A Microservices Architecture for Machine Learning Assisted Decision Support in a Real-Time Field Sensors Environment

Giovanni De Gasperis[1], Giuseppe Della Penna[1] and Sante Dino Facchini[1]

[1]*Università degli Studi dell'Aquila, Dipartimento di Ingegneria e Scienze dell'Informazione a Matematica, Via Vetoio, L'Aquila, 67100, Italy*

Abstract
In this paper we describe the design and development of a real-world software system that integrates machine learning augmenting a pre-existing remote surveillance framework. Machine learning was embedded as a service in the system, plugged-in between back-end data flux handlers; the system has been redesigned following a microservices architecture to make it scalable and to allow a progressive adoption of the machine learning-powered assistance in the event management process. A case study of the application in an actual security company is analysed and discussed, where we show how this innovation helped human operators to better shield themselves from the "information overloading".

Keywords
Real-Time Critical Systems, Machine Learning, Big Data, Microservices

1. Introduction

In this paper we describe the design and development of a real-world software system that integrates big data analytics and machine learning into a pre-existing remote surveillance framework operated by security company that monitors a number of sites through closed circuit and IP cameras, anti-theft sensors (e.g., volume and pressure sensors, door opening sensors, etc.) and also physical sensors (e.g., humidity and temperature).

Figure 1 shows a fragment of the process commonly followed to handle events and alarms coming from a surveillance network. When an alarm is received, first the operators check the surveillance videos. If such videos are not available or they do not clearly show the event, the operator requests an on-site check to the security staff. Such action and its outcome, as well as the outcome to all the actions taken during the process, is stored in the system database. Then, if the event is in progress, the operator starts the *true alarm* handling process. Otherwise, if the notified event is not actually in progress, the operator must check for other alarms on the same site and, if any, restart the handling process for such new events. If no other site alarms are active, the operator must try to understand the reason of the notified alarm. If it is recognized as a *false alarm*, the case is simply closed. On the other hand, if the alarm is *improper*, i.e., it is due a system anomaly, the operator starts an anomaly handling process.

The software adopted by the company to support such a process was a monolithic application that offered only basic functionalities such as collecting signals and data streams, presenting the events in a managing console and saving them in a persistent database. Therefore, most of the operations described by the event management process above required a substantial amount of manual work by the control center operators.

While *the human intervention cannot be avoided in such a context*, as in any security-related context, *machine learning can be exploited to assist the operators in several steps of the process*, leaving the humans with only the most critical steps to accomplish (see, e.g., [1, 2, 3, 4] for examples belonging to different surveillance contexts).

However, embedding machine learning in the company pre-existing software presented several challenges. First, we are modifying a production, real-time critical system, so we need to gradually add such a support, in order to let the operators adapt to the new functionalities while verifying their reliability without interrupting the company services. Second, the closed, monolithic architecture of the company software described above makes any modification to the pre-existing process very complex and error-prone. It is also worth noting that such a software, developed many years ago, was already not adequate to accomplish the current high QoS levels and to be compliant with the latest safety regulations.

Therefore, we decided to rebuild the system from scratch, extracting only some relevant modules/algorithms from the old software in order to embed it in

ECSA2021 Companion Volume, Robert Heinrich, Raffaela Mirandola and Danny Weyns, Växjö Sweden, 13-17 September 2021
giovanni.degasperis@univaq.it (G. De Gasperis); giuseppe.dellapenna@univaq.it (G. Della Penna); santedino.facchini@student.univaq.it (S. Facchini)
https://www.disim.univaq.it/main/home.php?users_username=giovanni.degasperis (G. De Gasperis); https://people.disim.univaq.it/~dellapenna (G. Della Penna)
0000-0001-9521-4711 (G. De Gasperis); 0000-0003-2327-9393 (G. Della Penna)

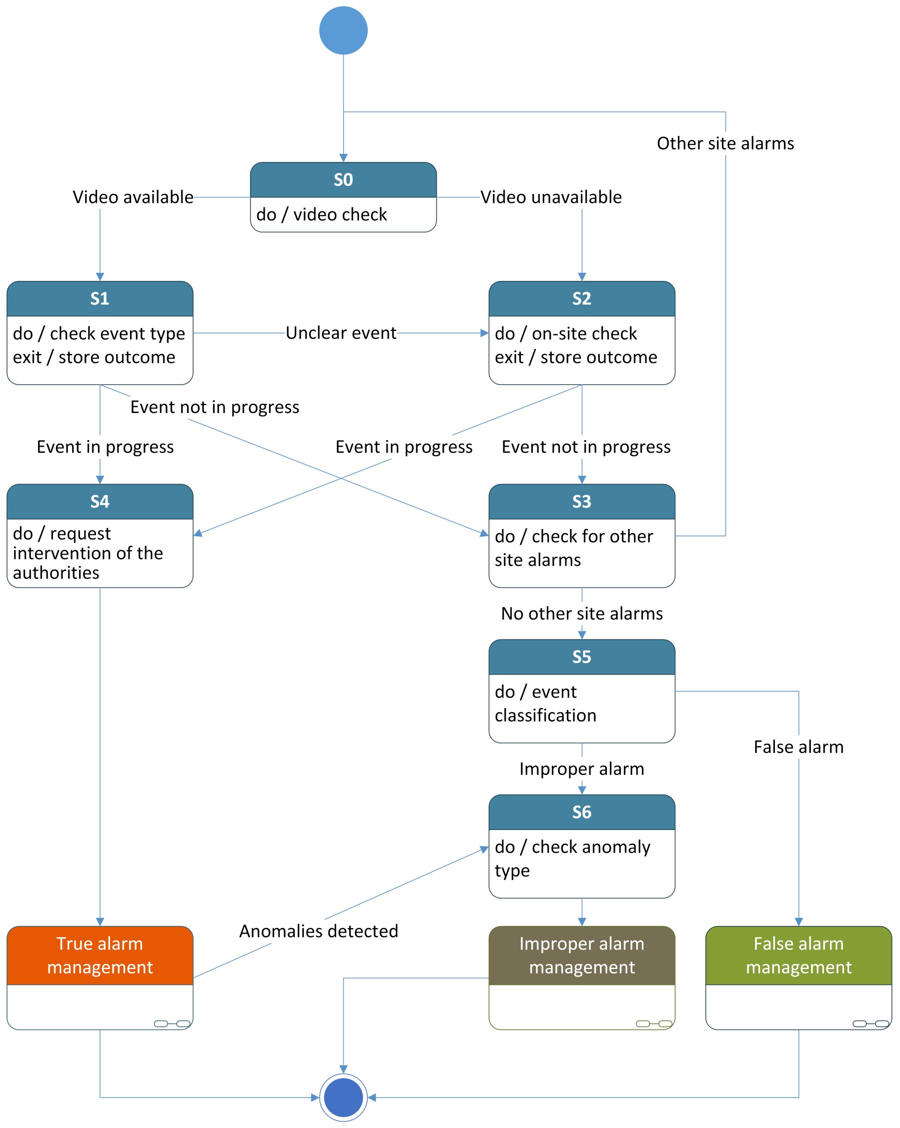


Figure 1: Surveillance event handling process

the new release, to maintain some kind of "continuity". In particular, while having a completely redesigned core, the system has been designed to offer the basic functionalities of the previous one (to maintain the above service continuity) and extend them in order to

- acquire data and information on objects and events from *multiple sources* such as IoT devices, open systems and mobile services;
- *combine and correlate the real-time big data streams* in order to make easier and faster the control center operators job, in particular the event classification, the data storage and the log certification;
- offer *decision support tools* to assist operators in the real-time management of events.

All the three extensions above could be based on simple (sometimes pre-existing) algorithms, but also take advantage from machine learning. Thus, *ML can be seen as a (plug-in) service in the overall system architecture.* This consideration, together with a number of technical aspects concerning the modular structure of the application and its deployment environment, made us opt for a service oriented architecture [5] as its development basis. More in detail, to be as lightweight as possible, we adopted a *microservices architecture* [6] to effectively split the complexity of the overall system into small specialized units, each with its REST API and containerized using Docker [7]. This allowed us to develop a *scalable*, *versatile*, and easily *maintainable* system, as opposed to the previous one. In particular, microservices allowed us to fragment the overall system functionality in a set of meaningful basic units that, apart from the known advantages in term of scalability and maintanability, make easier the progressive addition of ML support as a service (or set of services) interacting with the core system services to enhance their functionalities.

2. System Architecture for a Reliable Event Handling

At a macro level, the base requirements of the new system are the following:

1. Acquire data, events and states from an heterogeneous set of sources spread in a wide geographical area and connected through a digital network;
2. aggregate such data streams in a configurable and scalable way, since the number and type of sources may vary;
3. correlate events through time and space logical rules;
4. visualize all the collected and elaborated information on an interactive web dashboard;
5. store all the data, as well as all the actions taken by the control center operators, on a persistent memory;
6. provide a search engine with configurable queries to access such historical data.

Starting from such requirements, collected through interviews with the company staff and control center operators, we designed a microservice-based system architecture that aims to be easily testable, maintainable, and extensible. Figure 2 gives an overview of the developed system.

The data is pushed in the system by a number of specialized source services (drawn in green in the figure): in the currently deployed platform, we provide services which support reading data from sensors using MQTT [8] and SNMP [9] as well as from specific proprietary sensors such as Papago [10]. These services satisfy requirement 1.

The overall system leverages on three services that provide different "analysis horizons" on the event streams: the Prometheus [11] service (yellow), which wraps the corresponding software package, acts as the main data collector of the overall architecture, providing access to the events of the last three days gathered from all the sources and suitably aggregated (requirement 2). Prometheus is an open-source systems monitoring and alerting software, which provides a multi-dimensional data model with time series, and allows a variety of interactions with other third-party software components.

Next, there is the the Redis [12] (gray) service that wraps the well-known fast in-memory data store (supporting real-time data streams), which acts as a cache memory to provide the consumers a fast access to the event information streams. It essentially takes the role of "live working memory" for the events and the corresponding management procedures, which are stored in the database until they reach the closing state. It is worth noting that the output of sensors like temperature and humidity, whose handling does not require the human intervention (e.g., temperature sensor alarms are based on simple logic rules), are not stored in Redis.

Finally, the SQLite [13] service (gray) provides persistent storage for all the data flowing through Prometheus and Redis. This is mainly needed to later extract the evidences needed by the authorities (requirements 5 and 6).

To support requirement 4, the blue front-end services present the gathered and processed data in different formats, tailored for the specific needs of the different system users. In particular, three "*IG Server*" services handle the GUI for the operators, their supervisors and the system administrators. Such user interfaces present to the users the data streams coming from the sensors, read from the Redis service, as well as the data coming from Prometheus. A special "*configurator*" GUI is used to configure the system. The last two front-end services directly interact with Prometheus: the first is based on the *Grafana* software package [14], which provides advanced visualization and dashboards for the processed and aggregated data, whereas the *alertmanager* service pushes alerts automatically generated from Prometheus through PromQL queries directly into email and Telegram messages.

Finally, the red services are at the core of the architecture. In general, they read from both Prometheus and Redis and apply actions, possibly modifying the Redis data streams accordingly. In particular, *gest_notifier* is a critical module that notifies the supervisors (via text messages) about *event escalation*, i.e., alarms that are triggered by an event not being correctly and timely handled by the control center operators. On the other hand, the *gest_control* service manages the QoS by monitoring that the operator reaction times follow the company SLA, also generating alarms in case of inefficiencies (requirement 3). The *gcounter* service generates aggregate statistics from the Redis data and posts them back to Prometheus to support longer-term alarms (requirement 2). Finally, the *gest_source* service is the action actuator, i.e., it is

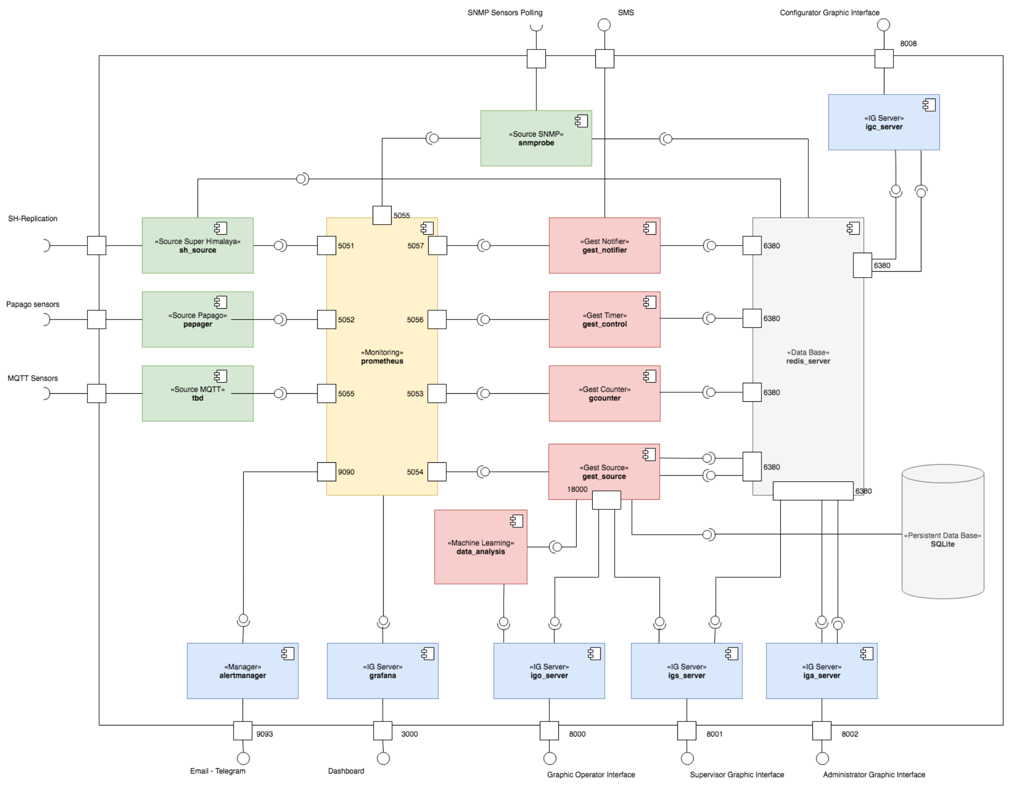


Figure 2: System architecture

called by the operator interfaces to actually apply the actions ("take charge", "start/pause/close workflow", etc.), executes them, and generates the action events that are stored back in Redis to log the handling process.

The *data_analysis* service, also drawn in red, is described in the next section. It provides ML support to the overall process, and in particular assists the operators by interacting with the graphical interfaces.

All the microservices above were developed in Python and containerized, to be easily deployable on the company's infrastructure through an overall container deployment script that, in particular, takes into account the service inter-dependencies.

3. Machine Learning Services Integration

The event handling process, in the critical context where our system works, must be timely and effective. The architecture described in the previous section (Figure 2) has been designed to be reliable and fast, but the process (Figure 1) still includes a number of checks, calls, and lengthy actions that require a substantial amount of work to the operators.

Clearly the human intervention cannot be avoided when trying to solve potentially dangerous events. However, ML can help the operators in many ways, as commented in the introduction. Thus, we initially focused on an aspect that is well known to benefit from automatic reasoning: mitigating the effects of an *event flooding* on the operators by pre-selecting or pre-classifying events.

Indeed, an operator that manages events is the classical FIFO order may spend too much time on less-significant events and delay the solution of the really critical ones. To avoid this, the Data Analysis service provides an adaptive event classification routine that prioritizes the events so that the ones that are considered more important, i.e., that may lead to real alarms, are presented first to the operators. Such a service stands between the *gest_source* and *lgo_server* services, learning from the operator actions and suitably modifying his data views in order to suggest the event classification.

Operators close each event handling process by labelling the events as true alarms, false alarms, inappropriate alarms (i.e., due to a system failure) or "other alarms" (typically due to test or maintenance). We focus on the true and false ones, since the other two types of events are a minority class that would be difficult and useless to consider in our context.

Thus, initially, we extracted a number of significant features from the events, relating them with the operator-assigned label. Such features include information such as the unique alarm ID, the alarm central where the event was generated, the related customer, the activated sensor name, the timestamp of the alarm, etc.

Then, we cleaned and refined these features to further focus on the information that seems to be more relevant for our classification task. As an example, we substituted the alarm timestamp, which conveys too much information, with the alarm weekday and the corresponding part of the day (morning, afternoon, evening, night). Moreover, we performed a K-means clustering [15] on the sensor names (suitably transformed in a numerical vector through a word embedding process), in order to extract an artificial "sensor type" feature. The significance of the selected features was validated by calculating the mutual information of each feature w.r.t. the classification label (see, e.g., [16] for an overview of mutual information applied to feature selection) on a set of 169,347 past events provided by the company.

Once the final 128 features were devised, we extracted an initial training dataset from the set of past events above. Unfortunately, the dataset was heavily imbalanced, since the false alarms were much more than the true ones [17]. Thus, we tried both a random undersampling of the majority class and the well-known Synthetic Minority Oversampling Technique (SMOTE, [18]) to re-balance it.

Finally, we built a deep neural network [19] with 128 input neurons (one for each feature), two hidden layers of 64 neurons each with *RELU* [20] activation function, and a single output neuron with *sigmoid* activation function. We trained the network on our dataset in order to obtain the correct classification given the event features. The classifier validation showed that the dataset re-balanced through random undersampling achieves a better overall performance in this context, reaching an accuracy of 0.91, a recall of 0.93, and a precision of 0.95, with a F1-Score of 0.92.

The trained network was then embedded in the Data Analysis service, where each new event is classified before being presented to the operators. However, since a wrong "false alarm" classification may always happen, we do not simply drop the events considered not harmful by the neural network from the stream, but rather we extract the classification probability that can be read from its output neuron and use it to alter the priority value that is used to sort the events on the operator dashboards. In this way, a possibly false alarm will be handled later, but never dropped. After the event is handled, the correct, final classification given by the operator is sent back to the neural network to fix its previsions, if needed.

4. Conclusions

Thanks to its design and to the use of ML, the developed system meets highest quality standards, in particular:

- it allows to to acquire, aggregate and process data and information from a variety of IoT devices, which means offering a better service in terms of quality and flexibility;
- it guarantees high scalability and easy configurability;
- it is fully compliant with data privacy, integrity and security regulations.

The project requirements foresaw to process about 300,000 events a year with the current number of operators, and manage an alarm within at most 30 seconds, as set by the standard regulations. Currently the system has been deployed and is being tested by the company in a control room operating 24/7 on three turns of eight hours each, with two or three operators per turn. The staff is managing about 32,500 anti-theft and intrusion detection sensors and over 500 environmental IoT devices mainly targeted to precision agriculture. The network connects about 30 clients and its nodes are deployed on more than 600 different sources.

Our initial statistics show that the staff is now able to manage an average of 1,000 events per day, thus yielding 365,000 events managed on yearly basis as a forecast, which doubles the performances achieved using the previous support software. The average time from the event arrival to its classification and taking charge is now 10 seconds, slightly better compared to the performance of the previous system but in a far more complex scenario.

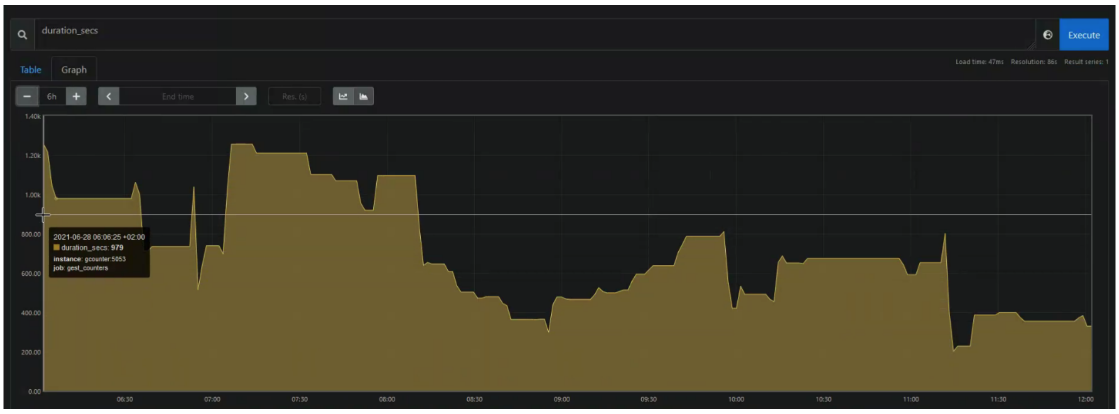
Finally, the average event management time, including classification, site operations, police calls, alarm closing and archiving, has been dramatically improved from 1,800 to 900 seconds (Figure 3 shows the current statistics generated by the Grafana service in the application), and the error ratio has been tackled almost to zero thanks to the ML priority classification system.

Acknowledgments

We would like to thank Mr. Andrea Perna who implemented the first prototype of the machine learning model as part of his Master's Thesis in Computer Science at our Department.

This work was funded by SPEE S.r.l. Company [1] in L'Aquila, ITALY.

[1]https://www.spee.it



```
duration_secs
```

Load time: 47ms Resolution: 86s Result series: 1

```
2021-06-28 06:06:25 +02:00
duration_secs: 979
instance: gcounter:5053
job: gest_counters
```

Figure 3: Average event management time

References

[1] A. Goyal, S. B. Anandamurthy, P. Dash, S. Acharya, D. Bathla, D. Hicks, A. Bhan, P. Ranjan, Automatic border surveillance using machine learning in remote video surveillance systems, in: T. Hitendra Sarma, V. Sankar, R. A. Shaik (Eds.), Emerging Trends in Electrical, Communications, and Information Technologies, Springer Singapore, Singapore, 2020, pp. 751–760.

[2] J. Albusac, J. Castro-Schez, L. Lopez-Lopez, D. Vallejo, L. Jimenez-Linares, A supervised learning approach to automate the acquisition of knowledge in surveillance systems, Signal Processing 89 (2009) 2400–2414. doi:https://doi.org/10.1016/j.sigpro.2009.04.008, special Section: Visual Information Analysis for Security.

[3] M. Elhoseny, Multi-object detection and tracking (modt) machine learning model for real-time video surveillance systems, Circuits, Systems, and Signal Processing 39 (2020) 611–630. doi:10.1007/s00034-019-01234-7.

[4] F. Opitz, K. Dästner, B. v. H. z. Roseneckh-Köhler, E. Schmid, Data analytics and machine learning in wide area surveillance systems, in: 2019 20th International Radar Symposium (IRS), 2019, pp. 1–10. doi:10.23919/IRS.2019.8768102.

[5] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari, A. R. B. C. Hussin, Understanding service-oriented architecture (soa): A systematic literature review and directions for further investigation, Information Systems 91 (2020) 101491. URL: https://www.sciencedirect.com/science/article/pii/S0306437920300028. doi:https://doi.org/10.1016/j.is.2020.101491.

[6] M. Fowler, Microservices: a definition of this new architectural term, 2014. URL: https://martinfowler.com/articles/microservices.html.

[7] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux journal 2014 (2014) 2.

[8] International Organization for Standardization, Information technology — message queuing telemetry transport (mqtt) v3.1.1 (iso/iec standard 20922:2016), 2016. URL: https://www.iso.org/standard/69466.html.

[9] D. Harrington, R. Presuhn, B. Wijnen, An architecture for describing simple network management protocol (snmp) management frameworks, 2002. doi:10.17487/RFC3411.

[10] Papouch, Papago sensor modules, 2021. URL: https://en.papouch.com/papago/.

[11] The Linux Foundation, many authors, Prometheus, 2021. URL: https://prometheus.io/.

[12] Redis Labs, S. Sanfilippo, Redis, 2021. URL: https://redis.io/.

[13] SQLite Consortium, many authors, SQLite, 2021. URL: https://www.sqlite.org/.

[14] Grafana Labs, many authors, Grafana: the open observability platform, 2021. URL: https://grafana.com/.

[15] A. K. Jain, R. C. Dubes, Algorithms for Clustering Data, Prentice-Hall, Inc., USA, 1988.

[16] I. Letteri, G. Della Penna, P. Caianiello, Feature selection strategies for http botnet traffic detection, in: IEEE Computer Society (Ed.), Proceedings of Workshop on Machine Learning for Cyber-Crime Investigation and Cybersecurity, Proceedings of 2019 IEEE European Symposium on Security and Privacy Workshops, 2019, pp. 202–210. doi:10.1109/EuroSPW.2019.00029.

[17] B. Krawczyk, Learning from imbalanced data: open

challenges and future directions, Progress in Artificial Intelligence 5 (2016) 221–232. doi:10.1007/s13748-016-0094-0.

[18] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: Synthetic minority oversampling technique, Journal of Artificial Intelligence Research 16 (2002) 321–357. URL: http://dx.doi.org/10.1613/jair.953. doi:10.1613/jair.953.

[19] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Networks 61 (2015) 85–117. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135. doi:https://doi.org/10.1016/j.neunet.2014.09.003.

[20] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, 2011, pp. 315–323.