# Data Collection and Machine Learning for Critical Cyber-Physical Systems

*Custom Anomaly Detector in Python*

*Lorenzo Bartolini* 7073016
lorenzo.bartolini8@edu.unifi.it
a.a. 2023/2024

Prof. *Andrea Ceccarelli*
Dott. *Tommaso Zoppi*

# Monitor, Fault Injection and Training Set generation

The monitor is built to perform the task of monitoring the system. The indicators have been chosen according to the anomaly that we want to detect.
We will analyze the CPU and RAM. The idea is to protect the target system from unexpected and wrong behaviors targeting these two components.
It's been used *psutil* to gather the information about CPU and RAM (note that the CPU indicators are collected for each physical core).

The monitor and fault injection tasks are performed by a single function that alternates two states: *Injection state* and *Rest state*.The alternation of the two states is done to gather data of the system under both stress and normal conditions.
During the *Injection state* the program launches an injection from the given set of injections. The duration of the injection is a parameter, such as the duration of the rest phase. During the injection, the monitor collects the data from the system at each time step. The data collected is extended with: the *time* at which it is generated and a *label* that tells the state the monitor is in. This label will be later used during the training of the ML Model.
Once the injection has ended, the *Rest state* begins: no injections are launched, the monitor continues collecting and storing data just like before. This is a cooldown phase for the system. The length of the *rest state* is a parameter of the monitor but can be modified. In fact it is randomly expanded or shrunk at the beginning of each *rest state* in order to add some variance to the training set and to better simulate the reality. This process ends once there are no more injections to be done.

During my test I used 3 types of injectors:

- CPU Single Core
    - It targets a single core, specified in the file *base_injectors.json*, at a constant load level across cores
- CPU Multicore
    - It targets all cores at the same time at a target load level, specified in the file *base_injectors.json*
- Memory
    - It simply fills up the RAM

The final training set contains 69 features (67 features+time+label) and over 11250 rows. During its generation on my machine, I voluntarily kept using the computer as usual to mimic the normal usage. For example I browsed the internet and used other small programs that would not cause any major stress for the system.

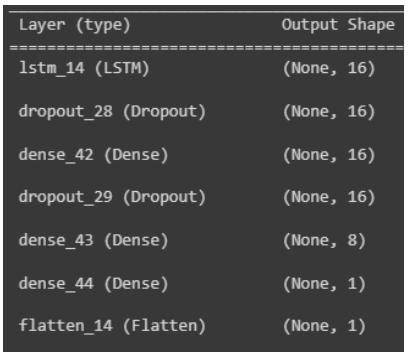
# Dataset Exploration and Manipulation

Before picking and training the models, it is important to analyze the obtained results exploring the dataset.
As I want to use a Neural Network, I have to check for categorical features. In this case there are none so I don't have to worry about adding embedding layers in the network.
The first thing to do is remove the features that do not help the model and we can find them studying the numbers outputted by the *describe* method of DataFrame, in particular *mean* and *std*. In fact we can drop all the columns that have a low std because they are not much useful to the model in order to distinguish an anomaly from a normal behavior.
A column that is important to eliminate is *time* because it can lead the model to learn by looking at the time at which a record is taken instead of learning the relationship between the features.
The last transformation to the dataset is to separate the input *X* and the labels *y*. The input is obtained removing the label column from the original dataset. For this project the goal is to do just Binary Classification so we don't need to store the individual injections made and to generate the labels we simply assign *zero* to rest and *one* to any fault injected.
To get better results the input is transformed using a StandardScaler; after doing so it is important to save the scaler in order to use it during the actual anomaly detection.
The *X* and *y* are then splitted using the *train_test_split* function in two ways: one split shuffling the data and the other without. The second one is needed to train the Recurrent Neural Network because it needs its input in the same order it has been recorded.

# Model Selection and Training

I chose to train 3 models and compared using the *accuracy* score and the *confusion matrix*:

- *RandomForestClassifier*(n_estimators=11)
    - This is the best model tested so the one used for the runtime anomaly detection
    - It achieved an accuracy on the test set of 98.1%
- *StackingClassifier*(LinearDiscriminantAnalysis, GaussianNB, DecisionTreeClassifier) with a RandomForest as the final estimator
    - I expected this to have better results compared to the one before
    - It achieved an accuracy on the test set of 96.4%
- *Recurrent Neural Network*
    - The thought process led me to the conclusion that a RNN could perform better at analyzing this dataset being a time serie but I had many problems optimizing it
    - After fine tuning the hyper parameters, the model achieved an accuracy on the test set of 93.8% and a loss of 0.25
    - The network analyzes 4 time steps at a time and it has been trained over 20 epochs




## First Test: Dataset without statistics about Virtual Memory

During this first test I tried to study how removing the statistics about virtual memory from the dataset affects the performance of the three models.

| **Model** | **Accuracy with Normal Dataset** | **Accuracy with No Memory Dataset** |
|---|---|---|
| *RandomForestClassifier* | 98.1% | 58.1% |
| *StackingClassifier* | 96.4% | 57.1% |
| *RNN* | 93.2% (**Loss** 0.58) | 94.7% (**Loss** 0.23) |

Overall we can see that removing the Virtual Memory information from the dataset is not a good choice.

## Second Test: Fine Tuning of the RNN

During this second test I explored two hyperparameters: the Dimension of Samples (set of time steps) and the Model Architecture.

### 1) Dimension of Samples

Note that the samples overlap. This way the model can learn better by looking at the same event both at the start and at the end of the sample. Also it is a good idea to overlap the samples to have a bigger dataset for training.
Training conducted for 20 epochs and batches of 15 samples.

| **Time Steps per Sample** | **Accuracy Train** | **Accuracy Test** |
|---|---|---|
| *1* | 98.6% (**Loss** 0.04) | 92.8% (**Loss** 0.45) |
| *2* | 99.1% (**Loss** 0.02) | 92.7% (**Loss** 0.70) |
| *4* | 99.0% (**Loss** 0.01) | 93.2% (**Loss** 0.58) |
| *8* | 98.1% (**Loss** 0.02) | 92.1% (**Loss** 0.57) |
| *16* | 96.3% (**Loss** 0.07) | 90.2% (**Loss** 0.44) |

We can see that using many time steps does not help, as a matter of fact, the best model was achieved with 4 time steps per Sample.

### 2) Model Architecture

During this test the optimizer and its parameters will remain the same.
Dropout layers and the last layer, that is always Dense(1), are not shown for simplicity.

| **Architecture** | **Dropout Rate** | *0.3* | *0.5* |
|---|---|---|---|
| *LSTM(8)*<br>*Dense(8)*<br>*Dense(4)* | | **Accuracy** 92.5%<br>(**Loss** 0.30) | **Accuracy** 91.8%<br>(**Loss** 0.23) |
| *LSTM(16)*<br>*Dense(16)*<br>*Dense(8)* | | **Accuracy** 92.6%<br>(**Loss** 0.32) | **Accuracy** 93.8%<br>(**Loss** 0.25) |
| *LSTM(32)*<br>*Dense(32)*<br>*Dense(16)* | | **Accuracy** 93.7%<br>(**Loss** 0.40) | **Accuracy** 92.8%<br>(**Loss** 0.38) |

# Runtime Anomaly Detection

My runtime anomaly detector is based on the RandomForest as it's the model that achieved the best results. To make a more sophisticated software I came up with the idea of *warning level*. It is a number that expresses the seriousness of the warning.
At each time step the system is monitored and the data is passed to the detector that tells if there is an anomaly or if everything seems correct.
If it detects an anomaly it raises the warning level, when the level passes the specified threshold a warning is activated. The program logs in a file the timestamp and the warning level and updates the console that is overwritten in case the warning level decreases under the threshold.
The speed at which the level decreases is exponential so it lowers very easily after a serious warning but the level increases linearly so it is not affected strongly by short bursts.
This means that the warning level can fluctuate on a value if the detector is unsure but it will rapidly go back to zero as soon as the detector is sure that there is not an anomaly.

# Bibliography

Keras Recurrent Layers, https://keras.io/api/layers/recurrent_layers/
LSTM for Anomaly Detection,
https://www.sciencedirect.com/science/article/pii/S240589631931554X