In [1]:

```
# ============================================================
# Notebook setup
# ============================================================

%load_ext autoreload
%autoreload 2

# Control figure size
figsize=(14, 4)

from util import util
import os
import numpy as np
from tensorflow.keras import callbacks
import warnings
import pandas as pd
warnings.simplefilter("ignore")

# Load data
data_folder = os.path.join('..', 'data')
data = util.load_cmapss_data(data_folder)

# Focus on a subset of the data
data_by_src = util.split_by_field(data, field='src')
dt = data_by_src['train_FD004']

# Split training and test machines
trs_ratio = 0.25
tru_ratio = 0.5

np.random.seed(42)
machines = dt.machine.unique()
np.random.shuffle(machines)

sep_s = int(trs_ratio * len(machines))
sep_u = int(tru_ratio * len(machines))
trsu_mcn = machines[:sep_s+sep_u]
ts_mcn = machines[sep_s+sep_u:]
trsu, ts = util.partition_by_machine(dt, trsu_mcn)

trs_mcn = trsu_mcn[:sep_s]
tru_mcn = trsu_mcn[sep_s:]
trs, tru_full = util.partition_by_machine(trsu, trs_mcn)

# Apply random censoring to the unsupervised data
tru = util.random_censoring(tru_full, rel_censoring_lb=0.5)

tr_mcn = np.hstack([trs_mcn, tru_mcn])
tr = pd.concat([trs, tru])

# Add time information
# Identify parameter and sensor columns
dt_in = list(data.columns[3:-1])
```

```
# Standardize parameters and sensors
trmean = tr[dt_in].mean()
trstd = tr[dt_in].std().replace(to_replace=0, value=1) # handle static field
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

# Normalize RUL and time (cycle)
trmaxrul = tr['rul'].max()
ts_s['cycle'] = ts_s['cycle'] / trmaxrul
tr_s['cycle'] = tr_s['cycle'] / trmaxrul
ts_s['rul'] = ts['rul'] / trmaxrul
tr_s['rul'] = tr['rul'] / trmaxrul

# Add time (cycle) to the input columns
dt_in = dt_in + ['cycle']
```

Survival Analysis using Neural Models

Open Issues with the Previous Approach

Our probabilistic RUL model worked quite well

...But it still has some weak spots

What if we are not confident about using a Normal?

- We could swap it for another distribution
- ...But it might not be easy to guess the correct choice

What if the RUL depends strongly on what happens in the future?

- Then, we would need a lot of runs to obtain a good marginalization
- ...And data availability is a critical issue in RUL estimation

The last observation deserves further attention

Censoring

In many domains, run-to-failure experiments are expensive to obtain

...But *partial runs* might abunant

- Broken industrial machines vs regularly maintained ones
- Deaths in organ transplant waiting lists vs alive patients

The C-MAPSS dataset is very unrealistic from this point of view

The simulator is good, but there are way too many experiments

- We can simulate limited availability of supervised data
- ...By randomly truncating a portion of the training set

In survival analysis, the lack of key events is known as *censoring*

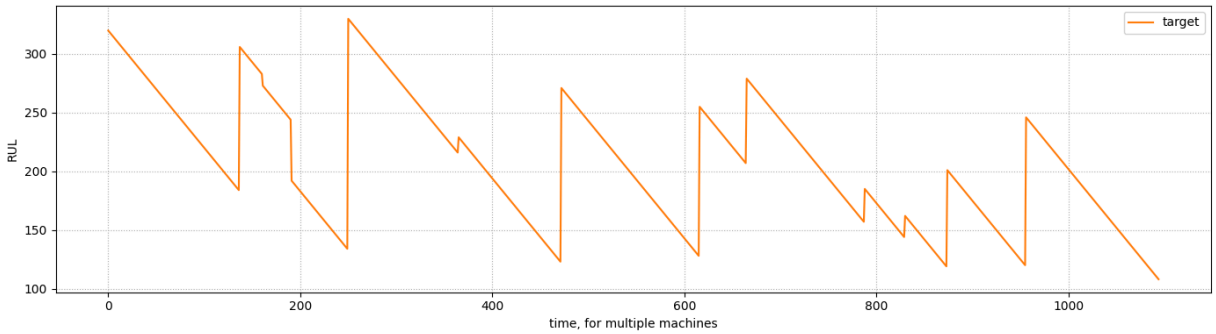
In [2]:
```
print(f'In this notebook, censoring was applied to {100*tru_ratio/(trs_ratio
```

```
In this notebook, censoring was applied to 67% of the training experiments
```

Censoring

In our plots, censoring will cause irregularities in the sawtooth pattern

In [3]:
```
stop = 1095
util.plot_rul(target=tru['rul'][:stop], figsize=figsize, xlabel='time, for m
```



- We still can plot the RUL values, but only since we used *simulated* censoring
- In a real use case, we would have *no RUL target for this data*

Can we still take advantage of this data? How?

Survival Function

We could study the distribution of T via its *survival function*

The survival function of a variable T is defined as:

$$S(t) = P(T > t)$$

I.e. it the probability that the entity "survives" at least until time t

- It is the complement of the cumulative probability function $F(t) = P(T \leq t)$

We can account for conditioning factors

...Which for the survival function only includes the *past* behavior

$$S(t, X_{\leq t}) = P(T > t \mid X_{\leq t})$$

- This means *it cannot account for the future*
- ...But also that *it cannot overfit due to poor marginalization*

...And Hazard Function

If we assume *discrete time*, then S can be *factorized*

$$S(t, X_{\leq t}) \simeq (1 - \lambda(t, X_t))(1 - \lambda(t-1, X_{t-1})) \dots$$

Where λ is called *hazard function*

The hazard function is *a conditional probability*

...That of not surving one more step. Formally:

- $\lambda(t, X_t)$ is the probability of *not surviving* at time t
- ...Given that the entity *has survived* until time $t - 1$. I.e.:

$$\lambda(t, X_t) = P(T > t \mid T > t - 1, X_t)$$

As a side effect, λ only depends on *one* observation

Our Plan

We will attempt to *train an estimator $\hat{\lambda}\theta(t, x_t)$_ for the hazard function*

- This requires *no assumption on the distribution* (besides that of using S)
- It does *not risk* overfitting due to *poor marginalization*
- And it makes sense even if we *do not observe a "death" event (censoring)*

As a side effect, we also *cannot account for future behavior*

Additionally, S and λ have more limited uses

We can still *define a threshold-based policy*, e.g. by checking whether:

$$\hat{\lambda}_\theta(t, x_t) \geq \varepsilon$$

...But we'll see that *making forecasts* is not trivial and requires approximations

Training a Hazard Estimator

Before we get that, we need a way to train our $\hat{\lambda}_\theta$ estimator

We can start by modeling the *probability of a survival event*

- Say the k-th experiment in our dataset ends at time e_k
- Then the corresponding probability according to our estimator is:

$$\hat{\lambda}_\theta(e_k, x_{k,e_k}) \prod_{t=1}^{e_k-1} (1 - \hat{\lambda}_\theta(t, x_{k,t}))$$

Where $x_{k,t}$ is the available input data for experiment k at time t

This is the probability of:

- Surviving all time steps from 1 to $e_k - 1$
- Not surviving at time e_k

Training a Hazard Estimator

We can now formulate a likelihood maximization problem

Assuming we have m experiments, we get:

$$\underset{\theta}{\operatorname{argmax}} \prod_{k=1}^{m} \hat{\lambda}_\theta(e_k, x_{k,e_k}) \prod_{t=1}^{e_k-1} (1 - \hat{\lambda}_\theta(t, x_{k,t}))$$

Then, let's rewrite the formula:

- Let $d_{kt} = 1$ iff $t = e_k$, i.e. if the experiment ends at time k
- ...And let $d_{kt} = 0$ otherwise. Then we can get:

$$\underset{\theta}{\operatorname{argmax}} \prod_{k=1}^{m} \prod_{t=1}^{e_k} d_{k,t}\hat{\lambda}_\theta(t, x_{k,t}) + (1 - d_{k,t})(1 - \hat{\lambda}_\theta(t, x_{k,t}))$$

Now the two products can be freely swapped

Training a Hazard Estimator

Starting from:

$$\underset{\theta}{\operatorname{argmax}} \prod_{k=1}^{m} \prod_{t=1}^{e_k} d_{k,t}\hat{\lambda}_\theta(t, x_{k,t}) + (1 - d_{k,t})(1 - \hat{\lambda}_\theta(t, x_{k,t}))$$

We obtain an equivalent problem through a log transformation:

$$\operatorname{argmax}_\theta \sum_{k=1}^{m} \sum_{t=1}^{e_k} \log\Big(d_{k,t}\hat{\lambda}_\theta(t, x_{k,t}) + (1 - d_{k,t})(1 - \hat{\lambda}_\theta(t, x_{k,t}))\Big)$$

Since either $d_{k,t} = 1$ or $d_{k,t} = 0$, we can also split the log argument:

$$\text{argmax}_\theta \sum_{k=1}^{m} \sum_{t=1}^{e_k} d_{k,t} \log \hat{\lambda}_\theta(t, x_{k,t}) + (1 - d_{k,t}) \log(1 - \hat{\lambda}_\theta(t, x_{k,t}))$$

Training a Hazard Estimator

Finally, with a sign switch we get:

$$\text{argmin}_\theta - \sum_{k=1}^{m} \sum_{t=1}^{e_k} d_{k,t} \log \hat{\lambda}_\theta(t, x_{k,t}) + (1 - d_{k,t}) \log(1 - \hat{\lambda}_\theta(t, x_{k,t}))$$

> **Does this remind you of something?**

This is a (binary) *crossentropy minimization* problem!

- $d_{k,t}$ has the same role as a class
- $\hat{\lambda}_\theta(t, x_{k,t})$ is the model output
- We have a sample for every experiment and time step (the double summation)

Training a Hazard Estimator

This means that our $\hat{\lambda}_\theta$ can be seen *as a classifier*

- We just need to consider all samples in our dataset individually
- Then attach to them a class corresponding to d_{kt}
- ...And finally we can train a neural classifier as usual

The model output will be *an estimate of the hazard function*

This is almost precisely *what we did in our classification approach*

...But now we have *a much better interpretation*

- We know how to define the classes
- We better know how to interpret the output
- We know the semantic for a threshold-based policy
- We know that we can safely deal with censoring

Classes and Models

Let's start by defining the classes

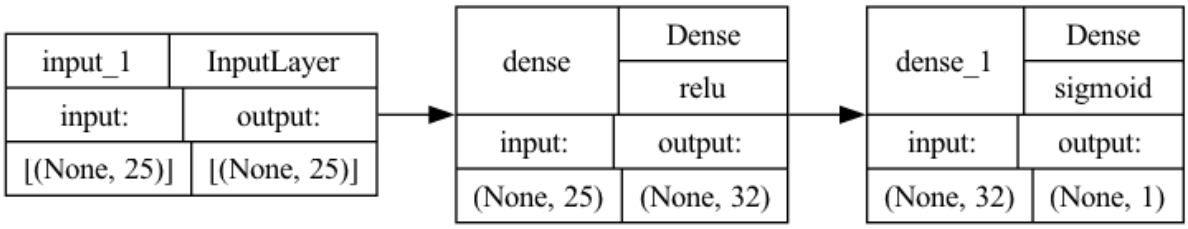
We check when the RUL is 0 (this the same as $t = e_k$)

In [4]:
```
tr_lbl = (tr['rul'] == 0)
ts_lbl = (ts['rul'] == 0)
```

Then we can build a (usual) classification model:

In [5]:
```
nnl = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidder
util.plot_nn_model(nnl)
```

Out[5]:



Effect on Censoring on the Distribution

The new approach allows us to use censored data

This is good, but it also has the effect of altering the distribution

- For end-to-failure experiments, are samples follow their natural distribution
- ...But censored data includes no end event, causing a skew

We can try to account for that by using *sample weights*

- Intuitively, if the censored data is equal to 100% of the sample with 0 label
- ...That will make the 0 label apparently twice as likely

Therefore, we can discount censored samples in the distribution:

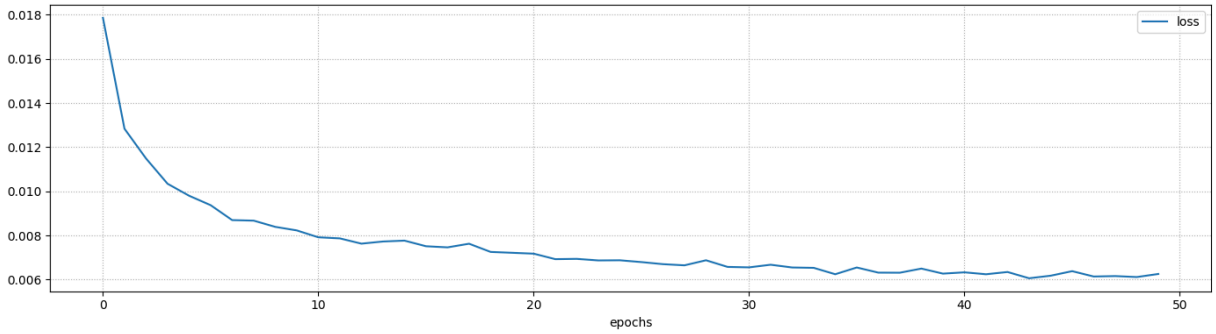
In [6]:
```
n_zeros = (tr_lbl == False).sum()
class_weights = np.array([(n_zeros - len(tru)) / n_zeros, 1.])
sample_weight = np.choose(tr_lbl, class_weights)
```

Training the Hazard Estimator

Then we train the hazard estimator as any other classifier

In [7]:
```
nnl = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidder
history = util.train_nn_model(nnl, tr_s[dt_in], tr_lbl, loss='binary_crosser
```

```
        verbose=0, patience=10, batch_size=32, validation_split=0.0, sample_
util.plot_training_history(history, figsize=figsize)
```



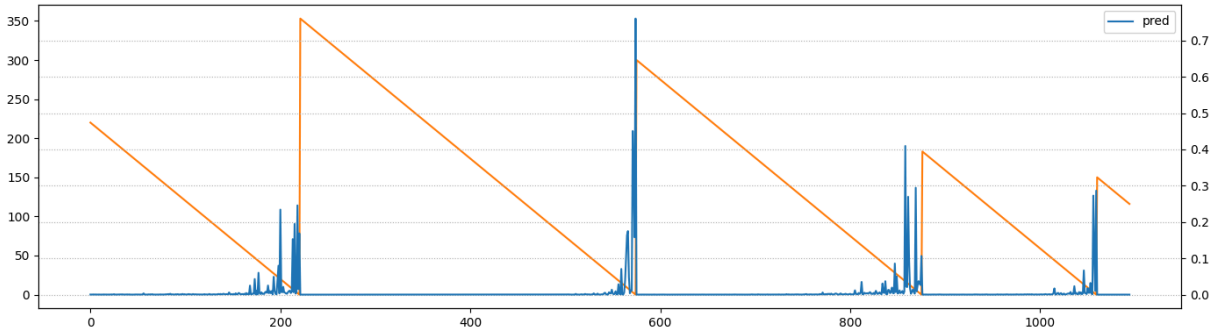

```
Final loss: 0.0063 (training)
```

Inspecting Hazards

We will start our evaluation by inspecting the hazard values

First for (part of) the *training set*:

In [8]:
```
tr_pred = nnl.predict(tr_s[dt_in], verbose=0).ravel()
stop = 1095
util.plot_rul(pred=tr_pred[:stop], target=tr['rul'][:stop], same_scale=False
```

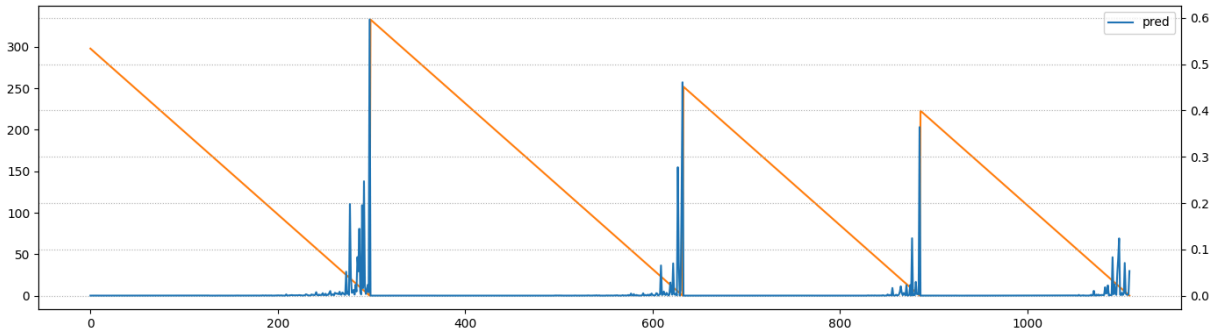


Inspecting Hazards

We will start our evaluation by inspecting the hazard values

...And here for (part of) the *test set*:

In [9]:
```
ts_pred = nnl.predict(ts_s[dt_in], verbose=0).ravel()
stop = 1110
util.plot_rul(pred=ts_pred[:stop], target=ts['rul'][:stop], same_scale=False
```



Hazard-based Policies

We can define a policy based on the $\hat{\lambda}_\theta$ estimator as usual

Namely, we trigger maintenance when:

$$\hat{\lambda}_\theta(t, x_t) \geq \varepsilon$$

The threshold can be defined again based on some cost metric

Some comments

- The old classifier-based approach still makes sense
- ...Though reasoning in terms of hazard function can be more versatile
- This approach can be combined with a sliding window input
- ...And smoothing might be be a good idea to avoid accidental triggering

Using Hazards for Forecasting

Additionally, we can use $\hat{\lambda}_\theta$ to perform forecasting

In particular, we know the probability of surving n *more* steps is given by:

$$\frac{S(t+n)}{S(t)} = \prod_{h=0}^{n}(1 - \lambda(t+h, X_{t+h}))$$

...Which we can approximate (for a run k) as:

$$\frac{S(t+n)}{S(t)} \simeq \prod_{h=0}^{n}(1 - \hat{\lambda}_\theta(t+h, x_{k,t+h}))$$

- In theory, we can forecast survival probabilities arbitrarily far
- ...But in practice there is an issue

Using Hazards for Forecasting

The formula requires access to *future values* of the X_t variable

$$\frac{S(t+n)}{S(t)} \simeq \prod_{h=0}^{n}(1-\hat{\lambda}_\theta(t+h, x_{k,t+h}))$$

- Unfortunately, we cannot access those in real life :-(
- We have two main options to deal with this

First, can *ignore time-varying input* in our estimator

Formally, this is the same as marginalizing out all time-varying factors

- $\hat{\lambda}_\theta(t, x_t)$ becomes $\hat{\lambda}_\theta(t, x)$, for a fixed x
- x represents some stable information, e.g. component type, genetics

In some cases, this is perfectly viable approach

Using Hazards for Forecasting

Second, we can attempt to *predict future* x_t values

This is viable as long as our predictions are good enough

- We can use a second ML estimator to predict x_t
- ...Or as a special case we can rely on the simple *persistence model*

In practice, we just assume x_t is stable for some time

With this simple assumption, we get:

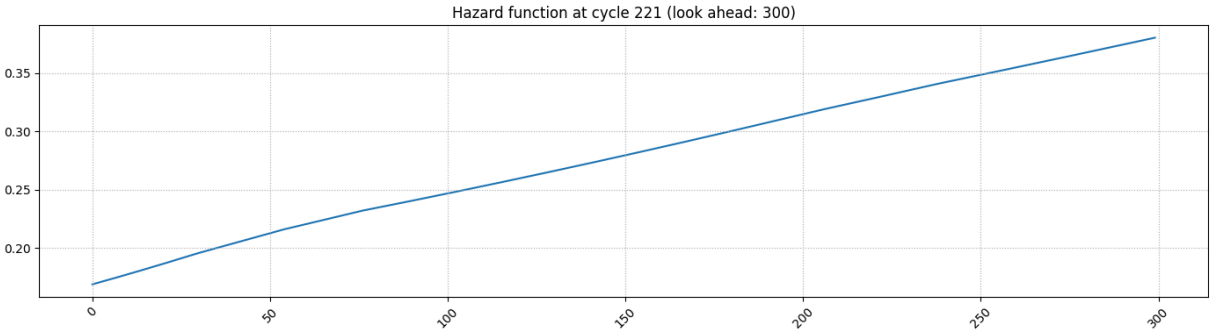
$$\frac{S(t+n)}{S(t)} \simeq \prod_{h=0}^{n}(1-\hat{\lambda}_\theta(t+h, x_{k,t}))$$

- Unlike the original expression, this is easy to compute
- ...And it might be a reasonable approximation for shorter time horizons

Approximate Future Hazard

Let's check this approximate future hazard for one of our test runs

In [10]:
```
ref_sample = tr_s.iloc[220]
look_ahead = 300
hazard = util.predict_cf(nnl, ref_sample[dt_in], columns='cycle',
                         values=ref_sample['cycle'] + np.arange(look_ahead)/
util.plot_series(hazard, figsize=figsize, title=f'Hazard function at cycle {
```

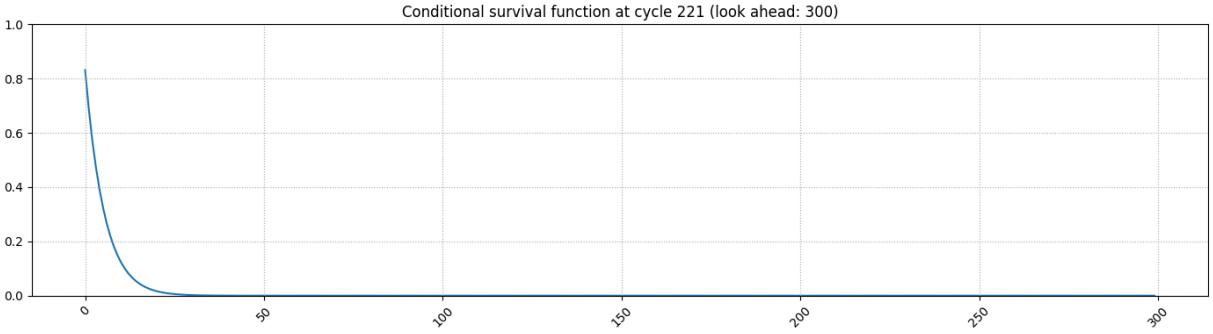


- The model has learned that time has an effect on λ

Approximate Conditional Survival

We can then estimate the conditional survival probability

In [11]:
```
survival = pd.Series(data=np.cumprod(1-hazard))
util.plot_series(survival, figsize=figsize, ylim=(0,1),
                 title=f'Conditional survival function at cycle {ref_sample[
```



The chance of being still running is smaller even in a few tens of steps

Approximate Conditional Survival

We can continuously compute n-step ahead conditional survival

Here's an example for 30-steps ahead, on the first test set experiment

In [12]:
```
ref_run = ts_s[ts_s['machine'] == ts_s.iloc[0]['machine']]
look_up_window = np.arange(30)/trmaxrul
rolling_survival = util.rolling_survival_cmapss(hazard_model=nnl, data=ref_r
rolling_survival.columns = [f'S(t+{h})/S(t)' for h in range(30)]
rolling_survival.head()
```

Out[12]:

	S(t+0)/S(t)	S(t+1)/S(t)	S(t+2)/S(t)	S(t+3)/S(t)	S(t+4)/S(t)	S(t+5)/S(t)	S(t+6)/S
321	0.999975	0.999949	0.999924	0.999898	0.999872	0.999846	0.999
322	0.999944	0.999887	0.999830	0.999772	0.999715	0.999657	0.999
323	0.999980	0.999960	0.999941	0.999921	0.999901	0.999880	0.999
324	0.999995	0.999990	0.999985	0.999980	0.999975	0.999970	0.999
325	0.999994	0.999989	0.999983	0.999977	0.999972	0.999966	0.999

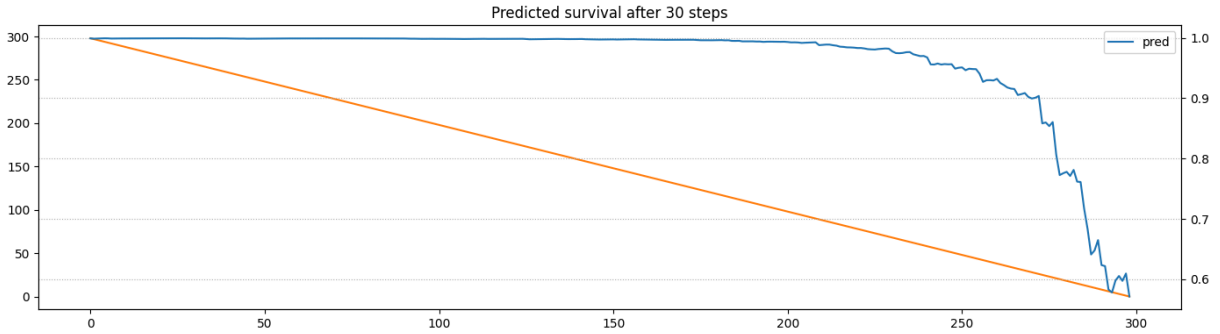
5 rows × 30 columns

- Each column contains the conditional survival h steps away

Approximate Conditional Survival

Here's a plot over time (after some smoothing)

In [13]:
```
rolling_survival_last = rolling_survival[rolling_survival.columns[-1]].ewm(1
util.plot_rul(pred=rolling_survival_last[:stop], target=ref_run['rul']*trmax
              figsize=figsize, title='Predicted survival after 30 steps')
```



- Remember that this is a stochastic phenomenon
- So, even an 80% chance is quite dangerous to take!

In Hindsight...

This whole lecture block was about *probabilistic models*

- The techniques we covered are interesting per-se
- ...And way more useful in practice than you might think

...But what the core message I hope you glimpsed is another

> **Machine Learning models are *not inflexible* tools**

- If you spot a limit, or a piece of information you can use
- ...And you know what you are doing

Then you can *dramatically change* their behavior!