

RUL Prediction as Classification



RUL Prediction as Classification

RUL-based maintenance can also be tackled **using a classifier**

- We build a classifier to determine whether a failure will occur in θ steps
- We stop as soon as the classifier outputs (say) a 0, i.e.

$$f_{\theta}(x, \lambda) = 0$$

- f is the classifier, with parameter vector λ
- θ is the horizon for detecting a failure

In a sense, we are trying to learn **directly a maintenance policy**

- The policy is the form "stop θ units before a failure"
- The classifier tries to learn it

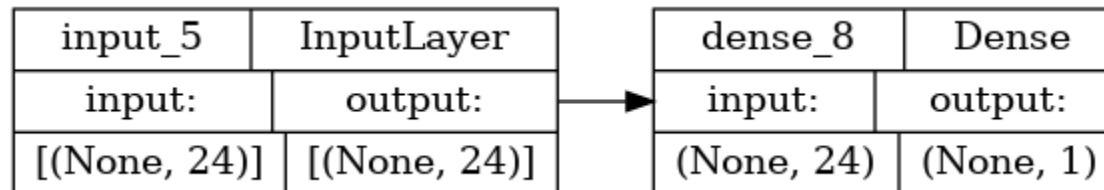


Classifier Architecture

We can therefore immediately define our classifier architecture:

```
In [13]: nn1 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[], output_activation='sigmoid')
util.plot_nn_model(nn1)
```

Out[13]:



- Like in the regression case, we use a Multilayer Perceptron
- The only difference is the use of a sigmoid activation in the output layer
- For `hidden = []` we get Logistic Regression
- ...Which of course is going to be our first model



Training

Before training, we need to define the classes

In turn, this requires to define the detection horizon θ :

```
In [14]: class_thr = 18  
tr_lbl = (tr['rul'] >= class_thr)  
ts_lbl = (ts['rul'] >= class_thr)
```

- The class is "1" if a failure is more than θ steps away
- The class is "0" otherwise

Classification problems **tend to be easier** than regression problems

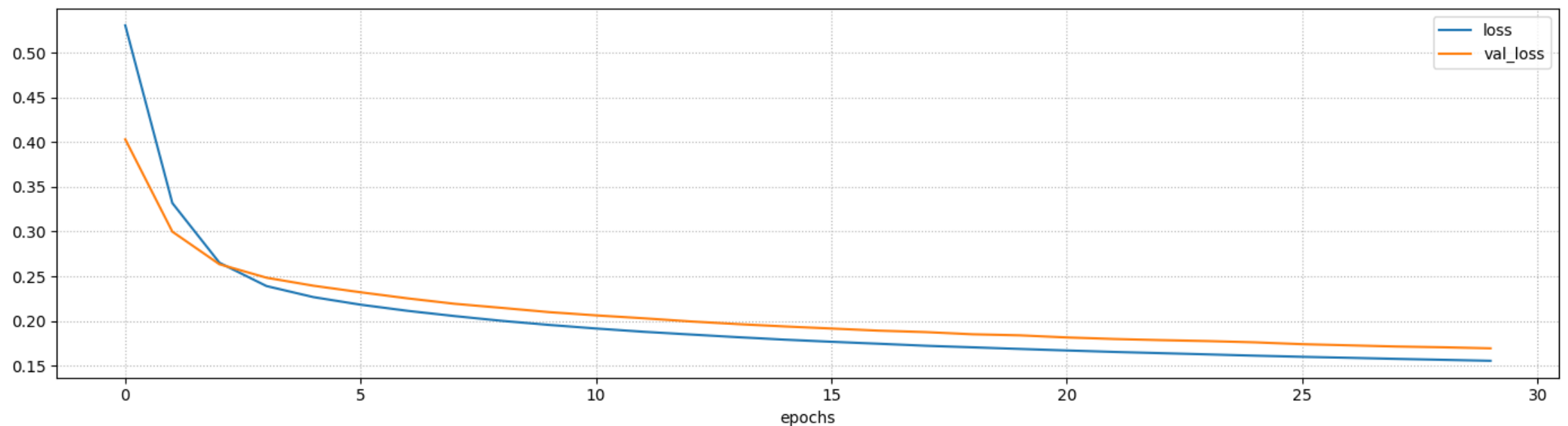
- On the other hand, learning the whole policy
- ...May be trickier than just estimating the RUL



Training

Let's start by training the simplest possible model

```
In [15]: nn1 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[], output_activation='sigmoid')
history = util.train_nn_model(nn1, tr_s[dt_in], tr_lbl, loss='binary_crossentropy', epochs=30,
                             verbose=0, patience=10, batch_size=32, validation_split=0.2)
util.plot_training_history(history, figsize=figsize)
```



Final loss: 0.1555 (training), 0.1695 (validation)

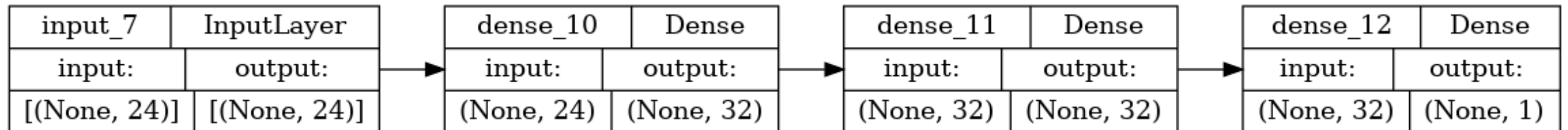


Training

Then let's try with a deeper model

```
In [16]: nn2 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32, 32], output_activation='sigmoid')
util.plot_nn_model(nn2)
```

Out[16]:



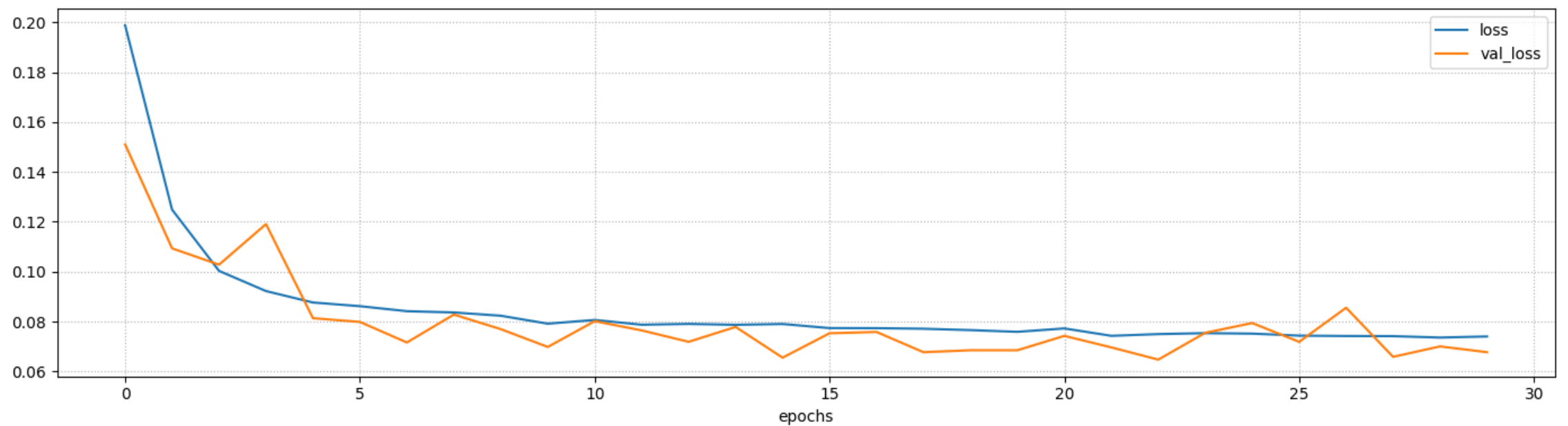
- Now we have two hidden layers
- ...Each with 32 neurons



Training

Let's train it and check the results

```
In [17]: nn2 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32, 32], output_activation='sigmoid')
history = util.train_nn_model(nn2, tr_s[dt_in], tr_lbl, loss='binary_crossentropy', epochs=30,
                             verbose=0, patience=10, batch_size=32, validation_split=0.2)
util.plot_training_history(history, figsize=figsize)
```



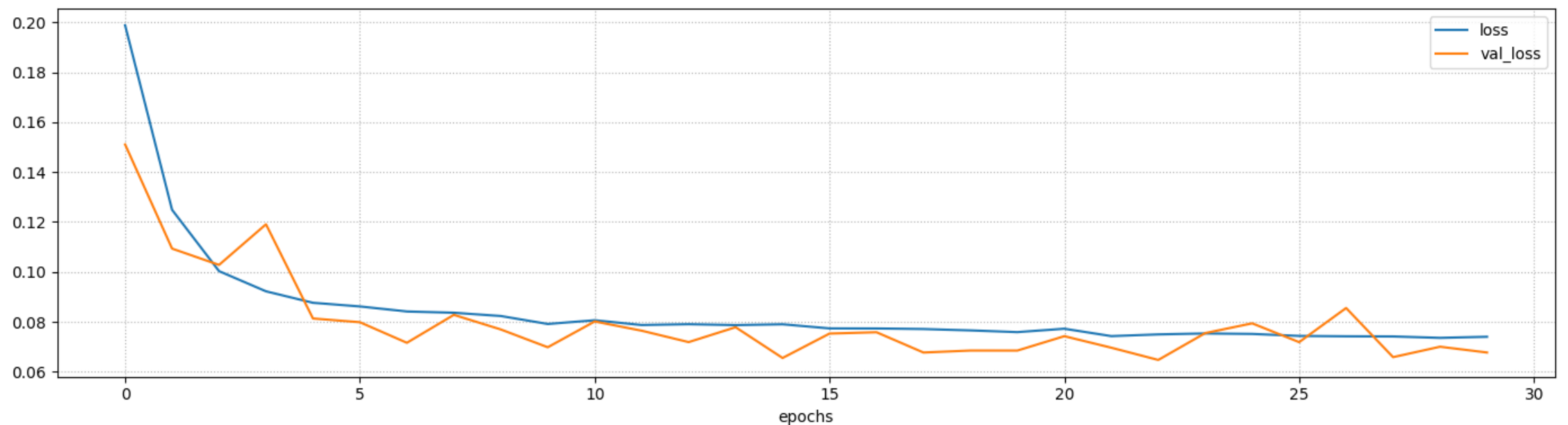
Final loss: 0.0740 (training), 0.0677 (validation)



Training

Let's train it and check the results

```
In [17]: nn2 = util.build_nn_model(input_shape=(len(dt_in), ), output_shape=1, hidden=[32, 32], output_activation='sigmoid')
history = util.train_nn_model(nn2, tr_s[dt_in], tr_lbl, loss='binary_crossentropy', epochs=30,
                             verbose=0, patience=10, batch_size=32, validation_split=0.2)
util.plot_training_history(history, figsize=figsize)
```



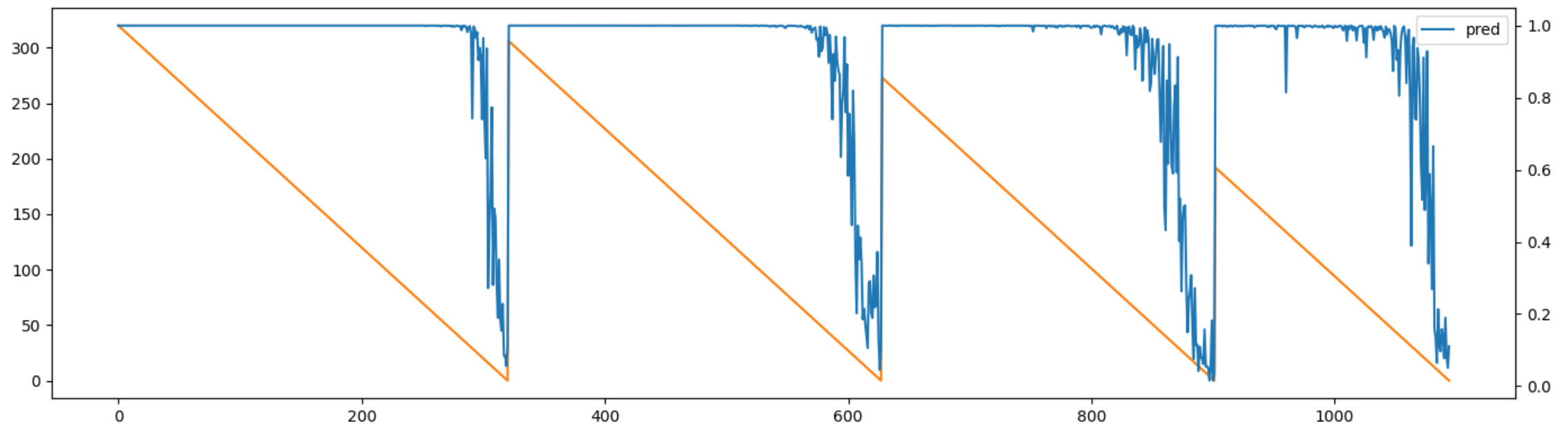
Final loss: 0.0740 (training), 0.0677 (validation)

 There is a significant improvement over Logistic Regression

Predictions

The model prediction can be interpreted as a probabilities of **not** stopping

```
In [18]: tr_pred2_prob = nn2.predict(tr_s[dt_in], verbose=0).ravel()  
stop = 1095  
util.plot_rul(tr_pred2_prob[:stop], tr['rul'][:stop], same_scale=False, figsize=figsize)
```



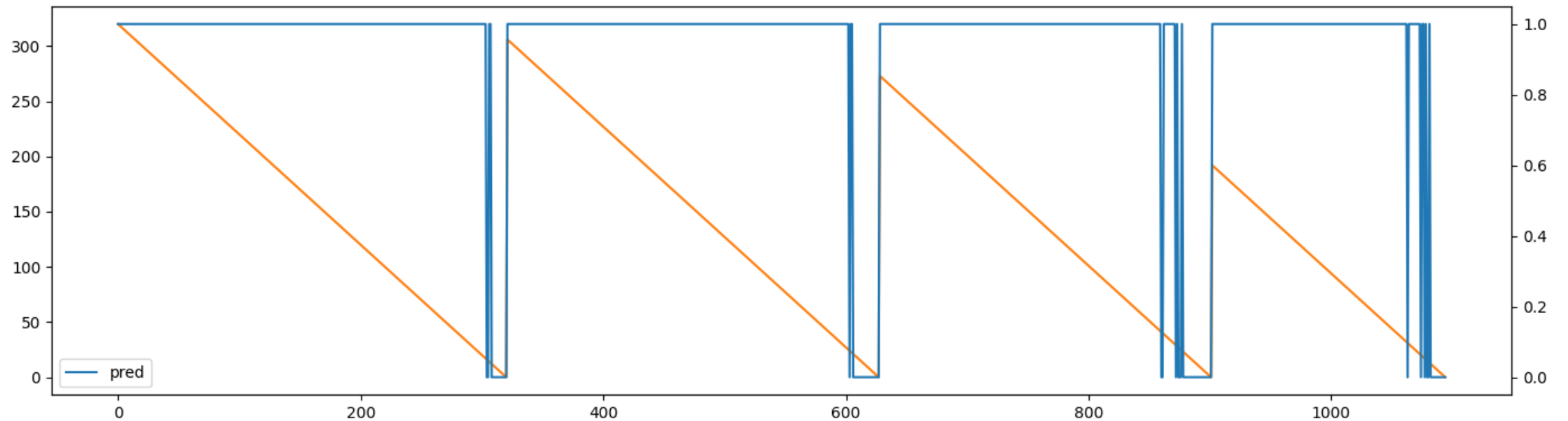
- The probability falls when closer to failures



Predictions

In practice, we'll need to convert the predictions into integers via rounding
...Unless we want to deal with one more threshold (in addition to θ)

```
In [19]: tr_pred2 = np.round(nn2.predict(tr_s[dt_in], verbose=0).ravel())  
util.plot_rul(tr_pred2[:stop], tr['rul'][:stop], same_scale=False, figsize=figsize)
```



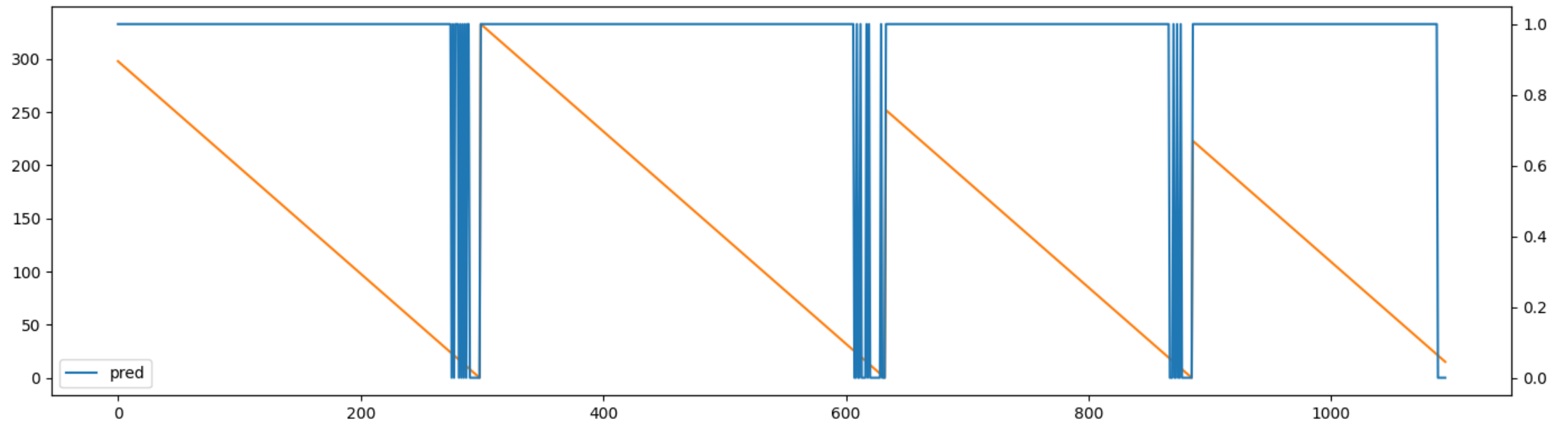
■ Still, the behavior seems to be reasonable



Predictions

Let's see the behavior on the test set

```
In [20]: ts_pred2 = np.round(nn2.predict(ts_s[dt_in], verbose=0).ravel())  
util.plot_rul(ts_pred2[:stop], ts['rul'][:stop], same_scale=False, figsize=figsize)
```



■ Apparently a decent degree of generalization



Evaluation

We can evaluate the classifier directly

...Because it defines the whole policy, with no need for additional calibration!

- On one hand this makes this stage of the process simpler
- ...On the other, this is (apparently) a missed opportunity

```
In [24]: tr_c2, tr_f2, tr_s2 = cmodel.cost(tr['machine'].values, tr_pred2, 0.5, return_margin=True)
         ts_c2, ts_f2, ts_s2 = cmodel.cost(ts['machine'].values, ts_pred2, 0.5, return_margin=True)
         print(f'Cost: {tr_c2/len(tr_mcn):.2f} (training), {ts_c2/len(ts_mcn):.2f} (test)')
         print(f'Avg. fails: {tr_f2/len(tr_mcn):.2f} (training), {ts_f2/len(ts_mcn):.2f} (test)')
         print(f'Avg. slack: {tr_s2/len(tr_mcn):.2f} (training), {ts_s2/len(ts_mcn):.2f} (test)')
```

```
Cost: -89.51 (training), -99.38 (test)
Avg. fails: 0.00 (training), 0.00 (test)
Avg. slack: 26.89 (training), 24.56 (test)
```

- Still pretty good results, but worse than the best regression approach



Why do you think this is the case?



Why do you think this is the case?

There are a few reasons, we will explore one



Uncalibrated Threshold

In the example from this notebook, we are defining the classes using:

```
class_thr = 18
tr_lbl = (tr['rul'] >= class_thr)
ts_lbl = (ts['rul'] >= class_thr)
```

- Like in the regression case, we are using a threshold θ
- ...But here θ is employed for defining the classes

This approach has both PROs and CONs

- PRO: we can (ideally) choose how close the failure we should stop
- CON: early signs of failure might not be evident in the chosen interval
- CON: we **did not calibrate** θ

The last point should be elaborated a bit more



Taking a Step Back

In the **regression** case, we are formally solving:

$$\begin{aligned} \operatorname{argmin}_{\theta} \sum_{k \in K} \operatorname{cost}(f(\hat{x}_k, \omega^*), \theta) \\ \text{s.t.: } \omega^* = \operatorname{argmin}_{\omega} L(f(\hat{x}_k, \omega), \hat{y}_k) \end{aligned}$$

- Where ω^* is the optimal parameter vector (i.e. the network weights)
- L is the loss function (i.e. the MSE), and **cost** is our cost model
- The threshold θ is chosen so as to minimize the cos

This is a **bilevel optimization** problem

- However, since θ appears neither in L nor in f

■ ...It can be **decomposed into two sequential subproblems**

Taking a Step Back

In the **classification** case, we are formally solving:

$$\begin{aligned} \operatorname{argmin}_{\theta} \sum_{k \in K} \operatorname{cost}(f(\hat{x}_k, \omega^*), 1/2) \\ \text{s.t.: } \omega^* = \operatorname{argmin}_{\omega} L(f(\hat{x}_k, \omega), \mathbb{1}_{y_k \geq \theta}) \end{aligned}$$

- We use a canonical threshold in the cost model (i.e. 0.5)
- L is again the loss function (binary cross entropy)
- $\mathbb{1}_{y_k \geq \theta}$ is the indicator function of $y_k \geq \theta$ (i.e. our class labels)

Unlike the previous one, this problem cannot be decomposed

...Because θ appears in the loss function!

- This means we need to **optimize θ and ω at the same time**

Black Box Optimization

Let's sketch a possible optimization approach

1. We search over the possible values of θ
2. For the given θ value, we compute $\mathbb{1}_{y_k \geq \theta}$ (i.e. the class labels)
3. We train the model to compute ω^*
4. Then we compute the cost
5. ...And finally we repeat, for the next value of θ

At the end of the process, we choose the configuration with the best cost

In principle we could use grid search again, but...

- Evaluating the cost is **slow**, since it requires retraining
- The search space is **grows exponentially** with the number of parameters

We need a better optimization method!

