AUTOMATED DEEP LEARNING FOR TIME SERIES OUTLIER DETECTION

A Thesis

by

WANGYANG HE

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Xia Hu
Co-Chair of Committee,	Frank Shipman
Committee Member,	Kevin Nowka
Head of Department,	Scott Schaefer

December 2022

Major Subject: Computer Science & Engineering

ABSTRACT

Time-series outlier detection reveals uncommon points or patterns with abnormal behaviors within time-series datasets and settings. It is a crucial research area to explore because it can be helpful for many real-world scenarios. Some popular fields, such as fraud detection, healthcare, cancer detection, cybersecurity attack detection, and fault detection, could benefit from time-series outlier detection. For example, in real-world fraud detection, millions of transactions are fed into the database daily. The outlier detection system needs to recognize a suspicious transaction or pattern as soon as possible. If we manually download the data daily to make predictions on it, this would take too much time and effort, and most importantly, it could potentially be too late to detect the fraud case. Therefore, in these real-world databases, time-series data becomes a real challenge for researchers to explore.

Oftentimes, engineers take a dataset and manually build a fixed-designed neural network for a specific task to predict and recognize outliers. However, this isn't the optimal strategy for treating time-series data. A fixed-designed neural network will not have enough power to capture all the details inside a time-series data. Each time-series outlier detection task will require different network architectures to detect outlier points and patterns accurately.

In general machine learning, researchers have found a way to search for the best model according to the unique behaviors of each dataset, which is Automated Machine Learning (AutoML). AutoML automates machine learning tasks and workflows with different techniques, which benefits non-experts to use machine learning models more quickly. Some standard methods for AutoML include hyperparameter optimization, meta-learning, and neural architecture search (NAS).

In this thesis proposal, by adapting and modifying the traditional NAS strategies, we propose a new method to construct a suitable search space with the proper size and combine the power of different deep learning time-series outlier detection algorithms with AutoML searching methods to search an effective neural network for time-series outlier detection automatically.

DEDICATION

To my family, mentors, friends, and everyone who supported me physically or mentally, I could not have done this without you all.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Xia Hu, Professor Frank Shipman from the Department of Computer Science & Engineering, and Professor Kevin Nowka from the Department of Electrical & Computer Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

The research work was supported under the Student Technician title from Texas A&M University.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

Time-series outlier detection tackles the challenges of finding abnormal points or patterns based on overall behaviors in a time series environment. This area of research [1] [2] started around the 1960s when researchers and scientists tried to solve outlier issues with single random samples and some standard linear model methods. Back in the 1970s, researchers [3] started to categorize different types of outlier points that can be found in time series based on different kinds of unique behaviors. By that time, time-series outlier detection had slowly become an essential task in the computer science and data mining field, which not only attracted more researchers to explore this area but also interested engineers to start applying it to different real-world applications, including fraud and fault detection tasks, etc.

As time went by, the development of this field grew extremely fast. Especially in the early 2010s, when hardware and software technologies became more advanced led, machine learning methods began to shine in almost every area. Researchers [4] started to apply different statistical and machine learning methods to time-series outlier detection, and this massively leveled up the potential and popularity in this area. Statistics methods like vector autoregression (VARMA), autoregressive moving average (ARMA), and clustering methods like k-Means, one-class SVM, or even unsupervised parametric methods like finite state automata (FSA) or Hidden Markov Models (HMMs), and many more, were all tested and applied into time-series outlier detection. These traditional methods and approaches all showed promising results on popular small academic datasets but faced challenges on large scaled multivariate time series datasets. As we all know by now, temporal data are getting collected every second in almost every single industry. By following the "big data" trend, it is really not an easy route for those traditional methods and approaches to be applied to large real-world data. Therefore, this has become a huge issue with time-series outlier detection, among all the other well-known issues like data labeling, data feature complexity, out-

lier categorization, etc. We slowly realize a model for time-series outlier detection might not be so easy to construct in order to take care of the complex nature of time-series data.

During the last several years, researchers adapted many deep learning methods to time-series outlier detection and exceeded many state-of-the-art results compared to traditional methods, especially in unsupervised settings. A popular method [5] [6] is autoencoder(AE) based on recurrent-based layers, which aims to extract and learn multivariate time series data with a large number of dimensions. While deep learning methods like AE could achieve wonderful results on time series data, it relies heavily on machine learning experts to construct such a model for each different case. While the number of machine learning experts has been growing extremely fast in recent years, it is still impossible to match the amount of data being collected, especially time-series data, are getting collected each second.

The purpose of collecting a massive amount of time-series data isn't just for the record itself. We would like to find some sort of trend, structure, or behavior from these data. Machine learning comes in place as the best tool to do so. However, there are only less than 1% of humans are machine learning experts, and they have a better sense of how to structure and initialize the machine learning settings and parameters. But what about the rest of 99% of the population that wants to use machine learning? Due to the nature of machine learning, in which it could be applied to any type of scenario and data set, this technique is now the state-of-the-art way to analyze and predict. Many non-ML experts from areas like healthcare, statistics, finance, business, etc., want to use machine learning in their own areas, but it would take a lot of time and effort to get their foot in the door to learn it. Even after learning the basics of machine learning, it is extremely difficult for them to try out all kinds of models and combinations of architectures and hyperparameters. This is why Automated Machine Learning (AutoML) is rapidly becoming the most user-friendly way of using machine learning.

The term "AutoML" is pretty easy to understand, making machine learning automated. A typical AutoML process would take the input data from the user, do some necessary preprocessing steps, and throw this data into hundreds or thousands of different combinations of machine learning

models. Through this process, the combinations of models are made based on the combinations of different hyperparameters. The AutoML process will stop once all hyperparameters reach their best performance, and this will output the best possible model for the user. The AutoML search process is thousands of times faster than real humans could possibly try and significantly reduces the chance of errors within its process.

There are many different types of AutoML, including automated feature engineering, automated model and hyperparameter tuning, automated deep learning, etc. Within each of these types of AutoML, there are different methods and techniques used for different purposes. For deep learning tasks, common methods such as Bayesian Optimization, Gradient Boost, Neural Architecture Search (NAS), etc. Our method will utilize the advantages of NAS to further explore and solve some limitations of existing AutoML techniques.

Neural Architecture Search (NAS) [7] refers to the process of automatically searching for a good architecture design of neural networks. It tends to try different combinations of model architectures, and the goal is to find the optimal structure for a given task or data. The initial purpose of AutoML is to reduce the time and complexity of human effort and reduce the chances of human error. It has been proved by many researchers that NAS solves these issues and greatly decreases human effort in selecting a good model architecture. However, new issues and limitations have occurred with using NAS.

It has shown that the computational cost for NAS is extremely expensive. This is because, in theory, NAS tries to search through every possible combination of model components, and this leads to very large search space. Therefore, for complex models, the size of search space for NAS grows at an exponential rate. Especially for time series data, its high dimensionalities and multivariate characteristic is the biggest enemy of NAS. Hence researchers are now exploring efficient ways to construct a suitable search space and more efficient methods to apply NAS.

To tackle the above challenge, we found a new way to solve the large search space size issue. Inspired by the autoencoder (AE) structure, we provided a way to construct an encoder-decoder "U" shaped architecture while keeping the advantages and flexibility of NAS. This approach proved

its value against many state-of-the-art traditional methods by either matching or outperforming them. While reducing significant time and effort by using AutoML in time-series outlier detection, we also output many different models tested during searching, which could be used for other purposes.

1.2 Literature Review

In this section, we will review existing methods and frameworks/tools in both time-series outlier detection and automated machine learning. Since our proposed method is a combination of these two areas, the goal of this section is to learn about the methods and successful works, to extract their advantages, and to see how they realize the tool in a technical way.

1.2.1 Time-Series Outlier Detection

There are now many successfully built Time-Series Outlier Detection frameworks and open source projects developed by machine learning experts and engineers, most of them require very low-code setup and easy-to-understand usage for users to understand extremely quickly. In this section, I will survey five existing papers on Time-Series Outlier Detection frameworks and/or projects, to analyze each of them in terms of their innovative ideas, methods, limitations, etc. In section 1.2.1.1, I will talk about each of them individually as a subsection, and in section 1.2.1.2, I will summarize the similarities, differences, pros/cons, and my personal thoughts.

1.2.1.1 Papers and Packages

TODS

TODS is an automated time series outlier detection system, its corresponding paper [8] describes this system as an open-sourced framework mainly for automated time series outlier detection tasks. It contains more than 70 python implementations of algorithms for time-series outlier detection, including methods for data processing, time-series processing, feature analysis/extraction, detection algorithms, and reinforcement module. It is very modulated and easy to use with low-code integration, and highly flexible on building a machine learning pipeline for time-series outlier detection. The goal for TODS is to become an end-to-end system for real-world time

series outlier detection. This package is open-sourced in Python language. Currently TODS does not support any AutoML functionality, however, this is what I have accomplished for this thesis work. TODS will be our main tool to develop and experiment on for this thesis.

telemanom

Telemanom [5] was originally proposed by researchers from NASA to handle outliers in telemetry data for spacecraft tasks. It uses LSTMs to catch information learned within the network so it will become a unsupervised setting, because the amount of expert-labeled telemetry anomaly data is very limited, so telemanom becomes very crucial for them. As we know, telemetry data contains huge portions of time series data with high dimensional features, therefore, telemanom has also quickly become a popular method in time-series outlier detection. It's code has now turned into an open-sourced framework for everyone to use, implemented in Python language. Packages like TODS also has developed telemanom as a primitive for everyone to use.

DeepADoTS

DeepADots is a repository maintained by KDD-OpenSource to evaluate deep learning methods in time-series outlier detection. It now has 7 implemented deep learning algorithms including LSTM-AD [9], LSTM-ED [10], Autoencoder [11], Donut [12], REBM [13], DAGMM [14], and LSTM-DAGMM. All of the 7 deep learning methods are available to use in the repository, fully implemented in Python language. One disadvantage of this repository is that it does not provide any toy dataset, only relies on MNIST within Tensorflow.

NAB

NAB [15] [16] stands for Numenta Anomaly Benchmark, which is a novel benchmark to use for streaming and real-time tasks in time-seires outlier detection. It contains more than 50 labeled real-world time-series outlier data and it has a scoreboard of different detectors with their performance scores. NAB supports both Python and Julia implementations, the current leader on the scoreboard is a Julia implementation while many other detectors are mainly Python.

AnomalyDetection

AnomalyDetection is an open-sourced package for outlier detection, it was developed by Twitter team back in early 2010s with many staistical methods. The underlying algorithm they used to detect outliers is is Seasonal Hybrid ESD. Even though this package stopped developing ever since 8 years ago, it it still one of the most popular package used for outlier detection in the R language. This is because nowadays the mainstream language for machine learning and outlier detection is Python instead of R.

1.2.1.2 Thoughts on Time-Series Outlier Detection

We analyzed five different frameworks and/or projects of time-series outlier detection. The goal of these frameworks concludes in several different directions, including different industries, different purposes, different languages, and different topics. Some of them are task-specific, while others are trying to build a unified framework. While most of the algorithms are similar to these frameworks, some of them are still in process of development, and some of them are capable of letting the users implement new algorithms. Some of these frameworks has almost all the state-of-the-art methods within itself, like TODS. While some others only uses deep leanring methods, like DeepADoTs. Some of these are for benchmark purpose, like NAB, while some are only for a certain industry, like telemanom for aerospace. While most of them can be used with Python, several had the capability to use them in other languages like Julia and R.

1.2.2 AutoML

There are now many successfully built AutoML frameworks and open source projects developed by machine learning experts and engineers, most of them require very low-code setup and easy-to-understand usage for users to understand extremely quickly. In this section, I will survey five existing papers on AutoML frameworks and/or projects, to analyze each of them in terms of their innovative ideas, methods, limitations, etc. In section 1.2.2.1, I will talk about each of them individually as a subsection, and in section 1.2.2.2, I will summarize the similarities, differences, pros/cons, and my personal thoughts.

1.2.2.1 Papers and Packages

AutoKeras

In AutoKeras [17] the authors focus on neural architecture search in deep neural networks. They think most of the algorithms and frameworks are limited by expensive computational costs, and they built an open-sourced AutoML system called AutoKeras. AutoKeras is designed to run the searching process in parallel on both CPU and GPU, and it will also automatically adapt the searching configuration based on the user's hardware memory limitation. The method behind this system enables Bayesian optimization to support the searching process, and it also optimizes the proposed acquisition function to build the search space. This framework is open-sourced with Python and is one of the most popular AutoML frameworks now. AutoKeras is now being developed by Google's official Keras team, and the authors have written the book called "Automated Machine Learning in Action" for AutoML learners. AutoKeras will also be an important tool to use for the experiment of this thesis, which I will modify parts of AutoKeras and merge into TODS to activate the AutoML functionalty in time-series outlier detection.

TPOT

In TPOT [18] the authors built a genetic programming-based AutoML system, which focuses on the optimization of a series of feature preprocessors and models. The goal of this system is to maximize the accuracy of supervised machine learning problems. The term TPOT stands for Tree-based Pipeline Optimization Tool, and it is a wrapper of the scikit-learn package. It has various operators and algorithms within the framework, including supervised algorithms like Decision Tree, Random Forest, XGBoost, etc; feature preprocessing algorithms like Standard Scaler, Min-Max Scaler, etc; and feature selection algorithms like Select K Best, Select Percentile, etc. The pipeline flow is the most important thing to use in this framework, the users need to optimize the best pipeline for the best outcome. This system was tested on 150 supervised classification data sets, and it performs extremely well with a good pipeline.

Auto-SKLearn

In Auto-SKLearn [19] the authors want to make a framework that can be used with low-code, low-requirement, for non-experts of machine learning. They developed a new AutoML system that is implemented based on the scikit-learn library, called Auto-SKLearn. This system won the first phase of the ChaLearn AutoML challenge, which was participated by the developer of many powerful AutoML frameworks. They define the AutoML problem as a CASH problem, which stands for Combined Algorithm Selection and Hyperparameter. This system was compared to Auto-WEKA and hyperopt-sklearn, which contains 21 datasets from the Auto-WEKA system. It tied the best optimizer in nine of the sixteen cases and lost the other ones. This system is open-sourced and implemented with Python.

H2O AutoML

In H2O AutoML [20] the authors designed a new platform called H2O AutoML, it is capable of dealing with very large data sets, and it has APIs for four different languages, such as R, Java, Python, and Scala. It also has a well-designed web GUI for diverse teams of users to use. The results of this framework present themselves as a "leaderboard" list, which ranks all the searched models based on a certain metric, and each model is exportable for users to reuse. The techniques used to search are fast random search and stacked ensembles, these techniques are semi-random with a goal of optimization of performance and time. This framework contains many algorithms and it is open-sourced with Python implementation.

Google Vizier

In Google Vizier [21] the authors describe Google Vizier, which is an internal AutoML framework for Google. Google Vizier is used for performing black box tasks to help tune parameters, and it works together with Google's Cloud Machine LEarning HyperTune subsystem to optimize their machine learning models. It is meant to be ease of use, minimal configuration, and setup, fast searching process, a wide range of algorithms, flexible algorithm implementation, etc. This

framework is currently implemented with C++, Python, and Golang. It has a flexible functionality for users to use their own arbitrary algorithms, called algorithm playground. It also has very impressive functionalities like automated early stopping, which makes the searching process stop when it reaches the best performance. It also supports transfer learning which will save time for advanced users. One rare capability of Vizier is it designs excellent cookies, which is important for back-tracking purposes.

AutoCompete

In AutoCompete [22] the authors proposed the framework called AutoCompete, which is an automated machine learning framework to use for machine learning competitions. The authors spent approximately two years developing this framework in online machine learning competition environments. The main contribution of this paper is the process of helping its users to identify data types, choose a well-fitted model, and tune the hyperparameters while avoiding overfitting and containing several popular evaluation metrics. The authors defined a "Stacker" which contains the pre-processing features and stacks them into the feature selection process, which will be later on sent to the "Selector" and the "Hyperparameter Selector" for the best performance model. This framework contains both classification and regression algorithms for users to pick, including many famous algorithms like Random Forest, Gradient Boosting, Logistic Regression, SVM, etc. This framework is written in Python and heavily relies on the scikit-learn library. The authors would like to add many features in the future and also enable the functionality for searching the optimized model with a certain evaluation metric.

ATM

In ATM [23] the authors presented ATM, which stands for Auto-Tuned Models. It is an automated machine learning system that can deliver read-to-predict models extremely fast. It has heavy attention to feature engineering, which is a crucial part of machine learning. The results can be displayed in the form of a confusion matrix, cross-validation results, and training times. The

authors tested this framework on 420 different datasets and trained more than 3 million classifiers for a few days. ATM by itself generated the largest single repository of trained models for users to use, and with only several days of training, it beat about 30% of the human-generated model's performances. Keep in mind that this accomplishment only used 1% of the time compared to human-performed models. This paper also presented a novel method for forming a hierarchical search space of different methods, and the parameter tree is the optimized structure for hyperparameter searching.

LEAF

In LEAF [24] the authors think that deep neural networks (DNNs) are usually not being used by their full potential, due to the fact that it is extremely hard to find the proper configuration. Therefore, they promoted a framework called LEAF, which optimizes the hyperparameters, and the network architectures with the most suitable size. This framework works based on evolutionary algorithms to produce the optimal model. It uses neural architectures search space as the main technique to form the proper space and to find the best result. All of the algorithms are based on the CoDeepNEAT algorithm and various versions of it. They tested this framework with Wikipedia comment toxicity classification data sets and medical data sets like X-rays. This framework heavily relies on Keras, which mainly runs with Python.

DeepArchitect

In DeepArchitect [25] the authors think that current frameworks are too focused on a specific use-case, and are not general enough. They developed a formal language to encode the search space of the AutoML process, which can be used with many different combinations of algorithms and settings with ease. The advantages of this language are: very similar to computational graph, where it is nothing new for users to learn; the reusable search space and modules are flexible enough to save users some time; easy to use, usable without needing a machine learning expert. This framework is written in Python and open-sourced for everyone to use.

Auptimizer

In Auptimizer [26] the authors want to speed up the model tuning process of the AutoML problem, which they developed Auptimizer. This framework is a general hyperparameter optimization framework in which the users will be able to use most of their computing resources available. One of the unique features of this framework is that it's able to play the role of a bookkeeper which will track the history of searching for users to see. This framework is also flexible enough for users to switch between algorithms with ease, or even integrate new algorithms into the framework. This framework is now free to use in Python and many other languages like MATLAB and R. There are both implementations of model frameworks for Tensorflow and PyTorch.

1.2.2.2 Thoughts on AutoML

We analyzed five different frameworks and/or projects of AutoML. The goal of these frameworks concludes in several different directions, including usability, faster speed, less hardware reliability, and even for internal organization use. Some of them are task-specific, while others are trying to build a unified framework. While most of the algorithms are similar to these frameworks, some of them are still in process of development, and some of them are capable of letting the users implement new algorithms. Some of these frameworks use the HPO (hyperparameter optimization) strategy, some use semi-random search algorithms, and some use NAS (neural architecture search space). While all of them can be used with Python, several had the capability to use them in other languages.

2.1 Baseline Methods

In this section, we will look at some baseline methods commonly used in time-series outlier detection, which categorizes into methods that include searching, and without searching. For methods without searching, we will describe LSTM AutoEncoder and LSTM RNN; for methods with searching, we will describe Basic Search and Random Search.

2.1.1 Methods without Searching

2.1.1.1 LSTM AutoEncoder

An LSTM AutoEncoder [10] stands for Long Short Term Memory Networks based Encoder-Decoder. It is a way to implement an autoencoder with Encoder-Decoder LSTM architecture. This is a two-part process, including the encoder and the decoder. The encoder will encode or compress input data into compressed representations. The final layer of the encoder will then become the initial layer of the decoder, where the decoder will try to reconstruct back to the input data from the compressed representations. During this process, it will learn the behavior of normal time-series data, and after reconstructing the data, it will detect outliers based on the reconstruction error. Oftentimes, it is easy to reconstruct the normal data points, while not so easy to reconstruct the abnormal data points, which tells us that this could be a potential outlier. LSTM AutoEncoders have been applied to different types of sequence data, including audio, video, text, and time series data.

2.1.1.2 RNN LSTM

RNN LTSM [27] [9] stands for Recurrent Neural Networks with Long Short Term Memory. LSTM networks can keep long-term memory information in their input, output, and forget gates with a particular unit called "memory cells". It has proven that LSTM shows its true power under sequences with long patterns or lengths, which means this advantage fits perfectly with the nature

of time series data. To achieve the RNN LSTM method, we need to stack up LSTM networks by fully connecting LSTM units in hidden layers with recurrent connections. This means every unit in the lower half of the LSTM hidden layer will be fully connected to every unit in the upper half of the LSTM hidden layer. Then, to predict outliers, we use the error vector to compute the prediction errors, which will help us find the abnormal results.

2.1.2 Methods with Searching

2.1.2.1 Basic Search

Basic search in AutoML, also refers to grid search [28], is the traditional method for hyperparameters optimization. It simply defines the possible search space of NAS as a grid that contains all possible hyperparameters and will evaluate every single possibility during the searching process. Grid search is great for small experiments to do a quick search of hyperparameters if you already expect some value of hyperparameters will perform well.

2.1.2.2 Random Search

Random search in AutoML is similar to basic search, except it randomly chooses hyperparameters instead of searching in a predefined order like grid search. Random search proves to perform better than grid search. However, it takes a longer time to finish the search process if you do not know what hyperparameters will perform well. A tree-based random search in AutoML [18] can also perform well, it is based on genetic programming methods, and the goal is to maximize accuracy in supervised classification tasks, represented by a popular framework called TPOT.

3.1 Method

Our goal is to tackle the biggest issue of AutoML in time-series outlier detection, which is to find a suitable search space with the proper size while keeping the flexibility of the NAS strategy. We propose a new method inspired by the encoder-decoder structure of autoencoders(AE), where our search space will have a similar "U" shaped predefined architecture before searching.

The advantage of NAS searching is to be able to search for all possibilities because, in theory, this will ensure an optimal solution. However, to control the size of the search space, we found a technique to semi-randomly design our architecture prior to searching. On top of the commonly used hyperparameters for NAS, like the number of layers, number of units, number of filters, etc. We also added two important hyperparameters to help us construct a suitable search space: the number of middle-layer units and layer multiplier:

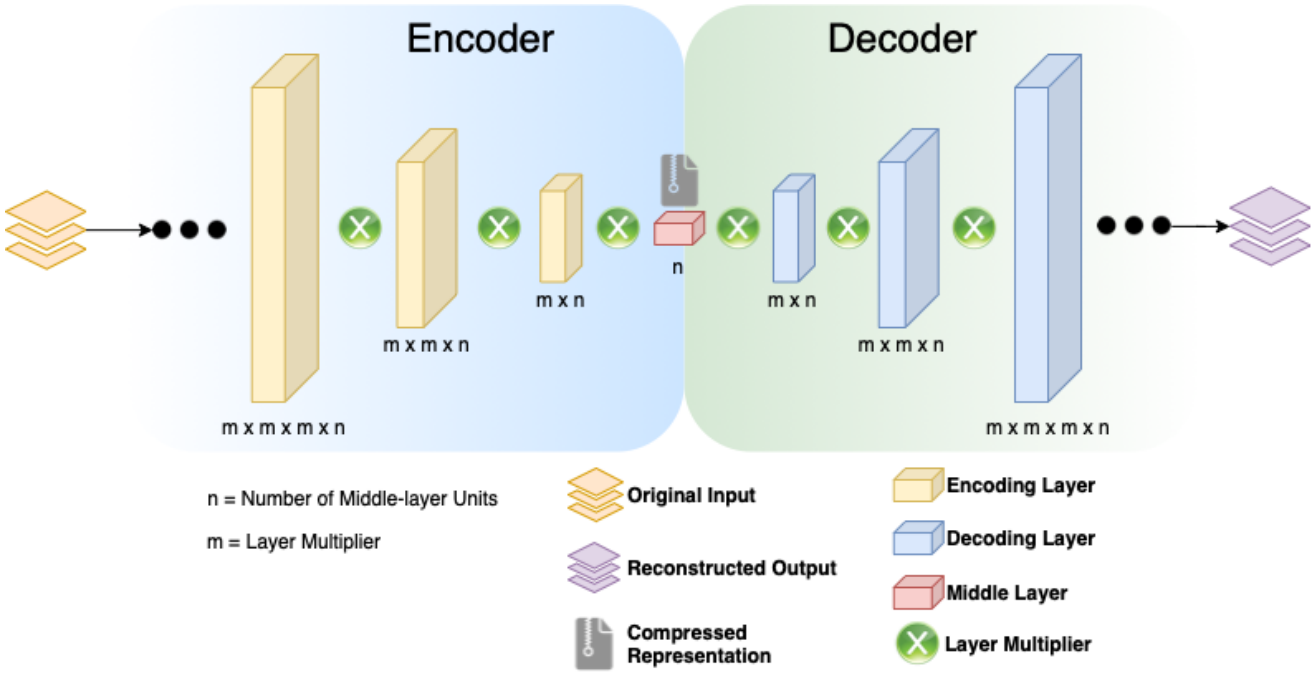

Figure 3.1: Proposed Neural Architecture Design

Number of Middle-layer Units: a hyperparameter that defines the number of units in the middle layer of all possible layers. This will always make the number of layers an odd number. The number of units in the middle layer will always be the lowest among all layers. In an autoencoder setting, this middle layer will become the last layer of the encoder and the first layer of the decoder.

Layer Multiplier: a hyperparameter that defines the multiplier for the number of units between each layer.

The two hyperparameters above will work together with the "number of layers" hyperparameter to construct a new architecture during each training step. As shown in Figure 3.1, based on an odd number "number of layers", the "number of middle-layer units" will start expanding the number of units per layer to its left and to its right based on a selected "layer multiplier", this process will end until the number of "number of layers" has been fulfilled. Then the final architecture for this specific training step will become:

$$[..., m^3n, m^2n, mn, n, mn, m^2n, m^3n, ...] \tag{3.1}$$

where n is the "number of middle-layer units" and m is the "layer multiplier".

3.1.1 Example Scenarios

Below I will list two possible scenarios on how this works:

Case 1:

First, NAS randomly selects the hyperparameter "number of layers" as 7, then our initial architecture will look like:

$$[l1, l2, l3, l4, l5, l6, l7] \tag{3.2}$$

where l means layer number. At this time, we are unsure how many units are in each of the layers yet.

Second, NAS randomly selects the hyperparameter "number of middle-layer units" as 4, from the above euqation, $l4$ is the middle-layer of this architecture, then our updated architecture will look like:

$$[l1, l2, l3, \mathbf{4}, l5, l6, l7] \tag{3.3}$$

in which now we know the number of units in the middle layer is 4. At this time, we are still unsure how many units are in each of the other layers yet.

Third, NAS randomly selects the hyperparameter "layer multiplier" as 2, which means starting from the middle layer, each layer to its left and right will be multiplied by 2 to get the number of units in that layer, then our updated architecture will look like:

$$[\mathbf{32}, \mathbf{16}, \mathbf{8}, \mathbf{4}, \mathbf{8}, \mathbf{16}, \mathbf{32}] \tag{3.4}$$

which this will be the final architecture for this specific training step.

Case 2:

First, NAS randomly selects the hyperparameter "number of layers" as 9, then our initial architecture will look like:

$$[l1, l2, l3, l4, l5, l6, l7, l8, l9] \tag{3.5}$$

where l means layer number. At this time, we are unsure how many units are in each of the layers yet.

Second, NAS randomly selects the hyperparameter "number of middle-layer units" as 8, from the above euqation, $l5$ is the middle-layer of this architecture, then our updated architecture will look like:

$$[l1, l2, l3, l4, \mathbf{8}, l6, l7, l8, l9] \tag{3.6}$$

in which now we know the number of units in the middle layer is 8. At this time, we are still unsure how many units are in each of the other layers yet.

Third, NAS randomly selects the hyperparameter "layer multiplier" as 3, which means starting from the middle layer, each layer to its left and right will be multiplied by 3 to get the number of units in that layer, then our updated architecture will look like:

$$[\mathbf{648}, \mathbf{216}, \mathbf{72}, \mathbf{24}, \mathbf{8}, \mathbf{24}, \mathbf{72}, \mathbf{216}, \mathbf{648}] \tag{3.7}$$

which this will be the final architecture for this specific training step.

3.1.2 Summary

As shown in the above section, we can see our method could potentially construct either small or large architectures. In Case 1, since our hyperparameters are commonly selected as smaller values, we resulted in a simple architecture as shown in Equation 3.4. However, in Case 2, our selected hyperparameters were bigger values than the ones selected in Case 1, and we resulted in a much more complex architecture, as shown in Equation 3.7.

Therefore, we can see that our proposed method did keep the flexibility of traditional basic search or random search methods. At the same time, it also has the capability of searching for both small and large architectures to ensure an optimal solution.

4. EXPERIMENTS

In this section, we will present a group of experimental studies to compare and evaluate the performance of our automated deep-learning methods in time-series outlier detection. We carefully set up our experiments and ignored problematic benchmark datasets mentioned in [29] and [30], where we picked two well-known benchmark datasets that are approved by many researchers in time-series outlier detection.

4.1 Experimental Setup

4.1.1 Environment

All experiments of this section are performed on an NVIDIA V100 GPU server with 8 GPUs, 64 CPUs, and a disk size of 1000GB. We believe it isn't necessary to use this high-performance environment for the experiment's purpose. However, it was used to decrease the running time.

4.1.2 Tools

To combine both AutoML and Time-series outlier detection, we utilized two open-source packages introduced in the Introduction sections. For automated machine learning, we used AutoKeras' AutoML backend framework[1], which relies heavily on TensorFlow and Keras. We used the "Regression Head" module from AutoKeras to enable the reconstruction functionality in TODS.

As we stated above, TODS is an automated time-series outlier detection system[2] that contains most of the state-of-the-art algorithms in outlier detection. Each algorithm is implemented into a "primitive" under the TODS unified framework. We took the "Dense Block" module and the "RNN Block" module from AutoKeras, modified both blocks, and merged them into TODS as new primitives called "AKAE" and "AKRNN".

[1] https://github.com/keras-team/autokeras
[2] https://github.com/datamllab/tods

AKAE is a new primitive in TODS, which stands for AutoKeras AutoEncoder. This primitive utilizes the original AutoKeras "Dense Block" which does not have the capability of an AutoEncoder. We added our newly designed method within the original "Dense Block" to predefine the Encoder-Decoder architecture before each training step.

AKRNN is also a new primitive in TODS, which stands for AutoKeras Recurrent Neural Network. This primitive utilizes the original AutoKeras "RNN Block" which stacks up many layers of LSTMs/GRUs. We added a dropout layer at the end to improve performance for time-series data. We added our newly designed method within the original "RNN Block" to predefine the neural architecture before each training step.

4.1.3 Datasets

We chose two datasets for testing our newly implemented primitives, which are The Server Machine Dataset(SMD) and the UCR benchmark dataset.

The Server Machine Dataset(SMD)[3] [6] is a real-world time-series public dataset. It was collected throughout a time period of 5 weeks from a large internet company. This dataset provides a training set, testing set, and labels for each corresponding machine, and they are all divided into the same size to better evaluate using different metrics. In total, it contains data from 28 server machines that monitor 33 metrics individually. We picked ten different machines to evaluate our method on and also compared to baseline methods performances on the same ten datasets selected.

The UCR Time Series Archive[4] [31] is another dataset we tested on. It was introduced in 2002 and has become one of the most used data sources for time-series-related research areas. This dataset contains 85 different time-series datasets on different tasks, and it gives information on the

[3] https://github.com/NetManAIOps/OmniAnomaly
[4] https://www.cs.ucr.edu/~eamonn/time_series_data/

number of training samples, testing samples, and anomaly ranges. To test our newly implemented primitives, we selected ten different datasets within this UCR archive and also used the same selected datasets to compare performance on selected algorithms.

4.1.4 Evaluation

The loss function used during model training is Mean Squared Error(MSE) 4.1, that is mainly because we currently only have regression based primitives. MSE can be defined as following:

$$\sum_{i=1}^{D}(x_i - y_i)^2 \tag{4.1}$$

We could also consider using Mean Absolute Error(MAE) 4.2, however it isn't as sensitive to time-series data as MSE. MAE can be defined as:

$$\sum_{i=1}^{D}|x_i - y_i| \tag{4.2}$$

The evaluation metrics used to compare results throughout all experiments are Precision, Recall, Accuracy, F1 score, and AUC score. In the resulting table, we reported only Accuracy, F1 score and AUC score, because both Precision and Recall are used to compute the F1 score.

In a confusion matrix, we would obtain information on the evaluated data with True Positive(TP) samples, True Negative(TN) samples, False Positive(FP) samples, and False Negative(FN) samples. In a time-series data setting, TP samples means the model detected normal sample points correctly compared to the ground truth labels; TN means the model detected outlier sample points correctly compared to the ground truth labels; FP means the model detected outliers as a normal point compared to the ground truth labels; FN means the model detected normal points as outliers compared to the ground truth labels. With the help of these information, we can compute the Precision as following:

$$Precision = \frac{TP}{TP + FP} \tag{4.3}$$

We can also compute the Recall as the following:

$$Recall = \frac{TP}{TP + FN} \tag{4.4}$$

On top of precision and recall, Accuracy can also be calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} * 100 \tag{4.5}$$

Now that we have both Precision and Recall, we can compute the F1 score 4.6:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{4.6}$$

Lastly, to compute the AUC score, we will need to use Sensitivity and Specificity, where Sensitivity can be computed similar to recall:

$$Sensitivity = Recall = \frac{TP}{TP + FN} \tag{4.7}$$

Specificity can be computed as:

$$Specificity = \frac{TN}{FP + TN} \tag{4.8}$$

Finally, Area Under Curve(AUC) can be computed as:

$$AUC = Sensitivity(TPR) - (1 - Specificity)(FPR) \tag{4.9}$$

Which can be expanded to:

$$AUC = (\frac{TP}{TP + FN})(TPR) - (1 - \frac{TN}{FP + TN})(FPR) \tag{4.10}$$

In my opionion the most important metric to look at is the F1 score, where we will be looking at the Macro-F1 average instead of the highest F1 score for each method. This is because the average would be more reasonable to compare since some algorithms aren't stable at all, they could have the highest F1 score on one dataset out of all the methods but also have the lowest F1 score on another dataset at the same time.

4.1.5 Baseline Comparison

We compared the neural architecture searched by AutoML for both AKAE and AKRNN with different related traditional approaches. Below here I will briefly summarize the traditional methods that we will be comparing to.

For AKAE, since it is very similar to the traditional AutoEncoder, just with architecture searching. We compared AKAE with AutoEncoder(AE) and VariationalAutoEncoder(VAE).

AutoEncoder

AutoEncoder(AE) is a neural network that can learn and can compress the input data, it will also learn how to reconstruct the data back to original input. The AutoEncoder contains two parts, the encoder, and the decoder. As shown in Figure 4.1, an image of number "2" from the MNIST dataset is fed into the encoder, the encoder will learn the image and compress it into compressed representations. The compressed representation will be the output of the encoder, and the input of

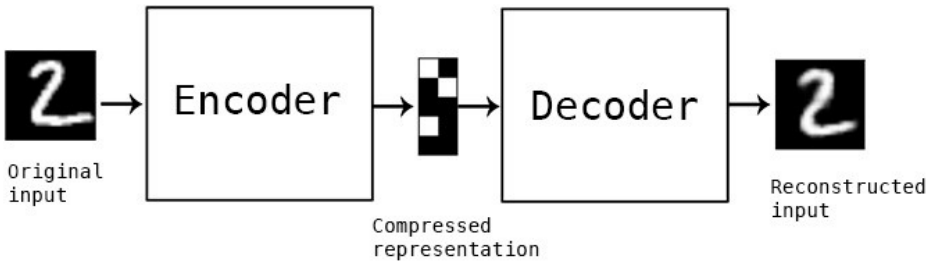

Figure 4.1: AutoEncoder for MNIST

the decoder. The decoder will take the compressed representation and try to reconstruct the image, which outputs a similar image like the original image.

VariationalAutoEncoder

VariationalAutoEncoder(VAE) is another popular neural network and it is an extension of AE. The only difference is that VAE addresses the non-regularized representations of AE so that now it is capable of randomly sampling vectors and generate data continuously. In VAE, to produce the latent value z:

$$z \sim q_{\mu,\sigma}(z) = \mathcal{N}(\mu, \sigma^2) \tag{4.11}$$

we need to sample:

$$\epsilon \sim \mathcal{N}(0, 1) \tag{4.12}$$

and finally z will be:

$$z = \mu + \epsilon \cdot \sigma \tag{4.13}$$

For AKRNN, it is a basic RNN network with either LSTM or GRU layers, it also can have bidirectional layers or dense layers based on searching results. We compare AKRNN to four different trational methods: Telemanom, DeepLog, SoGaal, and MoGaal.

Telemanom

Telemanom [5] uses LSTMs to detect anomalies under multivariate time-series data. The LSTM layers will learn the normal data points behaviors during the encoding process. It generates prediction errors during each single time step and compare it to the learned expected behavior. It is able to detect outlier points and even sequences in the time-series data. It was used initially by the NASA on aircraft telemetry environment, and now applied to many different time-series outlier detection tasks.

DeepLog

DeepLog [32] also uses LSTMs to detect outliers. It models a system log as a natural language sequence which allows DeepLog to learn the recorded system log patterns. The patterns learned will be compared to the patterns from normal executions to detect the outliers after comparison.

SoGaal

SoGaal [33] stands for Single-Objective Generative Adversarial Active Learning, it utilizes generative adversarial network(GAN) to adapt into outlier detection which directly generates informative possible outliers. It contains a generator and a discriminator to actively generate outliers. SoGaal heavily relies on prior knowledge to perform well on time-series outlier detection tasks.

MoGaal

MoGaal [33] stands for Multiple-Objective Generative Adversarial Active Learning, it utilizes generative adversarial network(GAN) to adapt into outlier detection which directly generates informative possible outliers. It contains a generator and a discriminator to actively generate outliers. It is an extension of the SoGaal method to prevent the generator from falling into the mode collapsing problem, in which MoGaal will generate a good distribution of references throughout the input dataset.

4.2 Results

As shown in Figure 4.2, we compared our AutoML AutoEncoder method to the vanilla AutoEncoder and VariationalAutoEncoder, and we compred our AutoML RNN method to the DeepLog, Telemanom, SoGaal, and MoGaal methods, on the Server Machine Dataset.

Also shown in Figure 4.3, we compared our AutoML AutoEncoder method to the vanilla AutoEncoder and VariationalAutoEncoder, and we compred our AutoML RNN method to the DeepLog, Telemanom, SoGaal, and MoGaal methods, on the UCR Time Series Dataset.

SMD Dataset

Metrics/Algorithms	AE	VAE	AE-AutoML	DeepLog	Telemanom	SoGaal	MoGaal	RNN-AutoML
Macro-F1	0.57	0.56	**0.65**	0.55	0.49	0.47	0.47	**0.65**
AUC	**0.72**	0.71	0.67	0.54	0.51	0.44	0.44	**0.67**
Accuracy	**95.0**	91.4	91.1	87.6	88.8	85.1	86.2	**90.7**

Figure 4.2: Performance Comparison on SMD Dataset

UCR Dataset

Metrics/Algorithms	AE	VAE	AE-AutoML	DeepLog	Telemanom	SoGaal	MoGaal	RNN-AutoML
Macro-F1	**0.49**	0.47	0.48	0.46	0.35	0.47	0.50	**0.51**
AUC	**0.49**	0.48	**0.49**	0.49	0.33	0.45	0.51	**0.58**
Accuracy	**97.5**	94.1	95.7	81.7	59.3	89.1	**91.2**	90.4

Figure 4.3: Performance Comparison on UCR Dataset

For each dataset, we provided three evaluation metrics to compare the results as shown in above

tables, including Macro-F1 score, area under curve score, and the accuracy score.

4.2.1 Performance Comparison

The experiment measures the use-fullness of AutoML in time-series outlier detection with efficient searching. We compare the best architecture found by AutoML after searching a certain number of trials. For testing purposes, we searched ten trails on each dataset. For each trial, the number of epochs is fixed at 30 for fairness. The batch size is also fixed at 128 for fairness.

For the comparison results shown in Figure 4.2, we compared AKAE against vanilla AE and VAE with three metrics. We compared AKAE with these two algorithms because they all shared the same Encoder-Decoder structure, which is significantly different from others. In the SMD dataset, AKAE had the best F1 score out of all three algorithms, with an increased performance of about 15%, which is the most important metric to look at. Additionally, AE performed the best on AUC and Accuracy metrics. However, it did not beat VAE or AKAE by a large margin, therefore are no significant differences between these two metrics.

For the comparison results shown in Figure 4.2, we compared AKRNN against DeepLog, Telemanom, SoGaal, and MoGaal, with three metrics. We compared AKRNN with these four algorithms because both DeepLog and Telemanom utilize LSTMs, which are similar to AKRNN. Additionally, SoGaal and MoGaal are GAN methods designed specifically for outlier detection, so I would like to see the comparison between these methods. In the SMD dataset, AKRNN had the best F1 score, AUC score, and Accuracy score, out of all five algorithms, with an increased performance of about 30% for the F1 score and AUC score and just a slight improvement in accuracy.

For the comparison results shown in Figure 4.3, we compared AKAE against vanilla AE and VAE with three metrics. We compared AKAE with these two algorithms because they all shared the same Encoder-Decoder structure, which is significantly different from others. In the UCR dataset, AKAE had the best AUC score out of all three algorithms. Additionally, AE also per-

formed the best on Macro-F1, AUC, and Accuracy metrics. However, the results for AE, VAE, and AKAE were extremely close to each other, and we are confident to say it all performed about the same. However, we were able to do it with less time and effort using NAS.

For the comparison results shown in Figure 4.3, we compared AKRNN against DeepLog, Telemanom, SoGaal, and MoGaal, with three metrics. We compared AKRNN with these four algorithms because both DeepLog and Telemanom utilize LSTMs, which are similar to AKRNN. Additionally, SoGaal and MoGaal are GAN methods designed specifically for outlier detection, so I would like to see the comparison between these methods. In the UCR dataset, AKRNN had the best F1 score and AUC score, and MoGaal was the second-best performing result. However, Telemanom and DeepLog had poor results compared to AKRNN. All the other four methods were designed with lots of time and effort to become one the state-of-the-art methods in time-series outlier detection, and we were able to beat them by simply enabling AutoML searching with RNN, which shows that our method is very promising.

4.2.2 Searching Path

4.2.2.1 General Searching Path

The searching path of AKAE and AKRNN follows the fashion of AutoKeras. We need to first set the max number of trials NAS can search until it stops and outputs the best model. For each trial it will train with user specified number of epochs and batch size. As show in Figure 4.4, this is the first trail of the searching process. On the left,it will show the hyperparameters chosen for this specific trail, in the middle it will show the best value so far for each single hyperparameter. Since this is the first trail, we do not have any values in the middle part yet. Lastly, on the right it shows the corresponding hyperparamter name within the search space.

Next, as shown in Figure 4.5, this is the output of the first trail, it shows the time taken for this

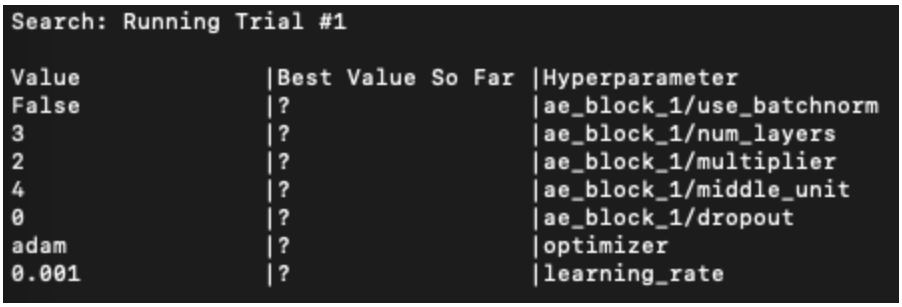

Figure 4.4: Step 1 of AutoML Searching Path

trail, and the MSE score for this trail. It also shows the best MSE so far, since we have only had one trail so far, the best MSE so far will be the MSE from the first trail. After this output shows up, the second searching trail will begin.

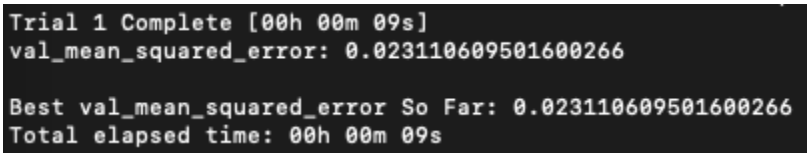

Figure 4.5: Step 2 of AutoML Searching Path

As shown in Figure 4.6 the next searching trail started and the hyperparameters on the left side changed for this trail. So far the architecture is still a small sized architecture.

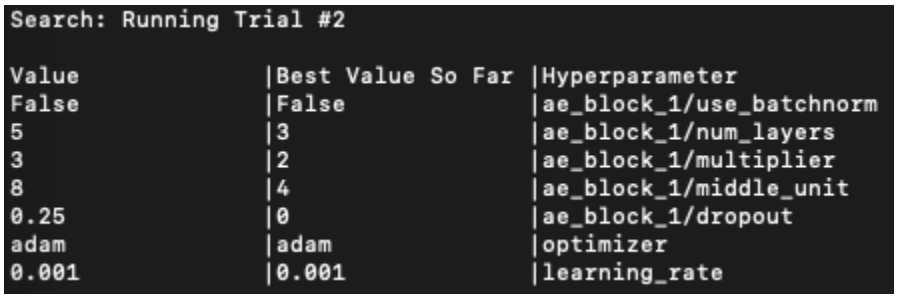

Figure 4.6: Step 3 of AutoML Searching Path

However, look at Figure 4.7 the architecture during the fifth trail has grew a lot bigger.

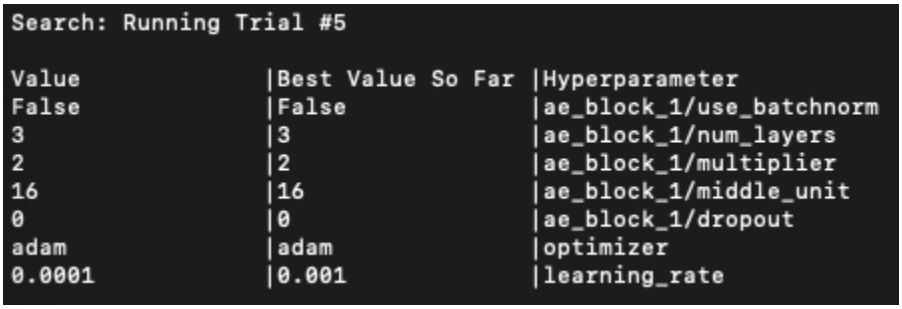

Figure 4.7: Step 4 of AutoML Searching Path

4.2.2.2 Example Searching Case

Let's take a look at a real process of the searching process, in this example we are using AKAE to demonstrate the searching process step by step:
As shown in Figure 4.8, this is the initial trail of the searching process, we have a predefined architecture for this trail with its corresponding hyperparameters like "num_layers", "multiplier" and "middle_unit". During this searching step, the predefined architecture is a pretty small sized architecture.

As shown in Figure 4.9, this is the 6th trail of the searching process, you can see that the best value so far has been changed from the first trail shown in 4.8, and the predefined architecture for this 6th trail is bigger than the 1st trail.

As shown in Figure 4.10, this is 9th trail of the searching process, the current trail has a very large predefined architecture and it is being compared to the architecture from the 6th trail shown in Figure 4.9, the model will be trained and compared its performance against the previous-best trail on this large predefined architecture.

```
Search: Running Trial #1

Value             |Best Value So Far |Hyperparameter
False             |?                 |ae_block_1/use_batchnorm
5                 |?                 |ae_block_1/num_layers
2                 |?                 |ae_block_1/multiplier
4                 |?                 |ae_block_1/middle_unit
0.25              |?                 |ae_block_1/dropout
adam              |?                 |optimizer
0.001             |?                 |learning_rate

Predefined Model Architecture for this Trial:
 [16, 8, 4, 8, 16]
Number of Units in this current layer:
 16
Number of Units in this current layer:
 8
Number of Units in this current layer:
 4
Number of Units in this current layer:
 8
Number of Units in this current layer:
 16
Epoch 1/30
593/593 [==============================] - 2s 2ms/step - loss: 0.0292
Epoch 2/30
593/593 [==============================] - 1s 1ms/step - loss: 0.0281
```

Figure 4.8: Step 1 of AutoML Searching Path Example

Lastly, as shown in Figure 4.11, after searching all trails, the best performance architecture is shown, we can see that the best isn't the biggest architecture we searched, it was the middle-sized architecture from the 6th trail. After all trails, it will trail on 30 epochs to evaluate the best performance architecture out of the 10 searching trails.

4.2.2.3 Searching Output

During the model searching process, for each trail searched, the information will be stored as output files in the current directory. As shown in Figure 4.12, the root folder named "auto_model" has the "best_model" folder which contains the best performing trail throughout the searching process. It also contains all information of the other searching trails, named "trail_num".

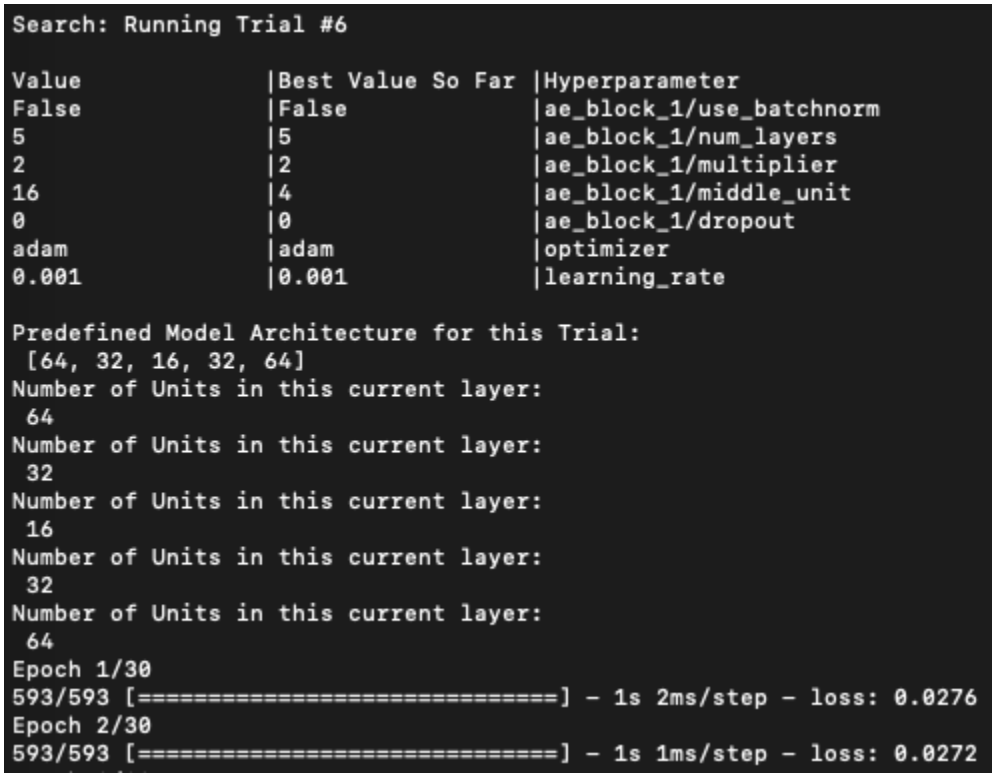

Figure 4.9: Step 2 of AutoML Searching Path Example

As shown in Figure 4.13, under the folder "best_model" there are files of the variables and the model to be used again. Under the "trail_num" folders, it contains the same information for this specific trail, and also the JSON information on the hyperparameters during this specific trail.

In Figure 4.14, we can see the JSON information on the hyperparameters for our model, which contains the default value and all other possible values.

Finally, in Figure 4.15, we can see the JSON information on the best performing trail with the best choices of the hyperparameters.

```
Search: Running Trial #9

Value             |Best Value So Far |Hyperparameter
False             |False             |ae_block_1/use_batchnorm
5                 |5                 |ae_block_1/num_layers
4                 |2                 |ae_block_1/multiplier
16                |16                |ae_block_1/middle_unit
0                 |0                 |ae_block_1/dropout
adam              |adam              |optimizer
0.001             |0.001             |learning_rate

Predefined Model Architecture for this Trial:
 [256, 64, 16, 64, 256]
Number of Units in this current layer:
 256
Number of Units in this current layer:
 64
Number of Units in this current layer:
 16
Number of Units in this current layer:
 64
Number of Units in this current layer:
 256
Epoch 1/30
593/593 [==============================] - 1s 1ms/step - loss: 0.0276
Epoch 2/30
593/593 [==============================] - 1s 1ms/step - loss: 0.0272
```

Figure 4.10: Step 3 of AutoML Searching Path Example

```
Best val_mean_squared_error So Far: 0.02306951768696308
Total elapsed time: 00h 02m 22s
Predefined Model Architecture for this Trial:
 [64, 32, 16, 32, 64]
Number of Units in this current layer:
 64
Number of Units in this current layer:
 32
Number of Units in this current layer:
 16
Number of Units in this current layer:
 32
Number of Units in this current layer:
 64
Epoch 1/30
741/741 [==============================] - 1s 868us/step - loss: 0.0280 - mean_squared_error: 0.0280
Epoch 2/30
741/741 [==============================] - 1s 893us/step - loss: 0.0274 - mean_squared_error: 0.0274
```

Figure 4.11: Step 4 of AutoML Searching Path Example

Figure 4.12: Output of Searching Results Part 1

4.3 Thoughts and Future Works

From our experiment results, we can confidently say that AutoML methods can either match or beat the state-of-the-art traditional methods. Combining AutoML with time-series outlier detection is a new research direction, and there aren't many findings in this area yet. The purpose of this thesis is to explore the potential and the limit of these two research areas combined together, to test the waters in this field. This was not just a research idea but also an engineering challenge since there are not any standard time-series outlier detection tools that have AutoML functionalities. We identified the challenges of combining these two topics together, and we solved part of the challenge by developing a method to predefine the architecture. I am very excited about the future of this area as I know many researchers are currently working on a better solution in AutoML for time-series outlier detection.

In the future, we could expand this work with more tasks. As we currently only support regres-

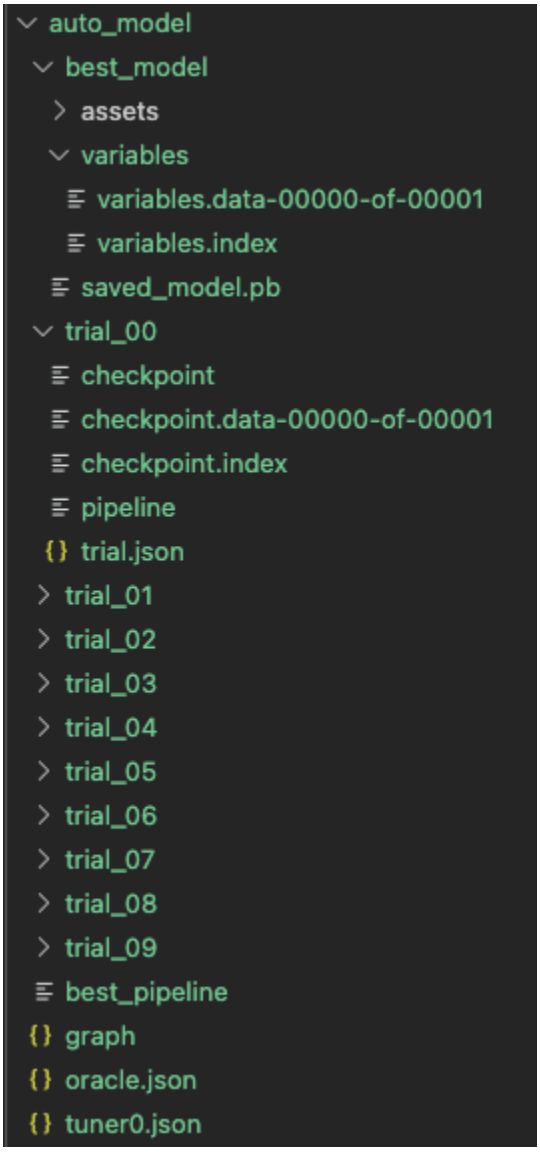

Figure 4.13: Output of Searching Results Part 2

sion tasks, we would like to add more tasks like classification. We would also like to expand our different types of neural networks, we only support AE and RNN so far, and something like CNN or Transformers would be great to have. Finally, a better solution to limit the suitable search space size for NAS is needed if AutoML in time-series outlier detection methods are being scaled up in the industry.

```
"hyperparameters": {
    "space": [
        {
            "class_name": "Boolean",
            "config": {
                "name": "ae_block_1/use_batchnorm",
                "default": false,
                "conditions": []
            }
        },
        {
            "class_name": "Choice",
            "config": {
                "name": "ae_block_1/num_layers",
                "default": 5,
                "conditions": [],
                "values": [
                    3,
                    5,
                    7
                ],
                "ordered": true
            }
        },
        {
            "class_name": "Choice",
            "config": {
                "name": "ae_block_1/multiplier",
                "default": 2,
                "conditions": [],
                "values": [
                    2,
                    3,
                    4
                ],
                "ordered": true
            }
        },
        {
            "class_name": "Choice",
            "config": {
                "name": "ae_block_1/middle_unit",
                "default": 4,
                "conditions": [],
                "values": [
                    4,
                    8,
                    16
                ],
                "ordered": true
            }
```


Figure 4.14: Output of Searching Results Part 3

```
"values": {
    "ae_block_1/use_batchnorm": false,
    "ae_block_1/num_layers": 5,
    "ae_block_1/multiplier": 2,
    "ae_block_1/middle_unit": 16,
    "ae_block_1/dropout": 0.0,
    "optimizer": "adam",
    "learning_rate": 0.001
}
```


Figure 4.15: Output of Searching Results Part 4

5. CONCLUSION

In this thesis, we proposed a new method to solve one of the biggest challenges for deep automated learning in time-series outlier detection, which is finding a suitable search space. We combined TODS and AutoKeras to enable new primitives such as AKAE and AKRNN to do regression tasks for time-series outlier detection tasks. We compared our method against many state-of-the-art traditional methods like AutoEncoder, VariationalAutoEncder, DeepLog, Telemanom, SoGaal, and MoGaal. Our results were as expected and very promising. Our method could either match or outperform most of the traditional methods based on comparing evaluation metrics like F1 score, AUC score, and accuracy with less effort and time taken.

REFERENCES

[1] F. J. Anscombe, "Rejection of outliers," *Technometrics*, vol. 2, no. 2, pp. 123–146, 1960.

[2] W. H. Kruskal, "Some remarks on wild observations," *Technometrics*, vol. 2, no. 1, pp. 1–3, 1960.

[3] A. J. Fox, "Outliers in time series," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 34, no. 3, pp. 350–363, 1972.

[4] M. Gupta, J. Gao, C. Aggarwal, and J. Han, "Outlier detection for temporal data," *Synthesis Lectures on Data Mining and Knowledge Discovery*, vol. 5, no. 1, pp. 1–129, 2014.

[5] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 387–395, 2018.

[6] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2828–2837, 2019.

[7] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[8] K.-H. Lai, D. Zha, G. Wang, J. Xu, Y. Zhao, D. Kumar, Y. Chen, P. Zumkhawaka, M. Wan, D. Martinez, *et al.*, "Tods: An automated time series outlier detection system," in *Proceedings of the aaai conference on artificial intelligence*, vol. 35, pp. 16060–16062, 2021.

[9] P. Malhotra, L. Vig, G. Shroff, P. Agarwal, *et al.*, "Long short term memory networks for anomaly detection in time series," in *Proceedings*, vol. 89, pp. 89–94, 2015.

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

[11] S. Hawkins, H. He, G. Williams, and R. Baxter, "Outlier detection using replicator neural networks," in *International Conference on Data Warehousing and Knowledge Discovery*, pp. 170–180, Springer, 2002.

[12] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 world wide web conference*, pp. 187–196, 2018.

[13] S. Zhai, Y. Cheng, W. Lu, and Z. Zhang, "Deep structured energy based models for anomaly detection," in *International conference on machine learning*, pp. 1100–1109, PMLR, 2016.

[14] B. Zong, Q. Song, M. R. Min, W. Cheng, C. Lumezanu, D. Cho, and H. Chen, "Deep autoencoding gaussian mixture model for unsupervised anomaly detection," in *International conference on learning representations*, 2018.

[15] A. Lavin and S. Ahmad, "Evaluating real-time anomaly detection algorithms–the numenta anomaly benchmark," in *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*, pp. 38–44, IEEE, 2015.

[16] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[17] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 1946–1956, 2019.

[18] R. S. Olson and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Workshop on automatic machine learning*, pp. 66–74, PMLR, 2016.

[19] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," *Advances in neural information processing systems*, vol. 28, 2015.

[20] E. LeDell and S. Poirier, "H2o automl: Scalable automatic machine learning," in *Proceedings of the AutoML Workshop at ICML*, vol. 2020, 2020.

[21] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1487–1495, 2017.

[22] A. Thakur and A. Krohn-Grimberghe, "Autocompete: A framework for machine learning competition," *arXiv preprint arXiv:1507.02188*, 2015.

[23] T. Swearingen, W. Drevo, B. Cyphers, A. Cuesta-Infante, A. Ross, and K. Veeramachaneni, "Atm: A distributed, collaborative, scalable system for automated machine learning," in *2017 IEEE international conference on big data (big data)*, pp. 151–162, IEEE, 2017.

[24] J. Liang, E. Meyerson, B. Hodjat, D. Fink, K. Mutch, and R. Miikkulainen, "Evolutionary neural automl for deep learning," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 401–409, 2019.

[25] R. Negrinho, M. Gormley, G. J. Gordon, D. Patil, N. Le, and D. Ferreira, "Towards modular and programmable architecture search," *Advances in neural information processing systems*, vol. 32, 2019.

[26] J. Liu, S. Tripathi, U. Kurup, and M. Shah, "Auptimizer-an extensible, open-source framework for hyperparameter tuning," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 339–348, IEEE, 2019.

[27] A. Nanduri and L. Sherry, "Anomaly detection in aircraft data using recurrent neural networks (rnn)," in *2016 Integrated Communications Navigation and Surveillance (ICNS)*, pp. 5C2–1, Ieee, 2016.

[28] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: a big comparison for nas," *arXiv preprint arXiv:1912.06059*, 2019.

[29] R. Wu and E. Keogh, "Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[30] K.-H. Lai, D. Zha, J. Xu, Y. Zhao, G. Wang, and X. Hu, "Revisiting time series outlier detection: Definitions and benchmarks," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[31] H. A. Dau, A. Bagnall, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, and E. Keogh, "The ucr time series archive," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 6, pp. 1293–1305, 2019.

[32] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pp. 1285–1298, 2017.

[33] Y. Liu, Z. Li, C. Zhou, Y. Jiang, J. Sun, M. Wang, and X. He, "Generative adversarial active learning for unsupervised outlier detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1517–1528, 2019.