Component Wear Anomalies

OCME Vega Shrinker

Let's consider the Vega skinwrapper family of packaging machines by OCME

- They work by wrapping products (bottles) in a plastic film
- ...Which is cut and heated, so that the film shrinks and stabilizes the content

OCME Vega Shrinker

A public dataset about one of their machines is publicly available from Kaggle

- The dataset contains a run-to-failure experiment
- I.e. the machine was left running until one of it components became unserviceable
  - Specificallly, it was the blade for cutting the film

This is an example of anomaly due to component wear

- It's a common type of anomaly
- ...And run-to-failure experiments are a typical way to investigate them

All problems in this class share a few properties

- The behavior becomes more and more distant from normal over time
- There is a critical anomaly at the end of the experiment

We will try to tackle the problem using the techniques we know

The Dataset

Let's have a first look at the dataset

In [2]:
```
print(f'Number of examples: {len(vs)}, number of inputs: {len(vs_in)}')
vs.head()
```

```
Number of examples: 1062912, number of inputs: 8
```

Out[2]:

	mode	segment	smonth	sday	stime	timestamp	pCut::Motor_Torque	pCut::CTRL_Position_controller::Lag_error	pCut::CTRL_Positi
0	1	0	1	4	184148	0.008	0.199603	0.027420	628392628
1	1	0	1	4	184148	0.012	0.281624	0.002502	628392625
2	1	0	1	4	184148	0.016	0.349315	-0.018085	628392621
3	1	0	1	4	184148	0.020	0.444450	-0.054680	628392617
4	1	0	1	4	184148	0.024	0.480923	-0.042770	628392613

- There aren't many columns, but there are many examples!
- The data refers to different measurement intervals (or "segments")
- Each segment contains data sampled every 4ms

The Dataset

Let's check some statistics

In [3]:
```
vs.describe()
```

Out[3]:

	mode	segment	smonth	sday	stime	timestamp	pCut::Motor_Torque	pCut::CTRL_Pos
count	1.062912e+06	1.062912e+06	1.062912e+06	1.062912e+06	1.062912e+06	1.062912e+06	1.062912e+06	1.062912e+06
mean	2.323699e+00	2.590000e+02	5.271676e+00	1.654143e+01	1.362122e+05	4.102069e+00	-1.206338e-01	-5.472746e-05
std	1.649207e+00	1.498222e+02	3.505212e+00	8.490150e+00	3.226381e+04	2.364827e+00	6.078708e-01	1.212122e-01
min	1.000000e+00	0.000000e+00	1.000000e+00	1.000000e+00	8.115800e+04	4.000000e-03	-6.560303e+00	-1.888258e+00
25%	1.000000e+00	1.290000e+02	2.000000e+00	9.000000e+00	1.113170e+05	2.056000e+00	-3.696310e-01	-2.201461e-02
50%	2.000000e+00	2.590000e+02	4.000000e+00	1.800000e+01	1.348180e+05	4.104000e+00	-1.187128e-01	6.456900e-04
75%	3.000000e+00	3.890000e+02	8.000000e+00	2.300000e+01	1.618270e+05	6.152000e+00	2.546913e-01	2.380830e-02
max	8.000000e+00	5.180000e+02	1.200000e+01	3.100000e+01	2.232490e+05	8.199999e+00	3.856873e+00	2.021531e+00

- The data is neither normalized nor standardized

The Dataset

Let's check for missing values

In [4]:
```
vs[vs_in].isnull().any()
```

Out[4]:
```
pCut::Motor_Torque                                      False
pCut::CTRL_Position_controller::Lag_error               False
pCut::CTRL_Position_controller::Actual_position         False
pCut::CTRL_Position_controller::Actual_speed            False
pSvolFilm::CTRL_Position_controller::Actual_position    False
pSvolFilm::CTRL_Position_controller::Actual_speed       False
pSvolFilm::CTRL_Position_controller::Lag_error          False
pSpintor::VAX_speed                                     False
dtype: bool
```

- There are none

The Dataset

And let's check the length of each segment

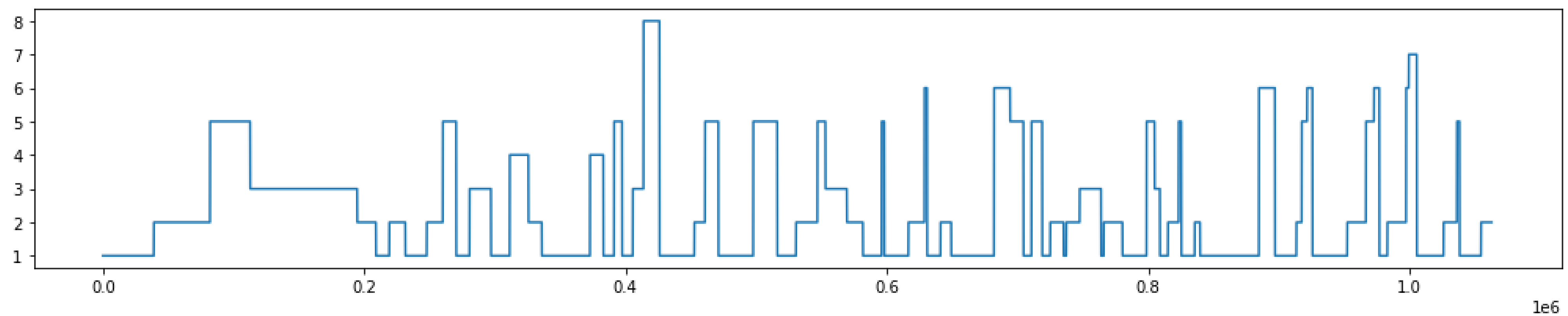
```
In [5]: vs.groupby('segment')['mode'].count().describe()

Out[5]: count     519.0
        mean     2048.0
        std         0.0
        min      2048.0
        25%      2048.0
        50%      2048.0
        75%      2048.0
        max      2048.0
        Name: mode, dtype: float64
```

- There are 519 segments overall
- ...Each with 2048 samples

The Dataset

The machine has multiple operating modes

In [6]:
```
nn.plot_series(vs['mode'], figsize=figsize)
```

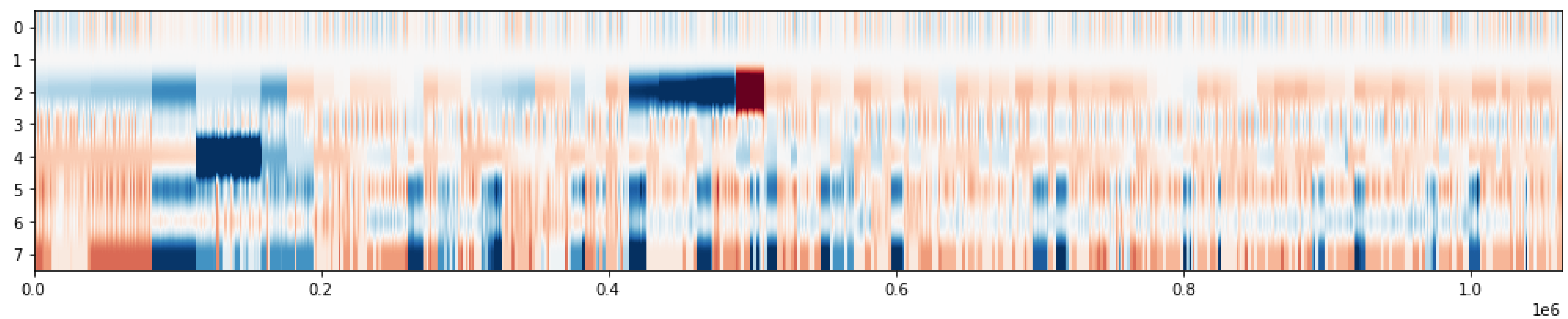


- The mode is a controlled parameter and does not change in the middle of a segment
- Intuitively, the mode has an impact on the machine behavior

The Dataset

Let's have a look at all the sensor readings

In [7]:
```
vss = vs.copy()
vss[vs_in] = (vss[vs_in] - vss[vs_in].mean()) / vss[vs_in].std()
nn.plot_dataframe(vss[vs_in], figsize=figsize)
```



Data Preparation

Binning

This dataset contain high-frequency data (4ms sampling period)

- In this situation, feeding the raw data to a model does not usually make sense
- Instead, we reduce the frequency of the data via a process called binning

A binning approach typically works as follows

We apply a sliding window, but so that its consecutive applications do not overlap

- Each window application is called a bin
- ...From which we extract one or more features

The result is series that contains a smaller number of samples

...But typically a larger number of features

Binning

There are two broad classes of features that are usually extracted

- Time-domain features (e.g. mean, standard deviation)
- Frequency-domain features (e.g. specific FFT amplitudes)

For this case, we will focus on time-domain features

As a first step, we defined which features we are going to extract

```
In [8]: functions = ['mean', 'std', 'skew', lambda x: x.kurtosis()]
        aggmap = {a: functions for a in vs_in}
        aggmap['mode'] = 'first'
        str(aggmap)
```

```
Out[8]: "{'pCut::Motor_Torque': ['mean', 'std', 'skew', <function <lambda> at 0x7fc0b5eb9268>], 'pCu
        t::CTRL_Position_controller::Lag_error': ['mean', 'std', 'skew', <function <lambda> at 0x7fc0b
        5eb9268>], 'pCut::CTRL_Position_controller::Actual_position': ['mean', 'std', 'skew', <functio
        n <lambda> at 0x7fc0b5eb9268>], 'pCut::CTRL_Position_controller::Actual_speed': ['mean', 'st
        d', 'skew', <function <lambda> at 0x7fc0b5eb9268>], 'pSvolFilm::CTRL_Position_controller::Actu
        al_position': ['mean', 'std', 'skew', <function <lambda> at 0x7fc0b5eb9268>], 'pSvolFilm::CTRL
        _Position_controller::Actual_speed': ['mean', 'std', 'skew', <function <lambda> at 0x7fc0b5eb9
        268>], 'pSvolFilm::CTRL_Position_controller::Lag_error': ['mean', 'std', 'skew', <function <la
        mbda> at 0x7fc0b5eb9268>], 'pSpintor::VAX_speed': ['mean', 'std', 'skew', <function <lambda> a
        t 0x7fc0b5eb9268>], 'mode': 'first'}"
```

Binning

Then we define bin numbers, and extract the features via a `groupby` operation

We need to take care so that no bin crosses between different segments

```
In [9]: %%time
binsize = 128
bins = []
for sname, sdata in vs.groupby('segment'):
    # Build the bin numbers
    sdata['bin'] = sdata.index // binsize
    # Apply the aggregation functions
    tmp = sdata.groupby('bin').agg(aggmap)
    bins.append(tmp)
vsb = pd.concat(bins)

CPU times: user 32 s, sys: 23.9 ms, total: 32 s
Wall time: 32.1 s
```

- This can be a relatively slow operation
- Bin numbers are usually easy to define using positional indexes and an integer division

Binning

If we choose the bin size correctly, we can speed up the operation

- In particular, if all segments have the same length...
- ...And we choose bin size that is a submultiple of the segment length

...Then we can avoid processing each segment separately:

```
In [10]: %%time
         # Build the bin numbers
         binsize = 128
         vsb = vs.copy()
         vsb['bin'] = vs.index // binsize
         vsb = vsb.groupby('bin').agg(aggmap)

         CPU times: user 16.5 s, sys: 46 ms, total: 16.5 s
         Wall time: 16.5 s
```

- This kind of approach is significantly faster

Binning

The resulting dataframe has a hierarchical column index

In [11]:
```python
vsb.iloc[:1]
```

Out[11]:

	pCut::Motor_Torque				pCut::CTRL_Position_controller::Lag_error				pCut::CTRL_Position_controller::Actual_po
	mean	std	skew	<lambda_0>	mean	std	skew	<lambda_0>	mean	std
bin										
0	0.475072	0.141935	-0.346041	-0.020202	0.000205	0.04027	0.069676	0.350389	6.283919e+08	539.217959

1 rows × 33 columns

It may be worth flattening, so as to simplify access:

In [12]:
```python
if isinstance(vsb.columns, pd.MultiIndex):
    vsb.columns = ['::'.join(c) for c in vsb.columns]
vsb.iloc[:1]
```

Out[12]:

	pCut::Motor_Torque::mean	pCut::Motor_Torque::std	pCut::Motor_Torque::skew	pCut::Motor_Torque:: <lambda_0>	pCut::CTRL_Position_controller
bin					
0	0.475072	0.141935	-0.346041	-0.020202	0.000205

1 rows × 33 columns

Standardization

Before we can train any model, we need some preparation

We will standardize sensor inputs (all except the mode) using the first third of the series

In [13]:
```
sep = int(np.round(len(vsb) * 0.34))
vsb_in = vsb.columns[:-1]
vsbs = vsb.copy()
tmp = vsbs[vsb_in].iloc[:sep]
vsbs[vsb_in] = (vsbs[vsb_in] - tmp.mean()) / tmp.std()
vsbs.iloc[:3]
```

Out[13]:

	pCut::Motor_Torque::mean	pCut::Motor_Torque::std	pCut::Motor_Torque::skew	pCut::Motor_Torque::<lambda_0>	pCut::CTRL_Position_controller
bin					
0	1.838916	-1.125481	0.478513	-0.549730	0.083036
1	-0.336049	-0.424273	0.757608	-0.689435	-0.595970
2	-1.199835	0.860690	-1.275360	1.098302	0.238978

3 rows × 33 columns

Categorical Mode

We will also adopt a categorical encoding for the operating mode

This is critical for neural network approaches in particular

In [14]:

```
from tensorflow.keras.utils import to_categorical
cmode = to_categorical(vsbs['mode::first'])
cols = [f'm{i}' for i in range(cmode.shape[1])]
cmode = pd.DataFrame(index=vsbs.index, data=cmode, columns=cols)
vsbs[cols] = cmode
vsbs.head()
```

Out[14]:

	pCut::Motor_Torque::mean	pCut::Motor_Torque::std	pCut::Motor_Torque::skew	pCut::Motor_Torque::<lambda_0>	pCut::CTRL_Position_controller
bin					
0	1.838916	-1.125481	0.478513	-0.549730	0.083036
1	-0.336049	-0.424273	0.757608	-0.689435	-0.595970
2	-1.199835	0.860690	-1.275360	1.098302	0.238978
3	-0.188107	-1.191551	0.250961	-0.443552	0.217620
4	0.252049	-0.947974	1.255687	-0.587060	-0.089580

5 rows × 42 columns

KDE Approach

KDE Approach

Now, let's try anomaly detection via KDE

First we estimate the optimal bandwidth:

In [15]:
```
%%time
vsbs_tr = vsbs.iloc[:sep]

params = {'bandwidth': np.linspace(0.2, 0.8, 10)}
opt = GridSearchCV(KernelDensity(kernel='gaussian'), params, cv=5)
opt.fit(vsbs_tr)
best_params = pd.Series(index=opt.best_params_.keys(), data=opt.best_params_.values())
print(best_params)
```

```
bandwidth    0.333333
dtype: float64
CPU times: user 8.4 s, sys: 0 ns, total: 8.4 s
Wall time: 8.41 s
```

KDE Approach

Then we can train an estimator

```
In [16]: h = opt.best_params_['bandwidth']
         kde = KernelDensity(bandwidth=h)
         kde.fit(vsbs_tr)

Out[16]: KernelDensity(bandwidth=0.33333333333333337)
```

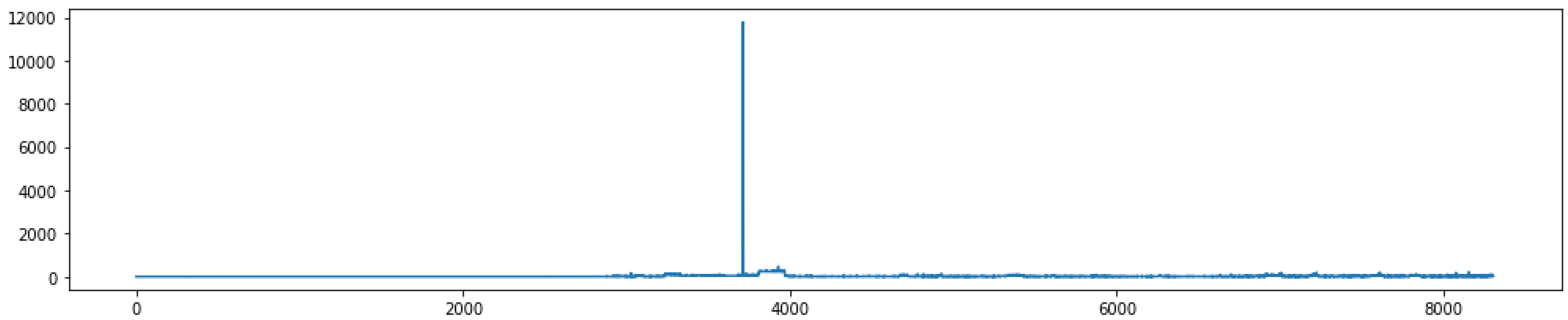
...And we can generate the alarm signal

```
In [17]: %%time
         ldens = kde.score_samples(vsbs)
         signal_kde = pd.Series(index=vsbs.index, data=-ldens)

         CPU times: user 3.24 s, sys: 6.64 ms, total: 3.25 s
         Wall time: 3.25 s
```

KDE Approach

Let's plot the signal

In [18]:
```
nn.plot_signal(signal_kde, figsize=figsize)
```

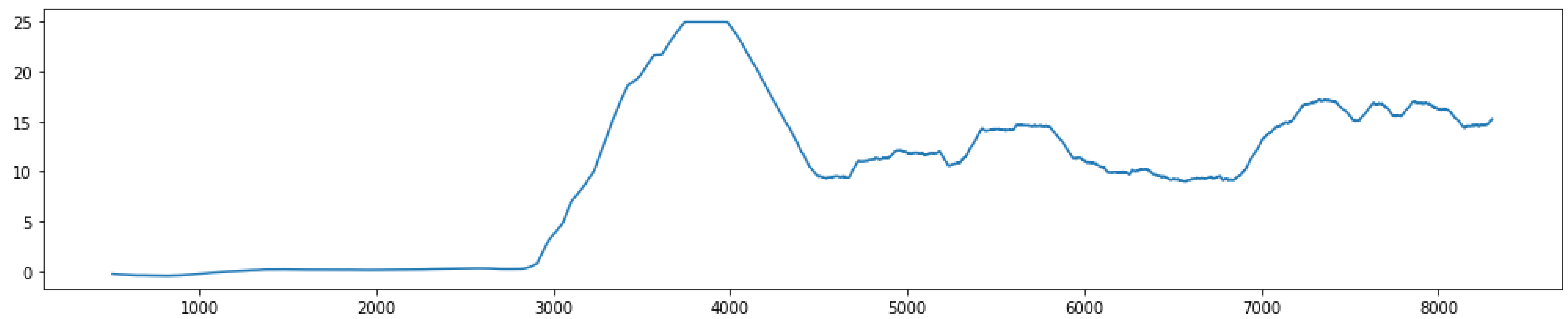


- A peak at mid run makes the signal difficult to read

KDE Approach

Since the anomaly arises slowly, it makes sense to clip and smooth the signal

In [19]:
```
nn.plot_signal(signal_kde.clip(upper=25).rolling(512).mean(), figsize=figsize)
```



- The smoothed signal has a tendency to grow
- There is a plateau in the middle that is difficult to explain

Autoencoder Approach

Autoencoder Approach

Let's repeat our analysis using an autoencoder

First we define its structure:

In [20]:
```
input_shape = vsbs.shape[1]
output_shape = len(vsb_in)
ae_x = keras.Input(shape=input_shape, dtype='float32')
ae_z = Dense(16, activation='relu')(ae_x)
ae_y = Dense(output_shape, activation='linear')(ae_z)
ae = keras.Model(ae_x, ae_y)
```

- The input includes the operating mode
- ...But the output does not!

We have no interest in reconstructing controlled parameters

Autoencoder Approach

Now we can perform training

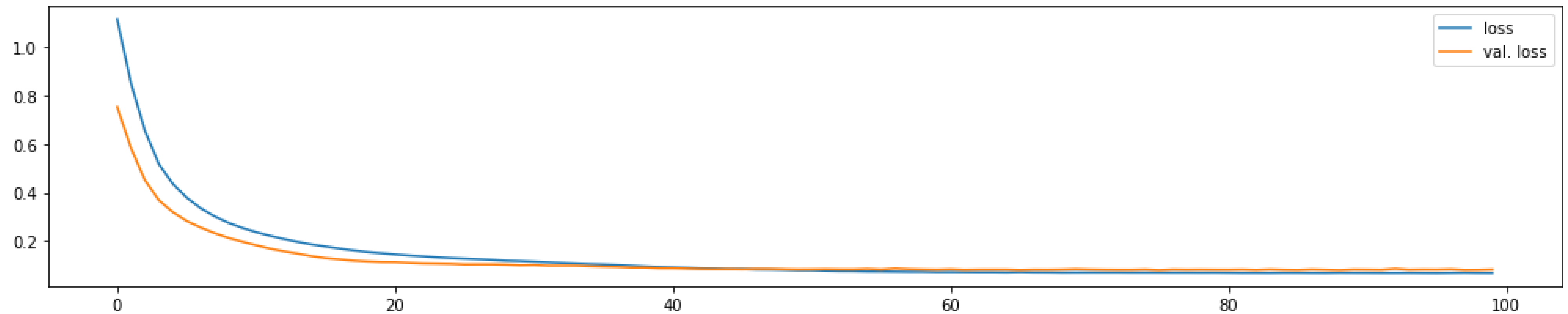
In [21]:
```
%%time
ae.compile(optimizer='RMSProp', loss='mse')
callbacks = [EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)]
history = ae.fit(vsbs_tr, vsbs_tr[vsb_in], validation_split=0.2, callbacks=callbacks,
                 batch_size=32, epochs=100, verbose=2)
```

```
Epoch 1/100
71/71 - 0s - loss: 1.1143 - val_loss: 0.7524
Epoch 2/100
71/71 - 0s - loss: 0.8494 - val_loss: 0.5817
Epoch 3/100
71/71 - 0s - loss: 0.6537 - val_loss: 0.4514
Epoch 4/100
71/71 - 0s - loss: 0.5175 - val_loss: 0.3687
Epoch 5/100
71/71 - 0s - loss: 0.4367 - val_loss: 0.3198
Epoch 6/100
71/71 - 0s - loss: 0.3797 - val_loss: 0.2831
Epoch 7/100
71/71 - 0s - loss: 0.3361 - val_loss: 0.2562
Epoch 8/100
71/71 - 0s - loss: 0.3025 - val_loss: 0.2329
Epoch 9/100
71/71 - 0s - loss: 0.2757 - val_loss: 0.2133
Epoch 10/100
71/71 - 0s - loss: 0.2545 - val_loss: 0.1975
Epoch 11/100
```

Autoencoder Approach

Let's see the loss evolution over time

In [22]:
```
nn.plot_training_history(history, figsize=figsize)
```



Autoencoder Approach

Then we obtain our predictions

In [23]:
```
%%time
preds = ae.predict(vsbs)
preds = pd.DataFrame(index=vsbs.index, columns=vsb_in, data=preds)
```

```
CPU times: user 182 ms, sys: 10.5 ms, total: 192 ms
Wall time: 157 ms
```

...And we generate the alarm signal:

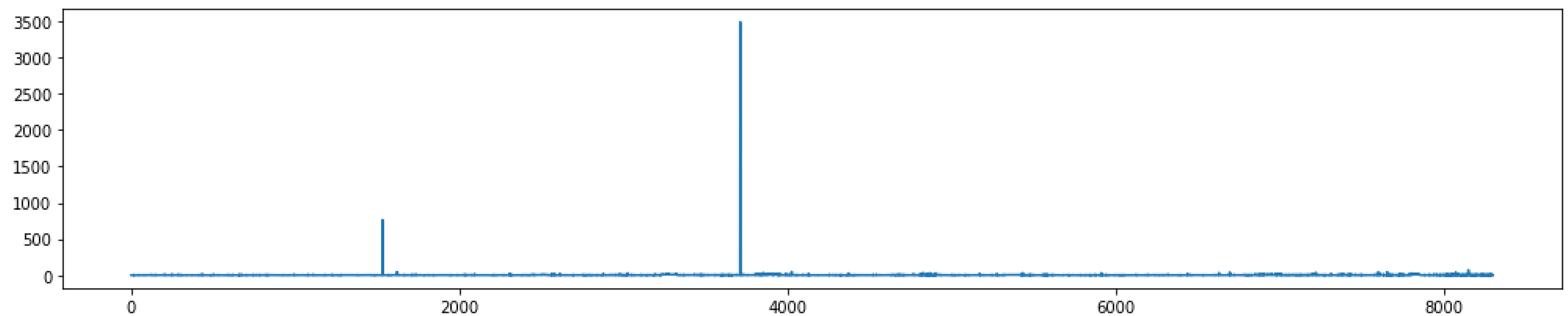
In [24]:
```
se = np.square(preds - vsbs[vsb_in])
sse = np.sum(se, axis=1)
signal_ae = pd.Series(index=vsbs.index, data=sse)
```

- We stored also the individual errors `se` for a later analysis

Autoencoder Approach

We can now plot the alarm signal

```
In [25]: nn.plot_signal(signal_ae, figsize=figsize)
```

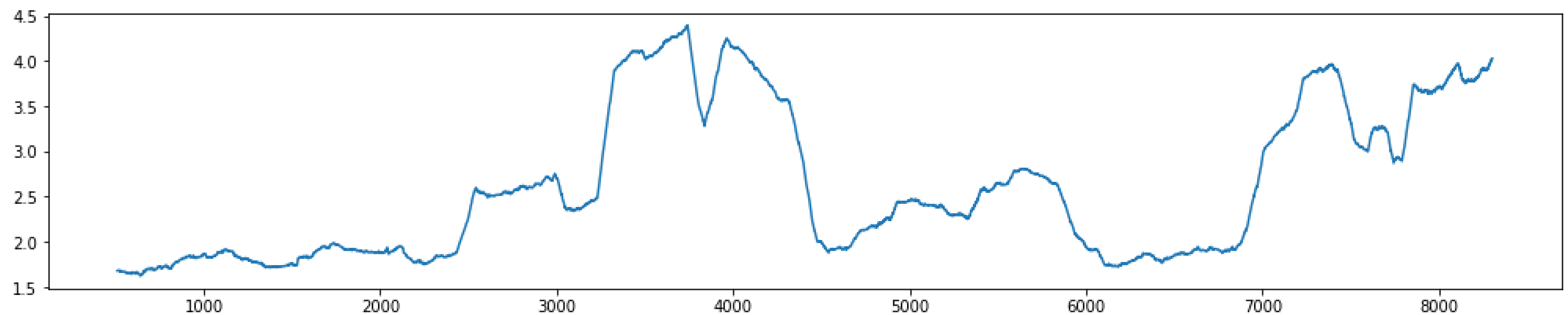


- Once again, the signal is difficult to read due to the peak

Autoencoder Approach

Let's apply clipping and smoothing

```
In [26]: nn.plot_signal(signal_ae.clip(upper=10).rolling(512).mean(), figsize=figsize)
```

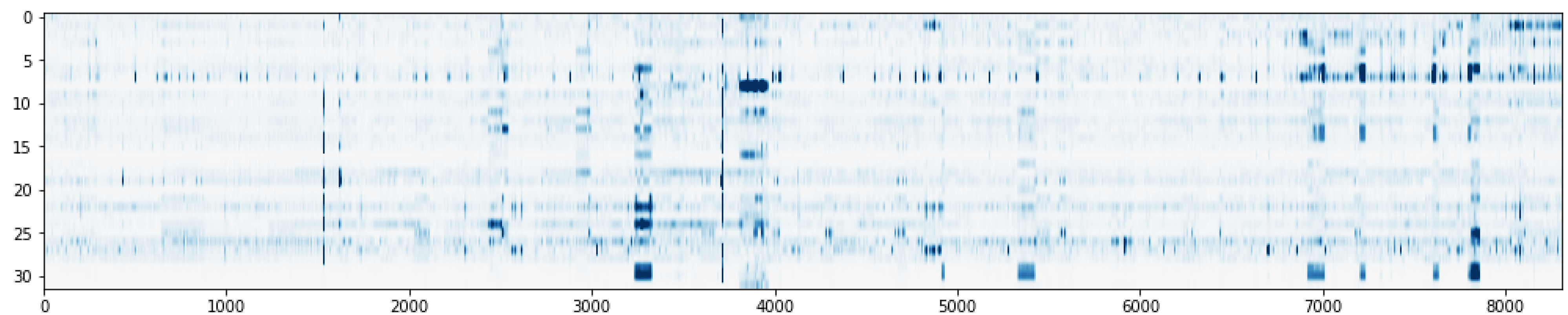


- Again, a tendency to grow (weaker than before)
- ...But this time we can try to explain the peak!

Mutiple Signal Analysis

Let's investigate the situation

In [27]:
```
signals_ae = pd.DataFrame(index=vsbs.index, columns=vsb_in, data=se)
nn.plot_dataframe(signals_ae, vmin=-1, vmax=1, figsize=figsize)
```

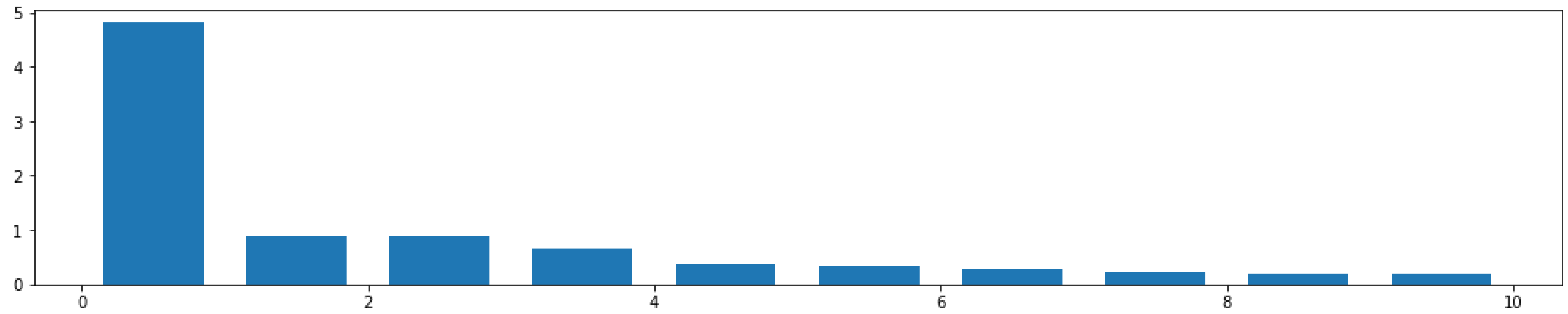


- As expected, errors are concentrated on a few features

Multiple Signal Analysis

Let's focus on the peak at mid run and check the largest errors

In [28]:
```
tmp = se.iloc[3500:4000].mean().sort_values(ascending=False)[:10]
nn.plot_bars(tmp, figsize=figsize, tick_gap=-1)
```



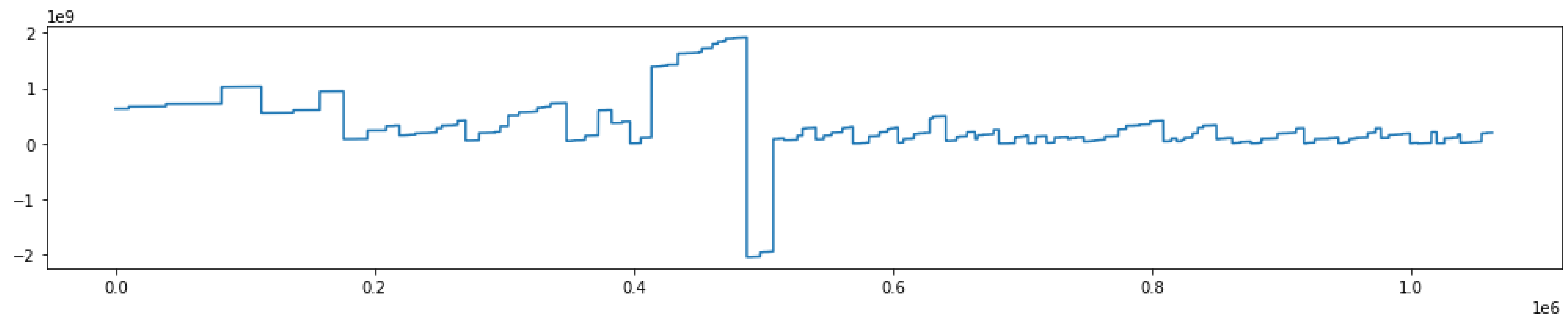
In [29]:
```
print(f'The largest error is on {tmp.index[0]}')
```

```
The largest error is on pSvolFilm::CTRL_Position_controller::Actual_position::<lambda_0>
```

Multiple Signal Analysis

Let's what was going on with the original series

```
In [30]: nn.plot_series(vs['pCut::CTRL_Position_controller::Actual_position'], figsize=figsize)
```



- Indeed, there is an unusual oscillation! A domain expert may make sense of that

RNVP Approach

RNVP Approach

Let's make an attempt with Real NVP

In [31]:

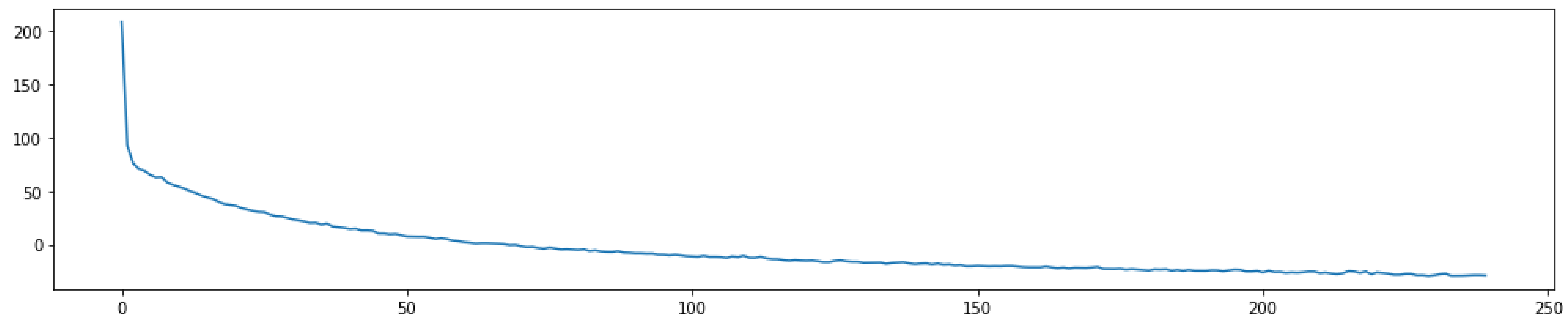
```
%%time
vs_rnvp = nn.RealNVP(input_shape=vsbs.shape[1],
                num_coupling=6, units_coupling=32, depth_coupling=1, reg_coupling=0.01)
vs_rnvp.compile(optimizer='Adam')
X = vsbs_tr.astype(np.float32).values
cb = [EarlyStopping(monitor='loss', patience=10, min_delta=0.001, restore_best_weights=True)]
history = vs_rnvp.fit(X, batch_size=256, epochs=300, verbose=1, callbacks=cb)
```

```
Epoch 1/300
12/12 [==============================] - 2s 3ms/step - loss: 207.9114
Epoch 2/300
12/12 [==============================] - 0s 3ms/step - loss: 92.4916
Epoch 3/300
12/12 [==============================] - 0s 4ms/step - loss: 75.9836
Epoch 4/300
12/12 [==============================] - 0s 4ms/step - loss: 70.8679
Epoch 5/300
12/12 [==============================] - 0s 3ms/step - loss: 69.0279
Epoch 6/300
12/12 [==============================] - 0s 3ms/step - loss: 65.2163
Epoch 7/300
12/12 [==============================] - 0s 3ms/step - loss: 62.8103
Epoch 8/300
12/12 [==============================] - 0s 3ms/step - loss: 63.1269
Epoch 9/300
12/12 [==============================] - 0s 3ms/step - loss: 58.0721
```

RNVP Approach

Let's have a look at the loss evolution over time

In [32]:
```
nn.plot_training_history(history, figsize=figsize)
```

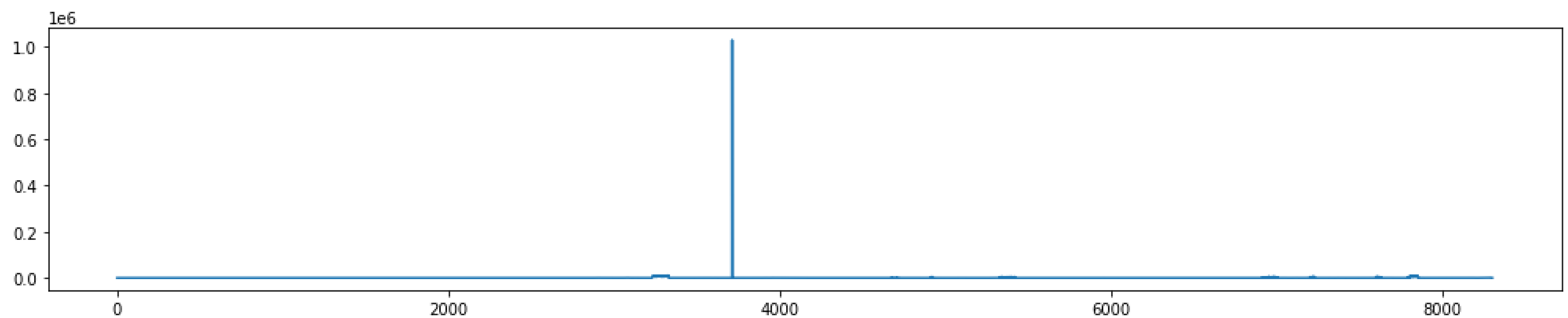


RNVP Approach

Now we can generate and plot the signal

In [33]:

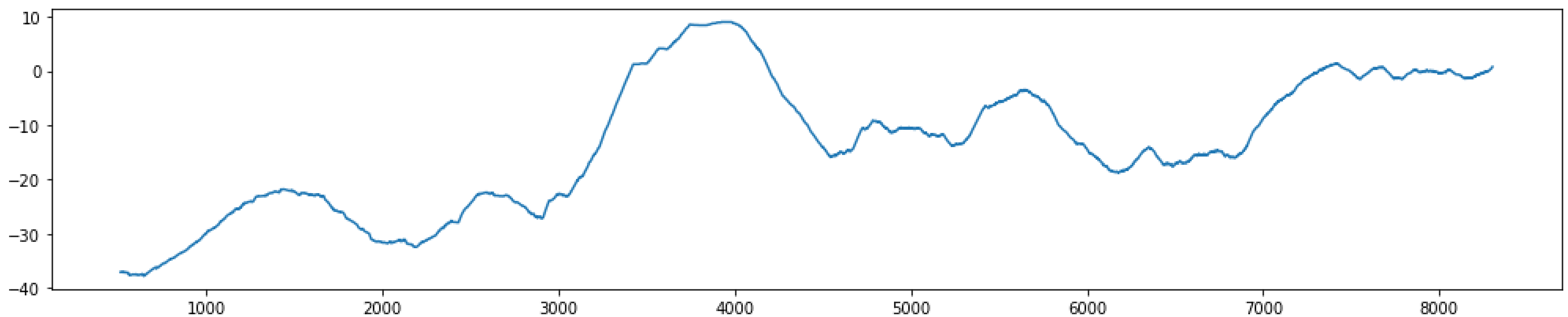
```
ldens = vs_rnvp.score_samples(vsbs.astype(np.float32).values)
signal_vs = pd.Series(index=vsbs.index, data=-ldens)
nn.plot_signal(signal_vs, figsize=figsize)
```



RNVP Approach

Finally, we apply clipping and smoothing

In [34]:
```
nn.plot_signal(signal_vs.clip(upper=10).rolling(512).mean(), figsize=figsize)
```



- This is the signal with the clearest trend so far
- Intuitively, component wear should grow progressively over time