Baseline Approach

Train/Test Split

We'll try to detect the component state by learning an autoencoder

- We'll train a model on the earlier data
- ...And then use the reconstruction error as a proxy for component wear

We start as usual by splitting the training and test set

In [2]:
```
tr_sep = int(0.5 * len(data_b))
data_b_tr = data_b.iloc[:tr_sep]
data_b_ts = data_b.iloc[tr_sep:]
```

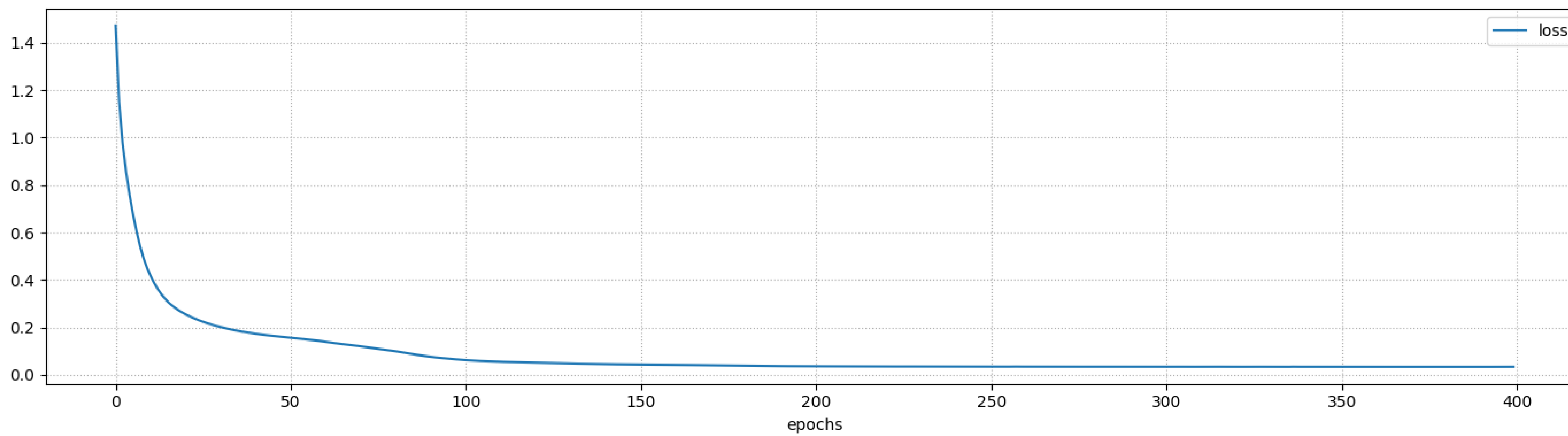
...And then by standardizing our data

In [3]:
```
scaler = StandardScaler()
data_b_s_tr = scaler.fit_transform(data_b_tr)
data_b_s_ts = scaler.transform(data_b_ts)
data_b_s = pd.DataFrame(columns=data_b.columns, data=np.vstack([data_b_s_tr, data_b_s_ts]))
```

Training and Autoencoder

Now we can build and train the autoencoder

```
In [4]: nn = util.build_nn_model(input_shape=len(data_b.columns), output_shape=len(data_b.columns),
                                 hidden=[len(data_b.columns)//2])
        history = util.train_nn_model(nn, data_b_s_tr, data_b_s_tr, loss='mse', validation_split=0.0
                                      batch_size=32, epochs=400)
        util.plot_training_history(history, figsize=figsize)
```



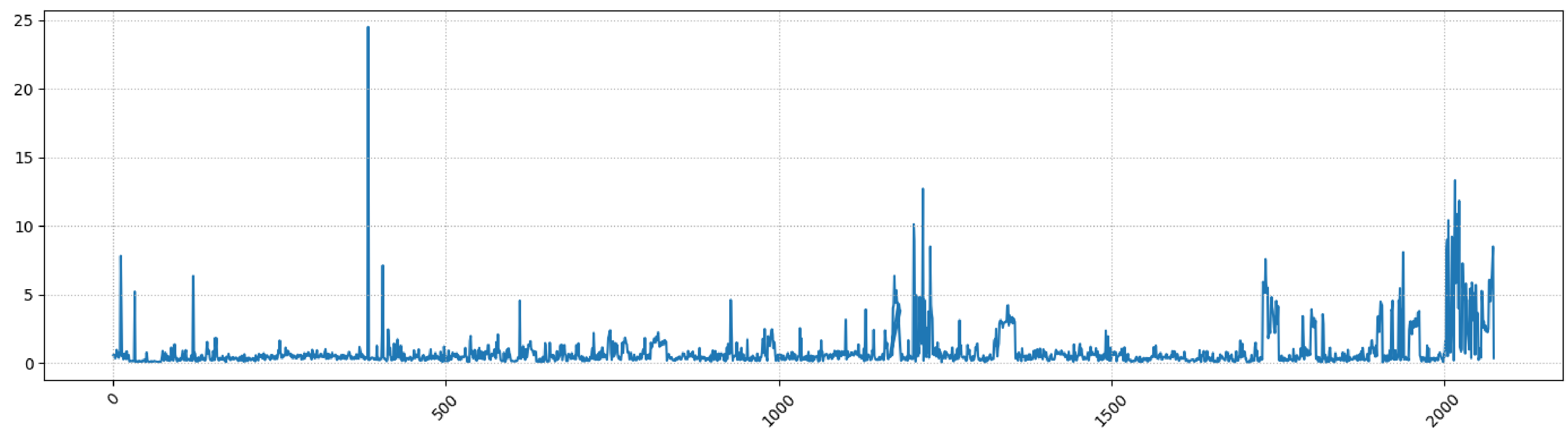

```
Final loss: 0.0347 (training)
```

- We need many epochs to compensate for the small number of samples

Evaluation

Let's check the reconstruction error

In [5]:
```
pred = nn.predict(data_b_s, verbose=0)
se = (data_b_s - pred)**2
sse = pd.Series(index=data_b.index, data=np.sum(se, axis=1))
util.plot_series(sse, figsize=figsize)
```

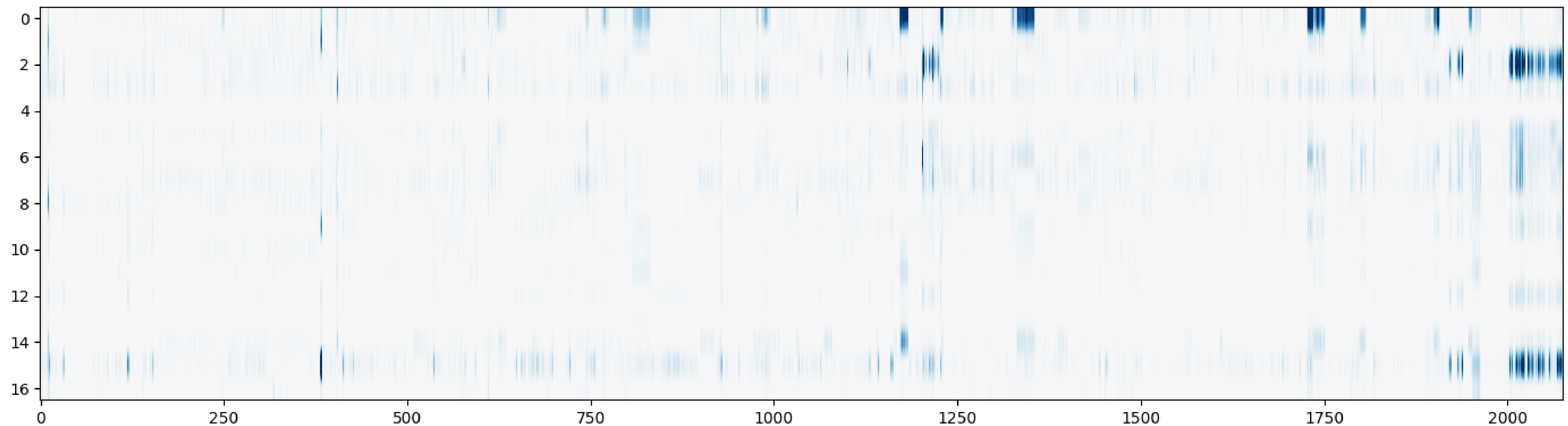


- Since we have a single run, we will limit ourselves to a visual inspection
- ...And the signal does not look very clear

Evaluation

We can gain more information by checking the individual errors

In [6]:
```
util.plot_dataframe(se, figsize=figsize)
```



- Reconstruction errors are large for different features over time

Do you think we can improve these results? How?

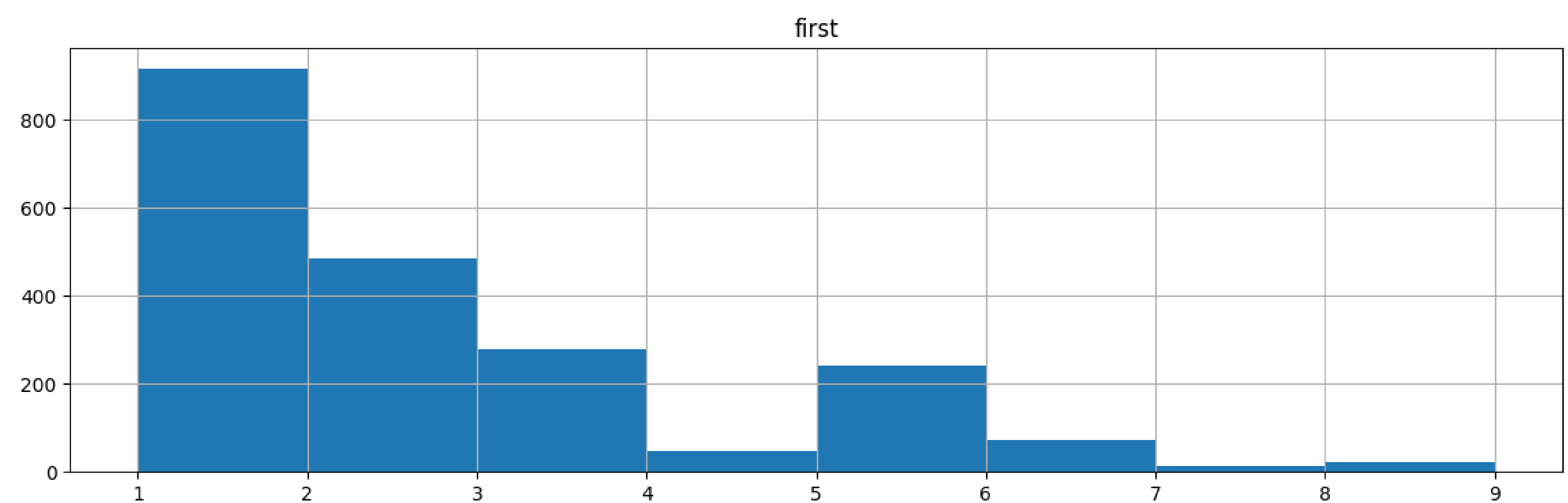
Altering the Training Distribution

Distribution Discrepancy

A major problem is related to the distribution balance

The modes of operation are not used equally often

```
In [8]: data_b['mode'].hist(figsize=figsize, bins=np.arange(1, 10));
```



- Moreover, the mode of operation is a controlled variable
- ...Hence its distribution might change a lot based on the workload

Distribution Discrepancy

In fact, there is a difference between the training and test distribution

```
In [9]: data_b_tr['mode'].hist(figsize=(figsize[0], figsize[1]/2), bins=np.arange(1, 10)); plt.title
        data_b_ts['mode'].hist(figsize=(figsize[0], figsize[1]/2), bins=np.arange(1, 10)); plt.title
```

Maximum Likelihood

This matters because we are training for maximum likelihood

...Ideally we would like to solve:

$$\operatorname*{argmax}_{\theta} \mathbb{E}_{x,y \sim P} \left[\prod_{i=1}^{m} f_\theta(y_i \mid x_i) \right]$$

- P represents the real (joint) distribution
- $f_\theta(\cdot \mid \cdot)$ is our estimated probability, with parameter vector θ
- I.e. an estimator for a conditional distribution
- We distinguish x (input) and y (output) to cover generic supervised learning
- ...Even if for an autoencoder they are the same

...And Empirical Risk

...But in practive we don't have access to the full distribution

So usually we employ a Monte-Carlo approximation:

$$\mathop{\mathrm{argmax}}_{\theta} \prod_{i=1}^{m} f_\theta(y_i \mid x_i)$$

- Typically, we consider a single sample $\boldsymbol{x}, \boldsymbol{y}$ (i.e. the training set)
- The resulting objective (i.e. the big product) is sometimes called empirical risk

...And Empirical Risk

...But in practive we don't have access to the full distribution

So usually we employ a Monte-Carlo approximation:

$$\mathop{\mathrm{argmax}}_{\theta} \prod_{i=1}^{m} f_\theta(y_i \mid x_i)$$

- Typically, we consider a single sample $\boldsymbol{x}, \boldsymbol{y}$ (i.e. the training set)
- The resulting objective (i.e. the big product) is sometimes called empirical risk

Problems arise when our sample is biased. E.g. because:

- We can collect data only under certain circumstances
- The dataset is the result of a selection process
- ...Or perhaps due to pure sampling noise

Handling Sampling Noise

So, let's recap

- Our issue is that the training sample is biased
- ...So that it is not representative of the true distribution

How can we deal with this problem?

Handling Sampling Noise

So, let's recap

- Our issue is that the training sample is biased
- ...So that it is not representative of the true distribution

How can we deal with this problem?

- A possible solution would be to alter the training distribution
- ...So that it matches more closely the test distribution

...And this is actually something we can do!

- E.g. we can use data augmentation, or subsampling
- ...Or we can use sample weights

Virtual Alterations to the Training Distribution

Let our training set consist of $\{(x_1, y_1), (x_2, y_2)\}$

The corresponding optimization problem would be:

$$\underset{\theta}{\operatorname{argmax}}\; f_\theta(y_1 \mid x_1) f_\theta(y_2 \mid x_2)$$

If sample #2 occurred twice in the training data, we would have

$$\underset{\theta}{\operatorname{argmax}}\; f_\theta(y_1 \mid x_1) f_\theta(y_2 \mid x_2)^2$$

Normalizing over the number of samples does not change the minima:

$$\underset{\theta}{\operatorname{argmax}}\; f_\theta(y_1 \mid x_1)^{\frac{1}{3}} f_\theta(y_2 \mid x_2)^{\frac{2}{3}}$$

Virtual Alterations to the Training Distribution

Let's generalize these considerations:

A general training problem based on Empirical Risk Minimization is the form:

$$\operatorname*{argmax}_{\theta} \prod_{i=1}^{m} f_\theta(y_i \mid x_i)$$

We can virtually alter the training distribution via exponents:

$$\operatorname*{argmax}_{\theta} \prod_{i=1}^{m} f_\theta(y_i \mid x_i)^{w_i}$$

- We can do this to make the training distribution more representative
- E.g. when we expect a discrepancy between the trainign and test distribution

Virtual Distribution and Sample Weights

When we switch to log-likelihood minimization

...The exponents become sample weights

$$\operatorname*{argmin}_{\theta} - \sum_{i=1}^{m} w_i \log f_\theta(y_i \mid x_i)$$

We can always view the weights as the ratio of two probabilities:

$$w_i = \frac{p_i^*}{p_i}$$

- p_i is the sampling bias that we want to cancel
- p_i^* is the distribution we wish to emulate

This approach is known as importance sampling

Canceling Sampling Bias in Our Problem

Let's apply the approach to our skinwrapper example

We know there's an unwanted sampling bias for some modes of operation

- Let $m(x_i)$ be the mode of operation for the i-th sample
- Then we can estimate p_i as a frequency of occurrence:

$$p_i = \frac{1}{n}|\{k : m(x_k) = m(x_i), k = 1..n\}|$$

We don't want out anomaly detector to be sensitive to the mode

- So we can assumption a uniform distribution for p_i^*:

$$p_i = \frac{1}{n}$$

Canceling Sampling Bias in Our Problem

By combining the two we get:

$$w_i = \frac{1}{|\{k : m(x_k) = m(x_i), k = 1..n\}|}$$

- I.e. the weight is just the inverse of the corresponding mode count

We can compute the weigths by first obtaining inverse counts for all modes

In [23]:
```
vcounts = data_b_tr['mode', 'first'].value_counts()
mode_weight = 1 / vcounts
```

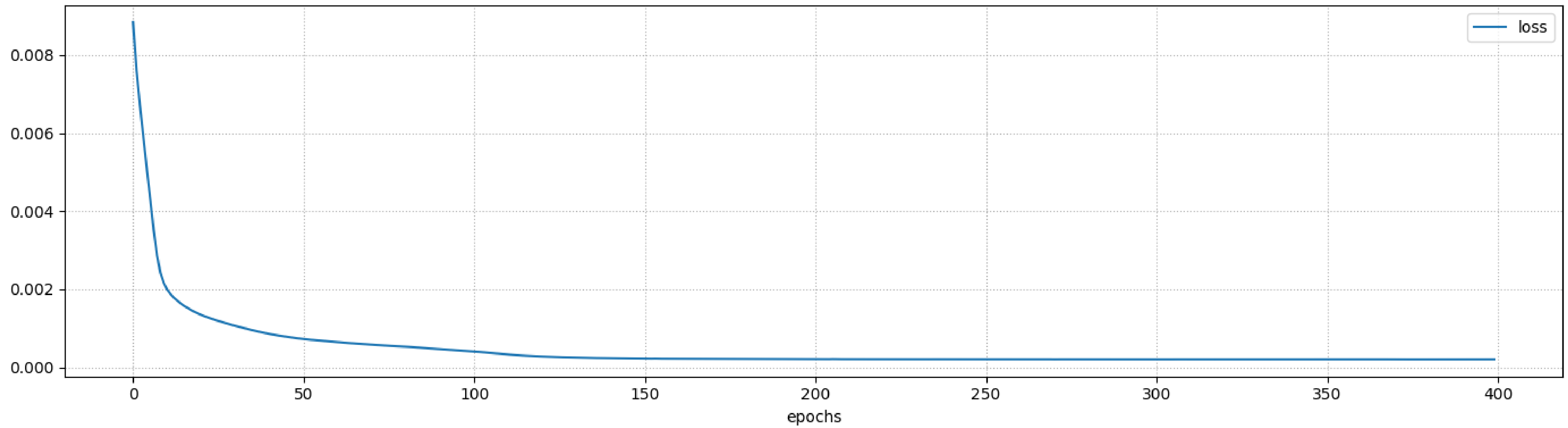
Then by associating the respective value to every sample:

In [24]:
```
sample_weight = mode_weight[data_b_tr['mode', 'first']]
```

Training with Sample Weights

Now we can pass training weights to the training algorithm

```
In [25]: nn2 = util.build_nn_model(input_shape=len(data_b.columns), output_shape=len(data_b.columns),
         history = util.train_nn_model(nn2, data_b_s_tr, data_b_s_tr, loss='mse', validation_split=0.
         util.plot_training_history(history, figsize=figsize)
```

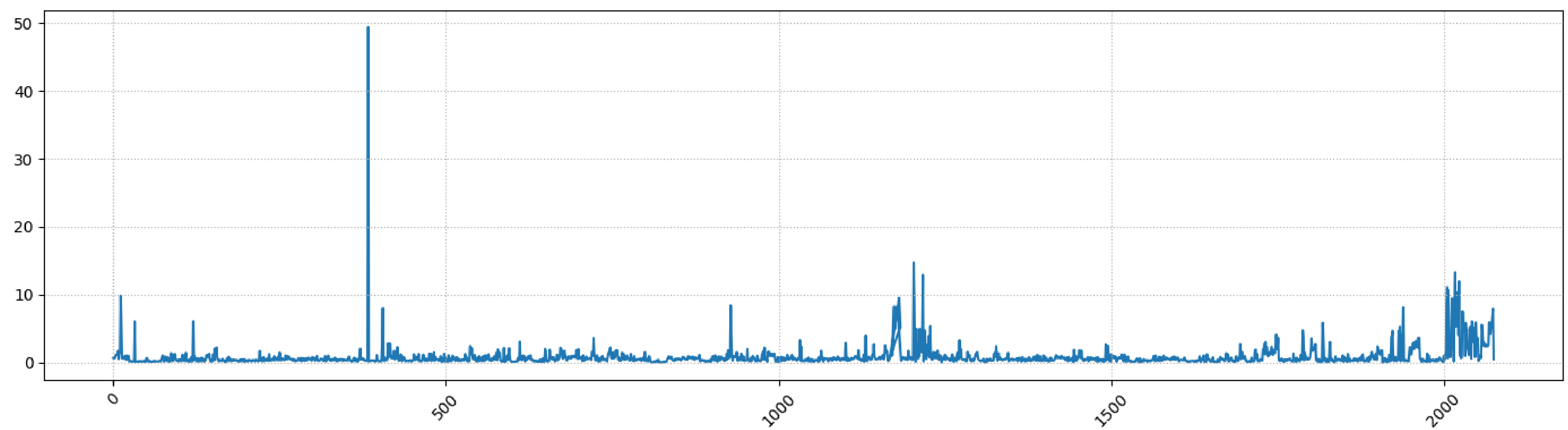



```
Final loss: 0.0002 (training)
```

Evaluation

Let's check the new reconstruction error

In [26]:
```
pred2 = nn2.predict(data_b_s, verbose=0)
se2 = (data_b_s - pred2)**2
sse2 = pd.Series(index=data_b.index, data=np.sum(se2, axis=1))
util.plot_series(sse2, figsize=figsize)
```

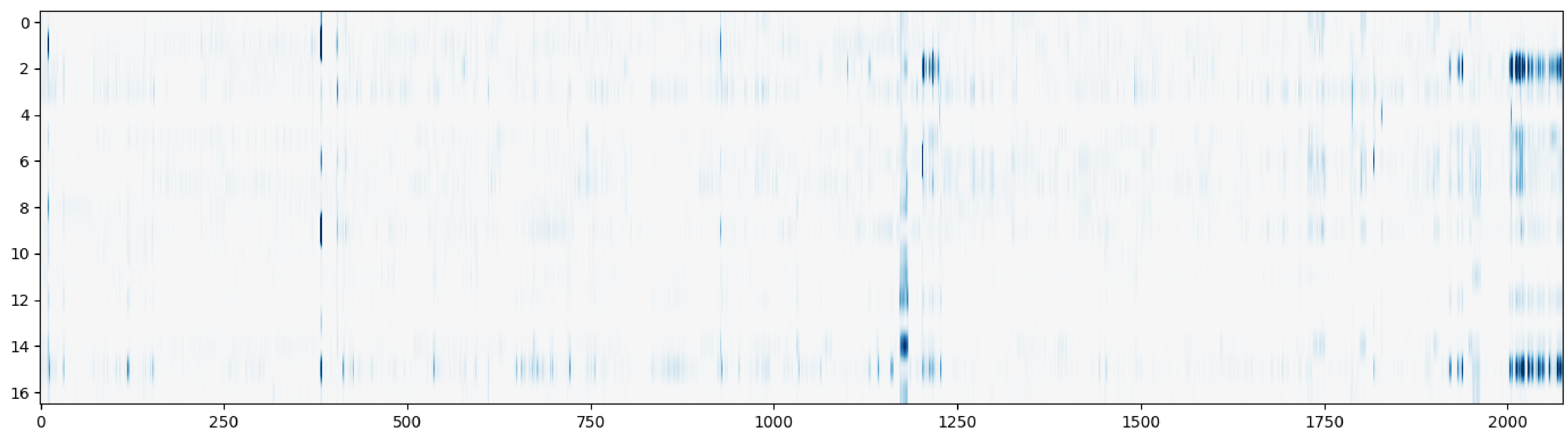


At a first glance, the change is not dramatic

Evaluation

...But the individual error components are very different

In [27]:
```
util.plot_dataframe(se2, figsize=figsize)
```



- Suspected anomalies in the middle sequence have almost disappeared
- ...And there is a much clearer plateau at the end of the signal

Applications of Importance Sampling

Despite its simplicity, importance sampling has many applications

Can you identify a few?

Class Rebalancing

The usual class rebalancing trick is a subcase of importance sampling

In this situation, we assume that some classes are over/under sampled

- Therefore, we estimate p_i using the class frequency
- We make a neutral (uniform) assumption on p_i^*
- ...And we define the sample weights for (x_i, y_i) as:

$$w_i = \frac{1}{n} \frac{n}{|\{k : y_k = y_i, k = 1..n\}|}$$

Watch out during evaluation!

- Evaluating via (e.g.) accuracy on the unmodified test set might be a mistake
- ...Since the weights alter the training distribution

Use a cost model instead, or just a confusion matrix

Removing Sampling Bias based on Continuous Attribute

The p_i and p_i^* values can be probability densities

...Meaning we can remove sampling bias over continuous attributes, e.g.:

- Continuous control variables (position, speed, etc.) in industrial machines
- Income or age in socio-economic applications
- Number of reviews in online rating systems

In this case:

We can first apply any density estimation approach

- The discrete attribute/class case is the same (we just use a histogram)

Then, it's a good idea to apply some clipping, i.e. $p_i = \max(l, \min(u, f(x_i, y_i)))$

- Densities can be very high/low, causing numerical instability

Removing Sampling Bias due to External Attributes

It is possible to remove sampling bias due to an "external" process

Consider an organ transplant program

- Candidate recipients are described by attribute x_i and wait in a queue
- ...from which they may be selected ($y_i = 1$) or not ($y_i = 0$) for surgery
- ...Surgey may then have a positive ($z_i = 1$) or negative ($z_i = 0$) outcome

Say we want to improve the outcome estimation using ML

...And possibly use to adjust the selection criterion

- The historical data will be subject to bias due to existing criteria
- ...But if we can estimate the current selection proabability $P(Y \mid X)$
- ...We can can use it as p_i for mitigating the bias!

Any classifier with probabilistic output can be used on this purpose

Sample-specific Variance

With an MSE loss, sample weights have also an alternative interpretation

In this case we have proved the training problem is equivalent to:

$$\operatorname*{argmin}_\theta - \sum_{i=1}^m \log k \exp\left(-\frac{1}{2}(y_i - h_\theta(x_i))^2\right)$$

- We have simply replaced the generic PDF with a Normal one
- We have $k = 1/\sqrt{2\pi}$ to simplify the notation

Let's now introduce sample weights, in the form as $1/\hat{\sigma}_i^2$

By doing so we get:

$$\operatorname*{argmin}_\theta - \sum_{i=1}^m \frac{1}{\sigma_i^2} \log k \exp\left(-\frac{1}{2}(y_i - h_\theta(x_i))^2\right)$$

Sample-specific Variance

Which can be rewritten as:

$$\underset{\theta}{\operatorname{argmin}} - \sum_{i=1}^{m} \log k \exp\left(-\frac{1}{2} \left(\frac{y_i - h_\theta(x_i)}{\sigma_i} \right)^2 \right)$$

- This means that sample weights with an MSE loss
- ...Can be interpreted as inverse sample variances

Sample-specific Variance

Which can be rewritten as:

$$\underset{\theta}{\operatorname{argmin}} - \sum_{i=1}^{m} \log k \exp\left(-\frac{1}{2} \left(\frac{y_i - h_\theta(x_i)}{\sigma_i} \right)^2 \right)$$

- This means that sample weights with an MSE loss
- ...Can be interpreted as inverse sample variances

This gives us a way to account for non-uniform measurement errors

- If we know that there is a measurment error with stdev $\boldsymbol{\sigma_i}$ on example $\boldsymbol{i}$
- ...We can account for that by using $1/\sigma_i^2$ as a weight

The result is analogous to using a separate variance model

Stochastic Differentiation in Reinforcement Learning

Importance sampling finds applications also in Reinforcement Learning

While the goal of statistical ML is usually maximize a likelihood, e.g.:

$$\underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{x,y\sim P}\left[f_\theta(y \mid x)\right]$$

...The goal of RL is to learn how to optimize a reward, e.g.:

$$\underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{x\sim P}\left[f(x, \pi_\theta(x))\right]$$

Where the presented formulation focuses on a single step (for simplicity)

- x represents an observable state
- $\pi_\theta : x \mapsto a$ is a parameterized policy outputing an action
- $f : x, a \mapsto r$ is a reward function

Stochastic Differentiation in Reinforcement Learning

In tipical RL settings, the reward function is non-differentiable

- In AlphaGo Zero, the ultimate reward is winning a game of Go
- For OpenAI Five the goal is winning a game of Dota 2
- In this research the goal is for a robot not to fall

If we still want to use a gradient method, we need to overcome this issue

- One way is approximating f via a differentiable critic (e.g. a NN)
- ...Another is using a stochastic policy

In the latter case, π_θ defines a probability distribution $\pi_\theta(a \mid x)$

- Given a state x, we might obtain different actions a
- ...Usually according to a Normal distribution (with fixed σ)

Stochastic Differentiation in Reinforcement Learning

With a stochastic policy, the training problem becomes:

$$\operatorname*{argmax}_{\theta} \mathbb{E}_{x \sim P, a \sim \pi_\theta(a|x)} [f(x, a)]$$

- The semantic is not the same as the original, but the goal is similar
- The problem contains now a double expectation

We could try to use a Monte Carlo approach with both

- There are well established techniques for sampling x
- ...And we could sample actions $\{a_k\}_{k=1}^m$ directly from $\pi_\theta(a \mid x)$, obtaining:

$$\mathbb{E}_{a \sim \pi_\theta(a|x)} [f(x, a)] \simeq \frac{1}{m} \sum_{k=1}^{m} f(x, a_k)$$

...But unfortunately this expression is again non-differentiable

Stochastic Differentiation in Reinforcement Learning

It is possible to circumvent the issue via importance sampling

We sample the actions uniformly at random, but then we alter their distribution

- All p_i are identical, due to the uniform assumption
- The p_i^* are given by the policy itself, leading to:

$$\mathbb{E}_{a \sim \pi_\theta(a|x)}\left[f(x, a)\right] \simeq \frac{1}{m} \sum_{k=1}^{m} \pi_\theta(a_k \mid x) f(x, a_k)$$

- While the $f(x, a_k)$ is still just a constant
- ...The probability $\pi_\theta(a_k \mid x)$ is now differentiable in θ

Intuitively: we train to increase the probability of good actions

Stochastic Differentiation in Reinforcement Learning

This differentiation trick via importance sampling is a bit crude

- Uniform sampling might generate actions with very low probability
- ...Leading to noisy estimates and numerical issues

In practice, it's not a good idea to use it direcly

...But it is the basis for some famous RL methods!

- It is used to derive the original REINFORCE algorithm
- It is central to the TRPO method
- ...And to the state-of-the-art Proximal Policy Optimization method