Walkowiak Tomasz, 0000-0002-7749-4251
Wrocław University of Science and Technology, Wrocław, Poland, Tomasz.Walkowiak{at}pwr.edu.pl

Mazurkiewicz Jacek, 0000-0002-7708-907X
Wrocław University of Science and Technology, Wrocław, Poland, Jacek.Mazurkiewicz{at}pwr.edu.pl

Sugier Jarosław, 0000-0003-1452-3067
Wrocław University of Science and Technology, Wrocław, Poland, Jaroslaw.Sugier{at}pwr.edu.pl

Śliwiński Przemysław, 0000-0003-3839-1580
Wrocław University of Science and Technology, Wrocław, Poland, Przemyslaw.Sliwinski{at}pwr.edu.pl

Performance analysis of intelligent agents in complex event processing systems

Keywords

complex event processing system, intelligent agent, deep learning, performance analysis

Abstract

The chapter discusses the performance aspects of intelligent agents in Complex Event Processing (CEP) systems. The contemporary solution for implementing CEP systems is based on available software components (Siddhi) and modern implementation techniques (Kubernetes). However, Siddhi lacks the implementation of modern deep learning algorithms. Hence, the concept of intelligent agent is introduced. A case study with a set of intelligent agents designed to handle real-world events related to environmental data monitoring is presented. The results of the case study discussion indicate a reasonable scale for tuning the Event Processing Element (EPA) topology with correct responses and the required output performance level. These results have important implications for the practical implementation of the EPA structure, i.e., the use of GPUs in CEP systems. Finally, the results of performance analysis of different implementations of intelligent agents are presented and discussed.

1. Introduction

The operation of various modern security monitoring systems is based on real time processing of stream of events coming from the environment. The functionality that the system must provide can be defined as detecting some specific temporal and semantic patterns in the stream of events, and then evaluating their various characteristics and generating appropriate responses because of their classification. It implies the usage of modern machine learning method as an intelligent element of the system. An event processing system forms a complex network of agents that process events and communicate with each other by sending generated event messages.

The ability to provide responses in a desired time frame and the ability to handle large, cumulative workloads (Sugier et al., 2019) is an important aspect of an event processing system, especially in the field of security monitoring of the environment characterized by different sources of events with various parameters.

Within this chapter the authors discus performance aspects of intelligent agents within Complex Event Processing (CEP) systems. The chapter is structured as follows. Firstly, we present the actual methods of deploying CEP systems. Next, we define the intelligent agent problem and the idea of CEP system creation using them. Section 4 presents the case study with the set of

Safety and Reliability of Systems and Processes, Summer Safety and Reliability Seminar 2022.
© Gdynia Maritime University. All rights reserved.
DOI: 10.26408/srsp-2022-14.

intelligent agents dedicated to real events caused the environmental data monitoring. Finally, performance analysis results are presented and discussed based on the working point of the system fixed by case study description.

2. Complex event processing system deployment

There are well-known methods and software environments for data streaming that can be useful for agent-to-agent dialogue in CEP systems. Siddhi is a native streaming and complex event processing engine in the cloud that understands streaming SQL queries to capture events from various data sources, process them, detect complex conditions, and publish the output to various endpoints in real-time. Siddhi is an event-driven system in which all the data it fetches, processes, and sends is modeled as events. Therefore, Siddhi can play an important role in any event-driven architecture. This software provides analytics operators to manage data flow, compute analytics results, and detect patterns on event data from a wide variety of live data sources, enabling developers to create applications that sense, think, and act in real time. So, we can create a type of processing where incoming event data is distilled down to more useful, higher-level complex event data that provides insight into what is happening. The computation is triggered by the reception of event data. They are used in highly demanding continuous intelligence applications that increase situational awareness and support real-time decision making. Streaming data integration is a way of integrating several systems by processing, correlating, and analyzing data in memory while continuously transferring real-time data from one system to another. Siddhi can continuously monitor event streams and send alerts and notifications. This paves the way for real-time dynamic decision making based on predefined rules and the current state of the connected systems (Suhothayan et al., 2011). Siddhi is equipped with a set of basic machine learning algorithms (Hastie et al., 2018), but their number is limited and do not include modern deep learning solutions. Therefore, there is a need to extend the Siddhi based CEP system with intelligent agents that could provide contemporary ML algorithms.
The CEP system can be seen as a collection of cooperating agents and therefore falls into the group of applications with microservice architecture (Wolff, 2016). The analyzed application area, i.e., security monitoring, requires high availability. A state-of-the-art solution for implementing microservices-based applications in high availability clusters is Kubernetes (Kubernetes, 2022). It is an open-source platform for automatically deploying, scaling, and managing containerized applications. Kubernetes provides the ability to heal deployed services by restarting failed containers and replacing or rescheduling containers when their hosts fail. However, restarting failed containers is not the only aspect of high availability. To handle large and cumulative workloads the performance of individual agents must also be ensured. This is especially important for intelligent agents executing computationally complex machine learning algorithms.

3. Types of intelligent agents

From the deployment and performance of CEP, intelligent agents could be divided into two groups: stateless and stateful. Stateless do not have any internal memory, they produce the output based on current inputs. Where stateful have an internal state that is changed during analysis of a new event and must be kept for processing of a new event.
Most of AI classification algorithms like multilayer perceptron, decision trees or SVM (Hastie et al., 2018) models are stateless agents. They are feed with input data and produce the output (i.e., the id of a class for classification tasks or a real number for regression problems). They have an internal state, the model fit during training, but its read-only and is not changing during generation of output (model inference).
The stateful agents works differently, they process a sequence of events by analyzing the current event and internal state produced during the previous event analysis. The Hidden Markov Models (Limnios & Oprisan, 2001) and LSTM recurrent networks (Greff et al., 2017) are examples of such style of processing. LSTM networks are very successful in any sequence analysis for example in natural learning processing (Peters et al., 2018), where text is seen a sequence of words of even letters. However, due to many hidden states require a lot of computing resources, especially in BI-LSTM (Greff et al., 2017) versions.
Note that stateless agents can work not only on the

current event, but also on a sequence of past events. Since most CEP engines feature event stream windowing (Suhothayan et al., 2011) it is possible to aggregate the historical event into the input to stateless intelligent agent. This can be done in sliding mode (overlapping windows) or in batch mode (windows with no overlapping events). This allows the intelligent agent to process data in stateless mode but make decisions based on a sequence of current and historical data. Feeding historical data to a classifier as separate values is a time series analysis scheme commonly used in machine learning. For example, if we assume that today temperature is a function of last n-days temperatures and the amount of precipitation in the previous days, we can feed the classifier with $2n$ inputs (vector consisting of past n-temperatures and past n-precipitation) (Abhishek et al., 2012). The SOTAs in image and text processing, i.e. Visual Transformers (Huang et al., 2020) and BERT (Devlin et al., 2019), are based on the transformer architecture. It extensively uses the attention mechanism that is based on generating output sequences as a weighted sum of processed sequence of inputs. So, a sliding window of historical input events allows to decide in a stateless mode.

As stated in the introduction, an important element of CEP is the speed of event processing. Algorithms used in intelligent agents require a lot of computation power Most of CEP are using messaging and queuing systems to manage communication between agents so stateless agents could be very easily scaled up and since the queuing systems works as a load balancer. However, it is not possible in case of stateful agents. So in this case, the performance is even more important aspect.

We propose to build intelligent agents – called EPA – Event Processing Element – as ontogenic neural networks. The main goal is to find an architecture that allows for the best generalization quality. The network must have a certain margin of freedom that lies in the adaptive parameters and allows the model states to change smoothly. Well-chosen margins of freedom together with model complexity control criteria also allow to fight the problem of local minima (Bonarini et al., 2003). A model changing its architecture moves to other adaptive parameter spaces with a different error function, where the learning process continues and new changes in the architecture are possible. In this way, such a learning model can explore different spaces in search of a certain optimum.

The methods for checking the complexity of network architectures can be divided into three groups:

- magnifying – these models include algorithms that allow to add new neurons or new connections among neurons,
- reducing – methods that remove unnecessary neurons or connections among neurons or algorithms that can join groups of neurons or connections between neurons,
- cooperative systems – groups of models, each to solve the subtask of the problem and the management system makes final decisions.

4. Case study

4.1. Topology tuning

Our intelligent agent – EPA – is based on reducing ontogenic neural network. The fully connected three-layer Multilayer Perceptron (MLP) is the starting point (Pratihar, 2009). The final topology is a result of neurons connection reducing based on the actual and previous answers of the EPA. It means we must store the history of the outputs and to take them into account for the decision which interconnection can be eliminated. The first possible approach for the reducing procedure is based on significance factor:

$$s_i = E(without_neuron_i) - E(with_neuron_i) \quad (1)$$

which determines the difference between a network error obtained without and with the participation of a neuron i. This method requires considerable calculation costs – to be determined for each coefficient s_i of equation (1) an error for the whole training set. Neurons with the less significance factors can be removed.

The similar – also passive – (2) way of the significance coefficients determining has been used in the FSM system (Feature Space Mapping) (Adamczak et al., 1997; Duch & Diercksen, 1997; Duch et al., 1995; Srivastava, 2008). Significance coefficients are determined for each hidden layer neuron after interrupting the learning process:

$$Q_i = C_i(\boldsymbol{X})/|\boldsymbol{X}| \quad (2)$$

where: $|\boldsymbol{X}|$ – number of training vectors, $C_i(\boldsymbol{X})$ – number of correct answers given by neuron i from

input set X. In FSM type network each neuron from hidden layer is responsible for the class. The neuron with Q_i close to zero is removed.
Methods that reduce the structure of a neural network can often be considered as regularization process. In the weight decay procedure (Hinton, 1987) for standard measure of error of model $E_0(f)$

$$E_0(f) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2 \quad (3)$$

the following factor is added (Weigend et al., 1990, 1991):

$$E_w(f) = E_0(f) + \lambda \sum_{i=1}^{M} \frac{w_i^2/w_0^2}{1+w_i^2/w_0^2} \quad (4)$$

where: w_0 – is constant parameter – the experiment shows – should be equal to one, if $/w_i/ >> w_0$ the factor goes to λ, if $/w_i/ << w_0$ goes to zero. The parameter λ can be tuned during learning process:

- $\lambda = \lambda + \Delta\lambda$ if $E_n < D$ or $E_n < E_{n-1}$,
- $\lambda = \lambda - \Delta\lambda$ if $E_n \geq E_{n-1}$ and $E_n \geq D$,
- $\lambda = 0.9\lambda$ if $E_n \geq E_{n-1}$ and $E_n < D$,

where: E_n – the last epoch error, D – final error for the training process. Finally, the training algorithm called Optimal Brain Damage (OBD) (Le Cun et al., 1990) looks as follows.

1. Set the starting topology.
2. Make the training process using classic gradient method until the error is acceptable and the changes are not important.
3. Calculate the significance factors taking into account the regularization parameters.
4. Remove the weights fixed to the extremely low values of significance factors. It means *turn-off* the neurons from the hidden layer.
5. If the weights are reduced go to Step 2.

Of course the reduced number of neurons is acceptable if the network answer is still correct from the functional point of view. If not it is obligatory to come back to the previous version of the topology. The OBD approach does not guarantee correct results of training procedure with the limited number of neurons (Weigend et al., 1990, 1991). It means the CEP construction have to provide the availability to the feedback signals and to preserve the earlier calculated results. In other words – our EPAs work not only on the current event but also on a sequence of past events and the dynamic structure of CEP is the key feature of the proposed solution.

4.2. EPA description

We decided to create four independent types of intelligent agents to deal with four types of event sensors:

- TS – two-states sensors responsible for simple events – like on-off, open-closed,
- AM – active movement sensors – output: distance to moving object,
- TM – temperature sensors – output: the actual temperature,
- BL – brightness level – output: actual brightness measured using the proper units.

Of course, these parameters can be exchanged to actual needs driven by the safety system features. Potential sources of events considered: access monitoring, area monitoring, mass messages, fire risk, internal notifications, video monitoring, communication channels, biometrics, access violation, temperature risk, gas hazard, acoustic threat, biological and medical risks, flood risk, open / close state, assembly / accumulation risk, system operator signal, user defined. A wide spectrum of potential sources of events does not exclude the use of a unified approach to the description of these events. We assume that the source – event generator – will determine the record of the given event in a kind of table. The number of description fields and their types are uniformly defined. Such approach will allow for the initial aggregation of events according to the reasons for their occurrence, as well as subsequent binding events in teaching vectors for intelligent information processing systems that will be used to accurately analyze the situation in the life of the system described by data recorded from many sensors.
The package of the following fields is stored in unified structure:

Event_ID, Source_Name, Source_ID,
Source_GPS, Object_ID, System_ID,
Event_Date, Event_Occurence, Event_Duration,
Event_Value, Event_Importance,
Event_Probability, Event_Type, Event_Info.

For the set of experiments 1000 records for each of four selected types of sensors were generated covering the wide spectrum of possible input data.

The data were used to prepare the training – 70% of population – and testing – 30% of population – vectors of 10 inputs:

Event_ID, Source_ID, Object_ID, System_ID, Event_Occurence, Event_Duration, Event_Value, Event_Importance, Event_Probability, Event_Type.

All introduced data were normalized – reflected from the original scale to [0,1] range. The training vectors were equipped with the correct answer – gradient methods of neural network training needs *the teacher* – as a source of expected output. We prepared four independent neural networks – four intelligent agents dedicated for each of four types of sensors. The output of each intelligent sensor is the value from 0 to 1 to describe the *importance level* of sensor reaction. The fifth neural network models the intelligent agent – as final voting element. Its input vector is created based on four intelligent agents responsible for sensors' data. This way the final output can be read as aggregated signal of alarm in the system. Of course, the final agent training procedure needs the expert answer if the actual input vector looks like the alarm situation. Such hierarchical structure of the single processing element gives a chance to create the flexible solution properly fixed to actual system needs (Fig. 1).

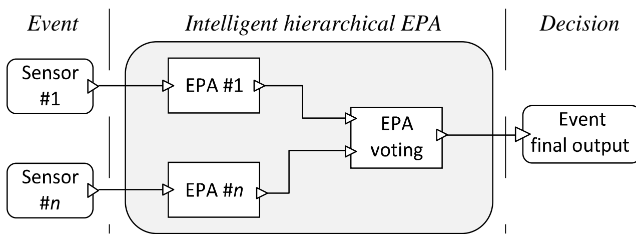

Figure 1. Intelligent hierarchical EPA.

4.3. EPA topology and training

The Event Processing Agents cooperating with the sensors are – initially – fully interconnected Multilayer Perceptron. Next, we transform them into ontogenic neural networks with flexible topology. The size of the input layer is equal to the size of the input vectors. It means we have 10 neurons there ready to input the float digits representing the components described in the previous subsection. The output layer has only one neuron to generate the answer of the net as the level of importance of the data driven by the input sensor. Of course, there is no problem to convert this fraction value to two state using simple threshold mechanism. The size of the single hidden layer is equal to 20 neurons as a starting value, but during the training procedure the number of active neurons is reduced by the Optimal Brain Damage (OBD) mechanism using regularization factors. This way the minimum number of working neurons in hidden layer is only 4. The training procedure was done for each EPA individually using the proper set of input vectors dedicated for each sensor. Each topology created as a result of OBD is trained again. The number of epochs is limited by the *no change* observation taking into account the network error minimizing. The final results for each EPA – and data from each sensor – are presented in Table 1.

Table 1. EPA correct answers [%] for different type of sensors and limited number of neurons in hidden layer as OBD mechanism result

Sensor	Distance type	Number of neurons in hidden layer – OBD result								
		4	6	8	10	12	14	16	18	20
TS	L_1	61	65	69	70	70	87	74	71	67
	L_2	56	56	60	62	67	76	71	73	69
	L_∞	47	51	53	57	62	65	61	57	55
AM	L_1	62	68	68	73	73	77	75	72	70
	L_2	55	58	58	63	66	72	72	70	68
	L_∞	45	45	52	52	58	63	63	62	60
TM	L_1	56	57	66	68	70	85	74	70	65
	L_2	56	56	57	62	66	74	69	71	69
	L_∞	43	45	50	55	61	67	61	55	53
BL	L_1	60	65	67	75	71	78	72	72	71
	L_2	52	55	59	65	66	76	71	70	68
	L_∞	43	43	49	51	51	64	63	61	60

The initial values of all weights are generated as random from [-1,1] range. The sigmoid transfer function is applied to all neurons from hidden and output layer. The training is done using Levenberg-Marquardt algorithm (Kung, 1993). Three different kinds of experimental distance have been used during error of model calculation (Kung, 1993).

The voting Event Processing Agents is – initially – also fully interconnected Multilayer Perceptron. The size of the input layer is equal to the size of the input vectors. It means we have 4 neurons

there ready to load the products of the EPA collaborating with the data taken from sensors. The output layer has only one neuron to generate the answer: one of two possible states: alarm / no alarm. The size of the single hidden layer is equal to 20 neurons as a starting value, but during the training procedure the number of active neurons is reduced by the Optimal Brain Damage (OBD) mechanism using regularization factors. This way the minimum number of working neurons in hidden layer is only 4.

The training procedure was done for using the set of vectors created based on the components aggregated as outputs from EPA cooperating with sensors. The number of epochs is limited by the *no change* observation taking into account the network error minimizing. The final results – the percentage of correct answers – are presented in Table 2. The initial values of all weights are generated as random from [-1,1] range.

The sigmoid transfer function is applied to all neurons from hidden and output layer. The training is done using Levenberg-Marquardt algorithm. Three different kinds of experimental distance have been used during error of model calculation (4).

Table 2. Voting EPA correct answers [%] for limited number of neurons in hidden layer as OBD mechanism result

Distance type	Number of neurons in hidden layer – OBD result								
	4	6	8	10	12	14	16	18	20
L_1	67	69	72	75	77	82	88	85	81
L_2	58	64	68	70	73	74	75	79	76
L_∞	53	55	58	62	65	71	73	70	70

4.4. Analysis and sensitivity discussion

The experimental results pointed in Table 1 show that intelligent hierarchical EPA is able to provide correct recognition of the importance level based on data from different types of sensors. Both bi-level and scaled continuous outputs from the sensors can be useful source of data for the proper decision. We find the answer of EPA cooperating with sensor as correct if it equals to required output within [-0.1, +0.1] range. The extremely reduced number of neurons according to OBD mechanism causes insufficient – incorrect from the functional point of view – answer of EPA. It seems natural, but on the other hand – the ontogenic approach to EPA construction – with *on-line* hidden layer tuning – looks very promising. We started with twenty neurons located in this layer, but this value – looking good as a-priori assumption – is not optimal. The best size is 14 neurons for all types of the sensors. Of course this optimal number of neurons can be different if we use other sets of input data or we redefine the input vectors. The ontogenic topology is caused by the data used for training procedure. The correct EPA answers available for all tested types of sensors is a kind of proof that the set of 1000 training vectors sounds sensible to create the required level of *recognition skills* of single EPA. By modelling the training vectors sets we can tune the level of EPA reaction for input as well as we can store in the EPA deeper and more or less detailed *history* of the system life. We know how important is the correct *teacher's* required output during the training process. This output should be based on the expert knowledge to finish the weights setting at the necessary level of details. The EPA outputs for all types of sensors look promising, but better results we find for TS – two-states sensors responsible for simple events and TM – temperature sensors. Maybe these types of data are more convenient for neural modeling, or the expert knowledge used during training is better, or other types of sensors need more epochs or more data to establish final values of weights. Table 1 also tells us that for all types of sensors the most classic approach to distance measure L_1 is the best for the task we discuss. It means the easiest implementation in the practical future of the system. The voting EPA results – Table 2 – looks also very promising. The final answer of the hierarchical EPA structure is the best for the topology with 16 neurons in the hidden layer. Again the OBD mechanism allows to reduce the size of this layer to the most suitable size. And the aggregation of the previous layer of EPAs' outputs is absolutely correct using the same L_1 type of distance during training procedure. The hierarchical construction of the intelligent EPA allows to create more sophisticated cascades of EPAs collaborating with sensors with the final decision block. This way we can decide about the components of the voting EPA answer, we can model the influence of the events for the next step of the safety system reaction.

During the last part of the experiment, we try to check the EPA sensitivity for the changes of the input vectors. Each unified input vector collects

the set of parameters describing the single sensor's actual state. Some of these parameters are constant or almost constant.

Table 3. EPA sensitivity [%] for different type of sensors and limited number of neurons in hidden layer as OBD mechanism result

Sensor	Distance type	Number of neurons in hidden layer – OBD result								
		4	6	8	10	12	14	16	18	20
AM	L_1	10	10	8	7	7	6	6	4	3
	L_2	13	13	11	10	9	7	6	5	4
	L_∞	15	15	12	10	10	9	7	6	5
TM	L_1	8	8	7	7	6	5	5	3	2
	L_2	10	10	11	8	9	6	6	5	4
	L_∞	12	14	12	10	10	8	7	6	5
BL	L_1	10	10	8	7	7	6	6	4	3
	L_2	13	13	11	10	9	7	6	5	4
	L_∞	15	15	12	10	10	9	7	6	5

The main changes are observable in these components which reflect the environmental feature tested by the sensor. This way we try to find if the EPA answer is really provoked by this leading value from the input vector. Result are presented in Table 3. We can easily notice that greater number of neurons provides the better sensitivity for input data. The net with the greater number of neurons in hidden layer can analyze the input vector in more detailed way. For TM sensors we find better sensitivity than for AM and BL sensors. The previous sentence is kind of analogy to the first part of the experiments, and we are not surprised about it. There is no sense to check the sensitivity parameter for TS sensors because inputs are binary. The L_1 distance type is the most suitable.

Results of the case study discussion show the sensible scale of the topology of EPA tunning taking into account the correct answers and required level of output sensitivity. These conclusions have the significant influence for the practical implementation of EPA structure – using GPU – as complete CEP system.

5. CEP with intelligent agents architecture

We propose a micro-service architecture (Wolff, 2016) for event processing with intelligent agents. The system consists of: NATS Streaming (Nats, 2022) for inter-component communication, Siddhi (Suhothayan et al., 2011) as CEP engine and intelligent agents itself. NATS Streaming due to its high performance and support from Siddhi acts as a communication middleware.

```
@sink(type='nats', @map(type='text',
@payload("{{json}}")), destination='agent_1_in',
bootstrap.servers='nats://saware.nats.streaming:4222
',client.id='agent_1_client_in',server.id='test-
cluster')
define stream AgentInputStream (json string);

@source(type='nats', @map(type='text',regex.B =
'.*', event.grouping.enabled= 'false',
@attributes(json = 'B[0]')),
destination='agent_1_out',
bootstrap.servers='nats://saware.nats.streaming:4222
',client.id='agent_1_client_out',server.id='test-
cluster')
define stream AgentOutputStream(json string);

from External_in insert into AgentInputStream;
from AgentOutputStream insert into External_out;
```

Figure 2. Siddhi agent configuration for communication with intelligent agent by NATS.

Intelligent agents are deployed as containers and act as NATS subscribers. They actively listen to NATS topics and receive messages. After processing the message using built-in ML algorithms, they send messages back to the Siddhi. We have created a simple Python library to facilitate the creation of intelligent agents. The programming interface consists of two methods: initialization, which is mainly used to load the model, and processing which is called when a message arrives and returns the result message.

The NATS script used to communicate with the intelligent agent is shown in Figure 2.

Intelligent agents are containerized using Docker (Merkel, 2014). This not only allows a basic version of the system to be quickly deployed on almost any computer (using Docker Compose), e.g., for testing and development, but also for automatic and continuous deployment using Kubernetes (Kubernetes, 2022). As mentioned in Section 2, Kubernetes provides scalability and high reliability, and enables easy management of a highly distributed event processing system. We maintain two versions of the containers, one for implementation on the CPU and one for implementation on the GPU, if needed.

6. Implementation of deep intelligent agents

Deep learning architectures (such as LSTM and transformers) require a lot of computation power even in the inference phase. Therefore, the most typical scenario is to use the GPU to compute the network results. However, the GPU is a limited resource, and it is worth investigating how a deep

network performs on the CPU alone. To analyze it and to select the optimal number of CPUs and the batch size, we conducted a set of experiments. We tested the PyTorch implementation of the BERT network with Hugging Face (Wolf et al., 2020) and the ONNX Runtime (Microsoft, 2022). The last uses an open-source machine-independent format and is widely used for exchanging neural network models (Bai et al., 2022). Both solutions allow the model to run on GPU or CPU. We trained the SBERT model (Reimers, Gurevych, 2019) and tested the longest sequences (512 tokens). In the tests, we used AMD Ryzen 9 3950X and NVIDIA GeForce RTX 3090 graphics cards. The results are shown in Table 4. All tests were repeated 100 times and the results represent the average value.

Table 4. BERT inference time per task in ms for different deployments. In case of a usage of batches the reported time is a time of processing a batch divided by the size of a batch

Batch size	PyTorch			ONNX		
	1	4	8	1	4	8
GPU	9.6	7.4	6.5	8.1	7.2	6.4
1xCPU	3103	3116	3071	968	1006	1103
2xCPU	1597	1611	1640	527	521	532
4xCPU	848	858	858	298	293	290
8xCPU	536	502	498	202	187	184
16xCPU	415	360	341	177	158	152

The results show that the ONNX implementation outperforms PyTorch in terms of inference time for both CPU and GPU cases. The tests were performed for different batch sizes. The results show that using batches speeds up the average inference time, but the efficiency of using batches is low. Therefore, the batch size must be carefully selected, especially for long processing times, as in the case of CPU-based implementations. Moreover, the use of batches for CPU is justified only when we have more CPU cores than the batch size, in other case there is no average speed-up. Even in such cases, the speed increase by using batches is not very large. For example, the differences between batch sizes of 4 and 8 (for more than 8 CPUs) are less than 2% and 4%, respectively. Thus, the use of batches should be limited to a small value and considered for reasons other than speeding up model inference; for example, event batches could speed up communication between the queueing system and the agent. The performance of multiprocessor inference decreases very quickly as a function of the number of cores used (see Table 5). Therefore, the only reason to use more than one CPU core for inference is to reduce the processing time per event. The PyTorch implementation gives better performance values, but as shown in Table 6, the ONNX implementations are 2-3 times faster on CPU than PyTorch.

Table 5. Multi CPU implementation efficiency. The ratio of the time required by one processor to the time required by *n*-processors divided by *n* (for batch of size 1)

Number of CPU	2	4	8	16
ONNX	0.918	0.812	0.599	0.341
PyTorch	0.971	0.914	0.772	0.467

Table 6. The ratio of ONNX processing time to PyTorch processing time. The 5th column of Table 4 divided by the 2nd column

GPU	CPU				
	1	2	4	8	16
1.18	3.2	3.0	2.84	2.65	2.34

Another important aspect of the deep agent implementation is the memory consumption and model loading time. The results for the GPU implementation are shown in Table 7 and for the CPU in Table 8. Again, it can be seen that the ONNX implementation is better than the PyTorch one. Both implementations allow the model to be removed from memory, but not to free all memory. Obviously, killing the process will free all memory, but given the architecture proposed in Section 5, this is not the case under study, since running a new pod in Kubernetes with the image and model downloading may take more than 6s. For the GPU, the time to reload the model after its release is approx. 3 times shorter than the first load. For the CPU, this phenomenon does not occur.

Table 7. Usage of memory and time of loading model in case of GPU implementation

Time / Memory	PyTorch	ONNX
Time of model loading	3.48 s	1.26 s
Time model reloading	0.88 s	0.46 s
Memory usage	2.6 GB	1.7 GB
Memory after freeing model	2 GB	0.6 GB

Table 8. Usage of memory and time of loading model in case of CPU (results for 8 cores and batch size 4)

Time / Memory	PyTorch	ONNX
Time of model loading	0.8-0.9 s	0.5 s
Memory usage	1.5 GB	1.1GB
Memory after freeing model	0.9 GB	0.4 GB

The performed analysis shows that deep agents should be implemented in ONNX runtime. In case of the lack of GPU the CPU implementation is possible but we have to use several CPUs, its number could be calculated looking at Table 4 results. Knowing the frequency of input events, we can estimate the maximum acceptable processing time for the agent and select the lowest number of CPU that allows processing time within limits.

7. Conclusion

The chapter discusses performance aspects of intelligent agents in Complex Event Processing (CEP) systems. Actual methods of implementing and deployment of CEP systems are presented. The intelligent agent problem and the idea of creating a CEP system with intelligent agents are defined. A case study with a set of intelligent agents designed to handle real events caused by environmental data monitoring is pointed out. The analysis performed shows that deep agents should be implemented in the ONNX runtime. In the absence of GPU, a CPU-based implementation is possible, but we need to use several CPUs. The frequency of input events is the main feature to estimate the maximum acceptable processing time of the agent and to choose the least number of CPUs to process the data in a given time.

Acknowledgment

This work presented results of the Smart Growth Operational Programme 2014-2020 grant No. POIR.01.01.01-00-0235/17 supported by the Polish National Centre for Research and Development (NCBR) as a part of the European Regional Development Fund (ERDF).

References

Abhishek, K., Singh, M.P., Ghosh, S. & Anand, A. 2012. Weather forecasting model using artificial neural network. *Procedia Technology* 4, 311–318.

Adamczak, R., Duch, W. & Jankowski, N. 1997. New developments in the feature space mapping model. *Third Conference on Neural Networks and Their Applications*. Kule, Poland, 65–70.

Bai, J., Lu, F., Zhang, K. et al. 2022. *ONNX: Open Neural Network Exchange*. GitHub https://github.com/onnx/onnxi. (accessed 01 June 2022).

Bonarini, A., Masulli, F. & Pasi, G. 2003. *Advances in Soft Computing, Soft Computing Applications*. Springer.

Devlin, J., Chang, M-W., Lee, K. & Toutanova, K. 2019. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* https://doi.org/10.48550/arXiv.1810.04805.

Duch, W. & Diercksen, G.H.F. 1994. Feature space mapping as a universal adaptive system. *Computer Physics Communications* 87, 341–371.

Duch, W., Jankowski, N., Naud, A. & Adamczak, R. 1995. Feature space mapping: a neurofuzzy network for system identification. *Proceedings of the European Symposium on Artificial Neural Networks*. Helsinki, 221–224.

Greff, K., Srivastava, R.K., Koutnik, J., Steunebrink, B.R. & Schmidhuber, J. 2017. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems* 28(10), 2222–2232.

Hastie, T., Friedman, J. & Tisbshirani, R. 2018. *The Elements of Statistical Learning: Data Mining, Inference and Prediction.* New York, Springer.

Hinton, G.E. 1987. Learning translation invariant recognition in massively parallel networks. *Proceedings PARLE Conference on Parallel Architectures and Languages Europe*. Berlin. Springer-Verlag, 1–13.

Huang, Z., Xu P., Liang, D., Mishra, A. & Xiang, B. 2020. *TRANS-BLSTM: Transformer with Bidirectional LSTM for Language Understanding*. https://doi.org/10.48550/arXiv.2003.07000.

Kubernetes. 2022. https://kubernetes.io/ (accessed 01 June 2022).

Kung, S.Y. 1993. *Digital Neural Networks*. Prentice-Hall.

Le Cun, Y., Denker, J. & Solla, S. 1990. *Optimal Brain Damage. Advances in Neural Information Processing Systems 2*. Morgan Kauffman. San Mateo CA.

Limnios, N. & Oprisan, G. 2001. *Semi-Markov Processes and Reliability*. *Birkhauser*, Boston.

Merkel, D. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014(239), 2.

Microsoft. 2022. *ONNX Runtime*, https://github.com/microsoft/onnxruntime (accessed 01 June 2022).

Nats. 2022. NATS.io; Cloud Native, Open Source, High-performance Messaging, https://nats.io/, https://nats.io/documentation/ (accessed 01 June 2022).

Peter, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, Ch., Lee, K. & Zettlemoyer, L. 2018. Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1, 2227–2237, New Orleans, Louisiana. Association for Computational Linguistics.

Reimers, N. & Gurevych, I. 2019. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*.3973–3983. 10.18653/v1/D19-1410.

Srivastava, A.K. 2008. Soft Computing. Narosa PH.

Sugier, J., Walkowiak, T., Mazurkiewicz, J., Śliwiński, P. & Helt, K. 2019. Performance evaluation of event-driven software applied in monitoring systems. I. Kabashkin, I. Yatskiv (Jackiva), O. Prentkovskis (Eds.). *Reliability and Statistics in Transportation and Communication. RelStat* 2018. Lecture Notes in Networks and Systems 68. Springer, Cham, 311–319.

Suhothayan, S., Gajasinghe, K., Narangoda, I.L., Chaturanga, S., Perera, S. & Nanayakkara, V. 2011. Siddhi: a second look at complex event processing architectures. *Proceedings of the 2011 ACM workshop on Gateway computing environments (GCE'11)*. Association for Computing Machinery, New York, NY, USA, 43–50.

Weigend, A.S., Rumelhart, D.E. & Huberman, B.A. 1990. Backpropagation, weight elimination and time series prediction. *Proceedings of the 1990 Connectionist Models Summer School*. Morgan Kaufmann, 65–80.

Weigend, A.S., Rumelhart, D.E. & Huberman, B. A. 1991. Generalization by weight elimination with application to forecasting. *Advances in Neural Information Processing Systems 3*. San Mateo CA. Morgan Kaufmann, 875–882.

Wolf, T., Debut, L. et al. 2020. Transformers: state-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45, Online. Association for Computational Linguistics.

Wolff, E. 2016. *Microservices: Flexible Software Architectures*. Addison-Wesley.