## 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 $\theta$ steps
- We stop as soon as the classifier outputs (say) a 0, i.e.

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

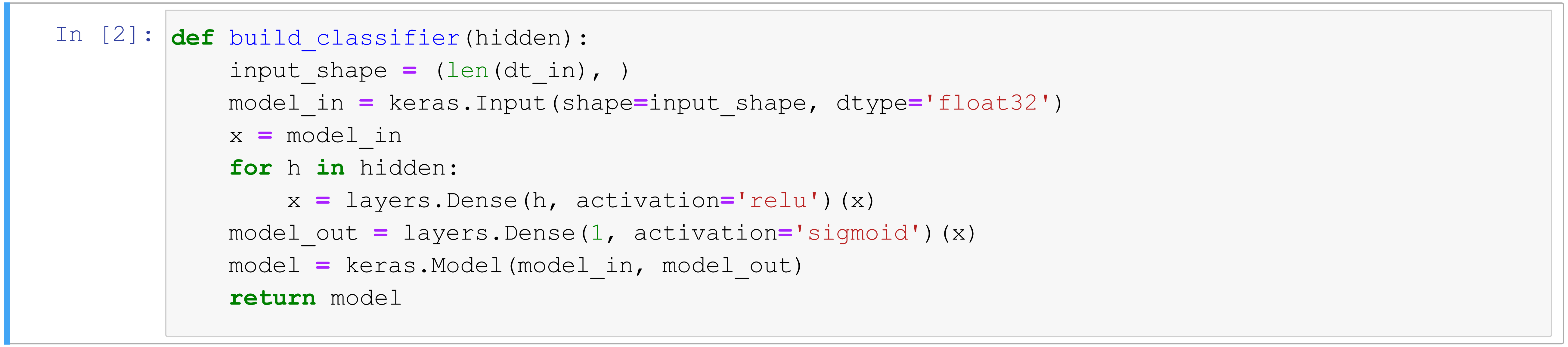
- $f$ is the classifier, with parameter vector $\lambda$
- $\theta$ is the horizon for detecting a failure

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

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

## Classifier Architecture

**We can therefore immediately define our classifier architecture:**



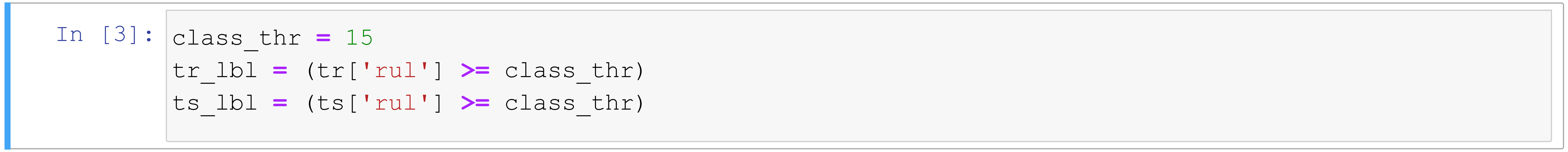
```
In [2]: def build_classifier(hidden):
            input_shape = (len(dt_in), )
            model_in = keras.Input(shape=input_shape, dtype='float32')
            x = model_in
            for h in hidden:
                x = layers.Dense(h, activation='relu')(x)
            model_out = layers.Dense(1, activation='sigmoid')(x)
            model = keras.Model(model_in, model_out)
            return model
```

- 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

## Training

**Before training, we need to define the classes**

In turn, this requires to define the detection horizon $\theta$:



```
In [3]: class_thr = 15
        tr_lbl = (tr['rul'] >= class_thr)
        ts_lbl = (ts['rul'] >= class_thr)
```

- The class is "1" if a failure is more than $\theta$ steps away
- The class if "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 [4]: nn1 = build_classifier(hidden=[])
        nn1.compile(optimizer='Adam', loss='binary_crossentropy')
        cb = [callbacks.EarlyStopping(patience=10, restore_best_weights=True)]
        history1 = nn1.fit(tr_s[dt_in], tr_lbl, validation_split=0.2,
                         callbacks=cb,
                         batch_size=32, epochs=20, verbose=1)
```

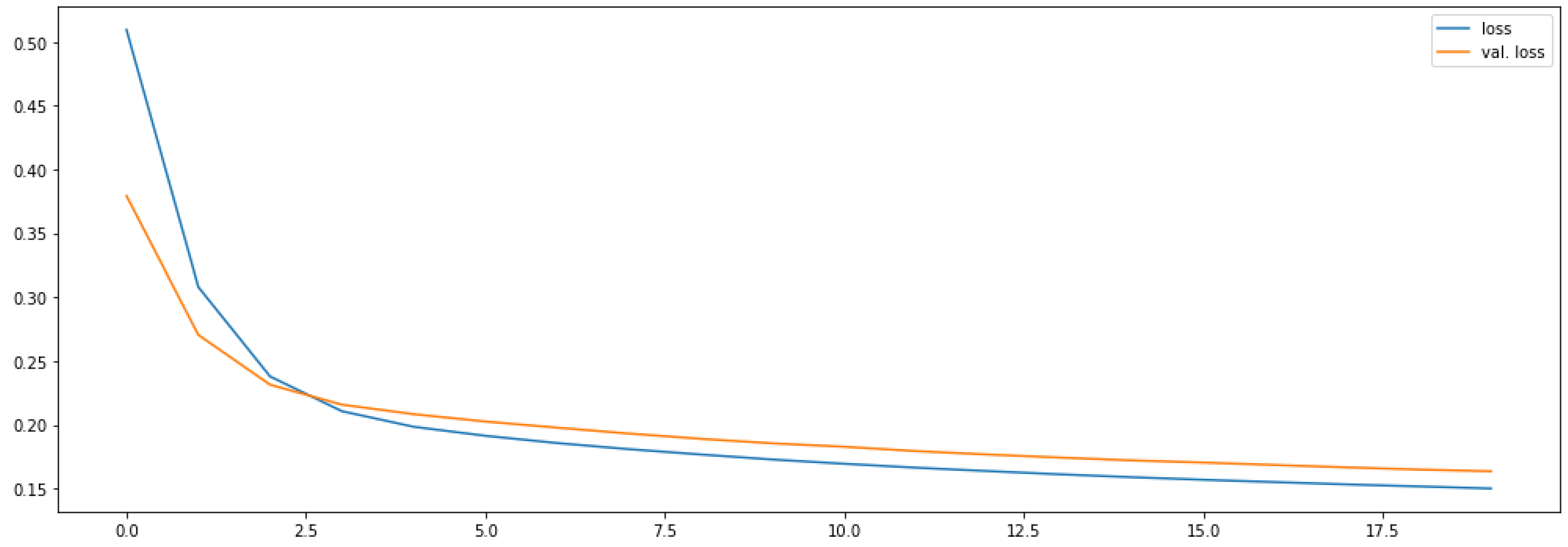
```
Epoch 1/20
1135/1135 [==============================] - 1s 636us/step - loss: 0.5097 - val_loss: 0.3793
Epoch 2/20
1135/1135 [==============================] - 1s 552us/step - loss: 0.3081 - val_loss: 0.2705
Epoch 3/20
1135/1135 [==============================] - 1s 549us/step - loss: 0.2378 - val_loss: 0.2313
Epoch 4/20
1135/1135 [==============================] - 1s 556us/step - loss: 0.2106 - val_loss: 0.2157
Epoch 5/20
1135/1135 [==============================] - 1s 551us/step - loss: 0.1983 - val_loss: 0.2082
Epoch 6/20
1135/1135 [==============================] - 1s 554us/step - loss: 0.1913 - val_loss: 0.2025
Epoch 7/20
1135/1135 [==============================] - 1s 556us/step - loss: 0.1856 - val_loss: 0.1976
Epoch 8/20
1135/1135 [==============================] - 1s 552us/step - loss: 0.1808 - val_loss: 0.1930
Epoch 9/20
1135/1135 [==============================] - 1s 550us/step - loss: 0.1766 - val_loss: 0.1889
Epoch 10/20
```

## Training

### Here's the loss evolution over time and its final value

```
In [7]: cmapss.plot_training_history(history1, figsize=figsize)
        tr1, vl1 = history1.history["loss"][-1], np.min(history1.history["val_loss"])
        print(f'Final loss: {tr1:.4f} (training), {vl1:.4f} (validation)')
```

```
Final loss: 0.1499 (training), 0.1634 (validation)
```



## Training

### Let's try with a deeper model

```
In [8]: nn2 = build_classifier(hidden=[32, 32])
        nn2.compile(optimizer='Adam', loss='binary_crossentropy')
        history2 = nn2.fit(tr_s[dt_in], tr_lbl, validation_split=0.2,
                         callbacks=cb,
                         batch_size=32, epochs=20, verbose=1)
```

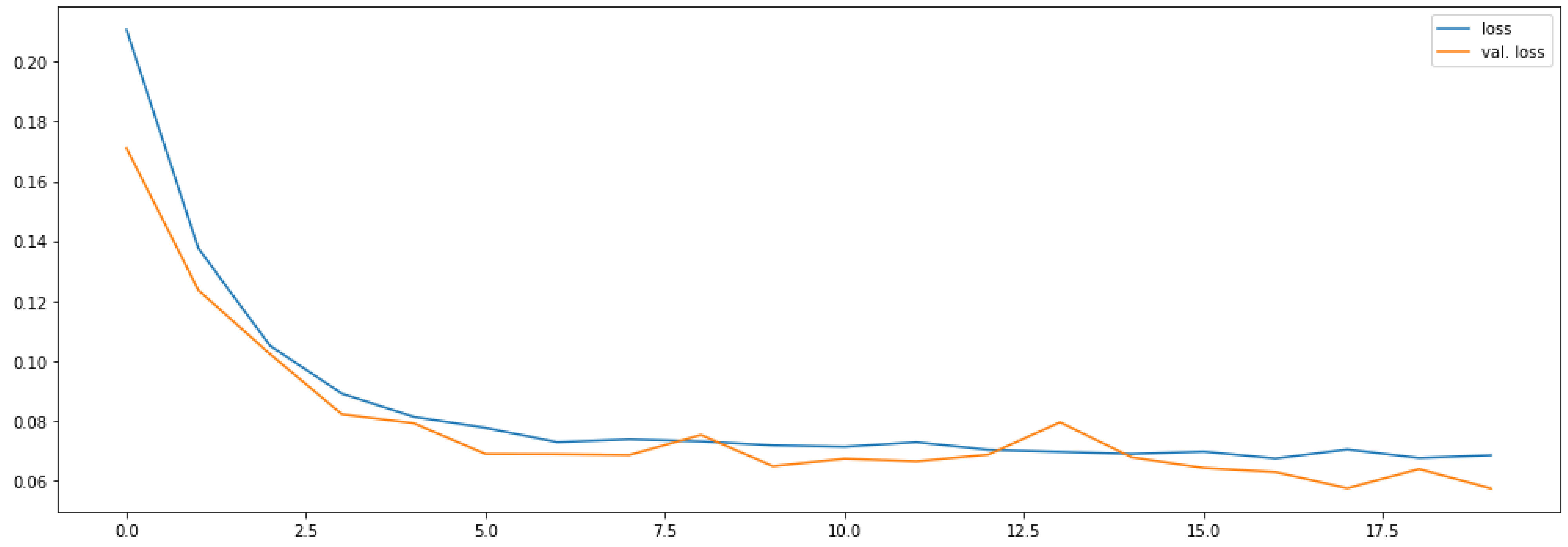
```
Epoch 1/20
1135/1135 [==============================] - 1s 733us/step - loss: 0.2107 - val_loss: 0.1711
Epoch 2/20
1135/1135 [==============================] - 1s 683us/step - loss: 0.1376 - val_loss: 0.1237
Epoch 3/20
1135/1135 [==============================] - 1s 691us/step - loss: 0.1050 - val_loss: 0.1022
Epoch 4/20
1135/1135 [==============================] - 1s 658us/step - loss: 0.0890 - val_loss: 0.0821
Epoch 5/20
1135/1135 [==============================] - 1s 654us/step - loss: 0.0813 - val_loss: 0.0791
Epoch 6/20
1135/1135 [==============================] - 1s 658us/step - loss: 0.0776 - val_loss: 0.0688
Epoch 7/20
1135/1135 [==============================] - 1s 656us/step - loss: 0.0728 - val_loss: 0.0687
Epoch 8/20
1135/1135 [==============================] - 1s 660us/step - loss: 0.0738 - val_loss: 0.0685
Epoch 9/20
1135/1135 [==============================] - 1s 739us/step - loss: 0.0731 - val_loss: 0.0752
Epoch 10/20
1135/1135 [==============================] - 1s 683us/step - loss: 0.0717 - val_loss: 0.0647
Epoch 11/20
```

## Training

### Here's the loss for the deeper model

```
In [9]: cmapss.plot_training_history(history2, figsize=figsize)
        tr2, vl2 = history2.history["loss"][-1], np.min(history2.history["val_loss"])
        print(f'Final loss: {tr2:.4f} (training), {vl2:.4f} (validation)')
```

```
Final loss: 0.0684 (training), 0.0573 (validation)
```

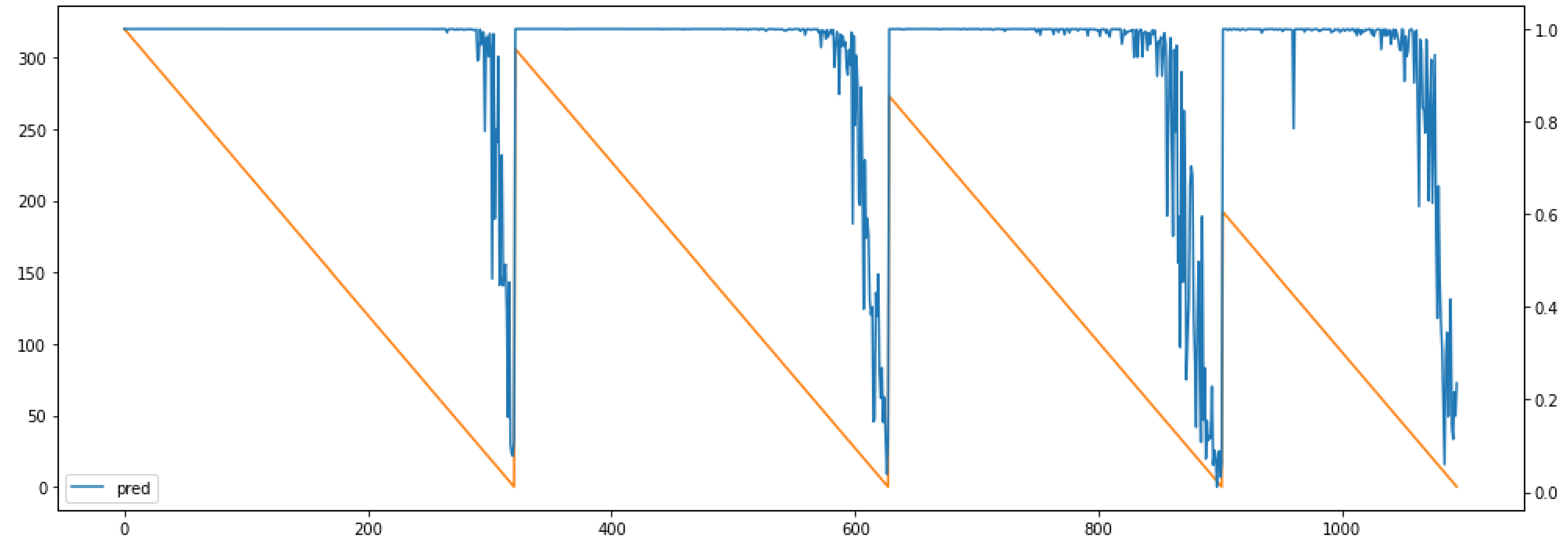


- Depth pays off big in this case

## Predictions

### Our predictions can be interpreted as probabilities (of advancing)

```
In [10]: tr_pred2_prob = nn2.predict(tr_s[dt_in]).ravel()
         stop = 1095
         cmapss.plot_rul(tr_pred2_prob[:stop], tr['rul'][:stop], same_scale=False, figsize=figsize)
```



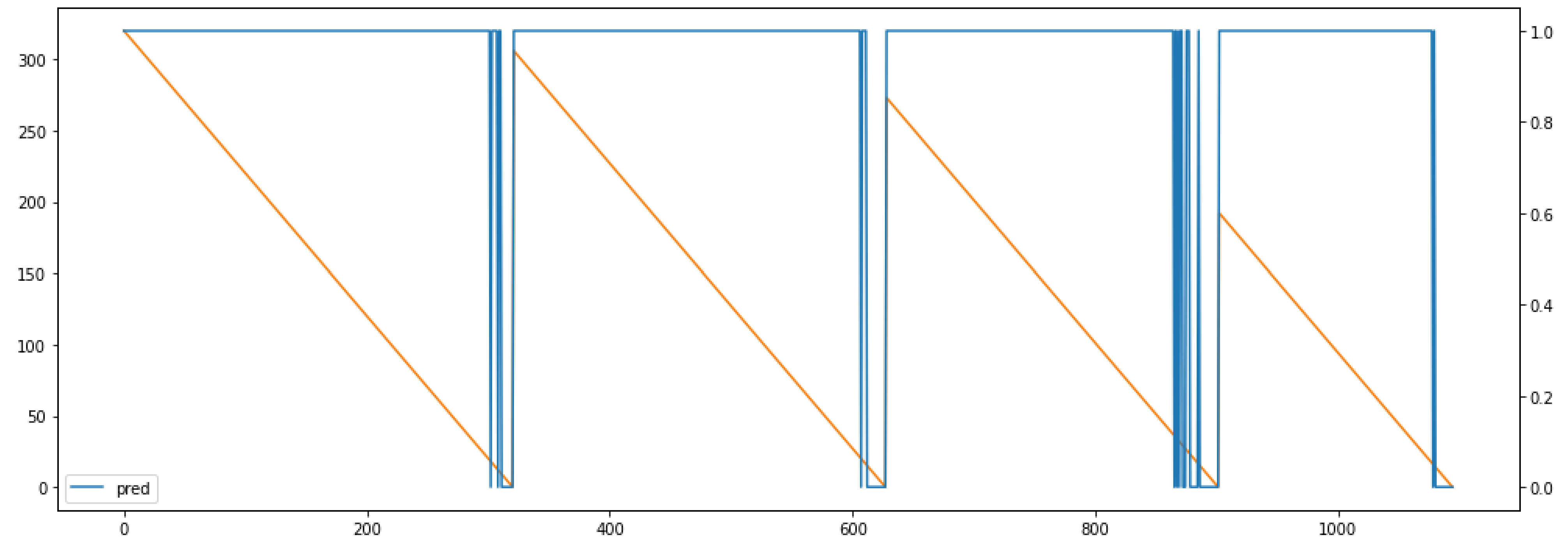
- The probability falls when closer to failures

## Predictions

**In practice, we need to convert the predictions to integers via rounding**

...Unless we want to deal with one more threshold (in addition to $\theta$)

```
In [11]: tr_pred2 = np.round(nn2.predict(tr_s[dt_in]).ravel())
         cmapss.plot_rul(tr_pred2[:stop], tr['rul'][:stop], same_scale=False, figsize=figsize)
```

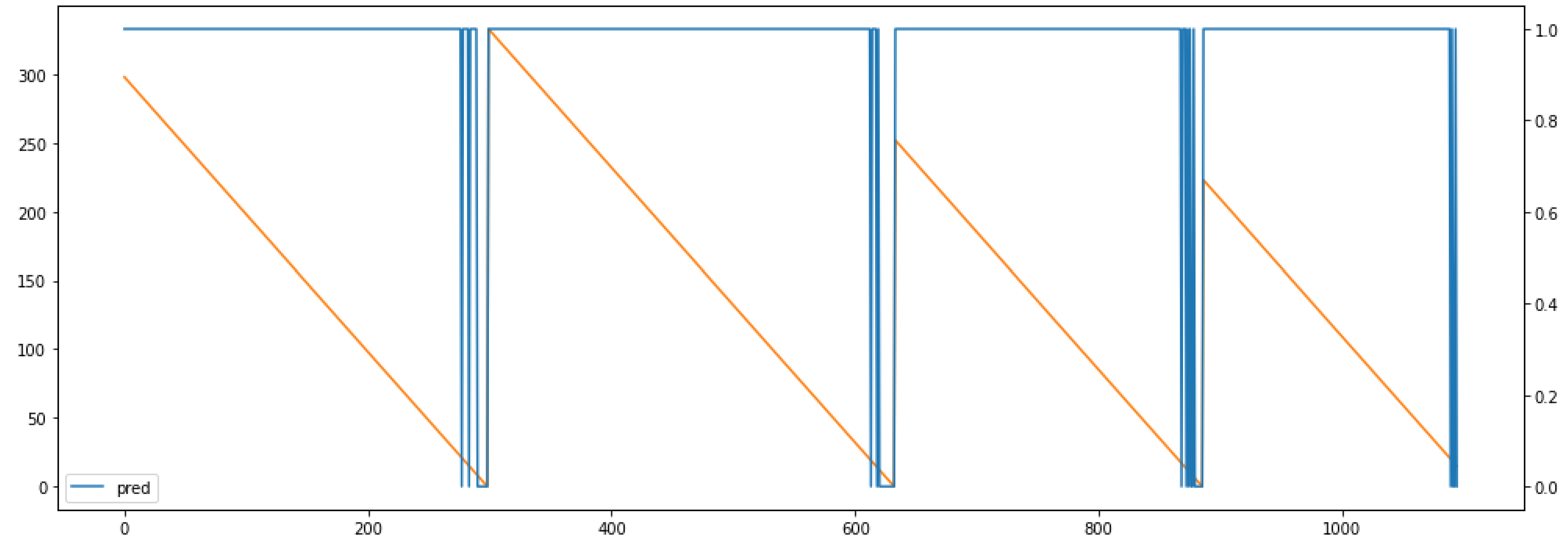


- Still, the behavior seems to be reasonable

## Predictions

### Let's see the behavior on the test set

```
In [12]: ts_pred2 = np.round(nn2.predict(ts_s[dt_in]).ravel())
         cmapss.plot_rul(ts_pred2[:stop], ts['rul'][:stop], same_scale=False, figsize=figsize)
```



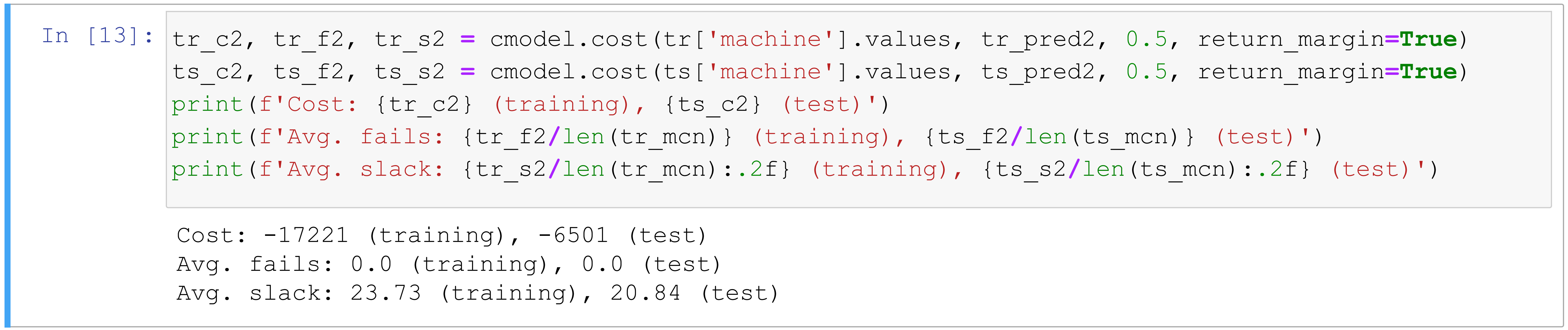
- Apparently a decent degree of generalization

## 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 [13]:
```
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} (training), {ts_c2} (test)')
print(f'Avg. fails: {tr_f2/len(tr_mcn)} (training), {ts_f2/len(ts_mcn)} (test)')
print(f'Avg. slack: {tr_s2/len(tr_mcn):.2f} (training), {ts_s2/len(ts_mcn):.2f} (test)')
```

```
Cost: -17221 (training), -6501 (test)
Avg. fails: 0.0 (training), 0.0 (test)
Avg. slack: 23.73 (training), 20.84 (test)
```

- Still pretty good results, but worse than the best regression approach
- We will investigate two potential reasons

## Balancing a Dataset

## Balancing a Dataset

**By construction, our dataset is likely to be strongly unbalanced**

```
In [14]: counts = tr_lbl.value_counts(normalize=True)
         counts

Out[14]: True     0.938526
         False    0.061474
         Name: rul, dtype: float64
```

- In these kind of situation, SGD optimization may have convergence issues
- The gradient will push strongly in the direction of the overrepresented class

**A common practice to address this issue is using class weights**

- Typically, we use weights inversely proportional to the counts
- ...So as to counter-balance the effect

```
In [15]: class_weight = {0: 1/counts[0], 1: 1/counts[1]}
```

## Class-Weights, Cross-Entropy, and Likelihood

**While are class frequencies typically used as weights?**

Let's say our classifier is trained via for weighted cross-entropy:

$$\min_y \left\{ -w_1 \sum_{\hat{y}_i=1} \log y_i - w_0 \sum_{\hat{y}_i=0} \log(1-y_i) \right\}$$

- Where $\hat{y}$ is the label vector and $y$ the vector of classifier outputs
- $w_0$ and $w_1$ are the weights for class 0 and 1

**If we apply an exponential and we switch optimization direction we get:**

$$\max_y \left\{ e^{w_1 \sum_{\hat{y}_i=1} \log y_i} e^{w_0 \sum_{\hat{y}_i=0} \log(1-y_i)} \right\}$$

- The optimal points are the same as before

## Class-Weights, Cross-Entropy, and Likelihood

**We with algebraic manipulations we finally obtain:**

$$\max_y \left\{ \prod_{\hat{y}_i=1} y_i^{w_1} \prod_{\hat{y}_i=0} (1 - y_i)^{w_0} \right\}$$

**When training, we are maximizing a likelihood expression!**

- To see that, just interpret $y_i$ as the (estimated) probability of the class being 1

**The weights act by multiplying the class probabilities**

...Which is equivalent to adjusting their frequencies!

- E.g. from a mathematical point of view, setting $w_0$ to 2...
- ...Is equivalent to making a copy of each example having label 0

Hence, inverse class frequencies are often used as weights for dataset rebalancing

## Training with Non-uniform Class Weights

### Let's repeat the training process

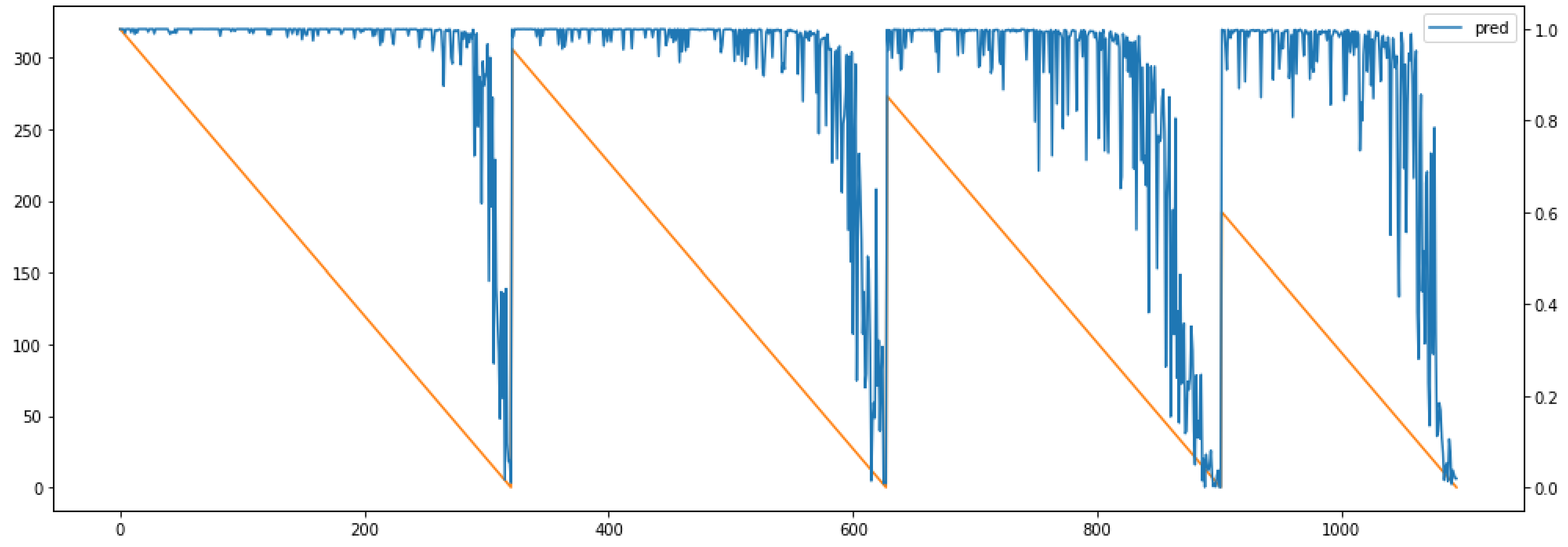
In [16]:
```
nn3 = build_classifier(hidden=[32, 32])
nn3.compile(optimizer='Adam', loss='binary_crossentropy')
history3 = nn3.fit(tr_s[dt_in], tr_lbl, validation_split=0.2,
                 callbacks=cb, class_weight=class_weight,
                 batch_size=32, epochs=20, verbose=1)
```

```
Epoch 1/20
1135/1135 [==============================] - 1s 773us/step - loss: 0.9785 - val_loss: 0.4846
Epoch 2/20
1135/1135 [==============================] - 1s 699us/step - loss: 0.5663 - val_loss: 0.2182
Epoch 3/20
1135/1135 [==============================] - 1s 698us/step - loss: 0.4288 - val_loss: 0.2280
Epoch 4/20
1135/1135 [==============================] - 1s 698us/step - loss: 0.3668 - val_loss: 0.1700
Epoch 5/20
1135/1135 [==============================] - 1s 695us/step - loss: 0.3452 - val_loss: 0.0957
Epoch 6/20
1135/1135 [==============================] - 1s 704us/step - loss: 0.3176 - val_loss: 0.1723
Epoch 7/20
1135/1135 [==============================] - 1s 704us/step - loss: 0.3070 - val_loss: 0.1606
Epoch 8/20
1135/1135 [==============================] - 1s 709us/step - loss: 0.2940 - val_loss: 0.3057
Epoch 9/20
1135/1135 [==============================] - 1s 718us/step - loss: 0.2820 - val_loss: 0.1319
Epoch 10/20
1135/1135 [==============================] - 1s 703us/step - loss: 0.2792 - val_loss: 0.1631
Epoch 11/20
```

## Predictions

**Let's check how the the raw predictions (probabilities) have changed**

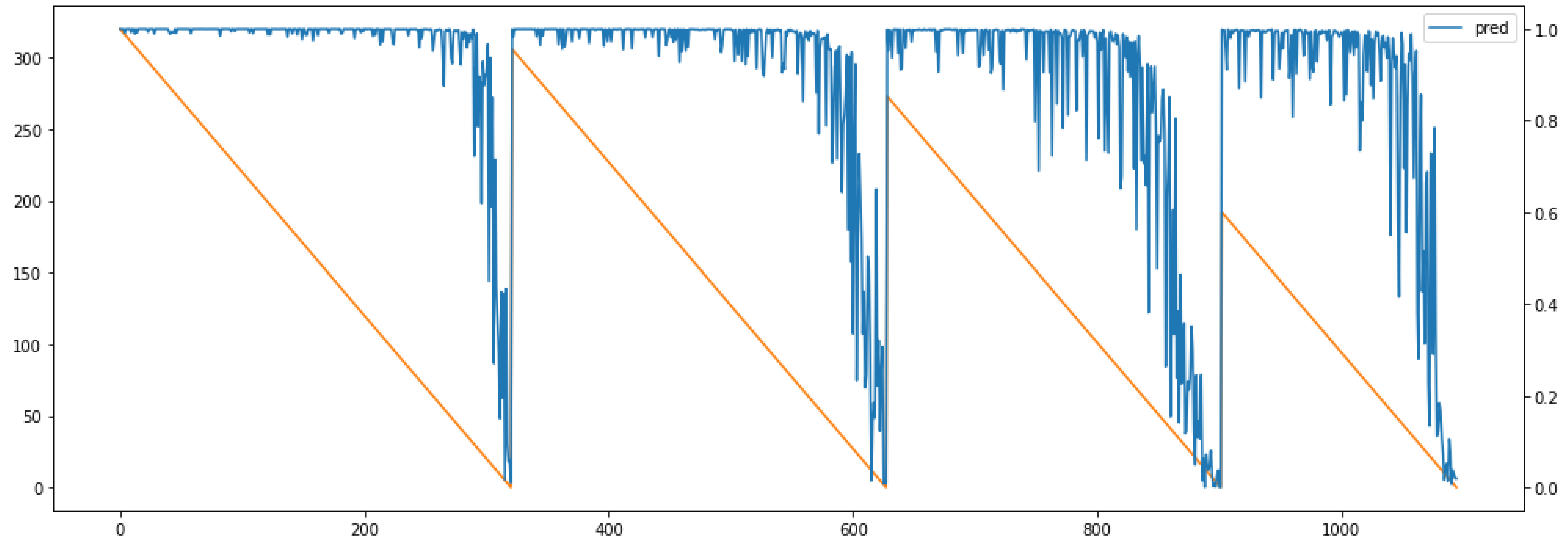
```
In [17]: tr_pred3_prob = nn3.predict(tr_s[dt_in]).ravel()
         cmapss.plot_rul(tr_pred3_prob[:stop], tr['rul'][:stop], same_scale=False, figsize=figsize)
```



## Predictions

### Let's check how the the raw predictions (probabilities) have changed

```
In [17]: tr_pred3_prob = nn3.predict(tr_s[dt_in]).ravel()
         cmapss.plot_rul(tr_pred3_prob[:stop], tr['rul'][:stop], same_scale=False, figsize=figsize)
```



- They have actually become worse: there are mistakes in the initial section

## Evaluation

**Let's check the model performance in terms of cost**

```
In [19]: tr_pred3 = np.round(nn3.predict(tr_s[dt_in]).ravel())
         ts_pred3 = np.round(nn3.predict(ts_s[dt_in]).ravel())
         tr_c3, tr_f3, tr_s3 = cmodel.cost(tr['machine'].values, tr_pred3, 0.5, return_margin=True)
         ts_c3, ts_f3, ts_s3 = cmodel.cost(ts['machine'].values, ts_pred3, 0.5, return_margin=True)
         print(f'Cost: {tr_c3} (training), {ts_c3} (test)')
         print(f'Avg. fails: {tr_f3/len(tr_mcn)} (training), {ts_f3/len(ts_mcn)} (test)')
         print(f'Avg. slack: {tr_s3/len(tr_mcn):.2f} (training), {ts_s3/len(ts_mcn):.2f} (test)')
```

```
Cost: -15327 (training), -5871 (test)
Avg. fails: 0.0 (training), 0.0 (test)
Avg. slack: 34.12 (training), 31.44 (test)
```

As expected, the results are worse

- The model now treats the two classes more fairly
- ...But this increases the chance of an (undesirable) early stop

**So, how shall we choose the balance?**

## Bayesian (Surrogate-Based) Optimization

## Taking a Step Back

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

$$\min_{\theta} \sum_{k \in K} cost(f(x_k\ \lambda^*), \theta)$$

$$\text{with: } \lambda^* = \text{argmin}_\lambda L(f(x_k, \lambda), y_k)$$

- Where $\lambda^*$ is the optimal parameter vector (i.e. the network weights)
- ...And $L$ is the loss function (i.e. the MSE), and $cost$ is our cost model
- $\theta$ is chosen so as to minimize the cost

**This is a bilevel optimization problem**

- However, since $\theta$ does 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:**

$$\min_\theta \sum_{k \in K} cost(f(x_k\ \lambda^*), 0.5)$$

$$\text{with: } \lambda^* = \text{argmin}_\lambda L(f(x_k, \lambda), \mathbf{1}_{y_k \geq \theta})$$

- In this case, we use a dummy threshold in the cost model (i.e. 0.5)
- ...And $\mathbf{1}_{y_k \geq \theta}$ is the indicator function of $y_k \geq \theta$ (i.e. our class labels)

**Unlike the previous one, this problem cannot be decomposed**

...Because $\theta$ appears in the loss function!

- This means we need to optimize $\theta$ and $\lambda$ at the same time
- This is obviously complicated, but it's also an opportunity
- ...Since we can adapt the training problem based on the threshold

## Taking a Step Back

**We can try to tackle the optimization process in a hierarchical fashion**

1. We search over the possible values of $\theta$
2. For the given $\theta$ value, we compute $\mathbf{1}_{y_k \geq \theta}$ (i.e. the class labels)
3. For the given $\theta$ and $\mathbf{1}_{y_k \geq \theta}$, we compute $\lambda^*$ (i.e. we train the model)
4. For the given $\theta$, $\mathbf{1}_{y_k \geq \theta}$, and $\lambda^*$, we compute the cost

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

**The method is reasonable, but slow**

- We could reduce the runtime by warm-starting each training attempt
- ...And by searching over $\theta$ in a smart way

## Bayesian Optimization

**We will use an approach known as Bayesian optimization**

- It's actually a family of surrogate-based optimization methods
- They are all designed to optimize blackbox functions
- I.e. functions with an unknown structure, that can only be evaluated

**Formally, they address problems in the form:**

$$\min_{x \in B} f(x)$$

- Where $\boldsymbol{B}$ is a box, i.e. a specification of bounds for each component of $\boldsymbol{x}$

In our case:

- The decision variable $\boldsymbol{x}$ would be $\boldsymbol{\theta}$
- The function to be optimized would be the cost

## Bayesian (or Surrogate-Based) Optimization

**Internally, methods in this family store:**

- A collection $\hat{x}$ of evaluated points
- A surrogate-model $\tilde{f}$ for $f$

**The basic loop of the algorithm is then as follow:**

Let $x^*$ be the current candidate optimum

- Based on the current $\tilde{f}$:
  - Determine whether there exists $x' \in B$ that may improve over $x^*$
  - If there is no such point, return $x^*$
  - Otherwise
    - Evaluate $f(x')$ and possibly replace $x^*$
    - Add $x'$ to $\hat{x}$, adjust $\tilde{f}$, and repeat

## Bayesian (or Surrogate-Based) Optimization

**A few considerations:**

In practice, the algorithm needs to balance exploration and exploitation

- We need to explore regions where we cannot make confident predictions
- ...But we need also to focus on regions with promising cost values

**Different Bayesian optimization algorithms:**

- Make use of different surrogate models
- Rely on different criteria for choosing $x'$
- Strike different trade-offs in terms of number of (expensive) evaluations of $f$
- ...And the quality of the obtained solutions

**For more information, see (e.g.) this tutorial**

## Our Chosen Optimizer

**We will use the `bayesian-optimization` python module**

- The solver is designed for maximization
- Decent documentation can be found on its github page
- It's not as fast as (e.g.) RBFOpt, but easier to install and configure

**The solver relies on Gaussian Processes as a surrogate model:**

GPs are guaranteed to (approximately) touch every point in the training set

- ...And behave reasonably well even with a simple RBF kernel
- ...Which is in fact the one used in this case

**GPs provide confidence intervals**

...Which the algorithm uses to balance exploration and exploitation

- Regions with wide C.I. may be worth exploring
- ...And regions with high (upper) C.I. may contain high-quality solutions

## Method Setup

**First, we need to define the function to be maximized**

This will be the (negative) cost, after retraining the classifier:

```
In [20]: nn4 = build_classifier(hidden=[32, 32])
         nn4.compile(optimizer='Adam', loss='binary_crossentropy')
         nn4.set_weights(nn2.get_weights())

         bo_weights = {}
         def classification_obj(class_thr, class0_weight):
             # Define new labels
             tr_lbl = (tr['rul'] >= class_thr)
             # Fit
             nn4.fit(tr_s[dt_in], tr_lbl, validation_split=0.2,
                              callbacks=cb, class_weight={1:1, 0:class0_weight},
                              batch_size=32, epochs=5, verbose=0)
             # Cost computation
             tr_pred = np.round(nn4.predict(tr_s[dt_in]).ravel())
             tr_cost = cmodel.cost(tr['machine'].values, tr_pred, 0.5)
             # Store the model weights
             bo_weights[class_thr, class0_weight] = nn4.get_weights()
             return -tr_cost
```

## Method Setup

**We rely on warm-starting to reduce the training time**

We start from (the weights of) our deep MLP

```
nn4 = build_classifier(hidden=[32, 32])
nn4.compile(optimizer='Adam', loss='binary_crossentropy')
nn4.set_weights(nn2.get_weights())
```

...So that we can train for just a few epochs

```
nn4.fit(tr_s[dt_in], tr_lbl, validation_split=0.2,
                 callbacks=cb, class_weight={1:1, 0:class0_weight},
                 batch_size=32, epochs=5, verbose=0)
```

- Since the training process is stochastic...
- The optimization result will be stochastic as well

## Method Setup

**We will optimize both $\theta$ and the class weights**

```
def classification_obj(class_thr, class0_weight):
    ...
    nn4.fit(tr_s[dt_in], tr_lbl, validation_split=0.2,
                     callbacks=cb, class_weight={1:1, 0:class0_weight},
                     batch_size=32, epochs=5, verbose=0)
    ...
```

- We can afford it, since we have a much better optimizer than grid search

**We need to store the weights of each model**

...So that we can recover the optimal configuration:

```
bo_weights[class_thr, class0_weight] = nn4.get_weights()
```

## Method Setup

**We now define the bounding box**

```
In [21]: box = {'class_thr': (1, 15), 'class0_weight': (1, 10)}
```

- We will consider stopping from 1 to 15 steps before a failure
- We will consider weights for class 0 from 1 to 10

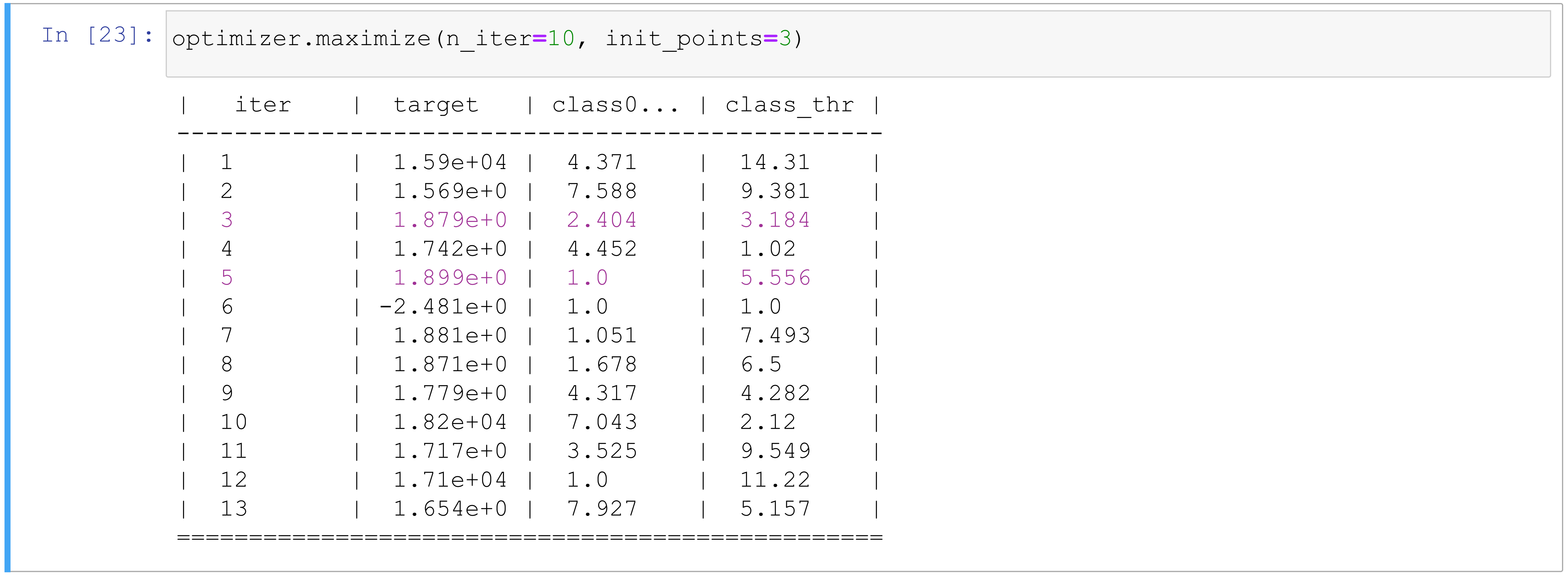
**Then we can configure the optimizer:**

```
In [22]: from bayes_opt import BayesianOptimization
         optimizer = BayesianOptimization(f=classification_obj,
                                          pbounds=box, random_state=42, verbose=2)
```

- We specify the function to be maximized, the box
- ...But also a random seed and the logging level

## Optimization

### Finally, we run the optimization process

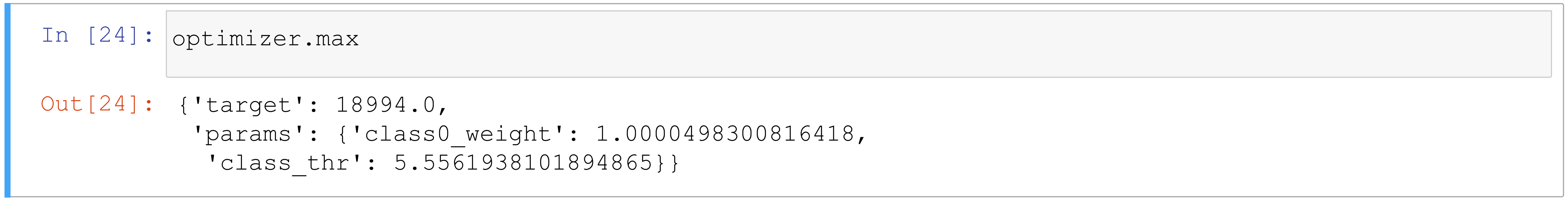


```
In [23]: optimizer.maximize(n_iter=10, init_points=3)
```

```
|   iter    |  target   | class0... | class_thr |
-------------------------------------------------
|  1        |  1.59e+04 |  4.371    |  14.31    |
|  2        |  1.569e+0 |  7.588    |  9.381    |
|  3        |  1.879e+0 |  2.404    |  3.184    |
|  4        |  1.742e+0 |  4.452    |  1.02     |
|  5        |  1.899e+0 |  1.0      |  5.556    |
|  6        | -2.481e+0 |  1.0      |  1.0      |
|  7        |  1.881e+0 |  1.051    |  7.493    |
|  8        |  1.871e+0 |  1.678    |  6.5      |
|  9        |  1.779e+0 |  4.317    |  4.282    |
|  10       |  1.82e+04 |  7.043    |  2.12     |
|  11       |  1.717e+0 |  3.525    |  9.549    |
|  12       |  1.71e+04 |  1.0      |  11.22    |
|  13       |  1.654e+0 |  7.927    |  5.157    |
=================================================
```

## Retrieving the Results

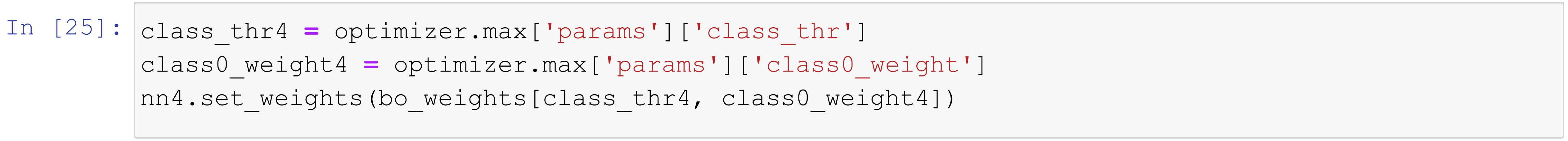
**We can access the best configuration via**



```
In [24]: optimizer.max

Out[24]: {'target': 18994.0,
          'params': {'class0_weight': 1.0000498300816418,
           'class_thr': 5.5561938101894865}}
```

Once we know the configuration, we can reconstruct the corresponding model:



```
In [25]: class_thr4 = optimizer.max['params']['class_thr']
         class0_weight4 = optimizer.max['params']['class0_weight']
         nn4.set_weights(bo_weights[class_thr4, class0_weight4])
```

- We do this by accessing the weights in the `bo_weights` dictionary

## Evaluation

**Finally, we can obtain the predictions**

```
In [26]: tr_pred4 = np.round(nn4.predict(tr_s[dt_in]).ravel())
         ts_pred4 = np.round(nn4.predict(ts_s[dt_in]).ravel())
```

...And we can evaluate the optimized model:

```
In [27]: tr_c4, tr_f4, tr_s4 = cmodel.cost(tr['machine'].values, tr_pred4, 0.5, return_margin=True)
         ts_c4, ts_f4, ts_s4 = cmodel.cost(ts['machine'].values, ts_pred4, 0.5, return_margin=True)
         print(f'Cost: {tr_c4} (training), {ts_c4} (test)')
         print(f'Avg. fails: {tr_f4/len(tr_mcn)} (training), {ts_f4/len(ts_mcn)} (test)')
         print(f'Avg. slack: {tr_s4/len(tr_mcn):.2f} (training), {ts_s4/len(ts_mcn):.2f} (test)')

          Cost: -18994 (training), -7127 (test)
          Avg. fails: 0.0 (training), 0.0 (test)
          Avg. slack: 13.95 (training), 10.68 (test)
```

- We are now on-par with the best regression based approaches!
- Actual results may be slightly better or worse due to randomness

## Considerations

**Why choosing a classification-based approach?**

- In some cases, classification can be easier than regression
- The threshold affects training, which is computationally heavier
- ...But it means that training can adapt to the threshold

**We we tackle the prediction and cost optimization stages in isolation:**

- The two stages are "glued" by the ML loss (MSE or cross-entropy)
- ....But the loss does not penalize mistakes based on their (economic) cost!
- Ideally, an end-to-end cost optimization approach should work better

**Bayesian/surrogate-based optimization can have many uses**

- Hyper-parameter optimization
- Fitting simulation models
- Optimizing equipment setpoints (e.g. Air Conditioning), based on simulation
- ...