RUL Base Maintenance

Problem and Data

Remaining Useful Life

The Remaining Useful Life is a key concept in predictive maintenance

The RUL refers to the time until a component becomes unusable

- If we can estimate the RUL of a component
- ...We can schedule maintenance operations only when they are needed

Current best practices are based on preventive maintenance

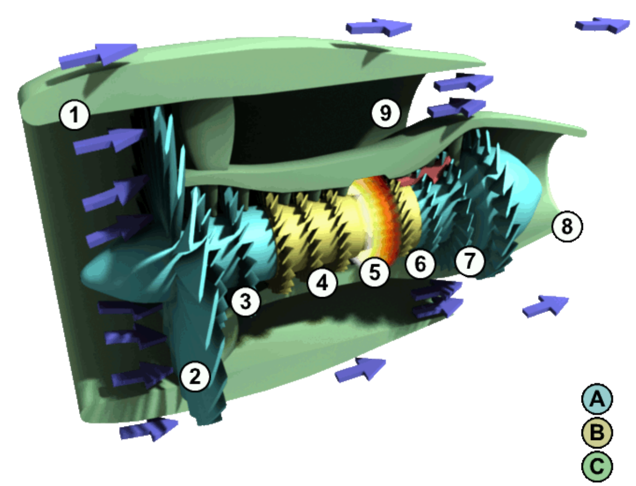
I.e. on having a fixed maintenance schedule for each component family

- RUL prediction can lead to significant savings
- ...By delaying maintenance operations w.r.t. the schedule
- ...But only as long as we are still able to prevent critical failures

The Dataset

We will consider the NASA C-MAPSS dataset

- The Modular Aero-Propulsion System Simulation (MAPSS)
- ...Is a NASA-developed simulator for turbofan engines



- It comes with both a Military (MAPSS) and commercial versionn (C-MAPSS)
- They different in the attributes of the considered engines

The Dataset

The C-MAPSS system can simulate a number of faults and defects

...And it was used to build a high-quality dataset for a competition

- The dataset consists of 4 "training set" files and 4 "test set" files
- The dataset differ by operating conditions (sea level only or different altitudes)
- ...And by fault types (High Pressure Compressor, fan)
- All engines are assumed to be healthy at the beginning of the simulation

We will focus on the hardest setup

- Multiple operating conditions
- Two fault types

Inspecting the Data

Let's have a look at the row data

In [2]:
```
data_raw = util.load_data(data_folder=os.path.join('..', 'data'))
data_dict = util.split_by_field(data_raw, field='src')
data = data_dict['train_FD004']
data.head()
```

Out[2]:

	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	s14	s15	s16	s1
0	train_FD004	1	1	42.0049	0.8400	100.0	445.00	549.68	1343.43	1112.93	...	2387.99	8074.83	9.3335	0.02	3
1	train_FD004	1	2	20.0020	0.7002	100.0	491.19	606.07	1477.61	1237.50	...	2387.73	8046.13	9.1913	0.02	3
2	train_FD004	1	3	42.0038	0.8409	100.0	445.00	548.95	1343.12	1117.05	...	2387.97	8066.62	9.4007	0.02	3
3	train_FD004	1	4	42.0000	0.8400	100.0	445.00	548.70	1341.24	1118.03	...	2388.02	8076.05	9.3369	0.02	3
4	train_FD004	1	5	25.0063	0.6207	60.0	462.54	536.10	1255.23	1033.59	...	2028.08	7865.80	10.8366	0.02	3

5 rows × 28 columns

- Columns "p1, p2, p3" refer to controlled parameters
- Columns "s1" to "s21" refer to sensor reading
- Binning has already been applied in the original dataset

Statistics

Let's check some statistics

In [3]:
```
dt_in = list(data.columns[3:-1]) # Exclude metadata
data[dt_in].describe()
```

Out[3]:

	p1	p2	p3	s1	s2	s3	s4	s5	
count	61249.000000	61249.000000	61249.000000	61249.000000	61249.000000	61249.000000	61249.000000	61249.000000	61
mean	23.999823	0.571347	94.031576	472.882435	579.420056	1417.896600	1201.915359	8.031626	11
std	14.780722	0.310703	14.251954	26.436832	37.342647	106.167598	119.327591	3.622872	5.4
min	0.000000	0.000000	60.000000	445.000000	535.480000	1242.670000	1024.420000	3.910000	5.6
25%	10.004600	0.250700	100.000000	445.000000	549.330000	1350.550000	1119.490000	3.910000	5.7
50%	25.001400	0.700000	100.000000	462.540000	555.740000	1367.680000	1136.920000	7.050000	9.0
75%	41.998100	0.840000	100.000000	491.190000	607.070000	1497.420000	1302.620000	10.520000	15
max	42.008000	0.842000	100.000000	518.670000	644.420000	1613.000000	1440.770000	14.620000	21

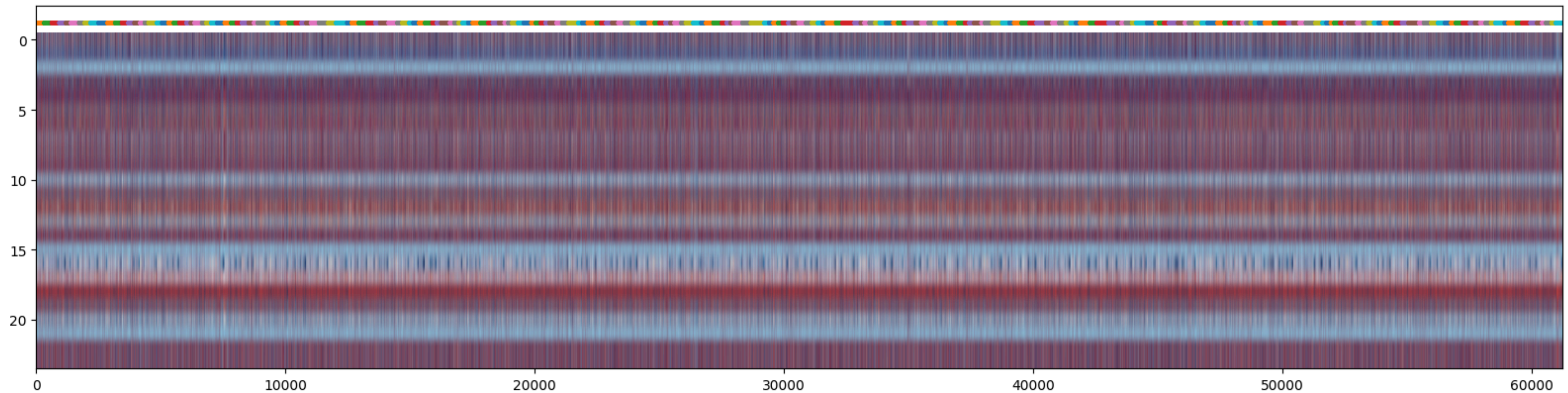
8 rows × 24 columns

There are appears to be no missing value

Heatmaps

We'll use a heatmap to get a glance of all data at once

In [4]:
```
util.plot_dataframe(data[dt_in], labels=data['machine'], figsize=figsize, vmin=-1, vmax=1)
```

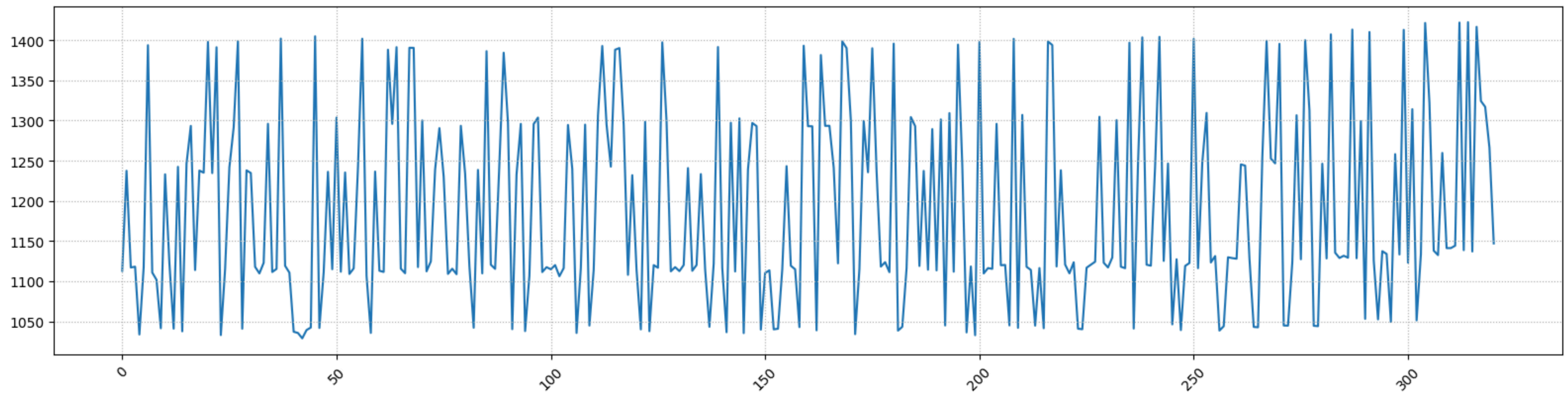


- Time is on the x-axis, every row corresponds to a table column
- Red = below average, blue = above average

A Sample Column

Let's plot one column in deeper detail for a single machine/experiment

```
In [5]: tmp = data[data['machine'] == data['machine'].iloc[0]]
        util.plot_series(tmp['s4'], figsize=figsize)
```

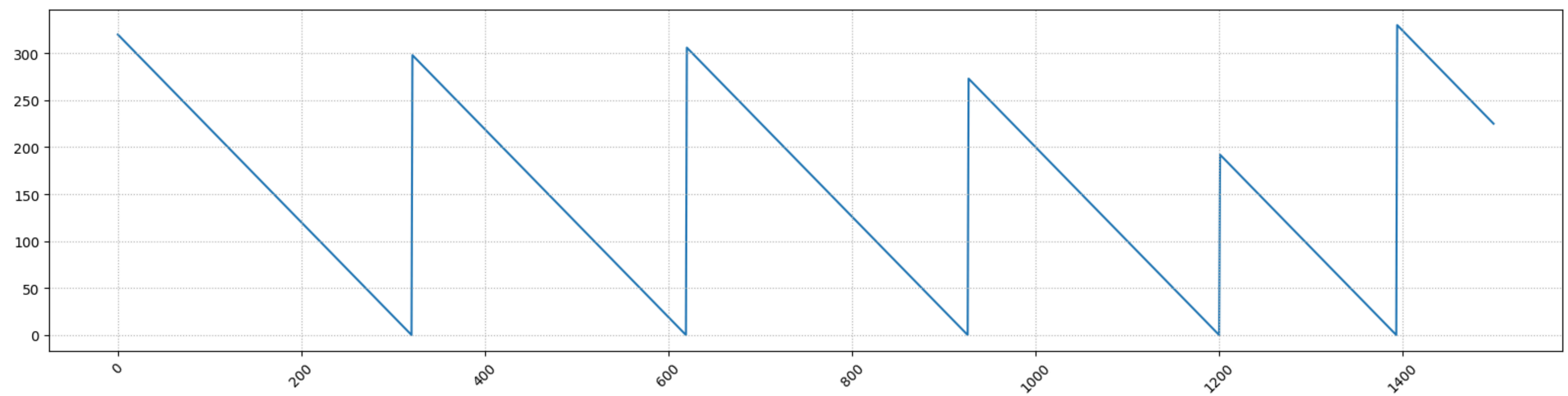


- There might be an increasing trend, but it's quit weak

Remaining Useful life

Let's have a look at the "rul" column

```
In [6]: util.plot_series(data['rul'].iloc[:1500], figsize=figsize)
```



- It has a saw-tooth pattern, since the duration of each experiment is known

RUL Base Maintenance

RUL Prediction

Say we want to define a RUL-based maintenance policy

How could we tackle that problem?

System Modeling

Let's start from modeling the system

We can view the RUL and the observed data as

$$X, R \sim P(X, R)$$

Since X is observed, we can actually focus on the conditional distribution of R:

$$R \sim P(R \mid X)$$

We can then define the expected RUL given observed values x for X:

$$f(x) = \mathrm{E}_{R \sim P(R|X=x)}\left[R\right]$$

This is exactly just the formalization for a classical regression problem

RUL Prediction as Regression

With this information, we can formulate a simple maintenance policy

We will train a regression model $\hat{y} = \hat{f}(x; \theta)$ to approximate $f(x)$

- We can use any regression approach in principle
- E.g. linear regression, Neural Networks, Random Forests, etc.

Then we trigger maintenance when the estimated RUL becomes too low, i.e.:

$$\hat{y} = \hat{f}(x; \theta) \leq \varepsilon$$

- θ is the vector of model parameters
- The threshold ε must account for possible estimation errors

We now need to define our training and test data

How do we proceed?

Training and Test Data

We now need to define our training and test data

In a practical setting:

- Some run-to-failure experiments will form the training set
- Others run-to-failure experiments will be used for testing

I.e. we split whole experiments rather than individual examples!

Each run-to-failure experiment in our data is associated to a machine

Let's check how many we have:

In [7]:
```python
print(f'Number of machines: {len(data.machine.unique())}')
```

```
Number of machines: 249
```

- This is actually a very large number (way more than typically available)

Training and Test Data

Let's use 75% of the machine for training, the rest for testing

First, we partition the machine indexes:

In [8]:
```
tr_ratio = 0.75
np.random.seed(42)
machines = data.machine.unique()
np.random.shuffle(machines)

sep = int(tr_ratio * len(machines))
tr_mcn = machines[:sep]
ts_mcn = machines[sep:]
```

Then, we partition the dataset itself:

In [9]:
```
tr, ts = util.partition_by_machine(data, tr_mcn)
```

Training and Test Data

Let's have a look at the training data

In [10]:
```
tr
```

Out[10]:

	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	s14	s15	s1
0	train_FD004	1	1	42.0049	0.8400	100.0	445.00	549.68	1343.43	1112.93	...	2387.99	8074.83	9.3335	0.0
1	train_FD004	1	2	20.0020	0.7002	100.0	491.19	606.07	1477.61	1237.50	...	2387.73	8046.13	9.1913	0.0
2	train_FD004	1	3	42.0038	0.8409	100.0	445.00	548.95	1343.12	1117.05	...	2387.97	8066.62	9.4007	0.0
3	train_FD004	1	4	42.0000	0.8400	100.0	445.00	548.70	1341.24	1118.03	...	2388.02	8076.05	9.3369	0.0
4	train_FD004	1	5	25.0063	0.6207	60.0	462.54	536.10	1255.23	1033.59	...	2028.08	7865.80	10.8366	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
60989	train_FD004	248	180	35.0019	0.8409	100.0	449.44	556.28	1377.65	1148.96	...	2387.77	8048.91	9.4169	0.0
60990	train_FD004	248	181	0.0023	0.0000	100.0	518.67	643.95	1602.98	1429.57	...	2388.27	8122.44	8.5242	0.0
60991	train_FD004	248	182	25.0030	0.6200	60.0	462.54	536.88	1268.01	1067.09	...	2027.98	7865.18	10.9790	0.0
60992	train_FD004	248	183	41.9984	0.8414	100.0	445.00	550.64	1363.76	1145.72	...	2387.48	8069.84	9.4607	0.0
60993	train_FD004	248	184	0.0013	0.0001	100.0	518.67	643.50	1602.12	1430.34	...	2388.33	8120.43	8.4998	0.0

45385 rows × 28 columns

Training and Test Data

...And at the test data

In [11]:
```
ts
```

Out[11]:

	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	s14	s15	s1
321	train_FD004	2	1	41.9998	0.8400	100.0	445.00	548.99	1341.82	1113.16	...	2387.98	8082.37	9.3300	0.0
322	train_FD004	2	2	9.9999	0.2500	100.0	489.05	604.23	1498.00	1299.54	...	2388.07	8125.46	8.6088	0.0
323	train_FD004	2	3	42.0079	0.8403	100.0	445.00	549.11	1351.47	1126.43	...	2387.93	8082.11	9.2965	0.0
324	train_FD004	2	4	42.0077	0.8400	100.0	445.00	548.77	1345.81	1116.64	...	2387.88	8079.41	9.3200	0.0
325	train_FD004	2	5	24.9999	0.6200	60.0	462.54	537.00	1259.55	1043.95	...	2028.13	7867.08	10.8841	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
61244	train_FD004	249	251	9.9998	0.2500	100.0	489.05	605.33	1516.36	1315.28	...	2388.73	8185.69	8.4541	0.0
61245	train_FD004	249	252	0.0028	0.0015	100.0	518.67	643.42	1598.92	1426.77	...	2388.46	8185.47	8.2221	0.0
61246	train_FD004	249	253	0.0029	0.0000	100.0	518.67	643.68	1607.72	1430.56	...	2388.48	8193.94	8.2525	0.0
61247	train_FD004	249	254	35.0046	0.8400	100.0	449.44	555.77	1381.29	1148.18	...	2388.83	8125.64	9.0515	0.0
61248	train_FD004	249	255	42.0030	0.8400	100.0	445.00	549.85	1369.75	1147.45	...	2388.66	8144.33	9.1207	0.0

15864 rows × 28 columns

Standardization/Normalization

We will use a Neural Network regressor

...Therefore, we need to make the range of each columns more uniform

- We will standardize all parameters and sensor inputs:

```
In [12]: trmean = tr[dt_in].mean()
         trstd = tr[dt_in].std().replace(to_replace=0, value=1) # handle static fields

         ts_s = ts.copy()
         ts_s[dt_in] = (ts_s[dt_in] - trmean) / trstd
         tr_s = tr.copy()
         tr_s[dt_in] = (tr_s[dt_in] - trmean) / trstd
```

- We will normalize the RUL values (i.e. our regression target)

```
In [13]: trmaxrul = tr['rul'].max()

         ts_s['rul'] = ts['rul'] / trmaxrul
         tr_s['rul'] = tr['rul'] / trmaxrul
```

Standardization/Normalization

Let's check the results

In [14]:
```
tr_s.describe()
```

Out[14]:

	machine	cycle	p1	p2	p3	s1	s2	s3
count	45385.000000	45385.000000	4.538500e+04	4.538500e+04	4.538500e+04	4.538500e+04	4.538500e+04	4.538500e+04
mean	122.490955	133.323896	2.894775e-16	1.302570e-16	1.178889e-16	4.664830e-15	2.522791e-15	1.727041e-15
std	71.283034	89.568561	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
min	1.000000	1.000000	-1.623164e+00	-1.838222e+00	-2.381839e+00	-1.055641e+00	-1.176507e+00	-1.646830e+00
25%	61.000000	62.000000	-9.461510e-01	-1.031405e+00	4.198344e-01	-1.055641e+00	-8.055879e-01	-6.341243e-01
50%	125.000000	123.000000	6.868497e-02	4.154560e-01	4.198344e-01	-3.917563e-01	-6.336530e-01	-4.718540e-01
75%	179.000000	189.000000	1.218855e+00	8.661917e-01	4.198344e-01	6.926385e-01	7.407549e-01	7.495521e-01
max	248.000000	543.000000	1.219524e+00	8.726308e-01	4.198344e-01	1.732749e+00	1.741030e+00	1.837978e+00

8 rows × 27 columns

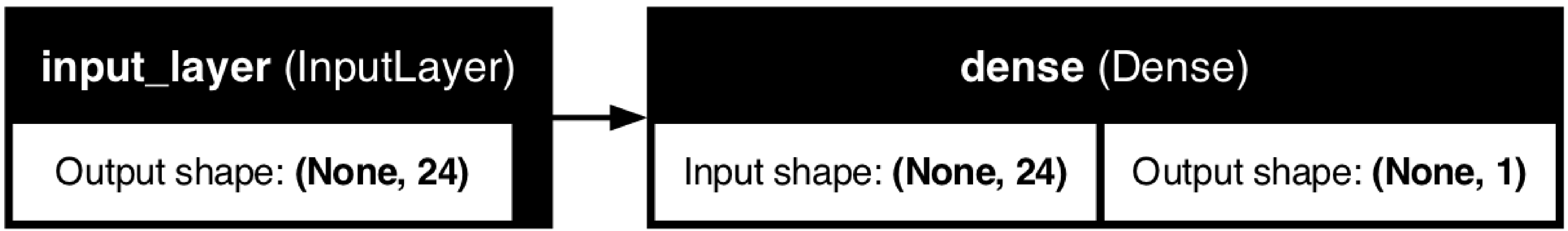
Regression Model

We will start with the simplest possible Neural Network

...Meaning a Linear Regressor!

In [15]:
```
nn1 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[])
util.plot_nn_model(nn1, dpi=150)
```

Out[15]:

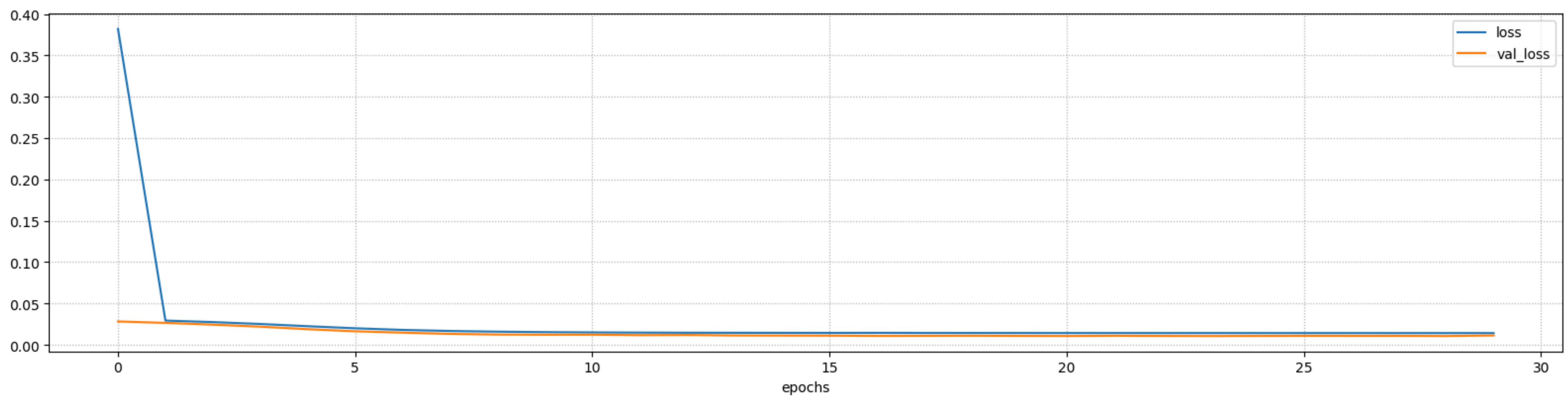


- We just need to specify that there are no hidden layers
- Why the simplest? As usual, due to Occam's razor

Training

We can now train our model

```
In [16]: nn1 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[])
         history = util.train_nn_model(nn1, tr_s[dt_in], tr_s['rul'], loss='mse', epochs=30, validat:
         util.plot_training_history(history, figsize=figsize)
```



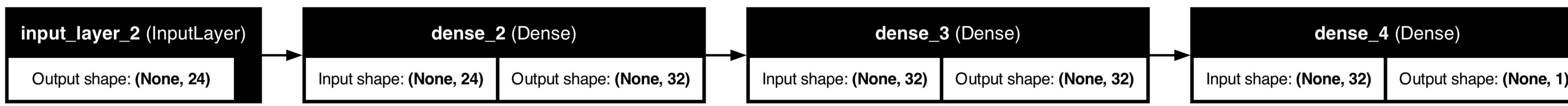

```
Final loss: 0.0142 (training), 0.0114 (validation)
```

Training

Let's try with a more complex model

In [17]:
```
nn2 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32, 32])
util.plot_nn_model(nn2, dpi=200)
```

Out[17]:

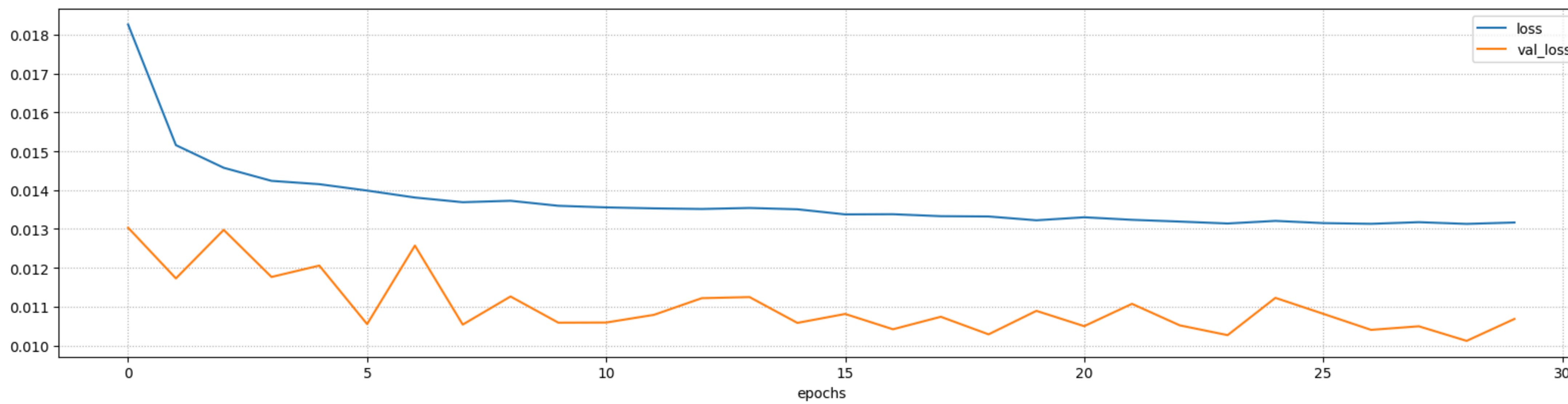


- Now we have two hidden layers
- ...Each with 32 ReLU neurons

Training

Let's check the loss behavior and compare it to Linear Regression

In [18]:
```
nn2 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32, 32])
history = util.train_nn_model(nn2, tr_s[dt_in], tr_s['rul'], loss='mse', epochs=30, validat
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.0132 (training), 0.0107 (validation)
```

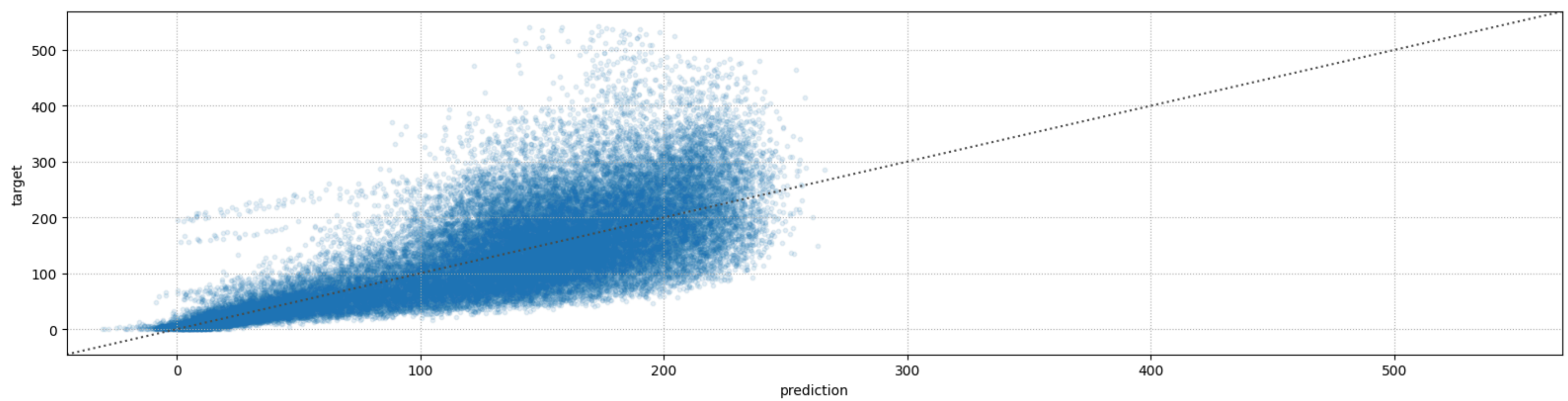
- There is a small improvement w.r.t. Linear Regression

Predictions

We can now obtain the predictions and evaluate their quality

```
In [19]: tr_pred = nn2.predict(tr_s[dt_in], verbose=0).ravel() * trmaxrul
         util.plot_pred_scatter(tr_pred, tr['rul'], figsize=figsize)
         print(f'R2 score: {r2_score(tr["rul"], tr_pred):.4f}')
```

```
R2 score: 0.5432
```



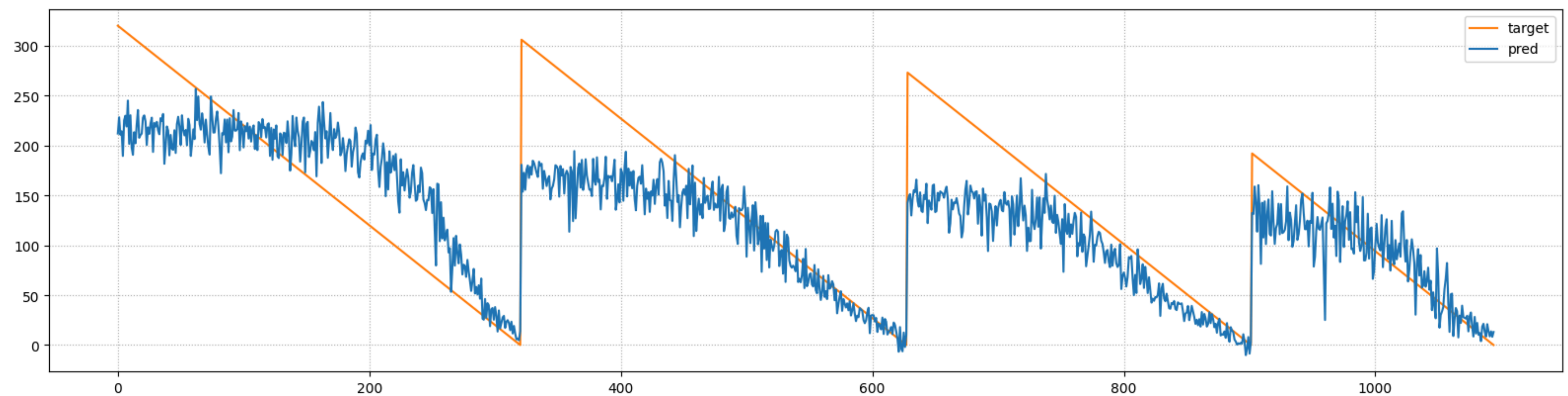
What do you think of these results?
Are they good or bad?

Predictions

The results so far are not comforting

...But it's worth seeing what is going on over time:

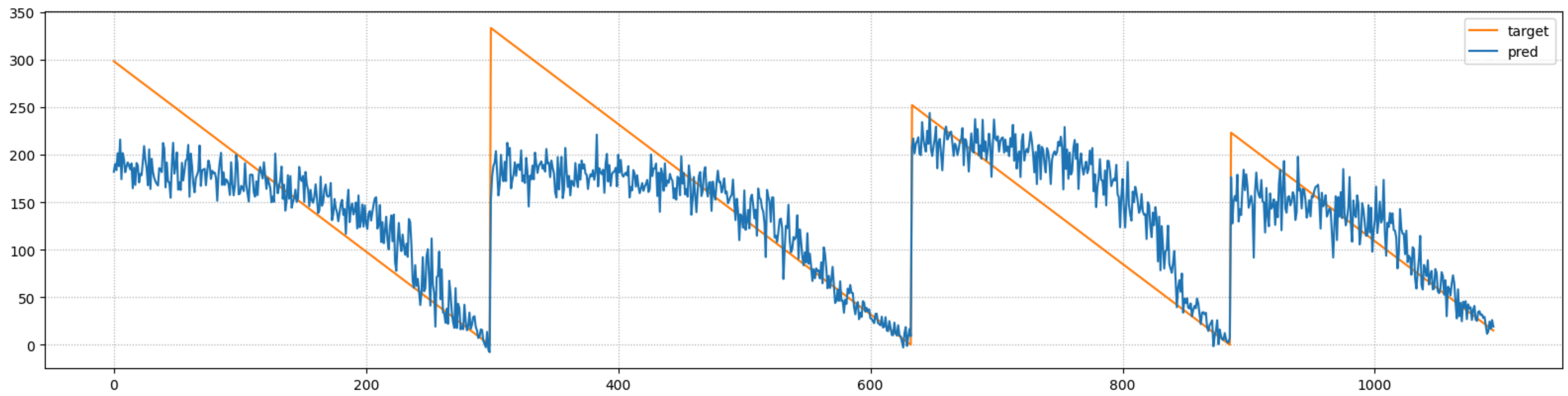
```
In [20]: stop = 1095
         util.plot_rul(tr_pred[:stop], tr['rul'][:stop], figsize=figsize)
```



Predictions

The situation is similar on the test set:

In [21]:
```
ts_pred = nn2.predict(ts_s[dt_in], verbose=0).ravel() * trmaxrul
util.plot_rul(ts_pred[:stop], ts['rul'][:stop], figsize=figsize)
```



Quality Evaluation

Let's try to recap the situation

Our accuracy is quite poor especially for large RUL values

- This may happens since large RUL value are somewhat scarce on the dataset
- ...Or because fault effects become noticeable only after a while

Quality Evaluation

Let's try to recap the situation

Our accuracy is quite poor especially for large RUL values

- This may happens since large RUL value are somewhat scarce on the dataset
- ...Or because fault effects become noticeable only after a while

But perhaps we don't care! Our goal is not a high accuracy

- We just need to stop at the right time
- ...And our model may still be good enough for that

For a proper evaluation, we need a cost model

Cost Model

We will assume that:

We consider one step of operation as our value unit

- ...So we can express the failure cost in terms of operating steps

Cost Model

We will assume that:

We consider one step of operation as our value unit

- ...So we can express the failure cost in terms of operating steps

Every run end with either failure or maintenance:

- Assuming that the failure cost is higher than maintenance cost
- ...We can diseregard the maintenance cost

Cost Model

We will assume that:

We consider one step of operation as our value unit

- ...So we can express the failure cost in terms of operating steps

Every run end with either failure or maintenance:

- Assuming that the failure cost is higher than maintenance cost
- ...We can diseregard the maintenance cost

A traditional preventive maintenance policy is also available

- We will never trigger maintenance ealier that such policy
- We only gain value if we beat such policy

Cost Model

The whole cost formula for a single machine will be:

$$cost(\hat{y}), \varepsilon) = op\_profit(\hat{y}, \varepsilon) + fail\_cost(\hat{y}, \varepsilon)$$

Where:

$$op\_profit(\hat{y}, \varepsilon) = -\max(0, stop\_time(\hat{y}, \varepsilon) - s)$$

$$fail\_cost(\hat{y}, \varepsilon) = \begin{cases} C \text{ if } \max(\hat{y}) \geq \varepsilon \\ 0 \text{ otherwise} \end{cases}$$

- If we fail, we pay C cost unit more than maintenance
- Profit is modeled as a negative cost
- We only make profit if we stop after s units

Cost Model

Normally, we would proceed as follows

- s is determined by the preventive maintenance schedule
- C must be determined by discussing with the customer

In our example, we will derive both from data

First, we collect all failure times

In [22]:
```
failtimes = data.groupby('machine')['cycle'].max()
failtimes.head()
```

Out[22]:
```
machine
1    321
2    299
3    307
4    274
5    193
Name: cycle, dtype: int64
```

Cost Model

Then, we define s and C based on statistics

```
In [23]: print(failtimes.describe())
         safe_interval = failtimes.min()
         maintenance_cost = failtimes.max()

count    249.00000
mean     245.97992
std       73.11080
min      128.00000
25%      190.00000
50%      234.00000
75%      290.00000
max      543.00000
Name: cycle, dtype: float64
```

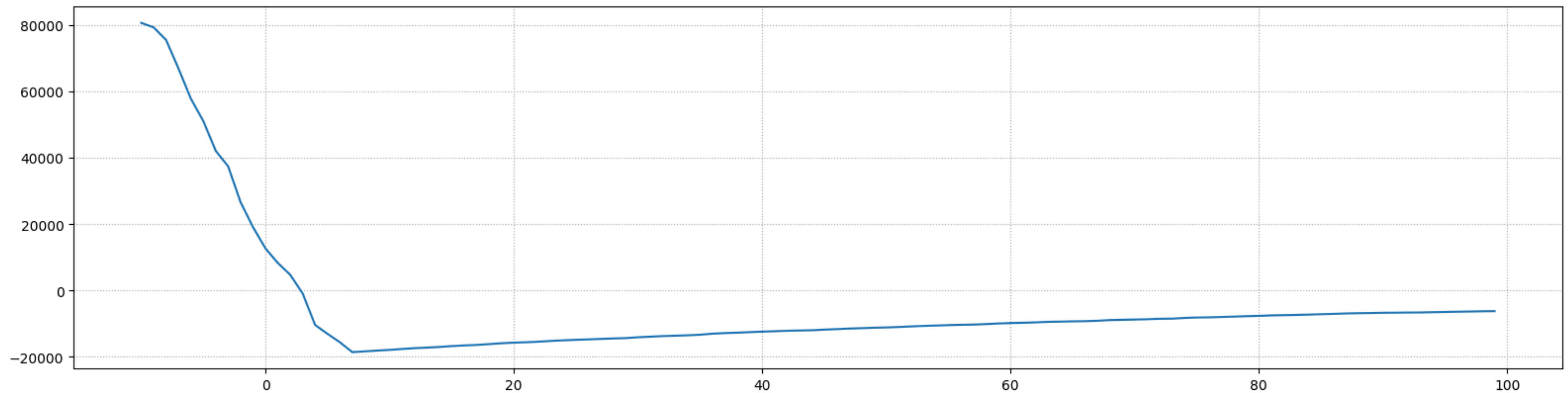
- For the safe interval s, we choose the minimum failure time
- For the maintenance cost C we choose the largest failure time

Threshold Choice

We can then choose the threshold θ as usual

```
In [24]: cmodel = util.RULCostModel(maintenance_cost=maintenance_cost, safe_interval=safe_interval)
         th_range = np.arange(-10, 100)
         tr_thr = util.opt_threshold_and_plot(tr['machine'].values, tr_pred, th_range, cmodel, figsi
         print(f'Optimal threshold for the training set: {tr_thr}')
```

```
Optimal threshold for the training set: 7
```



Evaluation

Let's see how we fare in terms of cost

```
In [25]: tr_c, tr_f, tr_sl = cmodel.cost(tr['machine'].values, tr_pred, tr_thr, return_margin=True)
         ts_c, ts_f, ts_sl = cmodel.cost(ts['machine'].values, ts_pred, tr_thr, return_margin=True)
         print(f'Avg. cost: {tr_c/len(tr_mcn):.2f} (training), {ts_c/len(ts_mcn):.2f} (test)')
```

```
Avg. cost: -100.15 (training), -110.10 (test)
```

We can also evaluate the margin for improvement:

```
In [26]: print(f'Avg. fails: {tr_f/len(tr_mcn):.2f} (training), {ts_f/len(ts_mcn):.2f} (test)')
         print(f'Avg. slack: {tr_sl/len(tr_mcn):.2f} (training), {ts_sl/len(ts_mcn):.2f} (test)')
```

```
Avg. fails: 0.00 (training), 0.00 (test)
Avg. slack: 16.11 (training), 13.71 (test)
```

- Slack = distance between when we stop and the failure
- The results are quite good and we also generalize fairly well