vehicle with 19 features and a CAN-bus. The vehicle has been run repeatedly on different paths to gather data with induced anomalies, such as loosened suspension and soldered resistor in series with the battery.

The experiments were carried out on an embedded device, STM32-F407. The results included comparing the distributions of the acquired fitness scores between the validation (normal) data and testing (faulty) data. The proposed encoder-decoder anomaly detection using linear genetic programming verifies anomalies by applying Welch's t-test on the two data sets.

To our knowledge there is no previous research where an encoder-decoder anomaly detection has been performed using linear genetic programming. The framework presented for detecting anomalies can be applied to a broader class of problems including sequence problems. This can open new opportunities of having an anomaly detection system in any vehicle, e.g., placing such a device in an electric vehicle and allowing it to develop a model for the precise car or driver.

Further experimental investigations include graduating from encoder-decoder anomaly detection to adversarial neural networks which may improve the ability to identify which anomaly has been triggered.

REFERENCES

Ahmed, M., Mahmood, A. N., and Islam, M. R. (2016). A survey of anomaly detection techniques in financial domain. *Future Generation Computer Systems*, 55:278–288.

Ahmed, R., El Sayed, M., Gadsden, S. A., Tjong, J., and Habibi, S. (2015). Automotive internal-combustion-engine fault detection and classification using artificial neural network techniques. *IEEE Transactions on vehicular technology*, 64(1):21–33.

An, J. and Cho, S. (2015). Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE*, 2:1–18.

Assunção, F., Lourenço, N., Machado, P., and Ribeiro, B. (2018). Evolving the topology of large scale deep neural networks. In *European Conference on Genetic Programming*, pages 19–34. Springer.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco.

Baxt, W. G. (1991). Use of an artificial neural network for the diagnosis of myocardial infarction. *Annals of internal medicine*, 115(11):843–848.

Brameier, M. and Banzhaf, W. (2001). A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26.

Brameier, M. F. and Banzhaf, W. (2007). Basic concepts of linear genetic programming. *Linear Genetic Programming*, pages 13–34.

Chandola, V., Banerjee, A., and Kumar, V. (2009).

Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Dabhi, V. K. and Chaudhary, S. (2015). Empirical modeling using genetic programming: a survey of issues and approaches. *Natural Computing*, 14(2):303–330.

Di Natale, M. (2000). Scheduling the can bus with earliest deadline techniques. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 259–268. IEEE.

Hangal, S. and Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 291–301. IEEE.

Hegde, R., Mishra, G., and Gurumurthy, K. (2011). An insight into the hardware and software complexity of ecus in vehicles. In *International Conference on Advances in Computing and Information Technology*, pages 99–106. Springer.

Hsu, K.-l., Gupta, H. V., and Sorooshian, S. (1995). Artificial neural network modeling of the rainfall-runoff process. *Water resources research*, 31(10):2517–2530.

Koza, J. R. and Rice, J. P. (1991). Genetic generation of both the weights and architecture for a neural network. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 397–404. IEEE.

Langdon, W. B. and Harman, M. (2015). Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135.

Lu, H., Li, Y., Mu, S., Wang, D., Kim, H., and Serikawa, S. (2018). Motor anomaly detection for unmanned aerial vehicles using reinforcement learning. *IEEE internet of things journal*, 5(4):2315–2322.

Malhotra, P., Ramakrishnan, A., Anand, G., Vig, L., Agarwal, P., and Shroff, G. (2016). Lstm-based encoder-decoder for multi-sensor anomaly detection. *arXiv preprint arXiv:1607.00148*.

Mehr, A. D., Kahya, E., and Olyaie, E. (2013). Streamflow prediction using linear genetic programming in comparison with a neuro-wavelet technique. *Journal of Hydrology*, 505:240–249.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al. (2019). Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

Nordin, P. and Banzhaf, W. (1995). *A genetic programming system learning obstacle avoiding behavior and controlling a miniature robot in real time*. Univ., Systems Analysis Research Group.

Olimex, L. (September, 2018). *STM32-E407 development board USER'S MANUAL*. Olimex LTD.

Pachauri, G. and Sharma, S. (2015). Anomaly detection in medical wireless sensor networks using machine learning algorithms. *Procedia Computer Science*, 70:325–333.

Rawal, A. and Miikkulainen, R. (2018). From nodes to networks: Evolving recurrent neural networks. *arXiv preprint arXiv:1803.04439*.

Sabokrou, M., Fayyaz, M., Fathy, M., and Klette, R. (2017). Deep-cascade: Cascading 3d deep neural networks for fast anomaly detection and localization in crowded scenes. *IEEE Transactions on Image Processing*, 26(4):1992–2004.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 1–37. IEEE.

Shao, L., Liu, L., and Li, X. (2014). Feature learning for image classification via multiobjective genetic programming. *IEEE Transactions on Neural Networks and Learning Systems*, 25(7):1359–1371.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2017). Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.

Taylor, A., Japkowicz, N., and Leblanc, S. (2015). Frequency-based anomaly detection for the automotive can bus. In *WCICSS*, pages 45–49.

Tsai, C.-F., Hsu, Y.-F., Lin, C.-Y., and Lin, W.-Y. (2009). Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000.

Tsai, J.-T., Chou, J.-H., and Liu, T.-K. (2006). Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Transactions on Neural Networks*, 17(1):69–80.

Xu, D., Ricci, E., Yan, Y., Song, J., and Sebe, N. (2015). Learning deep representations of appearance and motion for anomalous event detection. *arXiv preprint arXiv:1510.01553*.

Zhang, L. and Suganthan, P. N. (2016). A survey of randomized algorithms for training neural networks. *Information Sciences*, 364:146–155.