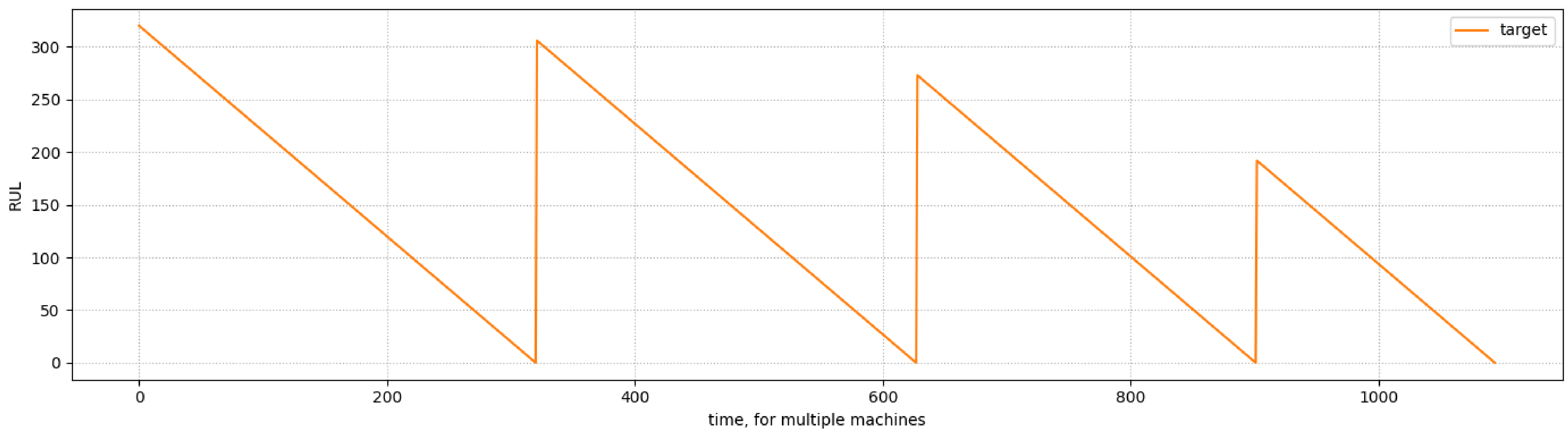
Survival Analysis using Neural Models

RUL Estimation, Again

For our RUL estimation problem, we tried two approaches

The first consisted in using a regressor to estimate this kind of function:

```
In [23]: stop = 1095
         util.plot_rul(target=tr['rul'][:stop], figsize=figsize, xlabel='time, for multiple machines', yl
```

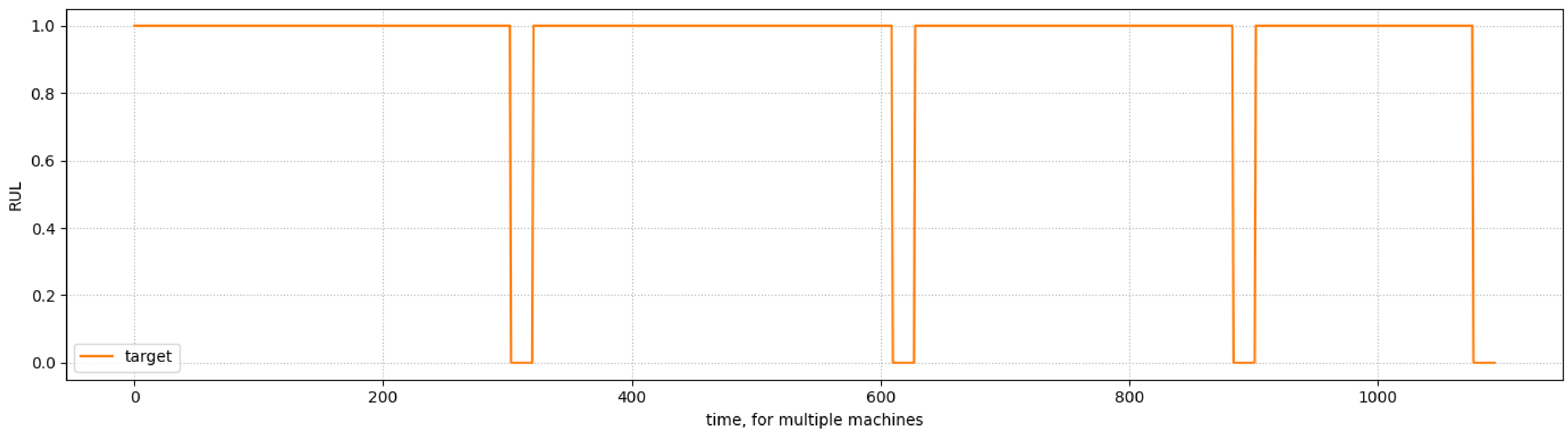


RUL Estimation, Again

For our RUL estimation problem, we tried two approaches

The second consisted in using a classifier to estimate this kind of function:

```
In [24]: stop = 1095
         util.plot_rul(target=(tr['rul'][:stop] >= 18), figsize=figsize, xlabel='time, for multiple machi
```



Limitations

Both approaches worked quite well on our dataset

...But there was a price to pay

Can you tell which one?

Limitations

Both approaches worked quite well on our dataset

...But there was a price to pay

Can you tell which one?

- We managed to obtain good maintenance policies
- ...But no well-grounded RUL estimate

There are a few reasons

- Failures are inherently stochastic
  - ...And we are treating them as deterministic phenomena
- We built our estimators without any underlying analysis
  - ...So the results are difficult to motivate and to interpret

Back to the Drawing Board

Here's what the correct approach should be:

- We start by defining a probabilistic model
- We use ML to approximate key components of such model
- We use the model + the approximators to make probabilistic predictions

This approach can be significantly more challenging

...But it comes with several benefits:

- You have both predictions and confidence
- You exploit a degree of domain knowledge
- You get a more interpretable model
- If you choose to ignore an element (e.g. because it is too difficult to model)
- ...At least you know that you have done so

(Conditional) Survival Analysis

We are interested in the "survival time" of an entity

We can start by modeling that as a single random variable T with unknown distribution

$$T \sim P(T) \qquad \text{(draft 1)}$$

- T (with support in $\mathbb{R}^+$) represents the survival time

To be specific, we want T to be remaining survival time

...With respect to time t when we perform the estimation. Formally:

$$T \sim P(T \mid t) \qquad \text{(draft 2)}$$

- Now the distribution is conditioned on t (which we can access)

(Conditional) Survival Analysis

Survival depends on additional factors

E.g. on how the lifestyle of a person, or on how industrial equipment is used

- We can model these factor as additional random variables
- We can distinguish between behavior in the past $X_{\leq t}$ and the future $X_{>t}$

Formally, we have:

$$T \sim P(T \mid X_{\leq t}, t, X_{>t}) \qquad \text{(draft 3)}$$

For now we focus on capturing the elements that affect the estimate

- We not not care (yet) about the fact that we can access them
- The idea is to focus on one problem at a time

(Conditional) Survival Analysis

...But of course whether a quantity can be accessed or not does matter

In particular, future behavior cannot be accessed at estimation time

- Intuitively, future behavior affects the estimate as noise
- Formally, we can average out its effect

This operation is called marginalization and leads to:

$$T \sim \mathbb{E}_{X_{>t}} \left[P(T \mid X_{\leq t}, t, X_{>t}) \right] \qquad \text{(draft 4)}$$

This is a good model for the distribution of the variable we wish to estimate

- The "sawtooth like" target that we used earlier for RUL regression
- ....Corresponds to samples from this distribution

In other words, we are saying our target was correct!

So, why did we get strange results in the RUL lecture?

Looking Back to Our Model

In the RUL lecture we trained a regressor

...With the current parameters/sensors as input nd an MSE loss

- Meaning the our estimator is making implicitly use of this model:

$$T \sim \mathcal{N}(\mu(X_t), \sigma)$$

- $\mathcal{N}$ denotes the Normal distribution, $\mu(\cdot)$ represents our old regressor

Looking Back to Our Model

In the RUL lecture we trained a regressor

...With the current parameters/sensors as input nd an MSE loss

- Meaning the our estimator is making implicitly use of this model:

$$T \sim \mathcal{N}(\mu(X_t), \sigma)$$

- $\mathcal{N}$ denotes the Normal distribution, $\mu(\cdot)$ represents our old regressor

Now, compare it with our "ideal" probabilistic model:

$$T \sim \mathbb{E}_{X_{>t}} \left[P(T \mid X_{\leq t}, t, X_{>t}) \right]$$

- Let's try to spot together any major difference

Implicit Assumptions

We made several implicit assumptions:

$$T \sim \mathcal{N}(\mu(X_t), \sigma) \quad \text{vs} \quad T \sim \mathbb{E}_{X_{>t}}\left[P(T \mid X_{\leq t}, t, X_{>t})\right]$$

Implicit Assumptions

We made several implicit assumptions:

$$T \sim \mathcal{N}(\mu(X_t), \sigma) \quad \text{vs} \quad T \sim \mathbb{E}_{X_{>t}}\left[P(T \mid X_{\leq t}, t, X_{>t})\right]$$

We considered a single X_t, rather than $X_{\leq t}$

- Actually, we tried that at least a bit (it helped, but not much)

Implicit Assumptions

We made several implicit assumptions:

$$T \sim \mathcal{N}(\mu(X_t), \sigma) \quad \text{vs} \quad T \sim \mathbb{E}_{X_{>t}}\left[P(T \mid X_{\leq t}, t, X_{>t})\right]$$

We considered a single X_t, rather than $X_{\leq t}$

- Actually, we tried that at least a bit (it helped, but not much)

We disregarded time as an input

- ...And thankfully this is easy to fix

Implicit Assumptions

We made several implicit assumptions:

$$T \sim \mathcal{N}(\mu(X_t), \sigma) \quad \text{vs} \quad T \sim \mathbb{E}_{X_{>t}}\left[P(T \mid X_{\leq t}, t, X_{>t})\right]$$

We considered a single X_t, rather than $X_{\leq t}$

- Actually, we tried that at least a bit (it helped, but not much)

We disregarded time as an input

- ...And thankfully this is easy to fix

We assumed a Normal distribution with fixed variance

- It's unclear how to relax the normality assumption
- ...But we know we can fix the variance this using a neuro-probabilistic model!

About Time

Let's fix one mistake by adding time as an input

In our dataset, time corresponds to the "cycle" field

In [25]:
```
# 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 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

# 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']
```

Estimated Variance

Then we can make our ML model capable of estimating variance

In particular, we can use a neuro-probabilistic ML model

- The underlying probabilistic model is:

$$T \sim \mathcal{N}(\mu(X_t, t), \sigma(X_t, t))$$

In practice:

- We use conventional ML model (a network) to estimate μ and σ
- ...Then we feed both parameters to a `DistributionLambda` layer

Our model will be able to learn how σ depends on the input

- This will be more challenging, but also more flexible
- ...And it will provide us confidence intervals

Building a Neuro-Probabilistic Model

Code to build the model can found in the `util` module

```
def build_nn_normal_model(input_shape, hidden, stddev_guess=1):
    model_in = keras.Input(shape=input_shape, dtype='float32')
    x = model_in
    for h in hidden:
        x = layers.Dense(h, activation='relu')(x)
    mu_logsigma = layers.Dense(2, activation='linear')(x)
    lf = lambda t: tfp.distributions.Normal(loc=t[:, :1], scale=tf.math.exp(t[:, 1:]))
    model_out = tfp.layers.DistributionLambda(lf)(mu_logsigma)
    model = keras.Model(model_in, model_out)
    return model
```

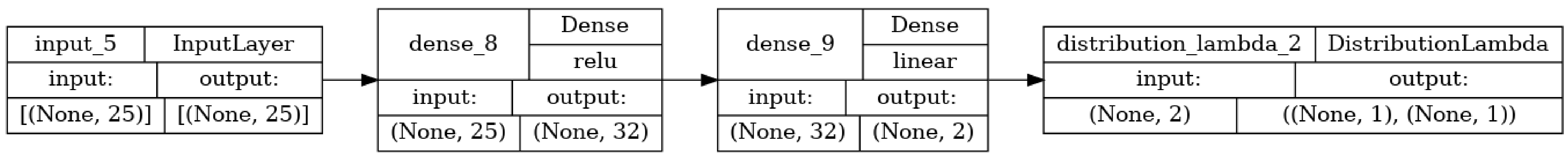
- Note the way the input tensor `t` is split in the `lambda` function
- That is needed to obtain the correct tensor shapes (columns)

Building a Neuro-Probabilistic Model

Let's build a simple neuro-probabilistic model

In [26]:
```
tr_rul_std = tr_s['rul'].std()
nnp = util.build_nn_normal_model(input_shape=(len(dt_in), ), hidden=[32], stddev_guess=tr_rul_st
util.plot_nn_model(nnp)
```

Out[26]:

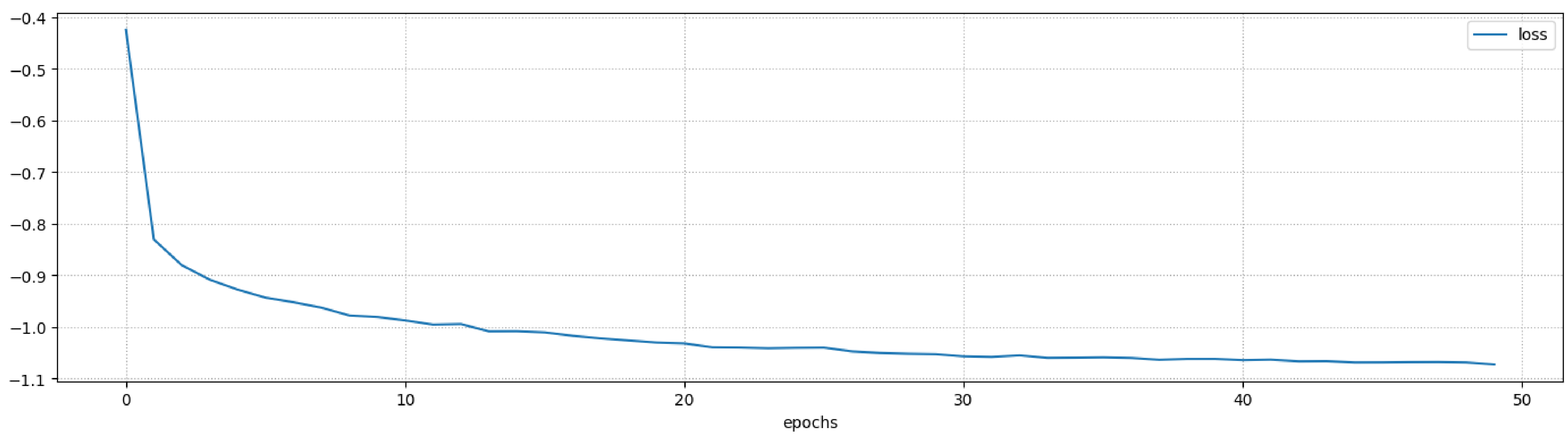


- There is a single hidden layer
- As a guess for σ, we provide the standard deviations over the training set

Training the Neuro-Probabilistic Model

We can train the model as in our previous example

In [27]:
```
negloglikelihood = lambda y_true, dist: -dist.log_prob(y_true)
nnp = util.build_nn_normal_model(input_shape=(len(dt_in), ), hidden=[32], stddev_guess=tr_rul_st
history = util.train_nn_model(nnp, tr_s[dt_in], tr_s['rul'], loss=negloglikelihood, epochs=50, v
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: -1.0728 (training)
```

Evaluation

We care about the estimated distributions (not about sampling)

...Therefore we call the model rather than using the `predict` method

```
In [28]: nn_pred_ts = nnp(tr_s[dt_in].values)
         nn_pred_ts

Out[28]: <tfp.distributions._TensorCoercible 'tensor_coercible' batch_shape=[45385, 1] event_shape=[] d
         type=float32>
```

From the distribution objects we can obtain means and standard deviations

```
In [29]: np_pred_ts_mean = nn_pred_ts.mean().numpy().ravel() * trmaxrul
         np_pred_ts_std = nn_pred_ts.stddev().numpy().ravel() * trmaxrul
```

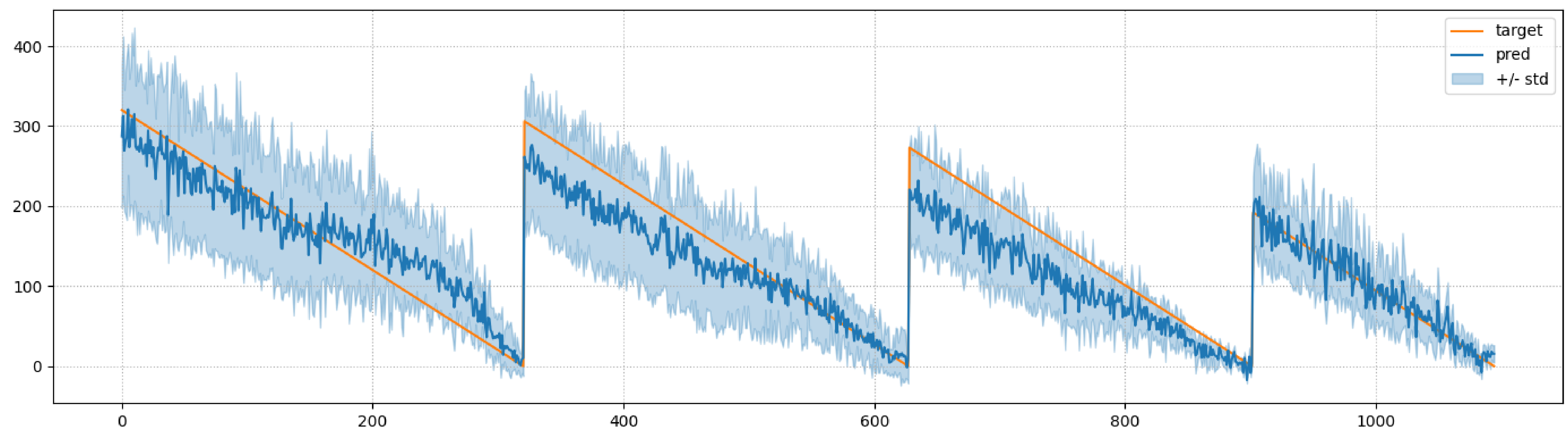
- For sake of keeping it short, we will just inspect the predictions
- ...Rather than making a full evaluation

That said, we could do it (and the results would be similar to the old ones)

Evaluation

Let's inspect the predictions on a portion of the test set

In [31]:
```
stop = 1095
util.plot_rul(target=tr_s['rul'].iloc[:stop]*trmaxrul, pred=np_pred_ts_mean[:stop],
              stddev=np_pred_ts_std[:stop], figsize=figsize)
```



- The initial plateaus in the predictions have disappeared
- ...And the true RUL is typically within 1σ from the predicted mean

Neuro-probabilistic Models vs Sample Weights

The approach we have seen works already very well

- We get a predicted mean (as usual)
- ...But also an input-dependent standard deviation

But can't we do the same with sample weights?

Neuro-probabilistic Models vs Sample Weights

The approach we have seen works already very well

- We get a predicted mean (as usual)
- ...But also an input-dependent standard deviation

But can't we do the same with sample weights?

Yes, but it's not the same

- Sample weights allow use to control the standard deviation with an MSE loss
- ...But we need to pre-compute them using another model (or assumption)

They cannot be learned in an end-to-end fashion!

Open Issues

But what if we are not confident about using a Normal?

We could build a histogram from the target values

- ...But that would not be a conditional distribution
- ...And what if it yields something strange (e.g. a multi-modal distribution)?

Open Issues

But what if we are not confident about using a Normal?

We could build a histogram from the target values

- ...But that would not be a conditional distribution
- ...And what if it yields something strange (e.g. a multi-modal distribution)?

And what if the RUL depends strongly on when defects arise?

- Then, in the early part of each run
- ...We might be accounting too much for what happened in the future

In practice, we risk overfitting (unless we have a lot of runs)

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 $F(t) = P(T \leq t)$

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

...And if we do it with S, we only care about 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

$$S(t, X_{\leq t}) = (1 - \lambda(t, X_t))(1 - \lambda(t-1, X_{t-1}))\ldots$$

Where λ is called hazard function

The hazard function is a conditional probability

...That of not making one more step, at time t. 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.:

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

Our Plan

We will attempt to train an estimator $\lambda(t, X_t, \theta)$ 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, it is not immediate to use λ to obtain a RUL estimate

...But we can use it to approximate the chance of surviving n steps from now

- In practice, we can approximately compute the conditional survival:

$$S(t+n)/S(t) = P(T > t+n \mid T \geq t-1)$$

- For many practical applications, this is enough

Training a Hazard Estimator

We still need to define how to train our λ estimator

...But at this point, we know enough to model 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:

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

This is the probability of:

- Surviving all time steps from 1 to $e_k - 1$
- Not surviving at time e_k
- $\hat{x}_{kt}$ is the available input data for experiment k at time t

Training a Hazard Estimator

We can now formulate a likelihood maximization problem

Assuming we have m experiments, we get:

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

- Let $\hat{d}_{kt} = 1$ iff $t = e_k$, i.e. if the experiment ends at time k
- ...And let $\hat{d}_{kt} = 0$ otherwise. Then we can rewrite the problem as:

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

Training a Hazard Estimator

Finally, with a log transformation and a sign switch we get:

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

This is a (binary) crossentropy minimization problem!

- We just need to consider all samples in our dataset individually
- Then attach to them a class given by $\hat{d}_{kt}$
- ...And finally we can train a classifier

This is almost precisely what we did in our classification approach

- But now we know exactly how to define the classes

Classes and Models

Let's start by defining the classes

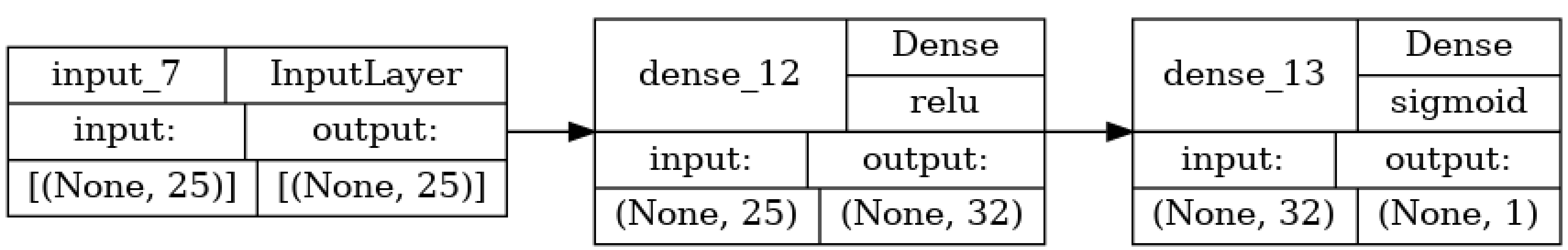
We check when the RUL is 0 (this the same as $t = e_k$)

In [32]:
```
tr_lbl = (tr['rul'] == 0)
ts_lbl = (ts['rul'] == 0)
```

Then we can build a (usual) classification model:

In [33]:
```
nnl = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32], output_activa
util.plot_nn_model(nnl)
```

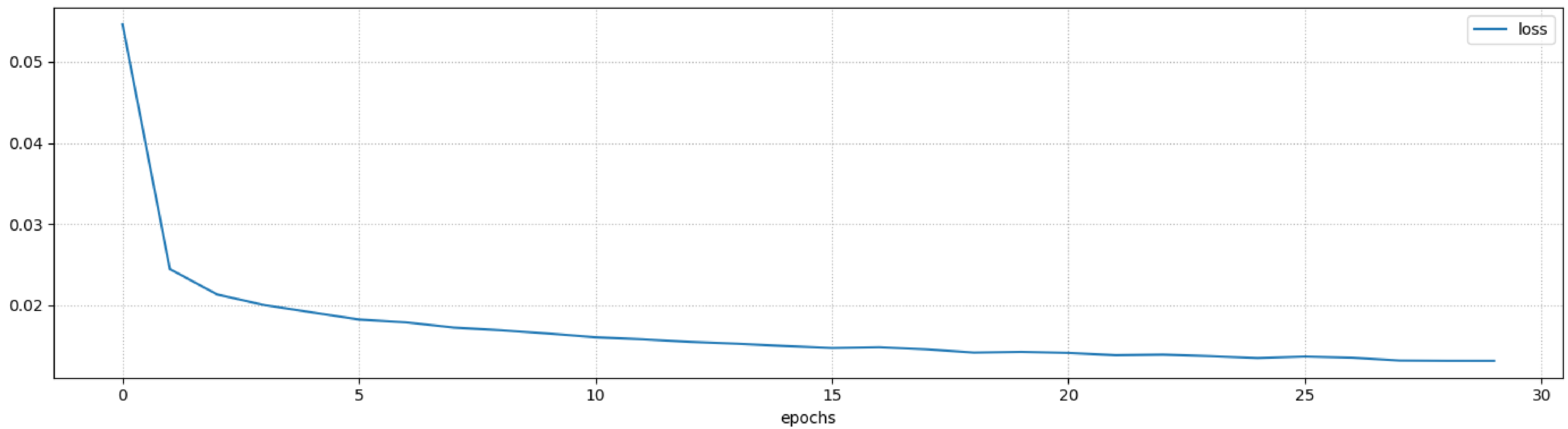
Out[33]:



Training the Hazard Estimator

We train the hazard estimator as any other classifier

In [34]:
```
nnl = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32], output_activa
history = util.train_nn_model(nnl, tr_s[dt_in], tr_lbl, loss='binary_crossentropy', epochs=30,
        verbose=0, patience=10, batch_size=32, validation_split=0.0)
util.plot_training_history(history, figsize=figsize)
```



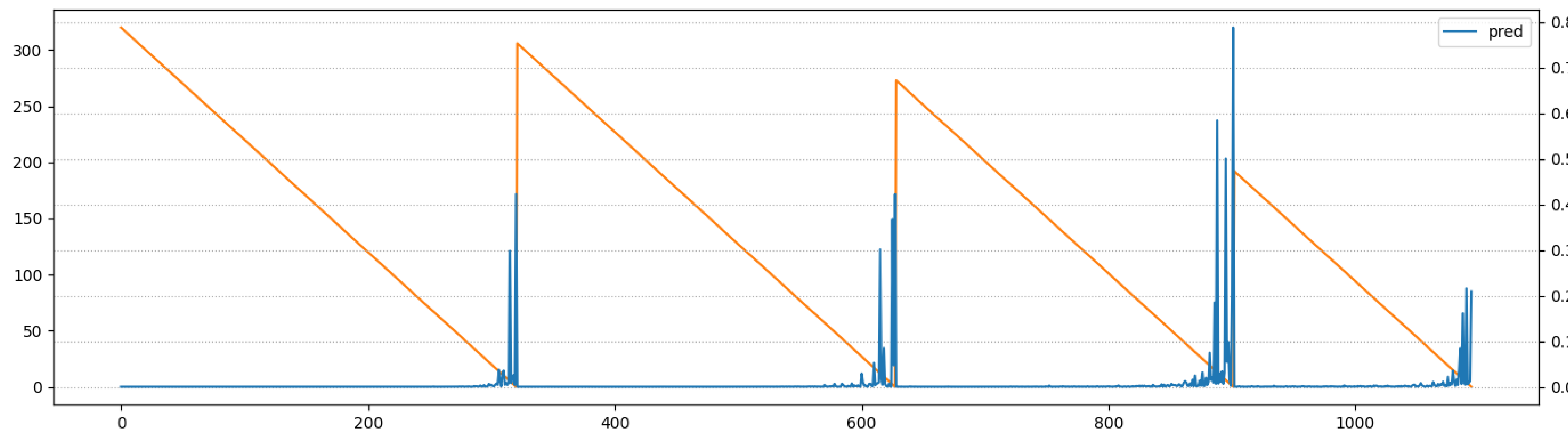

```
Final loss: 0.0131 (training)
```

Inspecting Hazards

We will start our evaluation by inspecting the hazard values

First for (part of) the training set:

In [35]:
```
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, figsize=figsize)
```

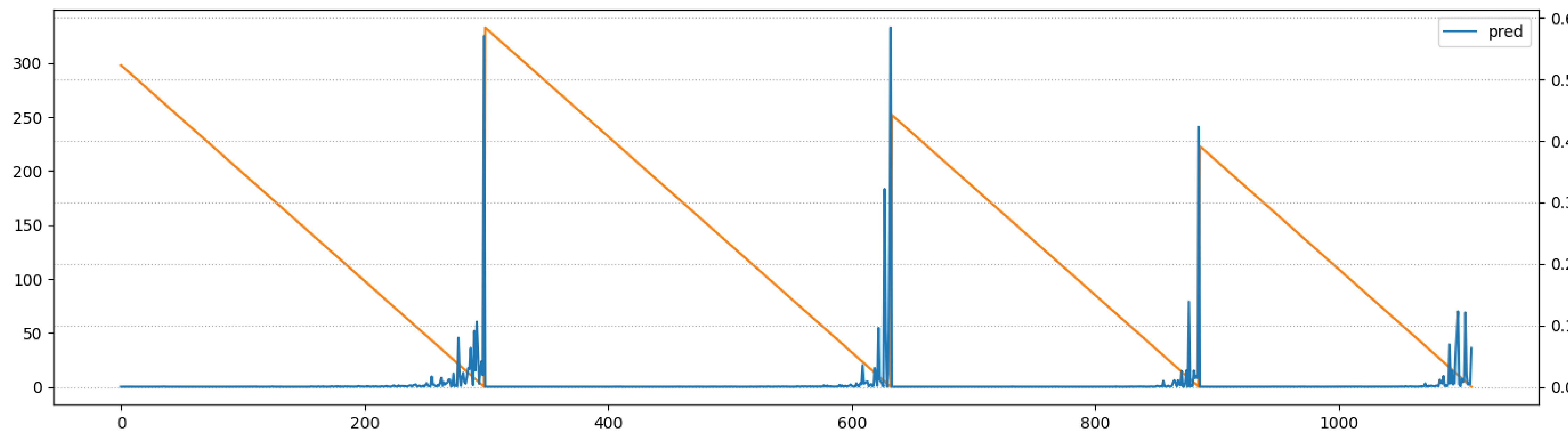


Inspecting Hazards

We will start our evaluation by inspecting the hazard values

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

```
In [36]: 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, figsize=figsize)
```



Beyond Simple Hazards

These are hazard values, i.e. they refer to single steps

- They could be used directly to formulate a maintenance policy
- To achieve that, we would just need to optimize a threshold

...But it is more interesting to stick to our plan:

We wish to compute the probability of being still up in n steps

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

By using our estimator (for a run k) we get:

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

Beyond Simple Hazards

The formula requires access to future values of the X_t variable

We cannot access those in real life, so we'll use an approximation:

$$S(t+n)/S(t) \simeq \prod_{h=0}^{n} (1 - \lambda(t+h, \hat{x}_{kt}, \theta))$$

- We keep the sample values $\hat{x}_{kt}$ fixed (e.g. parameters & sensors)
- ...And we just change the value of time

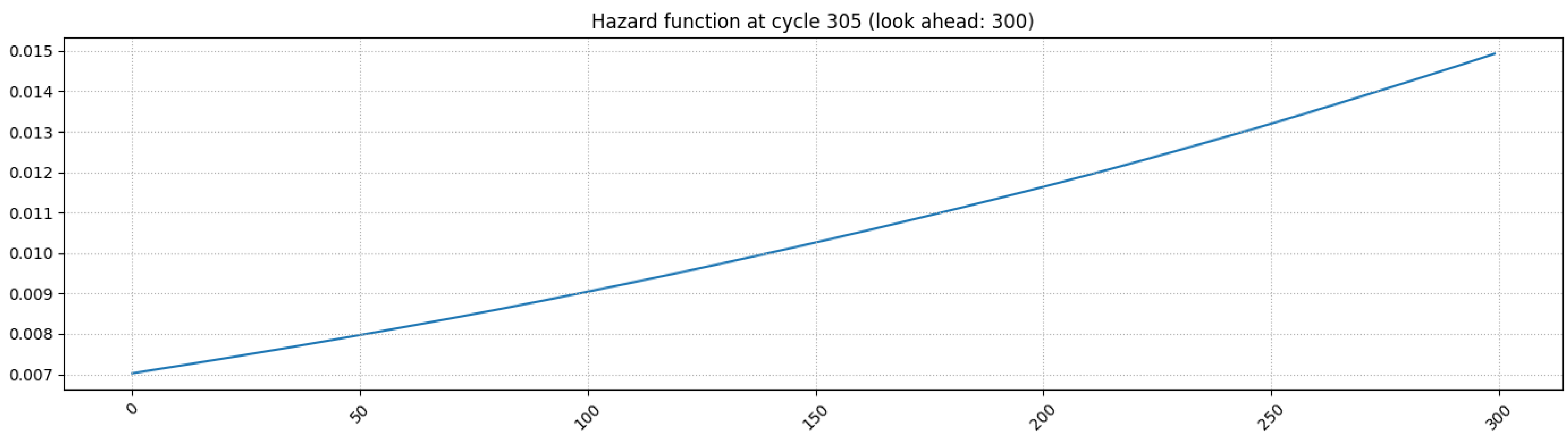
This trick has obvious limitations, however:

- On a short horizon (small n), the error is typically limited
- It allows us to investigate the impact of time on the hazard λ

Approximate Future Hazard

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

In [37]:
```
ref_sample = tr_s.iloc[304]
look_ahead = 300
hazard = util.predict_cf(nnl, ref_sample[dt_in], columns='cycle',
                         values=ref_sample['cycle'] + np.arange(look_ahead)/trmaxrul)
util.plot_series(hazard, figsize=figsize, title=f'Hazard function at cycle {ref_sample["cycle"]*
```

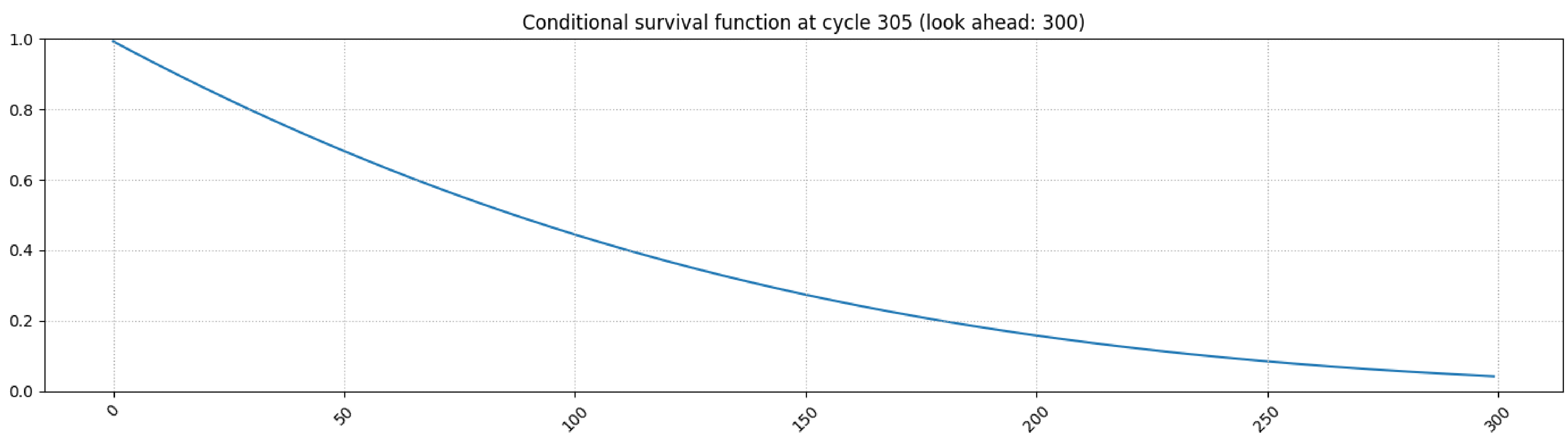


- The model has learned that time has a super-linear effect on λ

Approximate Conditional Survival

We can use the approximate hazard to estimate conditional survival

In [38]:
```
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["cycle"]*trmaxrul:.(
```



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

Approximate Conditional Survival

At deployment time, we could continuously compute conditional survival

...Over a fixed look ahead window (e.g. 30 steps)

- Let's do it for a single experiment from our dataset:

In [39]:
```
ref_run = tr_s[tr_s['machine'] == tr_s.iloc[0]['machine']]
look_up_window = np.arange(30)/trmaxrul
rolling_survival = util.rolling_survival_cmapss(hazard_model=nnl, data=ref_run[dt_in], look_up_w
rolling_survival.columns = [f'S(t+{h})/S(t)' for h in range(30)]
rolling_survival.head()
```

Out[39]:

	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(t)	S(t+7)/S(t)	S(t+8)/S(t)	S(t+9)/S(t)	...	S(t+20)/S(t)	S(
0	0.999998	0.999997	0.999995	0.999994	0.999992	0.999991	0.999989	0.999988	0.999986	0.999984	...	0.999967	0.
1	0.999998	0.999995	0.999993	0.999990	0.999988	0.999986	0.999983	0.999981	0.999979	0.999976	...	0.999949	0.
2	0.999998	0.999997	0.999995	0.999993	0.999992	0.999990	0.999988	0.999987	0.999985	0.999983	...	0.999964	0.
3	0.999998	0.999997	0.999995	0.999993	0.999992	0.999990	0.999988	0.999987	0.999985	0.999983	...	0.999964	0.
4	0.999971	0.999941	0.999911	0.999882	0.999852	0.999822	0.999792	0.999762	0.999732	0.999702	...	0.999365	0.

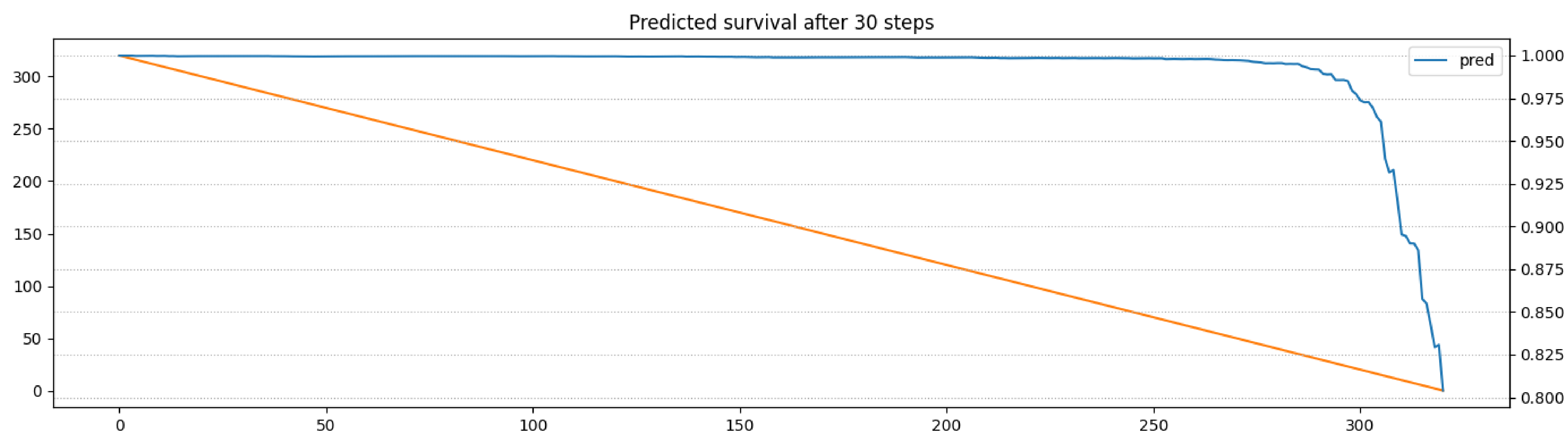
5 rows × 30 columns

- Each column contains the conditional survival h steps away

Approximate Conditional Survival

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

```
In [40]: rolling_survival_last = rolling_survival[rolling_survival.columns[-1]].ewm(30).mean()
         util.plot_rul(pred=rolling_survival_last[:stop], target=ref_run['rul']*trmaxrul, same_scale=Fals
                       figsize=figsize, title='Predicted survival after 30 steps')
```



- Remember that this is a stochastic phenomenon
- So, even a chance of 90% is quite dangerous to take!

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

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!