

# Autoencoders for Anomaly Detection

---

# High Performance Computing

## High Performance Computing

HPC refers to HW/SW infrastructures for particularly intensive workloads



# High Performance Computing

## HPC is (somewhat) distinct from cloud computing

- Cloud computing is mostly about running (and scaling services)
- ...HPC is all about **performance**

Typical applications: simulation, massive data analysis, training large ML models

## HPC systems follow a batch computation paradigm

- Users send **jobs** to the systems (i.e. configuration for running a program)
- Jobs end in one of several **queues**
- A **job scheduler** draws from the queue
- ...And dispatches jobs to computational **nodes** for execution

# High Performance Computing

## HPC systems can be large and complex

E.g. Leonardo, the 4-th most powerful supercomputer (as of June 2023).

4	<b>Leonardo</b> - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404
---	--	-----------	--------	--------	-------

- The system has 1,824,768 cores overall!

## Configuring (and maintaining the configuration) of these systems

- ...Is of very important, as it has an impact on the performance
- ...And very challenging, due to their **scale** and the to **node heterogeneity**

Hence the interest in **detecting anomalous conditions**

# The Dataset

**As an example, we will consider the DAVIDE system**

Small scale, energy-aware architecture:

- Top of the line components (at the time), liquid cooled
- An advanced monitoring and control infrastructure (ExaMon)
- ...Developed together with UniBo

The system went out of production in January 2020

**The monitoring system enables anomaly detection**

- Data is collected from a number of samples with high-frequency
- Long term storage only for averages over 5 minute intervals
- Anomalies correspond to unwanted configurations of the frequency governor
- ...Which can throttle performance to save power or prevent overheating

# A Look at the Dataset

Our dataset refers to the non-idle periods of a single node

```
In [2]: print(f'#examples: {hpc.shape[0]}, #columns: {hpc.shape[1]}')  
hpc.iloc[:3]
```

```
#examples: 6667, #columns: 161
```

Out[2]:

	timestamp	ambient_temp	cmbw_p0_0	cmbw_p0_1	cmbw_p0_10	cmbw_p0_11	cmbw_p0_12	cmbw_p0_13	cmbw_p0_14	cmbw_p0_15
0	2018-03-05 22:45:00	0.165639	0.006408	0.012176	0.166835	0.238444	0.230092	0.145691	0.227682	0.000094
1	2018-03-05 22:50:00	0.139291	0.007772	0.057400	0.166863	0.238485	0.230092	0.145691	0.227682	0.176855
2	2018-03-05 22:55:00	0.141048	0.000097	0.000000	0.166863	0.238444	0.230092	0.145691	0.227682	0.252403

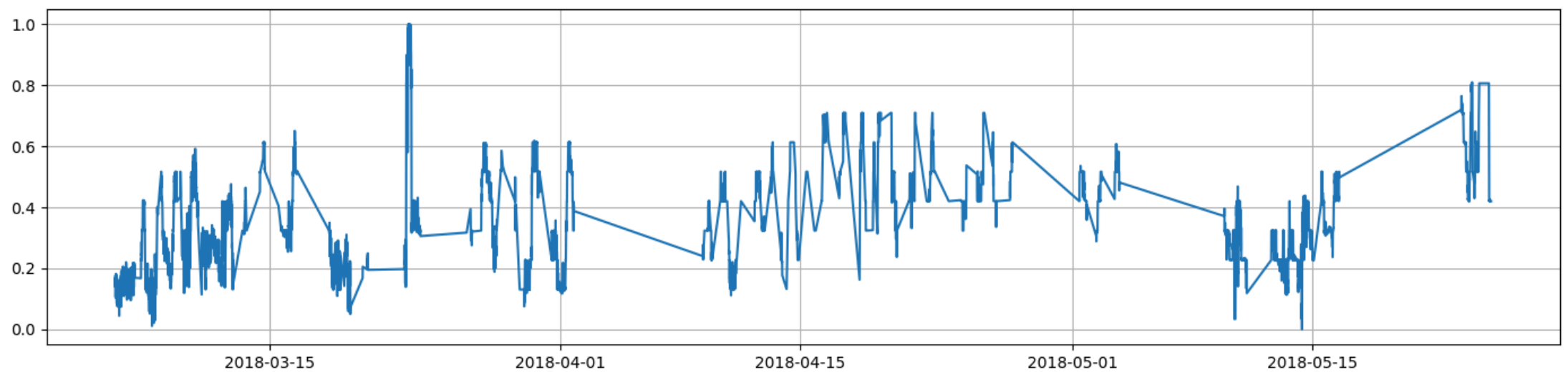
3 rows × 161 columns

- This still a time series, but a **multivariate** one

# A Look at the Dataset

How to display a multivariate series? Approach #1: showing **individual columns**

```
In [3]: tmp = pd.Series(index=hpc['timestamp'], data=hpc[inputs[0]].values)  
util.plot_series(tmp, figsize=figsize)
```



- The series contains significant gaps (i.e. the idle periods)

# A Look at the Dataset

## Approach #2: obtaining statistics

In [4]: `hpc[inputs].describe()`

Out [4]:

	ambient_temp	cmbw_p0_0	cmbw_p0_1	cmbw_p0_10	cmbw_p0_11	cmbw_p0_12	cmbw_p0_13	cmbw_p0_14	cmbw_p0_15
count	6667.000000	6667.000000	6667.000000	6667.000000	6667.000000	6667.000000	6667.000000	6667.000000	6667.000000
mean	0.357036	0.138162	0.060203	0.119616	0.160606	0.184970	0.118305	0.151434	0.143033
std	0.166171	0.128474	0.090796	0.098597	0.128127	0.163190	0.104490	0.120793	0.125052
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.227119	0.000073	0.000020	0.000000	0.000000	0.000000	0.000000	0.000000	0.000117
50%	0.323729	0.136095	0.000082	0.166835	0.238444	0.230092	0.145691	0.227682	0.174933
75%	0.470254	0.261908	0.134976	0.166984	0.238566	0.230406	0.145908	0.227779	0.251910
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

8 rows × 10 columns

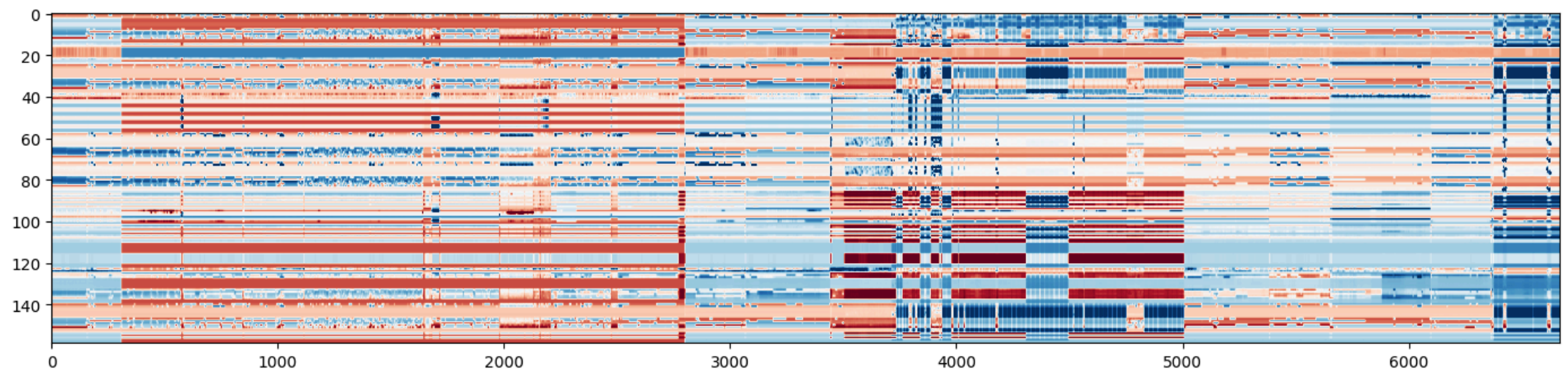
- No missing value, **normalized** data



# A Look at the Dataset

## Approach #3: standardize, then use a heatmap

```
In [5]: hpcsv = hpc.copy()  
hpcsv[inputs] = (hpcsv[inputs] - hpcsv[inputs].mean()) / hpcsv[inputs].std()  
util.plot_df_heatmap(hpcsv[inputs], figsize=figsize)
```



- White = mean, red = below mean, blue = above mean

# Anomalies

**There are three possible configurations of the frequency governor:**

- Mode 0 or "normal": frequency proportional to the workload
- Mode 1 or "power saving": frequency always at the minimum value
- Mode 2 or "performance": frequency always at the maximum value

**On this dataset, this information is known**

...And it will serve as our ground truth

- We will focus on discriminating normal from non-normal behavior
- I.e. we will treat both "power saving" and "performance" configurations as anomalous

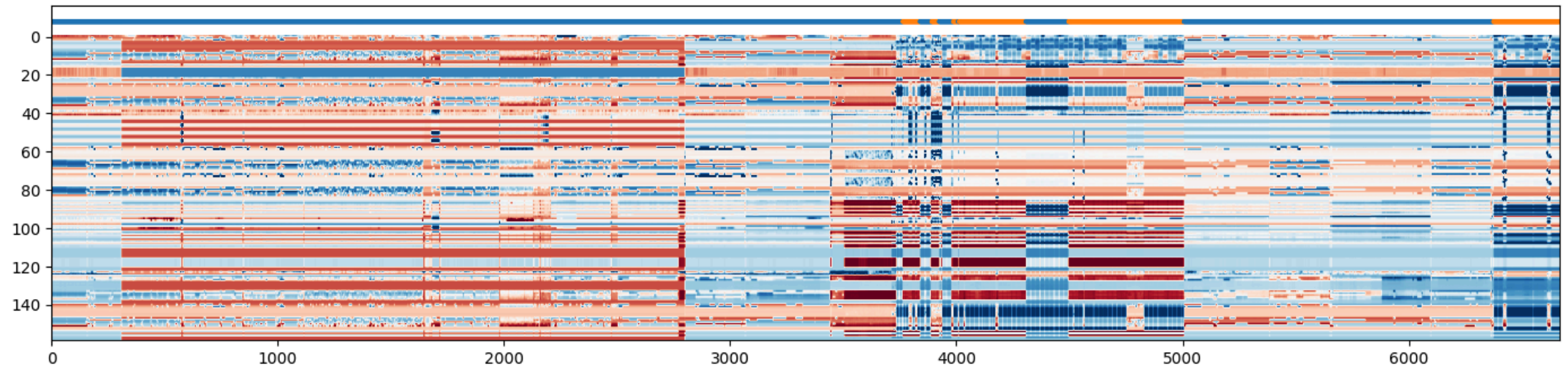
**Detecting them will be challenging**

- Since the signals vary so much when the running job changes

# Anomalies

We can plot the location of the anomalies:

```
In [6]: labels = pd.Series(index=hpcsv.index, data=(hpcsv['anomaly'] != 0), dtype=int)
util.plot_df_heatmap(hpcsv[inputs], labels, figsize=figsize)
```



- On the top, blue = normal, orange = anomaly

# Autoencoders for Anomaly Detection

---

# Autoencoders

An autoencoder is **a type of neural network**

The network is designed to **reconstruct its input vector**

- The input is some tensor  $\mathbf{x}$  and the output **should be** the same tensor  $\mathbf{x}$

**Autoencoders can be broken down in two halves**

- An encoding part, i.e.  $encode(\mathbf{x}, \theta_e)$ , mapping  $\mathbf{x}$  into a vector of **latent variables  $\mathbf{z}$**
- A decoding part, i.e.  $decode(\mathbf{z}, \theta_d)$ , mapping  $\mathbf{z}$  into reconstructed input tensor

**Autoencoders are trained so as to satisfy:**

$$decode(encode(\hat{x}_i, \theta_e), \theta_d) \simeq \hat{x}_i$$

- I.e. *decode*, when applied to the output of *encode*
- ...Should approximately return the input vector itself

A nice introduction and tutorial about autoencoders can be found [on the Keras](#)

# Autoencoders

Formally, we typically employ an MSE loss

$$L(\theta_e, \theta_d) = \sum_{i=1}^n \|\hat{x}_i - \text{decode}(\text{encode}(\hat{x}_i, \theta_e), \theta_d)\|_2^2$$

- This is trivial to satisfy if both *encode* and *decode* learn an identity relation
- ...So we need to prevent that

**There are two main approaches to avoid learning a trivial mapping**

- Using an *information bottleneck*, i.e. making sure that  $\mathbf{z}$  has fewer dimensions than  $\mathbf{x}$
- Use a regularization to enforce *sparse encodings*, e.g.:

$$L(\theta_e, \theta_d) = \sum_{i=1}^n \|\hat{x}_i - \text{decode}(\text{encode}(\hat{x}_i, \theta_e), \theta_d)\|_2^2 + \alpha \|\text{encode}(x, \theta_e)\|_1$$

# Autoencoders for Anomaly Detection

## Autoencoders can be used for anomaly detection

...By using the **reconstruction error as an anomaly signal**, e.g.:

$$\|x - \text{decode}(\text{encode}(x, \theta_e), \theta_d)\|_2^2 > \theta$$

## This approach has some PROs and CONs:

- Compared to KDE
  - Neural Networks have good **support for high dimensional data**
  - ...Plus **limited overfitting** and **fast prediction/detection time**
  - However, error reconstruction can be **harder than density estimation**
- Compared to autoregressors
  - Reconstructing an input is **easier than predicting the future**
  - ...So, we tend to get higher reliability

# Autoencoders in Keras

## Let's build an autoencoder in practice (with tensorflow 2.0 and keras)

First, we build the model

```
In [7]: input_shape = (len(inputs), )
        ae_x = keras.Input(shape=input_shape, dtype='float32')
        ae_z = layers.Dense(64, activation='relu')(ae_x)
        ae_y = layers.Dense(len(inputs), activation='linear')(ae_z)
        ae = keras.Model(ae_x, ae_y)
```

In this case, we used the keras functional API

- **Input** builds the entry point for the input data
- **Dense** builds a fully connected layer
- "Calling" layer A with parameter B attaches B to A
- **Model** builds a model object with the specified input and output



# Autoencoders in Keras

Then we compile (prepare for training) the model

```
In [8]: ae.compile(optimizer='adam', loss='mse')
```

Finally we can start training:

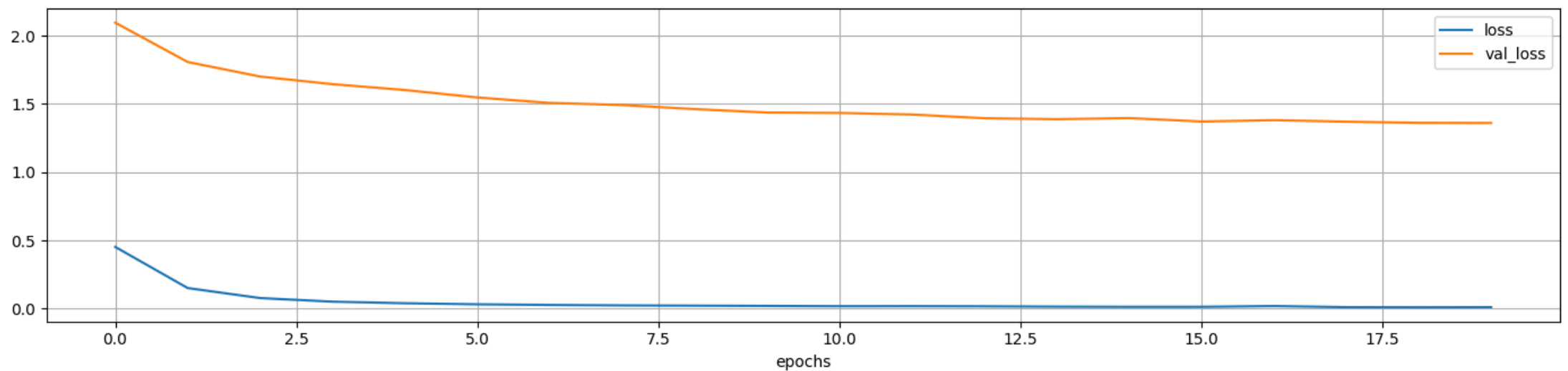
```
In [9]: cb = [callbacks.EarlyStopping(patience=3, restore_best_weights=True)]  
history = ae.fit(trdata[inputs], trdata[inputs], validation_split=0.1,  
                callbacks=cb,  
                batch_size=32, epochs=20, verbose=0)
```

- We are using a callback to stop training early
- ...If no improvement on the validation set is observed for 3 epochs

# Autoencoders in Keras

Let's have a look at the loss evolution over different epochs

```
In [10]: util.plot_training_history(history, figsize=figsize)
```



Final loss: 0.0116 (training), 1.3608 (validation)

# Autoencoders in Keras

Finally, we can obtain the predictions

```
In [11]: preds = pd.DataFrame(index=hpcs.index, columns=inputs, data=ae.predict(hpcs[inputs], verbose=0))
         preds.head()
```

Out[11]:

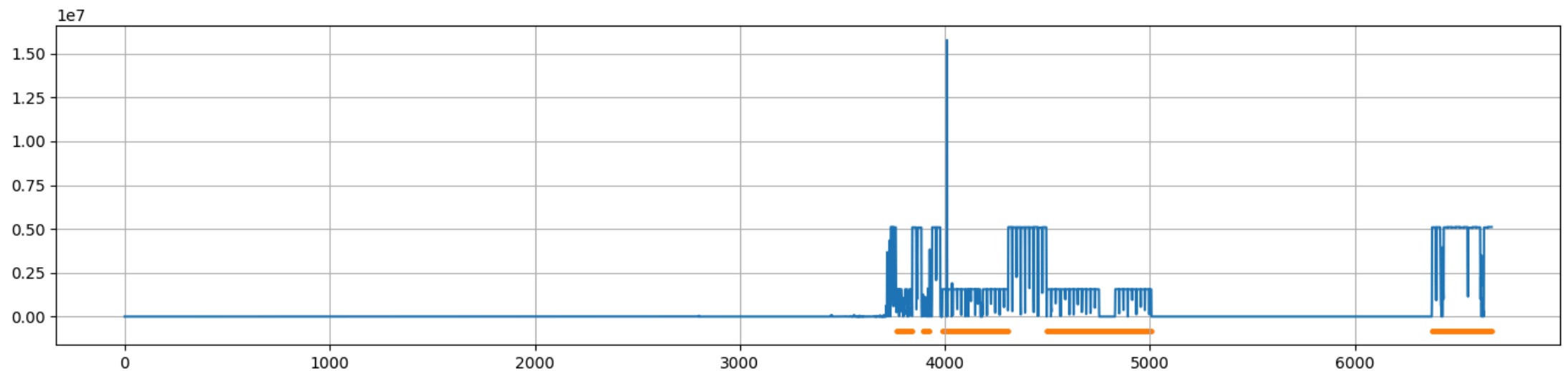
	ambient_temp	cmbw_p0_0	cmbw_p0_1	cmbw_p0_10	cmbw_p0_11	cmbw_p0_12	cmbw_p0_13	cmbw_p0_14	cmbw_p0_2	cmbw_p0_3
0	-1.412239	-0.843257	-0.401748	2.171150	1.943048	2.349579	2.030199	2.199551	-1.968331	-1.05065
1	-0.705542	-0.502070	0.019735	2.149246	2.061388	2.109589	2.245787	2.245715	0.423151	-0.38171
2	-1.127019	-0.847340	-0.590582	2.316129	2.215113	2.269886	2.223497	2.241434	0.319195	0.551385
3	-0.903677	-0.804386	-0.386561	2.175131	2.176953	2.173117	2.233175	2.217358	0.690856	0.987056
4	-1.165355	-0.784210	-0.561632	2.258652	2.137391	2.184276	2.231875	2.277363	0.757158	1.012378

5 rows × 159 columns

# Alarm Signal

We can finally obtain our alarm signal, i.e. the sum of squared errors

```
In [12]: sse = np.sum(np.square(preds - hpcs[inputs]), axis=1)
signal_ae = pd.Series(index=hpcs.index, data=sse)
util.plot_signal(signal_ae, labels, figsize=figsize)
```



- It is actually quite similar to the KDE signal

# Threshold Optimization

Then we can optimize the threshold as usual

```
In [13]: th_range = np.linspace(5e5, 1e6, 100)
th_ae, val_cost_ae = util.opt_threshold(signal_ae[tr_end:val_end],
                                       hpcs['anomaly'][tr_end:val_end],
                                       th_range, cmodel)

print(f'Best threshold: {th_ae:.3f}')
tr_cost_ae = cmodel.cost(signal_ae[:tr_end], hpcs['anomaly'][:tr_end], th_ae)
print(f'Cost on the training set: {tr_cost_ae}')
print(f'Cost on the validation set: {val_cost_ae}')
ts_cost_ae = cmodel.cost(signal_ae[val_end:], hpcs['anomaly'][val_end:], th_ae)
print(f'Cost on the test set: {ts_cost_ae}')
```

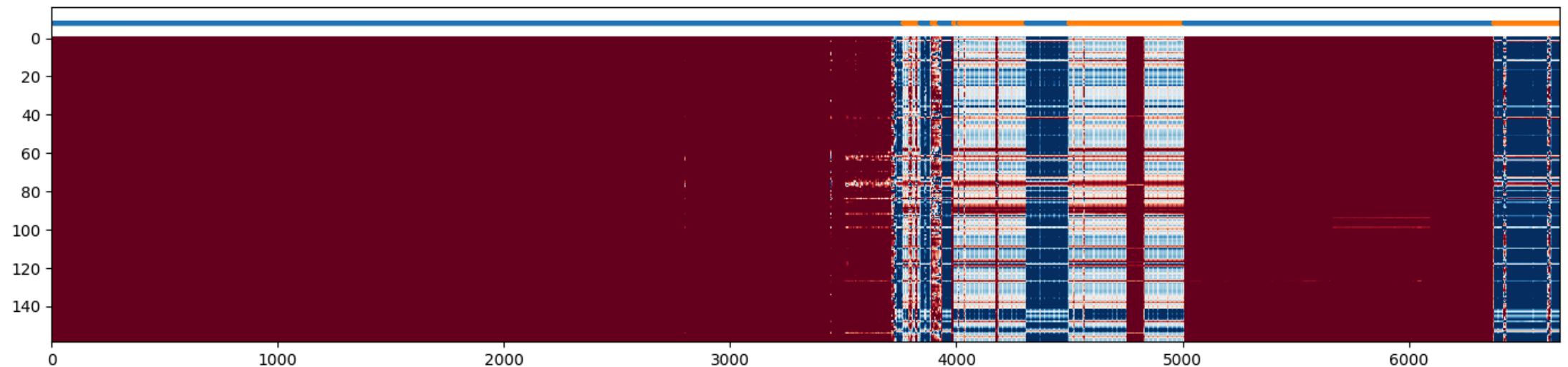
```
Best threshold: 974747.475
Cost on the training set: 0
Cost on the validation set: 249
Cost on the test set: 275
```

# Multiple Signal Analysis

But autoencoders do **more than just anomaly detection!**

- Instead of having a single signal we have **many**
- So we can look at the **individual** reconstruction errors

```
In [14]: se = np.sqrt(np.square(preds - hpcs[inputs]))  
signals_ae = pd.DataFrame(index=hpcs.index, columns=inputs, data=se)  
util.plot_df_heatmap(signals_ae, labels, vmin=np.quantile(se, 0.25), vmax=np.quantile(se, 0.95))
```

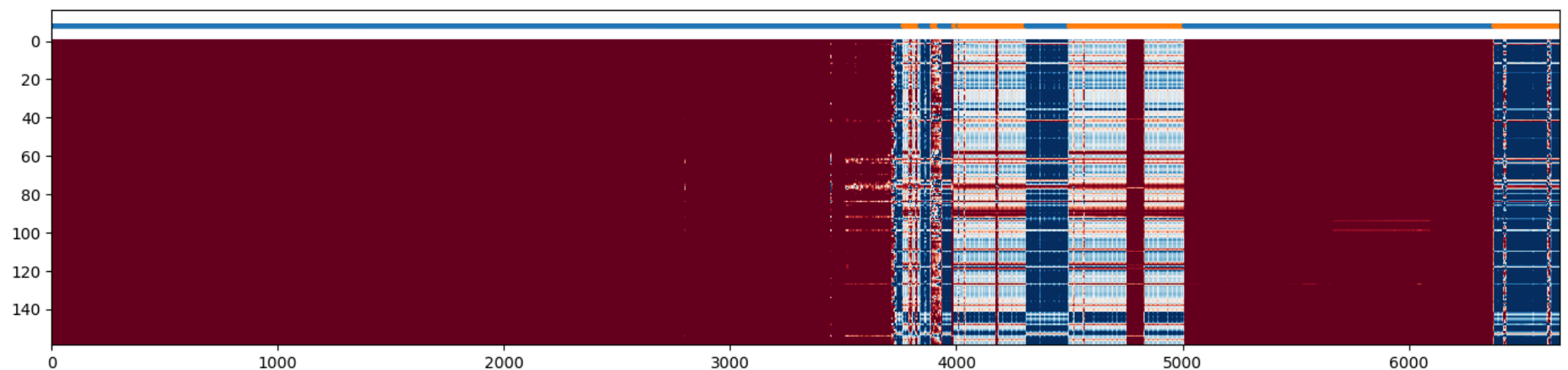


# Multiple Signal Analysis

## Reconstruction errors are often concentrated on a few signals

- These correspond to the properties of the input vector that were harder to reconstruct
- ...And often they are useful clues about the **nature of the anomaly**

```
In [15]: se = np.sqrt(np.square(preds - hpcs[inputs]))  
signals_ae = pd.DataFrame(index=hpcs.index, columns=inputs, data=se)  
util.plot_df_heatmap(signals_ae, labels, vmin=np.quantile(se, 0.25), vmax=np.quantile(se, 0
```



# Multiple Signal Analysis

Let's focus on the last **mode 1** anomaly ("power saving" mode)

Here are the 8 largest errors in descending order

```
In [16]: last_mode_1 = hpcs.index[hpcs['anomaly']==1][-1]
          se.iloc[last_mode_1].sort_values(ascending=False)[:8]
```

```
Out[16]: ips_p0_14      525.874402
          ips_p0_10      492.072065
          ips_p0_12      419.044014
          ips_p0_11      367.220421
          ips_p0_8       344.009226
          ips_p0_9       258.454398
          util_p0_12      228.343085
          util_p0_10      191.123226
          Name: 5006, dtype: float64
```

- They are mostly related to performance (e.g. "ips" - Instructions Per Second)
- ...As it should be!



# Multiple Signal Analysis

Now, let's move to the last **mode 2** anomaly ("performance" mode)

Here are the 8 largest errors in descending order

```
In [17]: last_mode_2 = hpcs.index[hpcs['anomaly']==2][-1]
         se.iloc[last_mode_2].sort_values(ascending=False)[:8]
```

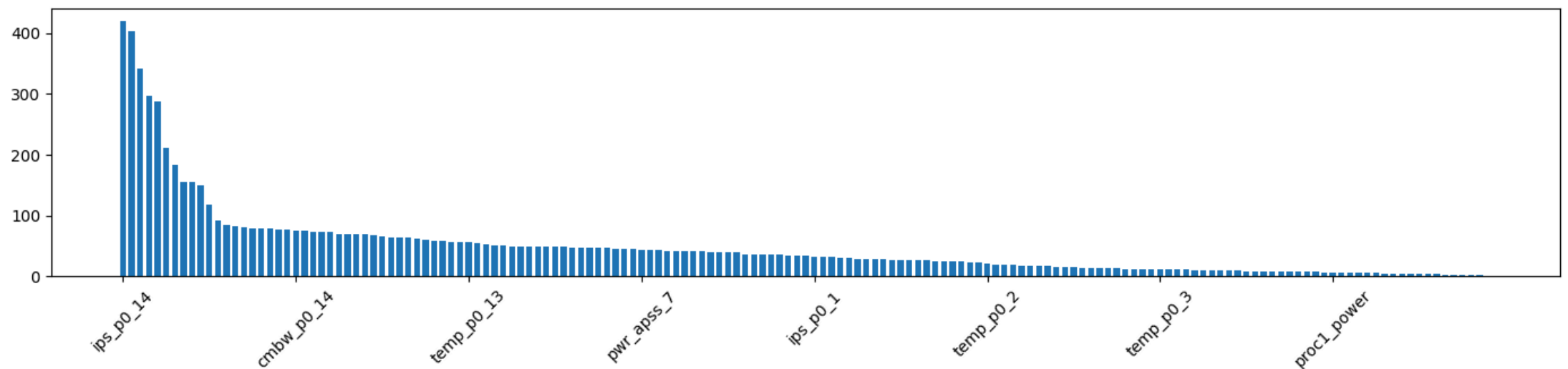
```
Out[17]: ips_p0_14      1022.077539
         ips_p0_10      958.192172
         ips_p0_12      833.876066
         ips_p0_11      734.644767
         ips_p0_8       673.767698
         ips_p0_9       537.029292
         ips_p0_13      233.014918
         util_p0_12     224.594563
         Name: 6666, dtype: float64
```

- Again, they are performance related

# Multiple Signal Analysis

Here are the **average errors** for mode 1 anomalies

```
In [18]: mode_1 = hpcs.index[hpcs['anomaly']==1]  
tmp = se.iloc[mode_1].mean().sort_values(ascending=False)  
util.plot_bars(tmp, tick_gap=20, figsize=figsize)
```

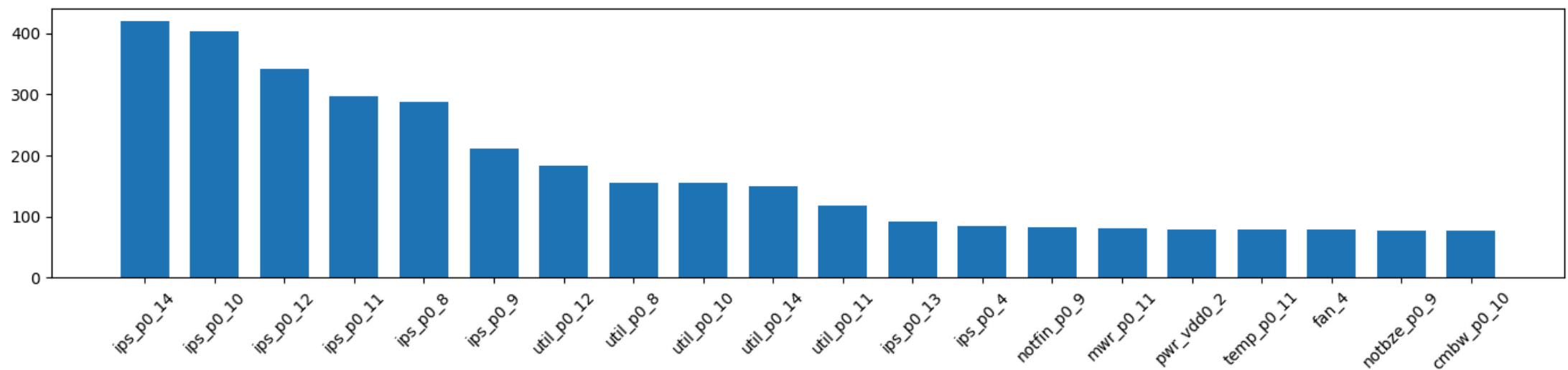


- Errors are concentrated on a small number of features

# Multiple Signal Analysis

These are the 20 **largest** average errors for **mode 1** anomalies

```
In [19]: mode_1 = hpcs.index[hpcs['anomaly']==1]
tmp = se.iloc[mode_1].mean().sort_values(ascending=False)
util.plot_bars(tmp.iloc[:20], figsize=figsize)
```

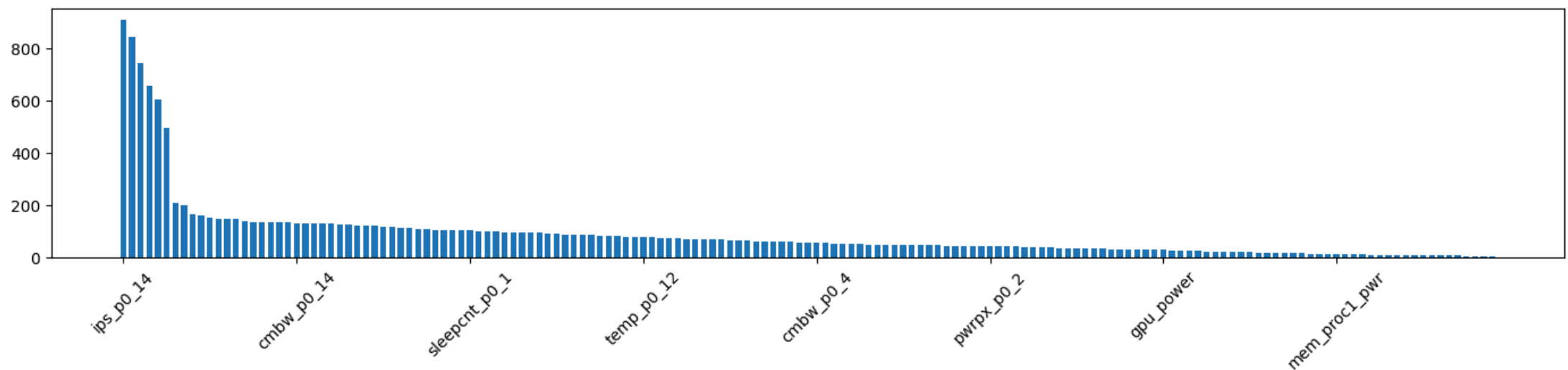


- The largest errors are on "ips", then on "util" (utilization)

# Multiple Signal Analysis

Let's repeat the analysis for **mode 2**. Here are the **average errors**

```
In [20]: mode_2 = hpcs.index[hpcs['anomaly']==2]  
tmp = se.iloc[mode_2].mean().sort_values(ascending=False)  
util.plot_bars(tmp, tick_gap=20, figsize=figsize)
```

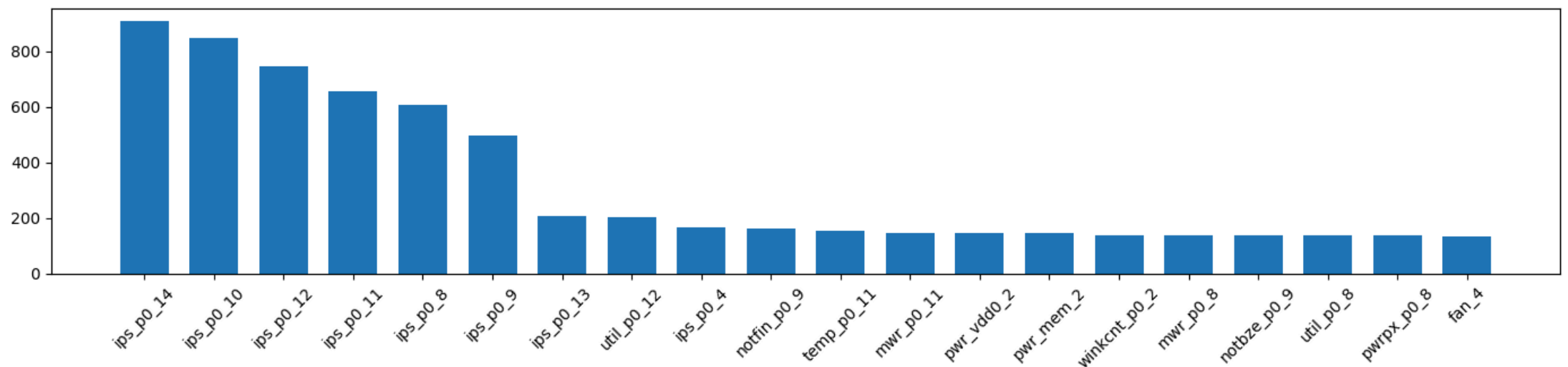


- The situation is similar to mode 1

# Multiple Signal Analysis

## The 20 largest average errors for mode 2

```
In [21]: mode_2 = hpcs.index[hpcs['anomaly']==2]
tmp = se.iloc[mode_2].mean().sort_values(ascending=False)
util.plot_bars(tmp.iloc[:20], figsize=figsize)
```



- The largest errors are on "ips", then on power signals

# Considerations

## **Autoencoders can be used for anomaly detection**

- They provide the usual benefits of Neural Networks
  - E.g. scalability, limited overfitting, limited need for preprocessing
- They tend to be more reliable than autoregressors
- They provide more fine grained information than density estimation
- ...And you can make them **deep**!

## **Analyzing individual efforts provides clues about the anomalies**

- In this case, we manage to focus on 10-20 features, rather than 160!

## **Density estimation is (usually) a bit better at pure anomaly detection**

- ...But there is no reason not to use both approaches!
- E.g. density estimation for detection, autoencoders for the analysis