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. Marconi-100 at CINECA, which was the 9-th most powerful supercompuer (as of June 2020)

9	**Marconi-100** - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476

- The system has 31,360 cores overall!

Configuring (and maintaining the configuration) of these systems

- ...Is of paramount important, as it has an impact on the performance
- ...Is very challenging, due to their large scale and the presence of 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 [3]:
```
print(f'#examples: {hpc.shape[0]}, #columns: {hpc.shape[1]}')
hpc.iloc[:3]
```

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

Out[3]:

	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_2
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

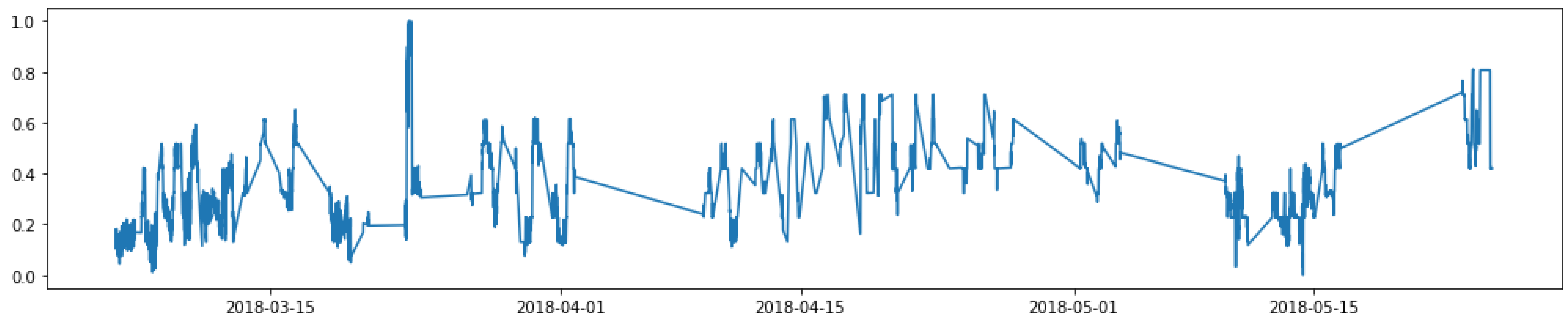
3 rows × 161 columns

- 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 [4]: tmp = pd.Series(index=hpc['timestamp'], data=hpc[inputs[0]].values)
        nn.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 [5]:
```
hpc[inputs].describe()
```

Out[5]:

	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
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

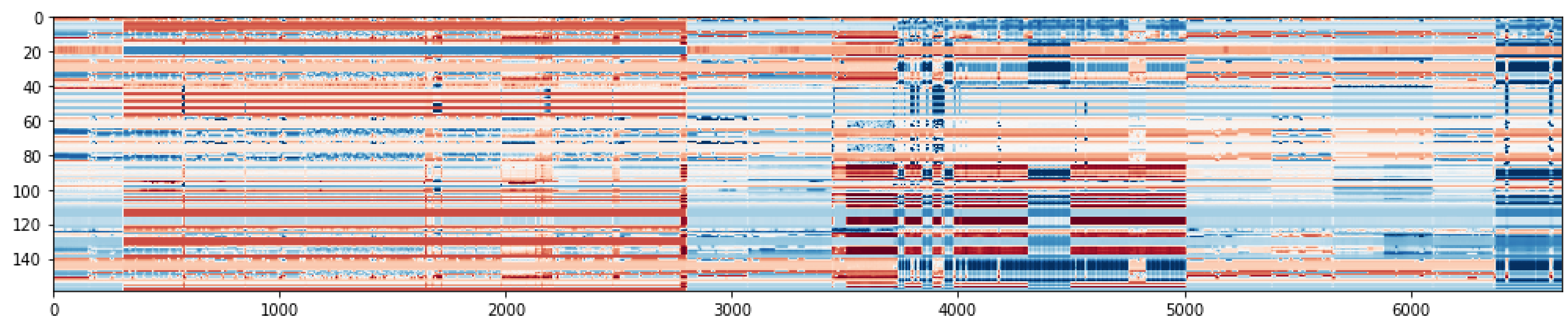
8 rows × 159 columns

- No missing value, normalized data

A Look at the Dataset

Approach #3: standardize, then use a heatmap

In [6]:
```
hpcsv = hpc.copy()
hpcsv[inputs] = (hpcsv[inputs] - hpcsv[inputs].mean()) / hpcsv[inputs].std()
nn.plot_dataframe(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

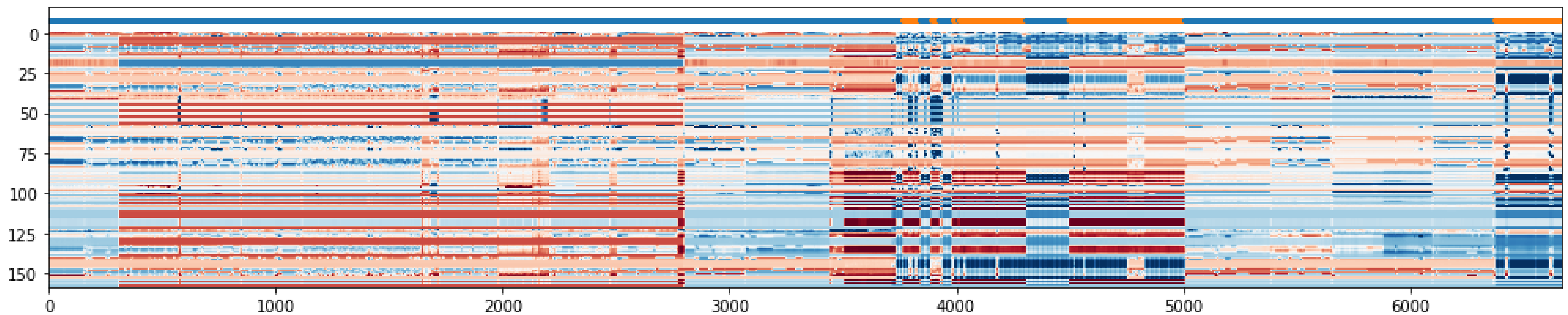
Detecting them will be challenging

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

Anomalies

We can plot the location of the anomalies:

In [7]:
```
labels = pd.Series(index=hpcsv.index, data=(hpcsv['anomaly'] != 0), dtype=int)
nn.plot_dataframe(hpcsv[inputs], labels, figsize=figsize)
```



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

A KDE Approach

KDE Approach

Let's try first a density estimation approach (once again using KDE)

First, we need to standardize the data again, based on training information alone

In [8]:
```
tr_end, val_end = 3000, 4500

hpcs = hpc.copy()
tmp = hpcs.iloc[:tr_end]
hpcs[inputs] = (hpcs[inputs] - tmp[inputs].mean()) / tmp[inputs].std()
```

- This is needed so that we do not accidentally exploit test set information
- The training set separator was chosen so as not to include anomalies

Then we can separate training, validation, and test data:

In [9]:
```
trdata = hpcs.iloc[:tr_end]
valdata = hpcs.iloc[tr_end:val_end]
tsdata = hpcs.iloc[val_end:]
```

A KDE Approach

Then we estimate the optimal bandwidth:

```
In [10]: params = {'bandwidth': np.linspace(0.1, 1, 10)}
         opt = GridSearchCV(KernelDensity(kernel='gaussian'), params, cv=5)
         opt.fit(trdata[inputs])
         opt.best_params_

Out[10]: {'bandwidth': 0.5}
```

...Ad we can train the estimator and generate the anomaly signal:

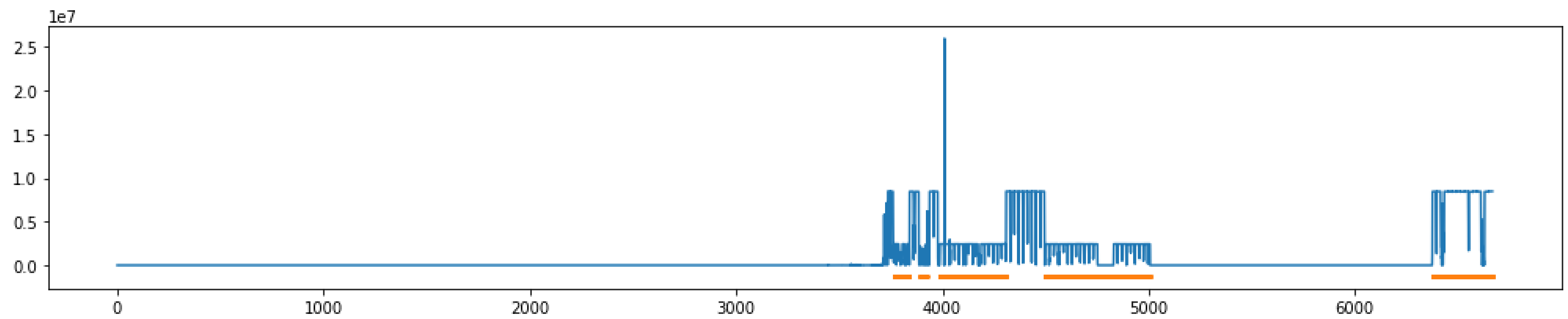
```
In [11]: h = opt.best_params_['bandwidth']
         kde = KernelDensity(bandwidth=h)
         kde.fit(trdata[inputs])
         ldens = kde.score_samples(hpcs[inputs])
         signal_kde = pd.Series(index=hpcs.index, data=-ldens)
```

- Tuning the bandwidth and obtaining densities are relatively expensive operations

KDE Approach

There is a good match with the anomalies, but also many spurious peaks

In [12]:
```
nn.plot_signal(signal_kde, labels, figsize=figsize)
```



- This is mostly due to the large variations due to job changes

KDE Approach

We then need to define the threshold, but for that we need a cost model

Our main goal is to detect anomalies, not anticipating them

- Misconfigurations in HPC are usually not critical
- ...And cause little issue, unless they stay unchecked for very long

We will use a simple cost model:

- c_{alarm} for false positive (erroneous detections)
- c_{missed} for false negatives (undetected anomalies)
- Detections are fine as long as they are within $tolerance$ units from the anomaly

In [13]:
```
c_alarm, c_missed, tolerance = 1, 5, 12
cmodel =  nn.HPCMetrics(c_alarm, c_missed, tolerance)
```

The implementation details can be found in the `nn` utility module

KDE Approach

We can now optimize the threshold over the validation set

```
In [14]: th_range = np.linspace(1e4, 1e5, 100)
         th_kde, val_cost_kde = nn.opt_threshold(signal_kde[tr_end:val_end],
                                                 valdata['anomaly'],
                                                 th_range, cmodel)
         print(f'Best threshold: {th_kde:.3f}')
         tr_cost_kde = cmodel.cost(signal_kde[:tr_end], hpcs['anomaly'][:tr_end], th_kde)
         print(f'Cost on the training set: {tr_cost_kde}')
         print(f'Cost on the validation set: {val_cost_kde}')
         ts_cost_kde = cmodel.cost(signal_kde[val_end:], hpcs['anomaly'][val_end:], th_kde)
         print(f'Cost on the test set: {ts_cost_kde}')

         Best threshold: 82727.273
         Cost on the training set: 0
         Cost on the validation set: 266
         Cost on the test set: 265
```

- The `opt_threshold` function runs the usual line search process
- In this case the training and validation set are completely separated

The Trouble with KDE

KDE-based approach works well, but have some issues

First, KDE itself runs into trouble with high-dimensional data:

- With a larger dimensionality, prediction times grows...
- ...And more data is needed to obtain reliable results
- We are not seeing that too much, but eventually it will become a problem

Second, KDE gives you nothing more than an anomaly signal

- Determining the cause of the anomaly is up to a domain expert
- ...Who needs to take a look at all the supposed anomalous data
- This is doable in low-dimensional spaces, but way harder on high-dimensional one

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 x and the output should be the same tensor x

Autoencoders can be broken down in two halves

- An encoding part, i.e. $encode(x, \theta_e)$, mapping x into a vector of latent variables z
- A decoding part, i.e. $decode(z, \theta_d)$, mapping 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

Autoencoders

Formally, we typically employ an MSE loss

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

- 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 z has fewer dimensions that x
- Use a regularization to enforce sparse encodings, e.g.:

Autoencoders for Anomaly Detection

Autoencoders can be used for anomaly detection

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

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

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 using (e.g.) the functional API

In [15]:
```
from tensorflow import keras
from tensorflow.keras import layers, callbacks

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)
```

- `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 can prepare our model for training

In keras terms, we compile it:

```
In [16]: ae.compile(optimizer='RMSProp', loss='mse')
```

- We are using the `RMSProp` optimizer (a variant of Stochastic Gradient Descent)

Then we can start training:

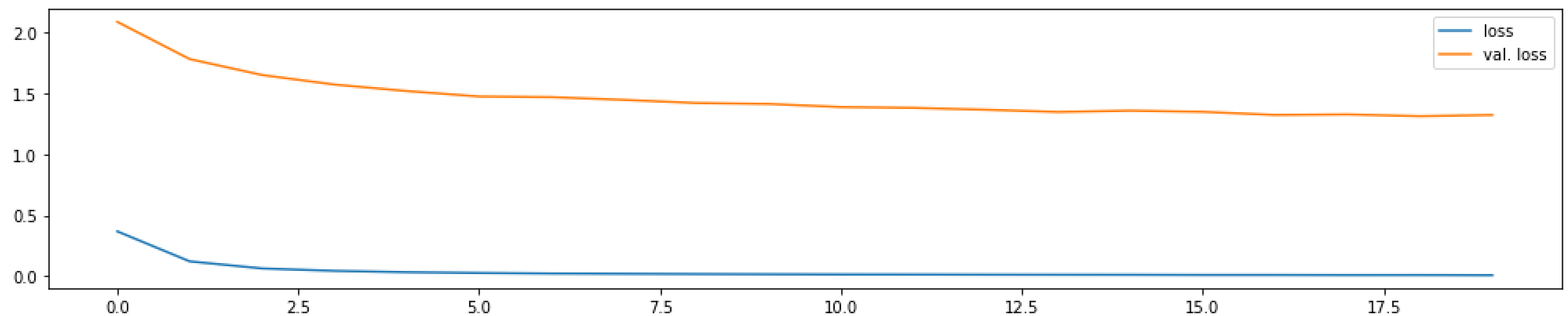
```
In [17]: cb = [callbacks.EarlyStopping(patience=3, restore_best_weights=True)]
         history = ae.fit(trdata[inputs], trdata[inputs], validation_split=0.1,
                          callbacks=cb,
                          batch_size=32, epochs=20, verbose=0)
```

- 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 [18]:
```
nn.plot_training_history(history, figsize=figsize)
```



Autoencoders in Keras

Finally, we can obtain the predictions

In [19]:
```
preds = pd.DataFrame(index=hpcs.index, columns=inputs, data=ae.predict(hpcs[inputs]))
preds.head()
```

Out[19]:

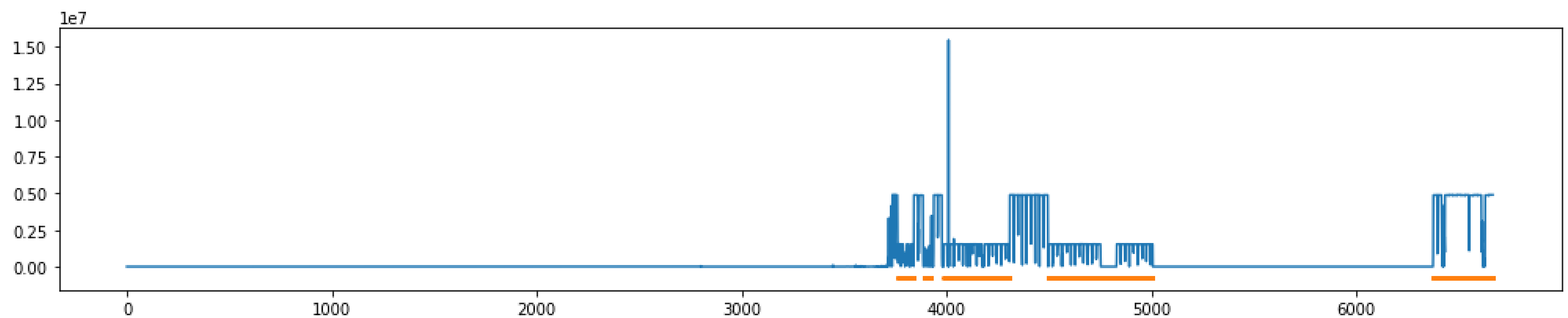
	ambient_temp	cmbw_p0_0	cmbw_p0_1	cmbw_p0_10	cmbw_p0_11	cmbw_p0_12	cmbw_p0_13	cmbw_p0_14	cmbw_p0_2	cmbw_p0_:
0	-0.243467	-0.644425	-0.085428	1.969099	2.288656	1.892106	2.233884	1.789767	-1.515644	-0.648156
1	-0.829325	-0.414412	-0.041981	2.225615	2.237881	2.219681	2.185933	2.292887	0.473269	-0.791788
2	-1.136139	-0.787518	-0.634200	2.250645	2.318117	2.340399	2.272343	2.270084	0.375756	0.450275
3	-0.797620	-0.639444	-0.686028	2.104995	2.209446	2.167086	2.186000	2.131691	0.641634	0.912332
4	-0.948215	-0.702033	-0.597398	2.151001	2.248978	2.192173	2.200649	2.205288	0.761485	0.783307

5 rows × 159 columns

Alarm Signal

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

In [20]:
```
sse = np.sum(np.square(preds - hpcs[inputs]), axis=1)
signal_ae = pd.Series(index=hpcs.index, data=sse)
nn.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 [21]: th_ae, val_cost_ae = nn.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: 93636.364
         Cost on the training set: 0
         Cost on the validation set: 264
         Cost on the test set: 265
```

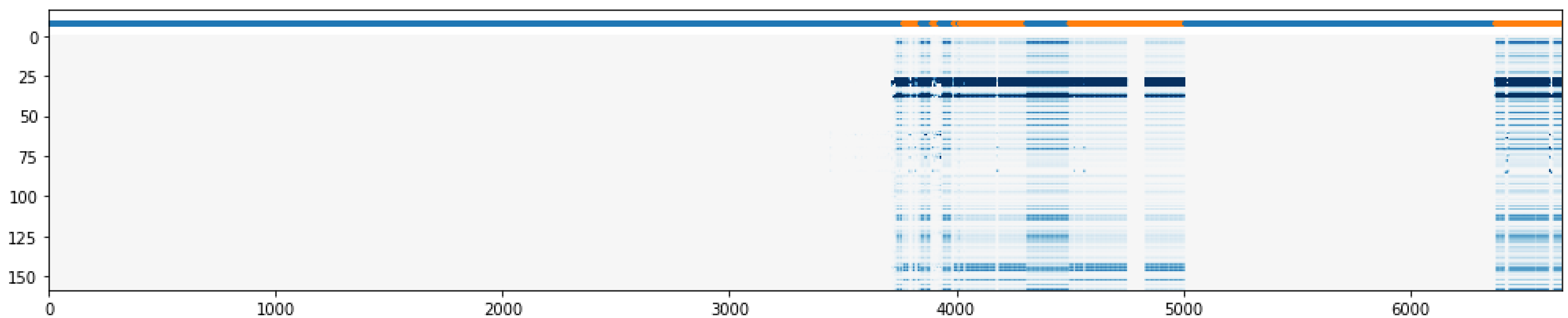
- We have more or less the same performance as KDE

Mutiple 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 [22]:
```
se = np.square(preds - hpcs[inputs])
signals_ae = pd.DataFrame(index=hpcs.index, columns=inputs, data=se)
nn.plot_dataframe(signals_ae, labels, vmin=-5e4, vmax=5e4, figsize=figsize)
```

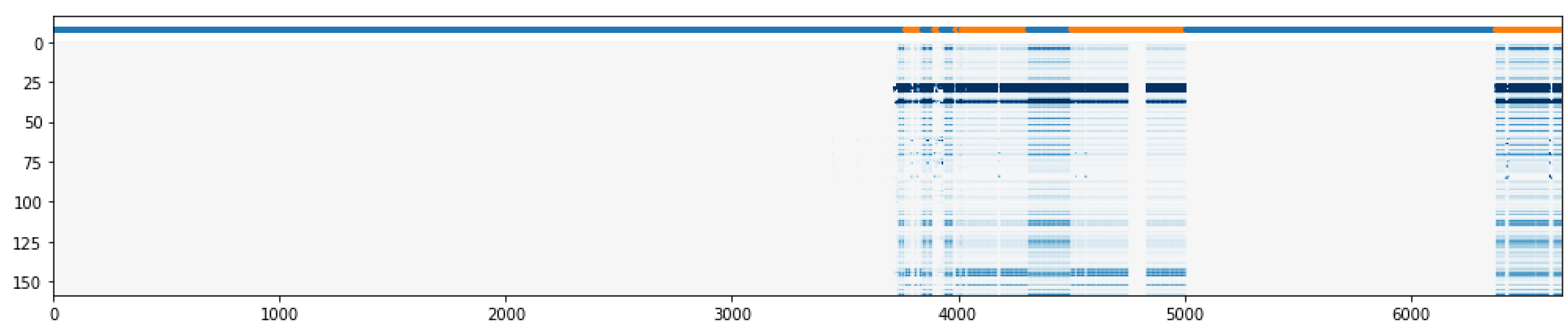


Mutiple Signal Analysis

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 [23]:
```
se = np.square(preds - hpcs[inputs])
signals_ae = pd.DataFrame(index=hpcs.index, columns=inputs, data=se)
nn.plot_dataframe(signals_ae, labels, vmin=-5e4, vmax=5e4, figsize=figsize)
```



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 [24]:
```
last_mode_1 = hpcs.index[hpcs['anomaly']==1][-1]
se.iloc[last_mode_1].sort_values(ascending=False)[:8]
```

Out[24]:
```
ips_p0_14      227888.971474
ips_p0_10      206312.780181
ips_p0_12      173627.325051
ips_p0_11      107111.104069
ips_p0_8        87152.130845
ips_p0_9        54111.813945
util_p0_14      38777.956789
util_p0_12      35632.388580
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 [25]:
```
last_mode_2 = hpcs.index[hpcs['anomaly']==2][-1]
se.iloc[last_mode_2].sort_values(ascending=False)[:8]
```

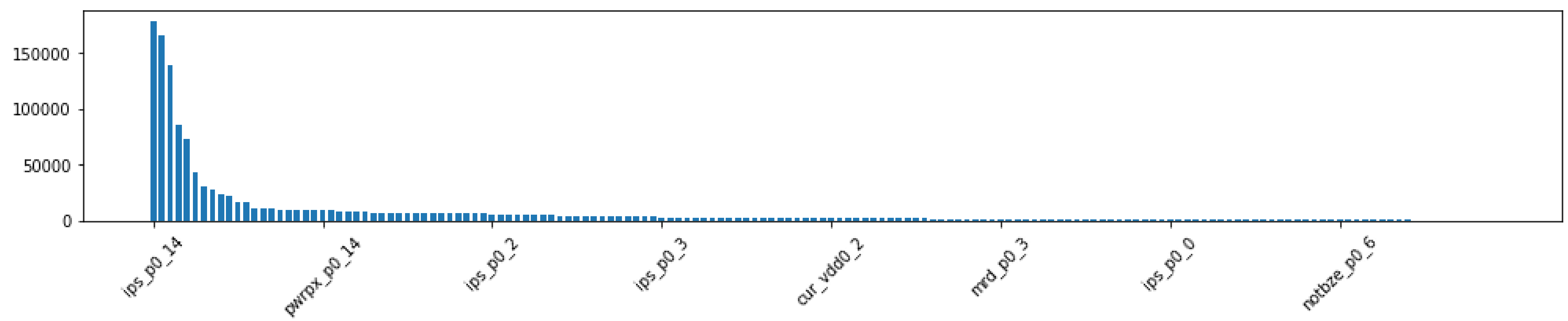
Out[25]:
```
ips_p0_14      896082.860429
ips_p0_10      814649.900693
ips_p0_12      674098.635930
ips_p0_11      449137.540408
ips_p0_8       338367.894462
ips_p0_9       212697.840918
ips_p0_13       44724.649477
cmbw_p0_11      39892.651824
Name: 6666, dtype: float64
```

- Again, they are performance related

Multiple Signal Analysis

Here are the average errors for mode 1 anomalies

In [26]:
```
mode_1 = hpcs.index[hpcs['anomaly']==1]
tmp =  se.iloc[mode_1].mean().sort_values(ascending=False)
nn.plot_bars(tmp, tick_gap=20, figsize=figsize)
```

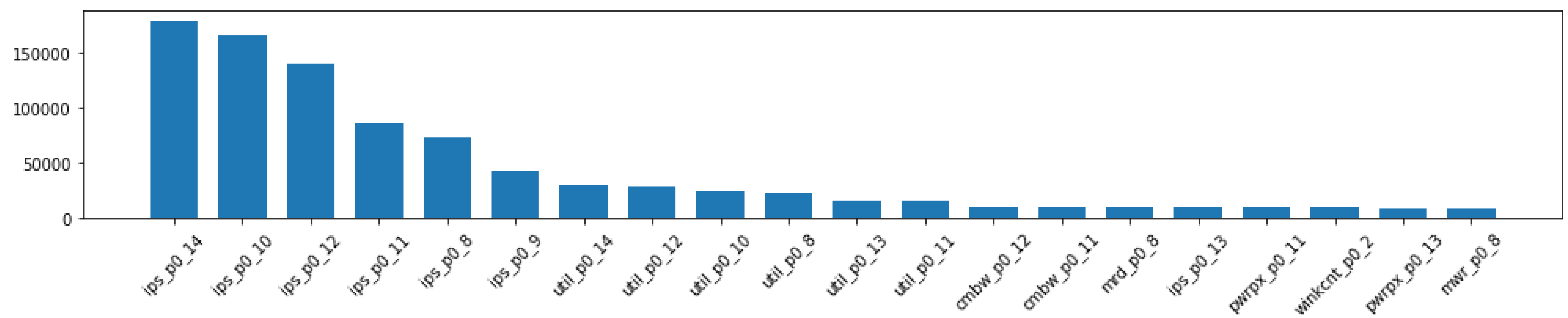


- Errors are concentrated on 10-20 features

Multiple Signal Analysis

These are the 20 largest average errors for mode 1 anomalies

In [27]:
```
mode_1 = hpcs.index[hpcs['anomaly']==1]
tmp =  se.iloc[mode_1].mean().sort_values(ascending=False)
nn.plot_bars(tmp.iloc[:20], figsize=figsize)
```

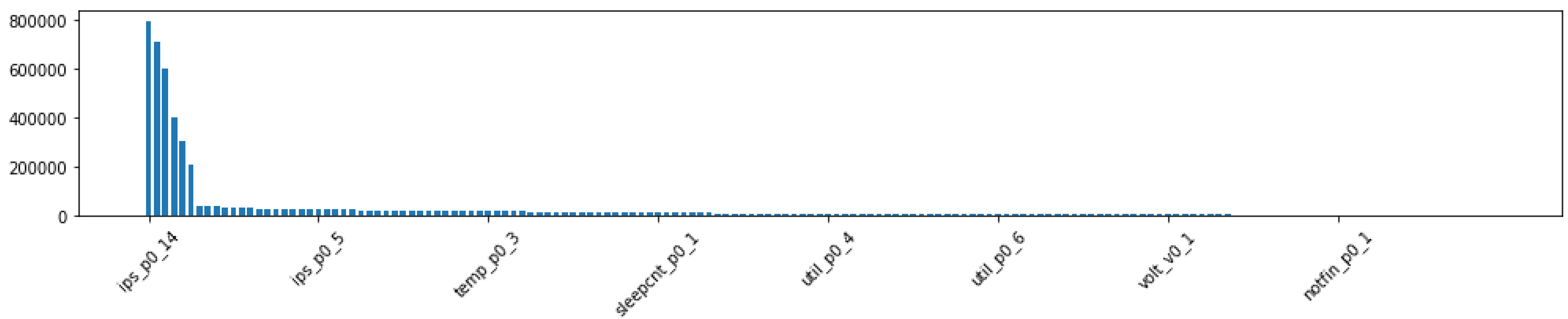


- The largest errors are on "ips", then on "util" (utilization)

Multiple Signal Analysis

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

```
In [28]: mode_2 = hpcs.index[hpcs['anomaly']==2]
         tmp =  se.iloc[mode_2].mean().sort_values(ascending=False)
         nn.plot_bars(tmp, tick_gap=20, figsize=figsize)
```

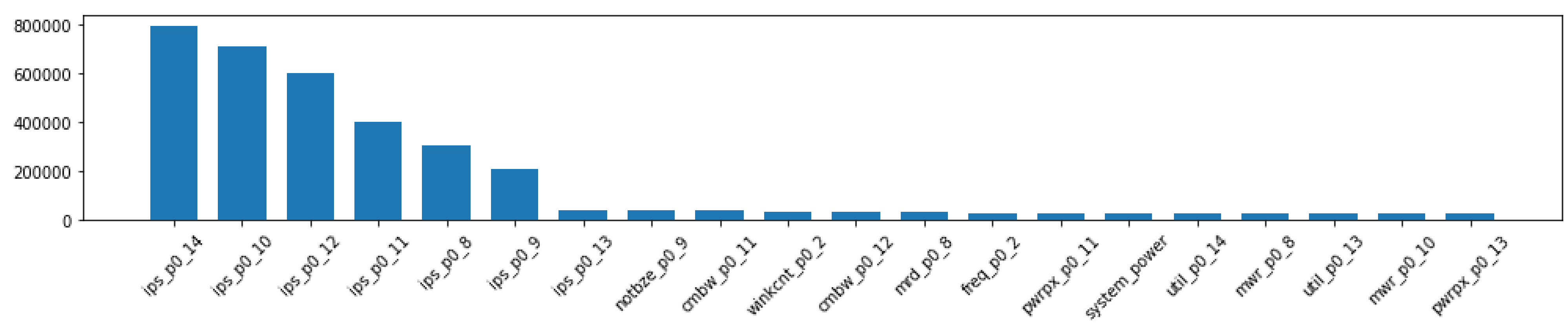


- The situation is similar to mode 1

Multiple Signal Analysis

The 20 largest average errors for mode 2

In [29]:
```
mode_2 = hpcs.index[hpcs['anomaly']==2]
tmp =  se.iloc[mode_2].mean().sort_values(ascending=False)
nn.plot_bars(tmp.iloc[:20], figsize=figsize)
```



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

Considerations

Autoenders can be used for anomaly detection

- The 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