Hybrid Deep Neural Networks to Infer State Models of Black-Box Systems

Anonymous Author(s)

ABSTRACT

Inferring behavior model of a running software system is quite useful for several automated software engineering tasks, such as program comprehension, anomaly detection, and testing. Most existing dynamic model inference techniques are white-box, i.e., they require source code to be instrumented to get run-time traces. However, in many systems, instrumenting the entire source code is not possible (e.g., when using black-box third-party libraries) or might be very costly. Unfortunately, most black-box techniques that detect states over time are either univariate, or make assumptions on the data distribution, or have limited power for learning over a long period of past behavior. To overcome the above issues, in this paper, we propose a hybrid deep neural network that accepts as input a set of time series, one per input/output signal of the system, and applies a set of convolutional and recurrent layers to learn the non-linear correlations between signals and the patterns, over time. We have applied our approach on a real UAV auto-pilot solution from our industry partner with half a million lines of C code. We ran 888 random recent system-level test cases and inferred states, over time. Our comparison with several traditional time series change point detection techniques showed that our approach improves their performance by up to 102%, in terms of finding state change points, measured by F1 score. We also showed that our state classification algorithm provides on average 90.45% F1 score, which improves traditional classification algorithms by up to 17%.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software reverse engineering**; **Requirements analysis**.

KEYWORDS

Recurrent Neural Network; Convolutional Neural Network; Deep Learning; Specification Mining; Black-box Model Inference; Time series;

ACM Reference Format:

Anonymous Author(s). 2020. Hybrid Deep Neural Networks to Infer State Models of Black-Box Systems. In *ASE 2020: IEEE/ACM International Conference on Automated Software Engineering September 21–25, 2018, Melbourne, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ASE 2020, 21 - 25 September 2020, Melbourne, Australia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automated specification mining or model inference [47] is the process of automatically reverse engineering a model of an existing software system. Behavioral models (e.g., state machines) are typically inferred from a running system by abstracting the execution traces. The inferred models are useful artifacts in many use cases where the actual behavior (abstracted as the inferred model) of the system is needed for analysis, such as debugging [2, 51, 68], testing [18, 57, 66, 75], anomalous behavior detection [74], and requirements engineering [20].

Inferring a behavior model of a system in a black-box manner is particularly interesting. In many real-world applications, the large-scale system is built by integrating many off-the-shelf libraries that are only available as binaries (no source code access). Thus, from a system's point of view, knowing the exact behavior of the system including all the interactions between black-box units are needed for most run-time analysis.

Most current behavioral model inference techniques are dynamic analysis methods (usually are more accurate than static analysis for run-time behavior inference) that require source code instrumentation to collect execution traces [47]. These methods are usually helpful in unit-level analysis where the instrumentation is not expensive and access to the code is allowed for the unit under study. However, in the system-level, thorough instrumentation is more expensive (not limited to one unit) and might not be even possible for some units (black-box libraries). Therefore, for use cases such as system-level anomaly detection, testing, and debugging a black-box behavior model inference that works on readily available input/outputs of the system is crucial.

In this paper, we propose a dynamic analysis method to detect the internal state and the state changes in a black-box software system using deep learning. We collected the numerical values of the inputs and outputs of the system, in regular time intervals to create a multivariate time-series. A hybrid deep learning model (including convolution and recurrent layers) was then trained on these time-series to predict the state of the system at each point in time. The deep learning model automatically performs feature extraction making it way more effective and flexible compared to traditional methods. In addition, we do not make any assumption about statistical properties of the data which makes it applicable to a wide range of subjects.

We applied and evaluated this method on an auto-pilot software (AutoPilot) used in an Unmanned Aerial Vehicle (UAV) system developed by our industry partner (called PARTNER from now on due to the double-blind restrictions). We evaluated the method from two perspectives: how well the model can detect the point in time when a state change happens? (RQ1: Change Point Detection (CPD)), and how accurately it can predict which state the system is in, during the execution? (RQ2: State Classification).

Comparing our approach with state-of-the-art alternatives, the results show that our approach performs better in both change point and state detection. We observed 88.00% to 102.20% improvement in the F1 score of our CPD, compared to traditional CPD techniques. In addition, we saw a 7.35% to 16.83% improvement in the F1 score of our state detection, compared to traditional classification algorithms on a sliding window, over the data.

The contributions of this paper can be summarised as:

- Introducing the first (to the best of our knowledge) deep learning architecture to infer behavior models from black-box software systems.
- Empirically evaluating the model and achieving very high accuracy compared to baselines using a real-world and large-scale case study on a UAV auto-pilot system developed by our industry partner.

Note that we have made all our source code, models, and execution scripts available online[1], however, due to confidentiality, we can not make our dataset public.

The rest of this paper is organized as follows: In section 2 we explain further how and in which contexts this research can be beneficial. Then in section 3 we briefly explain some background material this work is based on. In section 4, we explain how our proposed model is designed. The way it was evaluated and the results are explained in section 5. Finally, related work reviewed in section 6, and some final remarks about the future work are made in section 7.

2 MOTIVATION

Black-box components are ubiquitous in software development. Reusing high-quality black-box units generally offers a better overall system quality and a higher productivity [23]. The black-box units can be as small as a reusable library, or as large as a framework (such as .NET before going open-source), or a complete piece of software such as a remotely hosted web service. There are also scenarios where the unit's source code is not black-box in general, but not accessible to a specific team that wishes to perform the dynamic analysis.

One particular interesting use case of system-level black-box analysis is inferring run-time state model of a control software systems, where inputs/outputs are signals to/from the system. These inputs/outputs are typically multivariate time series, which are already logged in such systems (no overhead for instrumentation). The goal is automatically detecting the high-level system state and its changes, over time.

As discussed in section 1, we partnered with an auto-pilot manufacturer and performed this study on their auto-pilot software (called AutoPilot in this paper). The goal was to determine the state of AutoPilot from its input/output signals, over time. In this scenario, the inputs are the sensor readings going into AutoPilot and the outputs are command signals sent to controller motors of the aircraft, showing AutoPilot's reaction to each input at each state. A state in this example is the high-level stage of a flight and a state change happens when the current input values in the current state trigger a constraint in the implementation that changes the way the output signals are generated.

[1]https://github.com/anonymous-author-ase/hybrid-net

In this example, the training set will consist of input and output values recorded during one execution of the system, as a multivariate time series, along with state ids (as labels) per time stamp. One execution of the AutoPilot will be the whole flight process that may go through a "take-off" until a successful "landing". Depending on the flight plan, AutoPilot goes through states such as "acceleration", "take-off", "climbing", "turning", "descending", etc.

During a flight, the AutoPilot monitors changes in the input values and makes adjustments to its outputs in order to hold some invariants (predefined rules). For example, if AutoPilot is in the "hold altitude" mode, it monitors the altimeter's readings and when it goes out of the acceptable range, proportionate adjustments to the throttle or the nose pitch will be made to get it back to the desired altitude. This is basically how a typical feedback loop controller, such as PID or its variations work [61]. When AutoPilot's state changes from "hold altitude" to "descend to X ft" state, the set of invariants that AutoPilot is trying to hold are changed. It means its reactions to variations in inputs will be different. In this example, a decreasing altimeter reading will not trigger an increase in the throttle anymore.

Looking at the time series, a domain expert can identify what the state of AutoPilot is, at each point in time; the labeling process. Now the goal is to automate this task on a test set (in practice, future flights), assuming a training set is labeled by the experts (they only need to identify the state change time stamps, during a flight).

This problem can be tackled in two ways. The first solution is to identify the time stamp that the state change happens (i.e., Change Point Detection: RQ1); The more advanced solution is to predict the exact state per time stamp (i.e., State Classification: RQ2). The classic CPD techniques on time series [73] are mainly applicable on univariate data or put assumptions on the input/output distributions, thus not applicable in our case with multivariate inputs and no assumptions or knowledge about the states' distribution. The classic state classification techniques in time series are also weak in that they fail to balance between considering long-term relations or acting locally. The ones that use a sliding window, for example, do not have a long-term memory. The ones that act on the whole data on the other hand are too coarse-grained and inaccurate for this task.

Therefore, the motivation for this study is to provide a black-box technique that can be applicable on both CPD and state classification problems, and overcome the limitations of the existing techniques, in terms of capturing the non-linear correlation between multivariate inputs and outputs as well as learning patterns over a long period of time. Our proposal, which will be explained in detail in section 4, leverages the power of a deep neural network (DNN) with two types of layers that are particularly useful for this problem: a) convolutional layers which discover latent features from the data effectively through parameter sharing and b) recurrent layers that play a significant role in problems dealing with time series as they can learn long-term dependencies and seasonalities in the data.

Though our motivational example, as well as our case study, are from the UAV auto-pilot domain, our proposed method can be adapted to be applied to similar black-box control software systems in domains such as IoT, intelligent video surveillance, and self-driving cars.

3 BACKGROUND

Unlike the numerous techniques in the literature for behavior model inference [19, 40, 48, 76] which abstract a set of execution traces into states, our approach requires consuming a multivariate time-series and detect the state changes across time and predict the exact state labels. Thus, in this section, we briefly explain the two main sets of relevant existing techniques for "Change Point Detection" and "State Prediction" in time-series that can serve as background for our approach.

3.1 Change Point Detection

A fundamental tool in time-series data analysis is Change Point Detection (CPD). It refers to the task of finding points of abrupt change in the underlying statistical model or its parameters that could be a result of a state transition [3]. There are plenty of CPD algorithms; many of which perform effectively on a subset of CPD problems with some assumptions. The assumptions can be of various types. For example, one may assume the time series has only one input variable (univariate) [26], there is only one changing point [6], or the number of change points is known beforehand [42], or they might assume some statistical properties on the data [15]. These are limiting factors, since many of these assumptions do not necessarily hold in our case. CPD techniques are categorized into two main groups: a) online methods that process the data in real-time and b) offline methods that start processing the data after receiving all the values [73]. Since our model inference use case of CPD can afford waiting to collect all historical training data, we only considered offline techniques.

In general, CPD algorithms consist of two major components: a) the search method and b) the cost function [73]. Search methods are either exact or approximate. For instance, Pelt is the most efficient exact search method in the CPD literature, which uses pruning [37]. Approximate methods include window-based [8], bottom-up [33], binary segmentation [67], and more. In the window-based segmentation a sliding window is rolled over the data and then sum of costs of left and right half-windows is subtracted from the cost of the whole window. When the difference gets significantly high it means that the discrepancy between left and right half of the window is high and therefore a change point probably lies right in the middle of the window. In the bottom-up method, the input signal is split into multiple smaller parts, then using a similarity measure adjacent segments are merged until no more merges are feasible. The binary segmentation method finds one change point and splits the input into two parts around that point and then recursively applies the same method on each part.

The cost functions are also quite various, from simply subtracting each point from the mean to much more complex metrics, such as auto-regressive cost functions [4], and kernel-based cost functions. Kernel-based costs can have a wide variety, since the kernel function can be almost arbitrary, however a handful of them such as linear and Gaussian kernels are among the most popular ones [73].

In the context of our paper, we need a CPD method with no assumption on data distribution, number of change points, etc. In addition, our CPD method should work on multivariate data, and be able to capture non-linear relations between signals. It also needs to be resilient to time lags between an input signal change and its effect on the output signal (and the systems state). There is no traditional CPD algorithms that covers all these requirements. Therefore, we propose a novel CPD techniques that is based on Hybrid DNNs and compare it with several existing CPD techniques as our baselines, which are explained in details in section 5.

3.2 Convolutional and Recurrent Neural Networks

In both our problems (CPD and state classification), we can see that the changes in signals are more informative than their absolute values. Therefore, applying a derivation operation (or more generally a gradient) seems like necessary, at some point in the processing. Farid and Simoncelli listed some discrete derivation kernels in their study [25], but to have a more generalized and more flexible notion of discrete derivatives, convolutions seems like a better choice to apply. Nowadays, applying convolutional filters on signals is pretty much a standard process in signal processing studies that leverage deep learning [53, 82, 85]. Convolutional neural networks (CNNs) can learn to find features in a multidimensional input while being less sensitive to the exact location of the feature in the input [43]. In the forward pass of a convolutional layer, multiple filters are applied to the input. It means that in a trained neural net, multiple features can be leaned in one single convolutional layer.

Recurrent neural networks (RNN) have shown great performance in analysing sequential data such as machine translation, time-series prediction, and time-series classification [16, 54, 56, 79, 82, 86]. RNNs can capture long-term temporal dependencies which is quite useful for solving our problem. [14] For example, they might learn that "climb" state in a UAV auto-pilot usually follows "take off". Therefore, while it is outputting "take off" it anticipates what the next state will probably be and as soon as its input features start shifting, it detects the onset of a state change. It will help the model to better predict the system's behavior and be quicker to detect state changes in a way that could hardly be achieved with classic methods. Therefore, in this paper, we combine the CNNs and RNNs to create what is known as a hybrid deep neural network [79] to use for both CPD and state classification problems, in our context.

4 HYBRID NEURAL NETWORK FOR STATE INFERENCE

In this section, we describe our proposed deep learning approach for the black-box state inference task, in details.

4.1 The Model Architecture

The goal of this study is to infer the states of a running software system, over time. Given that our assumption is we don't have access to the source code (or part of it), we only leverage the values of inputs and outputs of the system, over time. As can be seen in figure 1, we capture all the inputs and outputs of the system as a time series and then process it in a DNN. The architecture of our proposed model is a hybrid DNN which is inspired by models proposed in the field of Human Activity Recognition (HAR). This task is quite similar to the subject of our paper in the sense that they both take in a multivariate time series-data (from sensor readings) and output the state of the system that generated those readings (see section 6.3 for more details on HAR papers). This DNN is made of

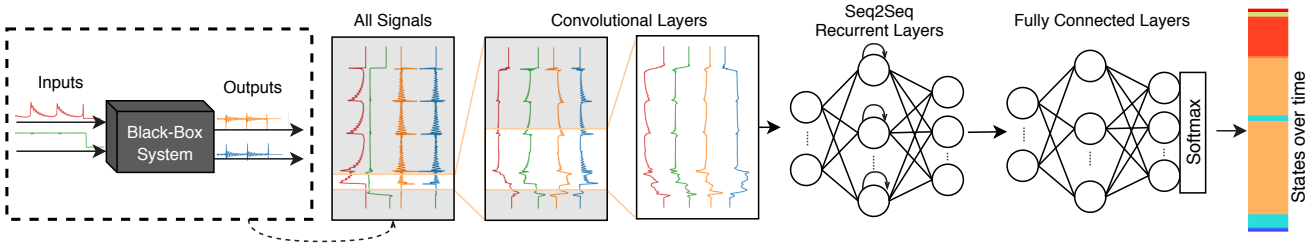

Figure 1: The input and output signals of the black-box system are captured as a multivariate time series; they are processed in a deep neural network that consists of 3 sections: convolutional, recurrent, and dense (fully connected) to predict the system's internal state and its changes over time.

three parts in sequence: 1) Convolutional, 2) Recurrent, and 3) Fully connected layers. This architecture addresses the aforementioned traditional methods' challenges; each part serves a different purpose in this process, as follows.

Convolutions, being more generalized than simple sliding windows, can discover patterns and features in the signals, both in temporal and in spatial (how signals affect each other) dimensions [79]. The convolutional layers' flexibility allows them to learn some typical preprocessing operations. For example a moving average or a discrete derivative can be learned as simple convolutional filters. They also help the model to be more resilient to varying time delays between noticing a deviation in input signals and the reaction that will appear in the output signals. Applying convolutional layers in sequence has been shown to result in each layer learning more complex features than the previous layers [84]. The number of layers, filters, and the kernel size are hyper-parameters that should be selected based on the size of data and the complexity of the system being modeled. Using a sequence of convolutions with a) increasing number of filters and the same kernel size, b) same number of filters and increasing kernel size, and c) decreasing filters with increasing kernel sizes are all different approaches that have been used in the literature by well-known architectures such as VGG and U-net [64, 69]. We will discuss more details of our CNN layers in Section 4.3.

Convolutions are quite powerful in discovering local features. To capture long-term features, recurrent layers which learn sequences of data are leveraged. For example, in our case, they can learn that "accelerate" and "take off" states only happen in the start of the states sequence, and each "take off" state is usually followed by a "climb" state. The type of recurrent cell to use (LSTM, GRU, etc.), how many cells to unravel in the layer, and the number of layers are also hyper-parameters that need to be tuned depending on the size and complexity of system under study.

Finally, one or more dense (fully connected) layers in the end are a common way of reducing the dimensions to match expected output dimensions. If there are only two states, the last layer can have a sigmoid activation function and be of shape L (the length of the input), otherwise, to match the one-hot encoding of labels, an output of shape $L \times N_s$ with softmax activation along the second axis (N_s) is required (N_s being the number of possible states).

In terms of loss function to optimize in the training process, a good choice is a dice overlap loss function, which is used in image semantic segmentation tasks, as well. An important property of this loss function is not getting negatively affected by class imbalances [52, 71].

4.2 Data Encoding

The input/output values of the black-box system create a multivariate time-series (T_k), which can be defined as a set of n univariate time series (V_i) of the same length l_k. Each V_i corresponds to the recorded values for one of the inputs or outputs of the system:

$$T_k = \{V_{1k}, V_{2k}, \ldots, V_{nk}\} \tag{1}$$

$$|V_{1k}| = |V_{2k}| = \ldots = |V_{nk}| = l_k \tag{2}$$

Note that, as figure 1 shows, we take both inputs and outputs as part of the time-series data to be fed as input into our deep learning models. This is to make sure we can model state-based behavior of the system, where the current state depends not only on the inputs, but also on the last state(s) (captured as previous outputs) of the system. As an example, from our case study, if the outputs are not taken into account a mid-flight "descend" state and the "approach" state right before landing are indistinguishable, using the sensor readings (inputs) alone.

Having such a time-series, the only remaining pieces from a training set are the labels. Unlike the input/output values (the features in the data set) the labels are not usually given. Our method to infer the labels is a supervised approach. Thus, we need the domain expert to manually label each individual time stamp with a state name/ID. In practice, what they would do is to identify the approximate time that a state change happens and assign the new state to one of the previous states labels or define a new label for this new state. Thus we encode the states information over time as a set of tuples in the form of (t_s, s) where t_s denotes the timestamp where the system entered state s. We show the set of all possible states with S ($s \in S$) and define N_s as the cardinality of this set.

$$\begin{aligned} CP_k &= \{(t_{s_1}, s_1), (t_{s_2}, s_2), \ldots, (t_{s_l}, s_l)\}, \; s_i \in S \\ N_s &= |S| \end{aligned} \tag{3}$$

So in summary, the dataset consists of N pairs of the I/O values as features and their state information as labels $\{(X = T_k, \; y = CP_k) | 1 \leq k \leq N\}$.

4.2.1 Data Preprocessing. Before being fed into the model $\mathcal{F}$ (as defined below), the inputs and labels need some preprocessing.

$$\mathcal{F}(\delta(T), m) \colon \mathbb{R}^{L \times n} \times \mathbb{R}^{L} \rightarrow S^{L}. \tag{4}$$

To run more efficiently, TensorFlow expects all the inputs to have the same length. To do that, the shorter T_ks should be zero-padded to length $L = max\{l_k\}$. The padding function δ does that. Therefore, eventually, the input to the model will be T_ks that are rearranged to form a tensor of shape $n \times L$ along with a padding mask (denoted with m). The mask tells the model where the tail starts so the model can ignore all the zeros from there on.

$$\begin{aligned} &\hat{O} = \langle \hat{o}_i \in S \rangle_{i=1}^{L} = \mathcal{F}\left(\left[\delta(V_1)^\intercal \ \delta(V_2)^\intercal \ \dots \ \delta(V_n)^\intercal\right], m\right) \\ &m = \delta(\vec{\mathbb{1}}_l) \quad \text{i.e.} \quad \langle m_j \rangle_{j=1}^{l} = 1\,,\ \langle m_j \rangle_{j=l+1}^{L} = 0 \end{aligned} \tag{5}$$

Here l denotes the length of the input before padding. It is equal to l_k for the kth training data (T_k).

As defined in (3), CP_ks are tuples of (t, s) which indicate the system have gone into state s at time t. To train the model, CP_k needs to be expanded into a vector of length L denoted by O where each element o_t holds the state at time t. To define it formally, the elements can be derived from CP_k using the following formula:

$$\begin{aligned} O = \langle \forall t \in \mathbb{N}_L : s_i \mid (t_{s_i}, s_i) \in CP_k \wedge \\ t_{s_i} = max\{t_{s_j} \mid (t_{s_j}, s_j) \in CP_k \wedge t_{s_j} \leq t\} \rangle \end{aligned} \tag{6}$$

For example: Suppose $L = 10$ and $CP = \{(0, a),\ (3, b),\ (5, c),\ (8, a)\}$ then $O = \langle a\ a\ a\ b\ b\ c\ c\ c\ a\ a \rangle$. If there are more than two possible states ($N_s > 2$), O needs to be one-hot encoded, at this stage.

4.3 The Model Implementation

The first few layers of the model are convolutional layers. We have used 5 convolutional layers with 64 filters each and a growing kernel size. The intuition behind this design is that starting with a small kernel guides the training in a way that the first layers learn simpler more local features that fits in their window (kernel size). Kernel sizes started with 3 since it is a common number in the literature for kernel sizes, then we used multiples of 5 from 5 to 20. The rationale behind choosing 5 is because the sampling frequency is 5, so each layer with a kernel size of $5n$ processes a whole n seconds worth of simulation data, in each step. Stopping at kernel size of 20 was a compromise between generalizability and model size. Generally, a larger model has more learning capacity, but it is also more prone to over-fitting. The current models are the smallest we could make the models (to avoid over-fitting), without compromising the performance.

Same compromise was made in the second section of the model (Recurrent layers), the sweet spot for hyper-parameters here was to use two GRU layers with 128 cells each. Their output was fed into a fully connected layer with 128 neurons with a leaky ReLU ($\alpha = 0.3$) activation function [50] and finally to a dense layer with $N_s = 25$ units with softmax activation. We used Adam optimizer [38] that could converge in 60-80 epochs, i.e. validation accuracy plateaued. The full architecture can be seen in figure 2.

5 EMPIRICAL EVALUATION

In this section, we explain our empirical evaluation of the proposed approach through a case study.

5.1 The Study Objectives

The goals of this study is to evaluate our proposed method in terms of change point detection and state inference, in comparison to traditional techniques in this domain. Therefore, our research questions are as follows:

5.1.1 RQ 1) How does our proposed technique perform in detecting the state changes? The goal of this RQ is to see how close the predicted state-change times are to the real state-change times. In other words, in RQ1, we do not predict the exact state labels and are only interested in predicting the change. To answer this question, we compare the performance of our proposed approach with several traditional baselines (see 5.3.1), in terms of modified precision, recall, and F1 scores that are introduced in section 5.2.1.

5.1.2 RQ 2) How well does our proposed technique predict the internal state of the system? In RQ1, we are only interested in detecting the time a state-change happens (binary classification), but here in RQ2, we extend that and are also interested in predicting the label of the new state that the system is going into (multi-class classification). Therefore, to answer this RQ, we change the labels from a Boolean (changed/not changed) to the actual collected labels.

Note that for both RQs, in our empirical study, to evaluate our approach, we use the source code to collect the exact time a state-change happens and the actual state labels (ground truth). However, in practice, labeling the training set is supposed to be done by the domain expert in a black-box manner. This is not an infeasible task or extra overhead. Monitoring the logs and identifying the current system state is in fact part of the developers/testers regular practice during inspection and debugging. All we provide here is a tool that given a partial labeling (only on the training set), automatically predict the state labels and the state-change times, for future flights. Also note that even though we use the source code to label the training set, we still look at the test set as a black-box and don't leak any information.

5.2 Evaluation Metrics

5.2.1 RQ1 (CPD) Performance Metrics. Given that in RQ1 there is an inherent class imbalance (there are far more points where a change has *not* happened compared to points with a state-change positive label), we avoid using accuracy and report both precision and recall. However, the original precision/recall metrics require some modifications due to the difficulty of predicting the exact time stamp that a state-change happened. To handle this, similar to related work [73], we use a tolerance margin τ. If a detected state-change ($\in \hat{CP}_k$) is within $\pm\tau$ of a true change ($\in CP_k$) , we call the prediction a True Positive, otherwise it is a False Positive. Similar adjustment to definition is applied for True Negative and False Negative. Formally speaking, we define predicted change points for k-th sample as:

$$\hat{CP}_k = \left\{(t, \hat{o}_t) \mid \hat{o}_t \neq \hat{o}_{t-1}\right\} \tag{7}$$

Please note that in (7), $\hat{o}_t$ refers to t-th element of output vector $\hat{O}$, as previously defined in (5). Based on that the confusion matrix elements are calculated as:

$$\begin{aligned} TP &= \left|\left\{(\hat{t}, \hat{s}_t) \in \hat{CP}_k \mid \exists\, (t, s_t) \in CP_k \text{ s.t. } |t - \hat{t}| < \tau\right\}\right| \\ FP &= \left|\left\{(\hat{t}, \hat{s}_t) \in \hat{CP}_k \mid \nexists\, (t, s_t) \in CP_k \text{ s.t. } |t - \hat{t}| < \tau\right\}\right| \\ FN &= \left|\left\{(t, s_t) \in CP_k \mid \nexists\, (\hat{t}, \hat{s}_t) \in \hat{CP}_k \text{ s.t. } |t - \hat{t}| < \tau\right\}\right| \end{aligned} \tag{8}$$

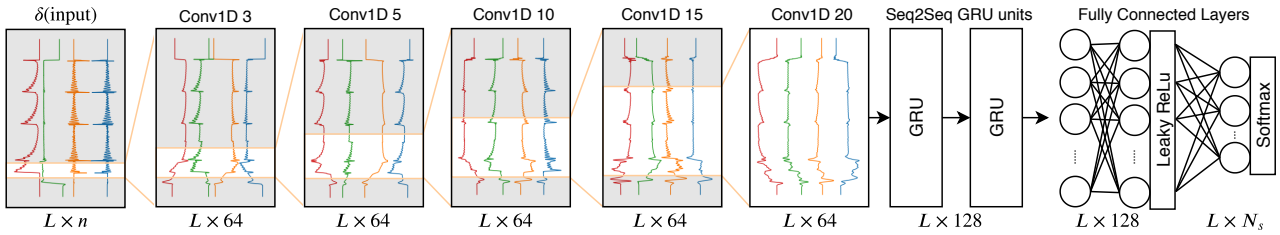

Figure 2: Model architecture in a nutshell. Tandem convolutional layers with increasing kernel size fed into two sequence-to-sequence recurrent layers with 128 GRU cells each, which is then fed into dense layers to output the predicted system state, as a list of one-hot encoded states. $\hat{O}$ will be the result of applying argmax operation on the last layer's output. $L = 18000$, $N_s = 25$

With these in mind, we measure precision, recall, and their harmonic mean F1 Score with three values for τ: 1, 3, and 5 seconds. The smaller the tolerance is the stricter the definitions become and the lower the numbers are.

5.2.2 RQ2 (State detection) metrics. In RQ2, we have a multi-class classification problem and thus multiple precisions/recalls will be calculated, one per class (state label). We then report the mean value across all classes.

$$\begin{aligned} P_s &= \{\hat{s}_t \in \hat{O}_k \mid \hat{s}_t = s\} \\ T_s &= \{s_t \in O_k \mid s_t = s\} \\ TP_s &= \{\hat{s}_t \in P_s \mid \hat{s}_t = s_t \in O_k\} \end{aligned} \tag{9}$$

$$Precision = \frac{1}{N_s}\sum_{s=1}^{N_s}\frac{|TP_s|}{|P_s|} \quad , \quad Recall = \frac{1}{N_s}\sum_{s=1}^{N_s}\frac{|TP_s|}{|T_s|}$$

5.3 Comparison Baselines

5.3.1 RQ1 (CPD) baselines. We used 'ruptures' library developed by authors of a recent CPD survey study [73]. It provides a modular framework for applying several CPD algorithms to univariate and multivariate data. As mentioned earlier two main elements of a CPD algorithm in their survey are the search method and the cost function.

We used Pelt [37] as the most efficient exact search method. As examples of approximate search methods, we applied bottom-up segmentation and window-based methods using a default window size of 100 [33]. However, after trying to run Pelt algorithm, we realized that it takes prohibitively longer to run compared to the approximate methods without providing much better results, so we only use the bottom-up and the window-based segmentation methods, as our CPD baselines.

For the cost function, we tried "Least Absolute Deviation", "Least Squared Deviation", "Gaussian Process Change", "Kernelized Mean Change", "Linear Model Change", "Rank-based Cost Function", and "Auto-regressive model change" as defined in the library. Their parameters were left as default. To optimize the number of change points a penalty value (linearly proportionate to the number of detected change points) is added to the cost function, which limits the number of detected change points, the higher the penalty the fewer reported change points. We tried three different ratios (100, 500, and 1000) for the penalty.

5.3.2 RQ2 (Multi-class classification) baselines. We used a sliding window of width w over the 10 time-series values and then flattened it to make a vector of size $10w$ as the features. For the labels, we used one-hot encoded state of the system. The window sizes were chosen as same as the sizes of convolutional layers' kernel sizes (3, 5, 10, 15, 20), to make the baselines better comparable with our method. We used Scikit-learn's implementation of the classification algorithms: A ridge classifier (Logistic regression with L2 regularization) and three decision trees. The ridge classifier was configured to use the built-in cross validation to automatically chose the best regularization hyper-parameter α in the range of 10^{-6} to 10^{6}. Each decision tree was regularized by setting "maximum number of features" and "maximum depth". For "maximum number of features" we tried no limits, $\sqrt{10w}$, and $\log_2 10w$. To find best "maximum depth" we first tried having no upper bound and observed how deep the tree grows; then we tried multiple numbers less than the maximum, until a drop in performance was observed.

5.4 Dataset and the Data Collection Process

We ran 948 existing test cases from our PARTNER's test repository using a software simulator[2] and collected the logged flight data, over time. The test cases are system-level tests. Each test case includes a flight scenario for various supported aircraft. A flight scenario goes through different phases in a flight such as "take off", "climb", "cruise", "hitting way points", and "landing". We sampled input and output values (listed in Table 1) at 5 Hz rate, which is the rate that AutoPilot reads the sensor values and performs the calculations required to update its output values at. Out of the 948 flight logs, we omitted 60 that were either too short or too long (shorter than 200 samples or longer than 20k samples). Figure 3 shows the distribution of the remaining log lengths. The maximum length (L) was 18,000 samples.

The dataset was randomly split into three chunks of 90%, 5%, and 5% for training, validation, and testing, where each sample corresponds to one test execution. Note that separate test and validation sets are needed to facilitate proper hyper-parameters tuning, without leaking information.

[2] It is developed by THE PARTNER COMPANY and provides an accurate simulation of the aerodynamic forces on the aircraft, the physical environment irregularities (e.g. unexpected wind gusts), and noises in sensor readings

[3] Above Ground Level

Table 1: The $n = 10$ collected I/Os of AutoPilot. The inputs are sensor readings and the outputs are the servo position update commands. All these I/Os over time are used as the inputs of the state prediction model.

Inputs	
Pitch	The angle that aircraft's nose makes with the horizon around lateral axis
Roll	The angle of aircraft's wings make with the horizon around longitudinal axis
Yaw	The rotation angle of aircraft around the vertical axis
Altitude	AGL[3] Altitude
Air speed	Speed of the aircraft relative to the air
Outputs	
Elevator	Control surfaces that control the Pitch
Aileron	Control surfaces that control the Roll
Rudder	Control surface that controls the Yaw
Throttle	Controller of engine's power, ranges from 0 to 1
Flaps	Surfaces of back of the wings that provide extra lift at low speeds, usually used during the landing

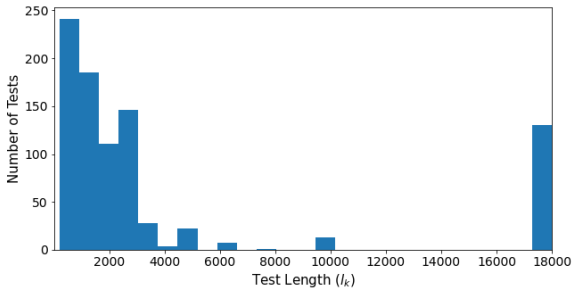

Figure 3: Distribution of flight log lengths for the $N = 888$ logs that were kept in the dataset ($200 \leq l_k \leq 20,000$)

5.5 Experiment Execution Environment

Training and evaluation of the deep learning model was done on a single node running Ubuntu 18.04 LTS (Linux 5.3.0) equipped with Intel Core i7-9700 CPU, 32 gigabytes of main memory, and 8 gigabytes of GPU memory on a NVIDIA GeForce RTX 2080 graphics card. The code was implemented using keras on TensorFlow 2.0.

The baseline models could not fit on that machine, so two nodes on Compute Canada's Beluga cluster, one with 6 CPUs and 75GiB of memory and one with 16 CPUs and 64GiB of memory, were used to train and evaluate them.

5.6 Results

In this section, we present the results of the experiments and answer the two research questions.

5.6.1 RQ1 Results: CPD Performance. Table 2 shows the results of running CPD algorithms for various configurations (as described in 5.3.1). For each search method and cost function pair only one of the penalty values which resulted in the highest F1 scores for all τ values is reported.

The first observation from the results is that as values of τ increases the scores get better. This was expected, since larger values relax the constraints on which detected change points are considered as a true positive. Another observation is that the bottom-up segmentation consistently outperforms the window-based segmentation method. We can also see that the linear cost function beats all the other ones, in terms of precision. The Gaussian cost function achieves much higher recall values costing it a huge loss in precision. It means this cost function results in detecting numerous change points spread across the time axis, so there is a good chance of having at least one change point predicted close to each true change point (hence the high recall), but also there are a lot of false positives, which leads to a low precision.

Measuring the same metrics on how our model performs on the test data shows better scores, almost twice the F1 score of the best performing baseline (see Table 3). Please note that unlike machine learning algorithms (such as ours), CPD algorithms do not have a separate training and testing phases. This fact works in their favor (by using the entire dataset for prediction and not just the training set), but still our model outperforms them.

In terms of execution cost, running all 42 different settings of CPD algorithms on the whole dataset took a bit over 12 hours in the cloud using 16 CPUs and 64GB of main memory. The deep learning model on the other hand takes about an hour to train (which only needs to be done once), on a smaller machine (see section 5.5). It made predictions on the whole dataset in less than a minute. So to answer RQ1, our method has shown $(66.18/32.73) - 1 = 102.20\%$ improvement in F1 score with $\tau = 1s$, $(78.06/41.52) - 1 = 88.00\%$ with $\tau = 3s$, and $(85.42/44.46) - 1 = 92.13\%$ with $\tau = 5s$; almost doubling the score compared to the baselines.

> Our model, which requires less memory compared to traditional CPD algorithms, improved their best performance by up to 102%, measured by F1 score, in less execution time.

5.6.2 RQ2 Results: Multi-class Classification Performance. To answer RQ2, we first compare different configurations of the baseline methods using the F1 score (harmonic mean of precision and recall) on the test data. The results are presented in Table 4.

Comparing the baseline methods with our proposed method (the last row) in Table 4 shows that our model outperforms all baselines. Comparing it with the model with the best F1-score shows a $(86.29/73.21) - 1 = 17.87\%$ improvement in precision as well as a $(95.04/82.16) - 1 = 15.68\%$ improvement in recall that means $(90.45/77.42) - 1 = 16.83\%$ overall improvement in F1-score.

To have a feeling of how good our predictions are in practice, Figure 4 shows the output of our model side by side with the ground truth. The horizontal axis shows sample ID (time) and the states are color coded. As it is seen, our algorithm performs better when the state changes are farther apart. Also there are some state changes that happen quite briefly which are not detected. That is not to

Table 2: Change point detection precision, recall, and F1-score calculated for the baseline methods using three values of tolerance (τ) for multiple configurations.

Cost Function	Search Method	Penalty	Prec.	Recall τ = 1s	F1	Prec.	Recall τ = 3s	F1	Prec.	Recall τ = 5s	F1
Autoregressive Model	Bottom Up	1000	10.43%	75.44%	18.33%	21.21%	80.32%	33.55%	28.94%	81.22%	42.68%
	Window Based	100	2.94%	3.98%	3.38%	8.53%	11.41%	9.76%	12.89%	17.54%	14.86%
Least Absolute Deviation	Bottom Up	500	7.32%	52.54%	12.85%	17.52%	87.73%	29.20%	25.02%	88.95%	39.05%
	Window Based	500	5.24%	8.31%	6.42%	15.20%	24.03%	18.62%	21.79%	38.25%	27.76%
Least Squared Deviation	Bottom Up	1000	7.44%	85.09%	13.68%	16.40%	89.81%	27.74%	24.16%	90.47%	38.14%
	Window Based	500	3.59%	6.79%	4.70%	10.27%	16.51%	12.66%	16.18%	26.84%	20.19%
Linear Model Change	Bottom Up	100	**37.59%**	28.98%	**32.73%**	**45.20%**	38.39%	**41.52%**	**48.07%**	41.36%	**44.46%**
	Window Based	500	6.70%	4.14%	5.12%	20.50%	13.05%	15.95%	38.78%	26.77%	31.67%
Gaussian Process Change	Bottom Up	100	3.77%	**92.23%**	7.25%	8.99%	**92.23%**	16.39%	13.53%	**92.23%**	23.60%
	Window Based	100	2.94%	3.95%	3.37%	8.69%	11.50%	9.90%	13.64%	18.30%	15.63%
Rank-based Cost Function	Bottom Up	100	13.45%	60.19%	21.98%	19.49%	80.10%	31.35%	22.98%	87.23%	36.38%
	Window Based	100	8.10%	13.70%	10.18%	15.72%	30.73%	20.80%	21.38%	46.64%	29.32%
Kernelized Mean Change	Bottom Up	100	4.13%	3.24%	3.63%	12.22%	8.14%	9.77%	15.38%	10.58%	12.54%
	Window Based	100	2.82%	3.00%	2.91%	10.14%	8.40%	9.19%	13.64%	12.61%	13.10%

Table 3: Change point detection precision, recall, and F1-score calculated, on the test data, for our proposed model, using three values of tolerance (τ) compared with the respective τ's best F1 score among baseline methods

τ	Prec.	Recall	F1 score	Baseline F1
1s	56.77%	79.32%	66.18%	32.73%
3s	69.58%	88.88%	78.06%	41.52%
5s	79.82%	91.87%	85.42%	44.46%

Table 4: Precision, recall, and F1 score of ridge classifiers (linear classifiers with L2 regularization) and decision tree classifiers (DT) with different sliding window widths (w). For each algorithm on each w several hyper-parameters were applied producing 152 different models. In this table, we only show the results of the best performing model in each group.

w	Classifier	Max Depth	Max Features	Prec.	Recall	F1
3	Ridge	-	-	71.39%	20.73%	32.13%
3	DT	-	-	69.21%	82.36%	75.21%
5	Ridge	-	-	69.15%	21.89%	33.26%
5	DT	100	-	68.37%	**83.16%**	75.04%
10	Ridge	-	-	71.97%	24.02%	36.02%
10	DT	260	-	67.94%	79.14%	73.12%
15	Ridge	-	-	76.87	25.90%	38.75%
15	DT	-	$\sqrt{10w}$	69.06%	80.76%	74.45%
20	Ridge	-	-	**80.38%**	26.50%	39.86%
20	DT	175	$\sqrt{10w}$	73.21%	82.16%	**77.42%**
	Our Proposed Method			**86.29%**	**95.04%**	**90.45%**

a great surprise since it takes some time for state changes to be reflected in the outputs and those might not have got any chance.

The classical models only see one window of the data at a time, convolutional layers on the other hand are more generalized and flexible since each filter in each layer is comparable to a sliding window. As we saw in Table 4, a larger window size means a higher performance. However, it gets significantly more difficult to train a model with large window sizes. In addition, convolutions can automatically learn preprocessing steps that could be beneficial such as a moving average. Each convolutional filter can learn a linear combination of its inputs. So when the convolutional layers are stacked on each other, with non-linear activation functions in between, the hypothesis space they can learn becomes quite large, probably much larger than most of the classical ML algorithms here. Also, they are still quite efficient (more efficient than baselines) due to parameter sharing and their high parallelizability.

The fact that the performance improves as the window size increases indicates the positive effect of being able to see longer-term relations in detecting the system's state. Recurrent cells (such as GRU) can capture long-term dependencies (that do not necessarily fall into one window) and learn sequences. This is one of the major differences between an RNN model and others, such as decision trees, which do not have such a notion of a "long-term memory" as LSTM/GRU neural networks do. All a decision tree could see is the values in a sliding window.

In terms of the training complexity (time and memory), our method is superior as well. That can largely be attributed to the use of deep learning. In baseline models, as the window size w grows the training and evaluation complexity grows, up to a point that they ran out of memory – consuming all the 47GB of main memory and swap area. This forced us to train them in the cloud. Meanwhile, as mentioned earlier, the deep learning model could be trained on a 8GB GPU in roughly an hour. (see section 5.5 for the machines' specs). Also, the decision tree training was not parallelized using

only one core of the CPU, while virtually all deep learning models can be heavily parallelized on a GPU/TPU.

> Our model, which requires less than half as many CPUs and 70% as much memory compared to the best performing classical ML model, improved their best performance by up to 17%, measured by F1 score, in less execution time.

5.7 Limitations and Threats to Validity

One of the limitations of our approach is that it might miss an input-output invariant correlation. It can happen when the input remains constant or it changes too little to reveal its relation with certain outputs. We assume that during the data collection, sampling happens in regular intervals; our approach probably will have a hard time achieving high performances, working on unevenly spaced time-series data.

In terms of construct validity, we are using standard metrics to evaluate the results. However, the use of tolerance margin should be taken with caution since it is a domain-dependant variable and can change the final results. To alleviate this threats, we have used multiple margins and reported all results. In terms of internal validity threats, we reduced the threat by not implementing the CPD baselines by ourselves and rather reusing existing libraries. In terms of conclusion validity threats, we have used many (888) real test cases from our PARTNER test repository and provided a proper train-validation-test split for training, tuning, and evaluation. Finally, in terms of external validity threats, our study suffers from being limited to only one case study. However, the study is a large-scale real-world study with many test cases. We plan to extend this work with more case studies from other domains, to increase its generalizability.

6 RELATED WORK

6.1 Time Series Change Point Detection

Change point detection is a well-studied subject due to its wide range of applications [8]. Several statistical and algorithmic methods have been tried to tackle several variations of this problem [15, 17, 28, 30, 41, 44, 55, 60, 63, 65, 78, 80, 81]. The models vary based on: whether the whole data is available at once (offline) or it is being generated on the go (online), whether there are statistical assumptions about the data distribution [31, 72], whether the number of change points is known [73], or whether we are dealing with a univariate or a multivariate time series, etc.

Ives and Dakos utilized locally linear models and used statistical significance test to determine at which point the changes in model parameters are large enough to signal a change in the state [32]. Blythe et al., used subspace analysis to reduce data dimensionality to keep the most non-stationary dimensions. This process helps detecting change points more effectively [13]. Several techniques have used penalty functions to find models that best fit each segment of the signal [34, 36, 41, 42, 58]. Desobry et al., and Hido et al., proposed methods to indirectly use classifiers such as SVM to detect change points [21, 29, 35]. We applied their approach on our data in early stages of the research but it could not perform as others. Lee et al., trained deep auto encoder networks that learns latent features in the data to detect change points [45]. Ebrahimzadeh et al., proposed what they call a pyramid recurrent neural network architecture, which is resilient to missing to detect patterns that are warped in time [22].

There are also a family of methods based on Bayesian models that focus on finding changes in parameters of underlying distributions of the data [1, 5, 7, 24, 45, 62].

Making assumptions about the data such as its distribution or the distribution of change points across the time and relying on basic statistical properties are the two major short comings of traditional CPD methods [45], which our proposed approach has overcome.

6.2 State Model Inference

Roughly speaking, dynamic EFSM[4] inference algorithms generally take a trace of "events" (along with perhaps some variable values) as their input [76] to infer a generalized finite state machine. They use the events to find the state transitions and the values for detecting invariants and generating the guard conditions on the transitions. k Tails, Gk Tail, EDSM, and MINT are examples of these algorithms, each improving upon the previous one [12, 40, 49, 76].

Walkinshaw et al., proposed an algorithm and developed a tool for state model inference [76]. Their work is based on previous endeavors on state merging algorithms such as gk-tail and k-tails [12, 49]. These methods require an execution trace of the program consisting of a list of "events" that occurred during program execution, such as function calls, system calls, transmitted network data, etc. Krka et al., performed an empirical study on 4 different categories of model inference algorithms to figure out what makes each group of methods more effective [39]. Beschastnikh et al., proposed a method to mine invariants from partially ordered logs from concurrent/distributed systems [10]. Invariants can be used to augment state models [9, 11]. Groz et al., use machine learning to heuristically infer state machine models of a un-resettable black-box system [27], however a significant difference between our method and theirs is that their method still relies on discrete events (such as HTTP request and responses) while our method does not assume that the input and outputs contain any kind of "events" happening at certain times. Our method aims to search for such events as change points in a continuous stream of data as time series.

6.3 Using Deep Learning on Time Series Data

Human activity recognition (HAR) is a well researched task which is quite relevant to the problem of black-box model inference. In HAR, just like in our context, a multivariate time series data is created from various sensors on a human body. The goal is to figure our what was the activity that human was performing in different time intervals. The sensors can be body worn accelerometers, or more generic sensors such as the ones found in a smart watch or a smartphone. Murad et al., [54] have shown deep RNNs outperform fully convolutional networks and deep belief networks in HAR task. Hybrid models are the combination of some deep architectures [77], such as a CNN + RNN or a CNN + a fully connected net. Morales

[4]Extended Finite State Machines, are special kind of state machines that have conditional expressions called "transition guards" on their transitions [49]. A state transition can only happen if the transition guard evaluates as true.

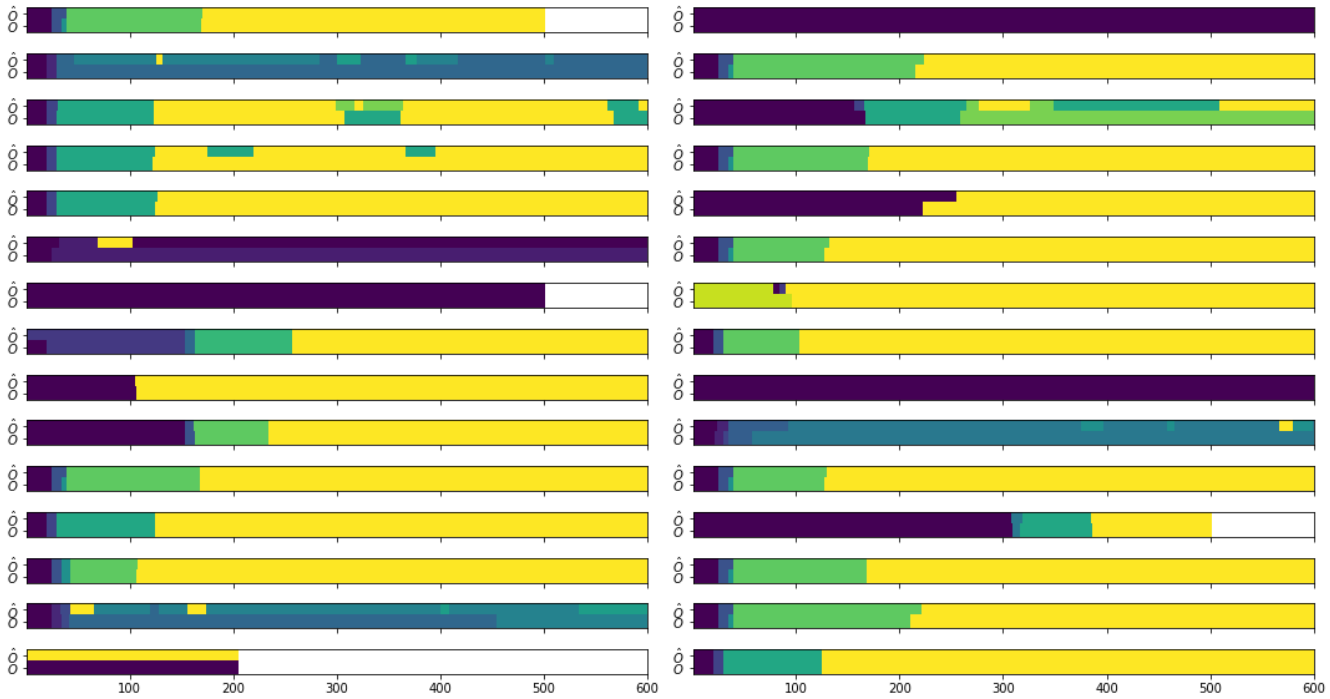

Figure 4: Evaluation of the model on 30 random test data. Each graph shows the states in one run of the system. The colors show the states. The top-half of each plot depicts model's prediction of the system states ($\hat{O}$) and the bottom-half shows the true labels(O). Since the output is one-hot encoded, the item with the most probability is used as the predicted label at each point in time. X-axis is the time axis. Only the first 600 samples (2 minute of simulation) are shown to improve legibility.

et al., have shown the former preforms better than the latter in HAR [53]. Yao et al., [83] introduced a CNN + RNN architecture that outperforms the state of the art both in classification and in regression tasks. Similar results have been shown in other works such as [56, 70, 87], as well.

Another related topic here is the time series classification. However, time series classification techniques often output only one label classifying entire data, thus not applicable in our context. What is more related to our problem is called "segmentation", using the computer vision terminology (not be confused with time series segmentation, such as [46]). U-net is one of the promising auto-encoder architectures for image segmentation [64]. Perslev et al., developed a similar idea for time-series to capture long-term dependencies and called it U-time [59]. It is fully convolutional and does not use memory cells (recurrent cells). A fully convolutional model can perform very well, since convolutions operate locally and image segments are large chunks of pixels in the 2D space and capturing local features using neighbouring pixels is quite useful. However, it cannot necessarily be as powerful on a more limited 1D data of time-series with different characteristics from an image. This study's design is optimized for the task of sleep phase detection, which does not have very clear boundaries between states and also the state changes are not very frequent. Therefore, the same method does not necessarily generalize to tasks such as ours, where we cannot make assumptions about frequency of state changes.

7 SUMMARY AND FUTURE WORK

In this paper, we introduced a hybrid CNN-RNN model that can be used for both CPD and state classification problems in multivariate time series. The proposed approach can be used as a black-box state model inference for variety of use cases such as testing, debugging, and anomaly detection in control software systems, where there are several input signals that control output states. We have evaluated our approach on a case study of a UAV auto-pilot software from our industry partner with 888 test cases and showed significant improvement in both change point detection and state classification. In the future, we are planning to extend this research with more case studies from open source auto-pilots. In addition, better tuning of hyper-parameters will be explored. Finally, we plan to examine the use of transfer learning to reduce the labeling overhead.

REFERENCES

[1] Ryan Prescott Adams and David JC MacKay. 2007. Bayesian Online Changepoint Detection. *stat* 1050 (2007), 19.
[2] Ziad A Al-Sharif. 2009. *An Extensible Debugging Architecture Based on a Hybrid Debugging Framework.* Ph.D. Dissertation. University of Idaho.
[3] Samaneh Aminikhanghahi and Diane J Cook. 2017. A survey of methods for time series change point detection. *Knowledge and information systems* 51, 2 (2017), 339–367.
[4] Daniele Angelosante and Georgios B Giannakis. 2012. Group lassoing change-points in piecewise-constant AR processes. *EURASIP Journal on Advances in Signal Processing* 2012, 1 (2012), 70.
[5] Jushan Bai. 1997. Estimation of a change point in multiple regression models. *Review of Economics and Statistics* 79, 4 (1997), 551–563.

[6] Jushan Bai, Robin L Lumsdaine, and James H Stock. 1998. Testing for and dating common breaks in multivariate time series. *The Review of Economic Studies* 65, 3 (1998), 395–432.

[7] Daniel Barry and John A Hartigan. 1993. A Bayesian analysis for change point problems. *J. Amer. Statist. Assoc.* 88, 421 (1993), 309–319.

[8] Michèle Basseville, Igor V Nikiforov, et al. 1993. *Detection of abrupt changes: theory and application*. Vol. 104. prentice Hall Englewood Cliffs.

[9] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. 468–479.

[10] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, Arvind Krishnamurthy, and Thomas E Anderson. 2011. Mining temporal invariants from partially ordered logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. 1–10.

[11] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 267–277.

[12] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers* 100, 6 (1972), 592–597.

[13] Duncan A.J. Blythe, Paul Von Bunau, Frank C. Meinecke, and Klaus Robert Muller. 2012. Feature extraction for change-point detection using stationary subspace analysis. *IEEE Transactions on Neural Networks and Learning Systems* 23, 4 (apr 2012), 631–643. http://ieeexplore.ieee.org/document/6151166/

[14] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. 2018. Recurrent Neural Networks for Multivariate Time Series with Missing Values. *Scientific Reports* 8, 1 (dec 2018), 1–12. arXiv:1606.01865

[15] Jie Chen and Arjun K Gupta. 2011. *Parametric statistical change point analysis: with applications to genetics, medicine, and finance*. Springer Science & Business Media.

[16] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[17] Md Foezur Rahman Chowdhury, S-A Selouani, and D O'Shaughnessy. 2012. Bayesian on-line spectral change point detection: a soft computing approach for on-line ASR. *International Journal of Speech Technology* 15, 1 (2012), 5–23.

[18] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. 2011. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering* 38, 2 (2011), 243–257.

[19] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*. 17–24.

[20] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel Van Lamsweerde. 2005. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering* 31, 12 (2005), 1056–1073.

[21] Frédéric Desobry, Manuel Davy, and Christian Doncarli. 2005. An online kernel change detection algorithm. *IEEE Transactions on Signal Processing* 53, 8 (2005), 2961–2974.

[22] Zahra Ebrahimzadeh, Min Zheng, Selcuk Karakas, and Samantha Kleinberg. 2019. *Deep Learning for Multi-Scale Changepoint Detection in Multivariate Time Series*. Technical Report. arXiv:1905.06913v1

[23] Stephen H Edwards. 2001. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability* 11, 2 (2001), 97–111.

[24] Chandra Erdman and John W Emerson. 2008. A fast Bayesian change point analysis for the segmentation of microarray data. *Bioinformatics* 24, 19 (2008), 2143–2148.

[25] Hany Farid and Eero P Simoncelli. 2004. Differentiation of Discrete Multidimensional Signals. *IEEE TRANSACTIONS ON IMAGE PROCESSING* 13, 4 (2004), 496–508.

[26] Piotr Fryzlewicz et al. 2014. Wild binary segmentation for multiple change-point detection. *The Annals of Statistics* 42, 6 (2014), 2243–2281.

[27] Roland Groz, Adenilso Simao, Nicolas Bremond, and Catherine Oriat. 2018. Revisiting AI and testing methods to infer FSM models of black-box systems. In *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*. IEEE, 16–19.

[28] Abeer Hasan, Wei Ning, and Arjun K Gupta. 2014. An information-based approach to the change-point problem of the noncentral skew t distribution with applications to stock market data. *Sequential Analysis* 33, 4 (2014), 458–474.

[29] Shohei Hido, Tsuyoshi Idé, Hisashi Kashima, Harunobu Kubo, and Hirofumi Matsuzawa. 2008. Unsupervised change analysis using supervised learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 148–159.

[30] DA Hsu. 1982. A Bayesian robust detection of shift in the risk structure of stock market returns. *J. Amer. Statist. Assoc.* 77, 377 (1982), 29–39.

[31] Tsuyoshi Idé and Koji Tsuda. 2007. Change-point detection using krylov subspace learning. In *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM, 515–520.

[32] Anthony R. Ives and Vasilis Dakos. 2012. Detecting dynamical changes in nonlinear time series using locally linear state-space models. *Ecosphere* 3, 6 (jun 2012), art58. http://doi.wiley.com/10.1890/ES11-00347.1

[33] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2001. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining*. IEEE, 289–296.

[34] Hossein Keshavarz, Clayton Scott, and XuanLong Nguyen. 2018. Optimal change point detection in Gaussian processes. *Journal of Statistical Planning and Inference* 193 (2018), 151–178.

[35] Haidar Khan. 2019. *Predicting Change Points in Multivariate Time Series Data*. Ph.D. Dissertation. Rensselaer Polytechnic Institute.

[36] Haidar Khan, Lara Marcuse, and Bülent Yener. 2019. Deep density ratio estimation for change point detection. (2019). arXiv:1905.09876 [cs.LG]

[37] Rebecca Killick, Paul Fearnhead, and Idris A Eckley. 2012. Optimal detection of changepoints with a linear computational cost. *J. Amer. Statist. Assoc.* 107, 500 (2012), 1590–1598.

[38] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[39] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 178–189.

[40] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. 1998. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*. Springer, 1–12.

[41] Marc Lavielle. 1999. Detection of multiple changes in a sequence of dependent variables. *Stochastic Processes and their Applications* 83, 1 (sep 1999), 79–102. https://www.sciencedirect.com/science/article/pii/S030441499900023X

[42] Marc Lavielle. 2005. Using penalized contrasts for the change-point problem. *Signal processing* 85, 8 (2005), 1501–1510.

[43] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[44] Wei-Han Lee and Ruby B Lee. 2017. Implicit smartphone user authentication with sensors and contextual machine learning. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 297–308.

[45] Wei-Han Lee, Jorge Ortiz, Bongjun Ko, and Ruby Lee. 2018. *Time Series Segmentation through Automatic Feature Learning*. Technical Report. arXiv:1801.05394v2

[46] Daniel Lemire. 2007. A better alternative to piecewise linear time series segmentation. In *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM, 545–550.

[47] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. 2011. *Mining software specifications: methodologies and applications*. CRC Press.

[48] David Lo, Shahar Maoz, and Siau-Cheng Khoo. 2007. Mining Modal Scenario-Based Specifications from Execution Traces of Reactive Systems. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 465–468. https://doi.org/10.1145/1321631.1321710

[49] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*. 501–510.

[50] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, Vol. 30. 3.

[51] Mohammad Jafar Mashhadi, Taha R Siddiqui, Hadi Hemmati, and Howard Loewen. 2019. Interactive Semi-automated Specification Mining for Debugging: An Experience Report. *arXiv preprint arXiv:1905.02245* (2019).

[52] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. 2016. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)*. IEEE, 565–571.

[53] Francisco Javier Ordóñez Morales and Daniel Roggen. 2016. Deep convolutional feature transfer across mobile activity recognition domains, sensor modalities and locations. In *Proceedings of the 2016 ACM International Symposium on Wearable Computers*. 92–99.

[54] Abdulmajid Murad and Jae-Young Pyun. 2017. Deep recurrent neural networks for human activity recognition. *Sensors* 17, 11 (2017), 2556.

[55] Kyong Joo Oh and Kyoung-jae Kim. 2002. Analyzing stock market tick data using piecewise nonlinear model. *Expert Systems with Applications* 22, 3 (2002), 249–255.

[56] Francisco Ordóñez and Daniel Roggen. 2016. Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition. *Sensors* 16, 1 (jan 2016), 115. http://www.mdpi.com/1424-8220/16/1/115

[57] Petros Papadopoulos and Neil Walkinshaw. 2015. Black-box test generation from inferred models. In *Proceedings - 4th International Workshop on Realizing*

Artificial Intelligence Synergies in Software Engineering, RAISE 2015. IEEE, 19–24. http://ieeexplore.ieee.org/document/7168327/

[58] Florian Pein, Hannes Sieling, and Axel Munk. 2017. Heterogeneous change point inference. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 79, 4 (2017), 1207–1227.

[59] Mathias Perslev, Michael Jensen, Sune Darkner, Poul Jørgen Jennum, and Christian Igel. 2019. U-Time: A Fully Convolutional Network for Time Series Segmentation Applied to Sleep Staging. In *Advances in Neural Information Processing Systems*. 4417–4428.

[60] Rychelly Glenneson da S Ramos, Paulo Ribeiro, and José Vinícius de M Cardoso. 2016. Anomalies Detection in Wireless Sensor Networks Using Bayesian Changepoints. In *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 384–385.

[61] Karl J (Karl Johan) Åström. 2008. *Feedback systems : an introduction for scientists and engineers*. Princeton University Press, Princeton.

[62] Bonnie K Ray and Ruey S Tsay. 2002. Bayesian methods for change-point detection in long-range dependent processes. *Journal of Time Series Analysis* 23, 6 (2002), 687–705.

[63] Jaxk Reeves, Jien Chen, Xiaolan L Wang, Robert Lund, and Qi Qi Lu. 2007. A review and comparison of changepoint detection techniques for climate data. *Journal of applied meteorology and climatology* 46, 6 (2007), 900–915.

[64] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[65] David Rosenfield, Enlu Zhou, Frank H Wilhelm, Ansgar Conrad, Walton T Roth, and Alicia E Meuret. 2010. Change point analysis for longitudinal physiological data: detection of cardio-respiratory changes preceding panic attacks. *Biological psychology* 84, 1 (2010), 112–120.

[66] I. Schieferdecker. 2012. Model-Based Testing. *IEEE Software* 29, 1 (Jan 2012), 14–18. https://doi.org/10.1109/MS.2012.13

[67] Andrew Jhon Scott and M Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* (1974), 507–512.

[68] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 402–411.

[69] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[70] Monit Shah Singh, Vinaychandran Pondenkandath, Bo Zhou, Paul Lukowicz, and Marcus Liwickit. 2017. Transforming sensor data to the image domain for deep learning—An application to footstep detection. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2665–2672.

[71] Carole H Sudre, Wenqi Li, Tom Vercauteren, Sebastien Ourselin, and M Jorge Cardoso. 2017. Generalised dice overlap as a deep learning loss function for highly unbalanced segmentations. In *Deep learning in medical image analysis and multimodal learning for clinical decision support*. Springer, 240–248.

[72] Jun-ichi Takeuchi and Kenji Yamanishi. 2006. A unifying framework for detecting outliers and change points from time series. *IEEE transactions on Knowledge and Data Engineering* 18, 4 (2006), 482–492.

[73] Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2018. Selective review of offline change point detection methods. (jan 2018). arXiv:1801.00718 http://arxiv.org/abs/1801.00718

[74] Alfonso Valdes and Keith Skinner. 2000. Adaptive, model-based monitoring for cyber attack detection. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 80–93.

[75] Neil Walkinshaw. 2018. *Testing Functional Black-Box Programs Without a Specification*. Springer International Publishing, Cham, 101–120. https://doi.org/10.1007/978-3-319-96562-8_4

[76] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853.

[77] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. 2019. Deep learning for sensor-based activity recognition: A survey. *Pattern Recognition Letters* 119 (2019), 3–11.

[78] Yao Wang, Chunguo Wu, Zhaohua Ji, Binghong Wang, and Yanchun Liang. 2011. Non-parametric change-point method for differential gene expression detection. *PloS one* 6, 5 (2011).

[79] Zhiguang Wang, Weizhong Yan, and Tim Oates. 2017. Time series classification from scratch with deep neural networks: A strong baseline. In *2017 International joint conference on neural networks (IJCNN)*. IEEE, 1578–1585.

[80] Yao Xie and David Siegmund. 2013. Sequential multi-sensor change-point detection. In *2013 Information Theory and Applications Workshop (ITA)*. IEEE, 1–20.

[81] Kenji Yamanishi, Jun-Ichi Takeuchi, Graham Williams, and Peter Milne. 2004. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Data Mining and Knowledge Discovery* 8, 3 (2004), 275–300.

[82] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. 2015. Deep convolutional neural networks on multichannel time series for human activity recognition. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[83] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. 2017. DeepSense: a Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing. (2017). http://dx.doi.org/10.1145/3038912.3052577

[84] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[85] Ming Zeng, Le T Nguyen, Bo Yu, Ole J Mengshoel, Jiang Zhu, Pang Wu, and Joy Zhang. 2014. Convolutional neural networks for human activity recognition using mobile sensors. In *6th International Conference on Mobile Computing, Applications and Services*. IEEE, 197–205.

[86] Jia-Shu Zhang and Xian-Ci Xiao. 2000. Predicting chaotic time series using recurrent neural network. *Chinese Physics Letters* 17, 2 (2000), 88.

[87] Yi Zheng, Qi Liu, Enhong Chen, Yong Ge, and J Leon Zhao. 2016. Exploiting multi-channels deep convolutional neural networks for multivariate time series classification. *Frontiers of Computer Science* 10, 1 (2016), 96–112.