Domain Knowledge Injection via SBR

Scarce Labels in RUL Predictions

In our RUL use case we had access to many run-to-failure experiments

In practice, this is seldom the case

- Run-to-failure experiments are time consuming
- They may not be viable for large and complex machines
- Typically, only a few runs are available

However, data about normal operation may still be abundant

- This may come from test runs, installed machines, etc.
- It looks exactly like the input data for our RUL prediction model
- ...And it will still show sign of component wear

However, the true RUL value in this case will be unknown

Can we still take advantage of this data?

Domain Knowledge in Machine Learning

We can take an anomaly detection approach

- We can use an AE or a density estimator to generate an anomaly signal
- Then we can optimize a threshold based on the few run-to-failure experiments

This approach may work, but:

- Signals for different machines may grow at different rates
- ...Thus making the generalization difficult to achieve

We can resort to autoencoders and use semi-supervised learning:

- We train an autoencoder on the unsupervised data, then remove the decoder
- ...We replace them it with classification layers, trained on the supervised data

Another viable technique, but with one drawback

- Since the AE is trained for a task very different from RUL prediction
- ...There is no guarantee that the learned encoding is well suited for that

Domain Knowledge in Machine Learning

We will investigate here a different approach

- We will use domain knowledge to get information from the unsupervised data
- We will then inject such information in the model by means of constraints

This approach introduces a new source of information

- The domain knowledge may be provided by experts
- ...Or it may be a second, heterogeneous model (e.g. a physical model)

In the remainder of the notebook

- We will first address the problem using only the supervised information
- ...Then we will see how to use domain knowledge to exploit unsupervised data

The approach is not limited to RUL prediction

- The techniques we will see work for a wide variety of constraints

Our Baseline Approach

Data Loading and Preparation

Let's start by loading our old dataset

We will focus once again on the FD004 data:

```
In [3]: data_by_src = cst.split_by_field(data, field='src')
        dt = data_by_src['train_FD004']
        dt[dt_in] = dt[dt_in].astype(np.float32)
```

We then simulate the scarcity of run-to-failure experiments:

```
In [4]: trs_ratio = 0.03 # Ratio of supervised experiments
        tru_ratio = 0.75 # Ration of supervised and unsupervised data

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

        sep_trs = int(trs_ratio * len(machines))
        sep_tru = int(tru_ratio * len(machines))

        trs_mcn = list(machines[:sep_trs])
        tru_mcn = list(machines[sep_trs:sep_tru])
        ts_mcn = list(machines[sep_tru:])
```

Data Loading and Preparation

Let's check how many machines we have in each group

In [5]:
```
print(f'Num. machine: {len(trs_mcn)} (supervised), {len(tru_mcn)} (unsupervised), {len(ts_mcn)}
```

```
Num. machine: 7 (supervised), 179 (unsupervised), 63 (test)
```

We can then splid the dataset according to this machine groups:

In [6]:
```
tr, ts = cst.partition_by_machine(dt, trs_mcn + tru_mcn)
trs, tru = cst.partition_by_machine(tr, trs_mcn)
```

Let's check the number of examples for each group:

In [7]:
```
print(f'Num. samples: {len(trs)} (supervised), {len(tru)} (unsupervised), {len(ts)} (test)')
```

```
Num. samples: 1376 (supervised), 44009 (unsupervised), 15864 (test)
```

Data Loading and Preparation

The we standardize the input data

In [8]:
```
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 = trs.copy()
tr_s[dt_in] = (tr_s[dt_in] - trmean) / trstd
trs_s = trs.copy()
trs_s[dt_in] = (trs_s[dt_in] - trmean) / trstd
tru_s = tru.copy()
tru_s[dt_in] = (tru_s[dt_in] - trmean) / trstd
```

...And we normalize the RUL values

In [9]:
```
trmaxrul = tr['rul'].max()

ts_s['rul'] = ts['rul'] / trmaxrul
tr_s['rul'] = tr['rul'] / trmaxrul
trs_s['rul'] = trs['rul'] / trmaxrul
tru_s['rul'] = tru['rul'] / trmaxrul
```

MLP with Scarce Labels

We can now train again our old MLP model

In this case, we have wrapped its code in a class:

```
class MLPRegressor(keras.Model):
    def __init__(self, input_shape, hidden=[]):
        super(MLPRegressor, self).__init__()
        self.lrs = [layers.Dense(h, activation='relu') for h in hidden]
        self.lrs.append(layers.Dense(1, activation='linear'))

    def call(self, data):
        x = data
        for layer in self.lrs: x = layer(x)
        return x
```

MLP with Scarce Labels

The model can be trained as usual

```
In [10]: nn = cst.MLPRegressor(input_shape=len(dt_in), hidden=[32, 32])
         nn.compile(optimizer='Adam', loss='mse')
         history = nn.fit(trs_s[dt_in], trs_s['rul'], batch_size=32, epochs=20, verbose=1)
```

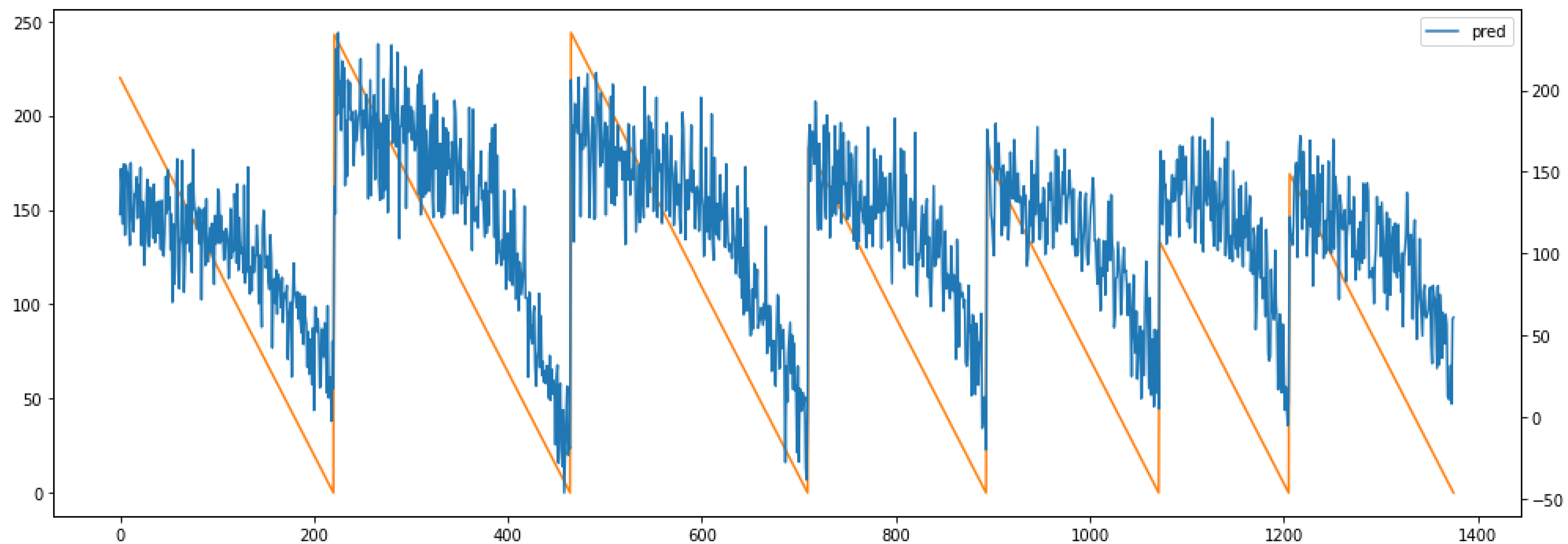
```
Epoch 1/20
43/43 [==============================] - 0s 851us/step - loss: 0.0795
Epoch 2/20
43/43 [==============================] - 0s 887us/step - loss: 0.0139
Epoch 3/20
43/43 [==============================] - 0s 800us/step - loss: 0.0111
Epoch 4/20
43/43 [==============================] - 0s 796us/step - loss: 0.0105
Epoch 5/20
43/43 [==============================] - 0s 867us/step - loss: 0.0100
Epoch 6/20
43/43 [==============================] - 0s 832us/step - loss: 0.0091
Epoch 7/20
43/43 [==============================] - 0s 822us/step - loss: 0.0085
Epoch 8/20
43/43 [==============================] - 0s 816us/step - loss: 0.0084
Epoch 9/20
43/43 [==============================] - 0s 889us/step - loss: 0.0077
Epoch 10/20
43/43 [==============================] - 0s 883us/step - loss: 0.0072
Epoch 11/20
43/43 [==============================] - 0s 857us/step - loss: 0.0071
```

Evaluation

The RUL Predictions follow the trend already identified

...But they are much more noisy, due to the small size of the training set

In [11]:
```
trs_pred = nn.predict(trs_s[dt_in]).ravel() * trmaxrul
stop = 1400
cst.plot_rul(trs_pred[:stop], trs["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```

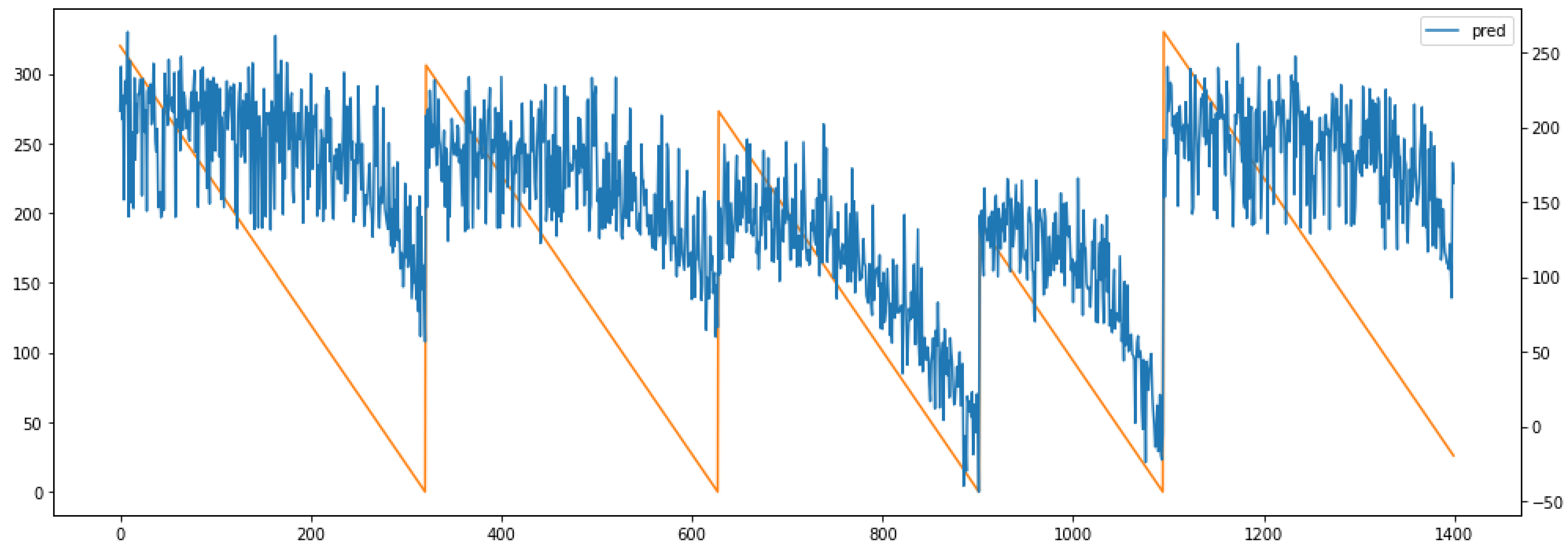


Evaluation

The behavior on the unsupervised data is very similar

...And similarly noisy

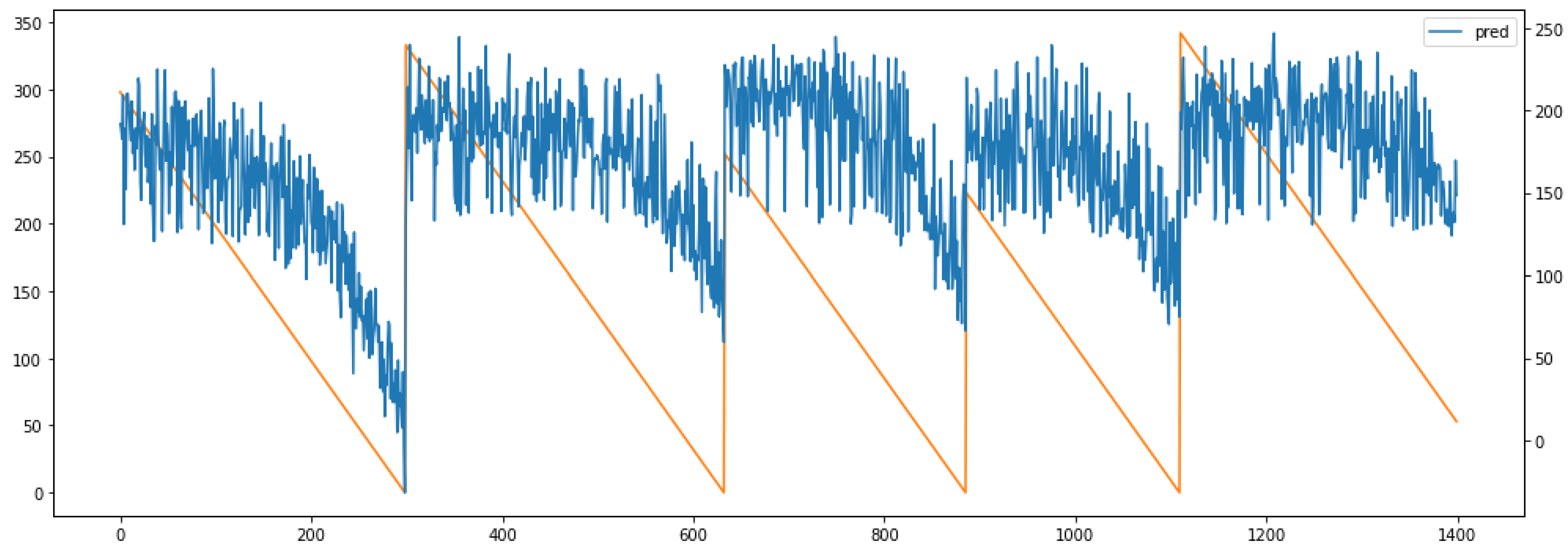
In [12]:
```
tru_pred = nn.predict(tru_s[dt_in]).ravel() * trmaxrul
cst.plot_rul(tru_pred[:stop], tru["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```



Evaluation

The same goes for the data in the test set

In [13]:
```
ts_pred = nn.predict(ts_s[dt_in]).ravel() * trmaxrul
cst.plot_rul(ts_pred[:stop], ts["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```



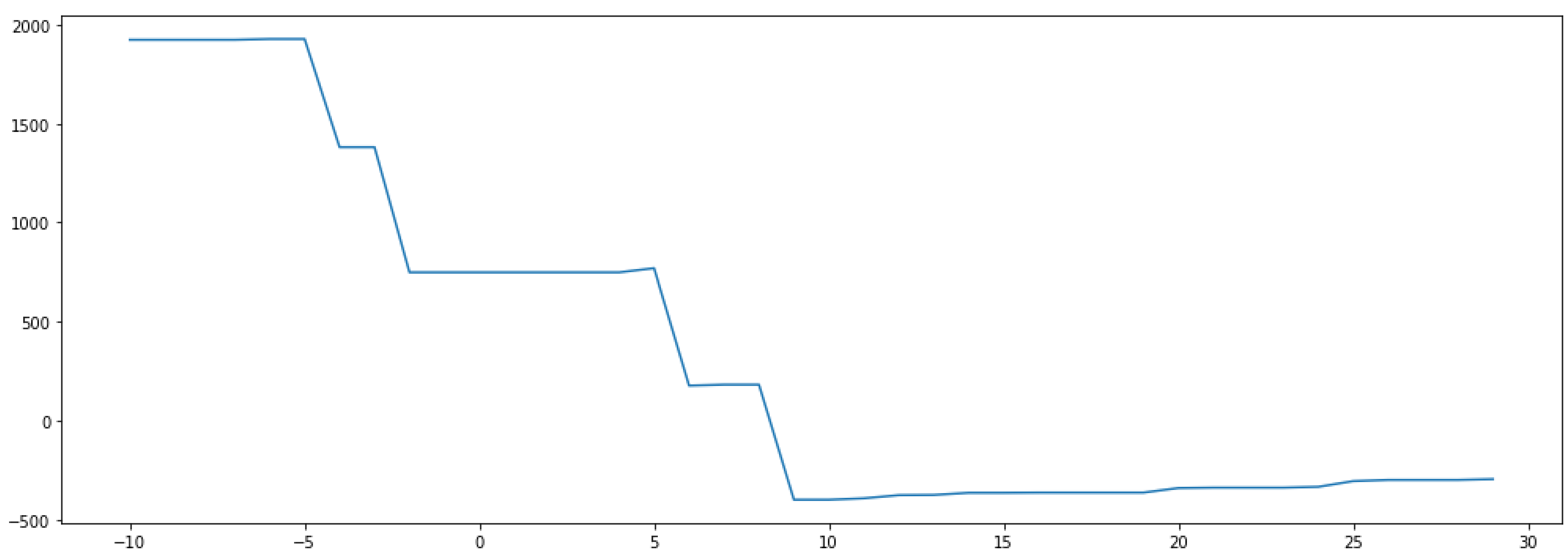
Cost Model and Threshold Optimization

We then proceed to define a cost model

```
In [14]: failtimes = dt.groupby('machine')['cycle'].max()
         safe_interval, maintenance_cost = failtimes.min(), failtimes.max()

         cmodel = cst.RULCostModel(maintenance_cost=maintenance_cost, safe_interval=safe_interval)
         th_range = np.arange(-10, 30)
         trs_thr = cst.opt_threshold_and_plot(trs_s['machine'].values, trs_pred, th_range, cmodel, figsiz
         print(f'Optimal threshold for the training set: {trs_thr}')
```

```
Optimal threshold for the training set: 9
```



Cost Results

The cost on the training set is still good...

...But that is not true for the unsupervised experiments and the test set

```
In [15]: trs_c, trs_f, trs_sl = cmodel.cost(trs_s['machine'].values, trs_pred, trs_thr, return_margin=Tru
         tru_c, tru_f, tru_sl = cmodel.cost(tru_s['machine'].values, tru_pred, trs_thr, return_margin=Tru
         ts_c, ts_f, ts_sl = cmodel.cost(ts['machine'].values, ts_pred, trs_thr, return_margin=True)
         print(f'Cost: {trs_c} (supervised), {tru_c} (unsupervised), {ts_c} (test)')

         Cost: -400 (supervised), 35858 (unsupervised), 15540 (test)
```

```
In [16]: trs_nm, tru_nm, ts_nm = len(trs_mcn), len(tru_mcn), len(ts_mcn)
         print(f'Avg. fails: {trs_f/trs_nm:.2f} (supervised), {tru_f/tru_nm:.2f} (unsupervised), {ts_f/ts
         print(f'Avg. slack: {trs_sl/trs_nm:.2f} (supervised), {tru_sl/tru_nm:.2f} (unsupervised), {ts_sl

         Avg. fails: 0.00 (supervised), 0.45 (unsupervised), 0.52 (test)
         Avg. slack: 11.43 (supervised), 7.22 (unsupervised), 7.11 (test)
```

- In particular, there is a very high failure rate on unseen data

Domain Knowledge as Constraints

Domain Knowledge as Constraints

We know that the RUL decreases at a fixed rate

- After 1 time step, the RUL will have decreased by 1 unit
- After 2 time steps, the RUL will have decreased by 2 units and so on

In general, let $\hat{x}_i$ and $\hat{x}_j$ be the i-th and j-th samples for a given component

Then we know that:

$$f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) = j - i \qquad \forall i, j = 1..m \text{ s.t. } c_i = c_j$$

- $\boldsymbol{c_i}, \boldsymbol{c_j}$ are the components for (respectively) sample $\boldsymbol{i}$ and $\boldsymbol{j}$
- Samples are assumed to be temporally sorted
- The left-most terms is the difference between the predicted RULs
- $\boldsymbol{j - i}$ is the difference between the sequential indexes of the two samples
- ...Which by construction should be equal to the RUL difference

Domain Knowledge as Constraints

The relation we identified is a constraint

$$f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) = j - i \qquad \forall i, j = 1..m \text{ s.t. } c_i = c_j$$

It represents domain knowledge that should (in principle) hold for our problem

- We don't need strict satisfaction: we can treat it as a soft constraint
- The constraint involves pairs of example, i.e. it is a relational constraint

A simple approach: use the constraint to derive a semantic regularizer

This approach is sometimes known as Semantic Based Regularization

- The regularizer represents a constraint that we think should generally hold
- ...It is meant to assist the model by ensuring better generalization
- ...Or by speeding up the training process
- ...Or by allowing one to take advantage of (otherwise) unsupervised data

Our Regularizer

We need to design a regualizer for our constraint

$$f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) = j - i \qquad \forall i, j = 1..m \text{ s.t. } c_i = c_j$$

The regualizer should penalize violations of the constraint, e.g.

$$\lambda \Big(f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) - (j - i) \Big)^2$$

- Using the absolute value (h1 norm) may also work

In principle, we should consider all valid pairs

Such an approach would lead to the following loss function:

$$L(\hat{x}, \omega) + \lambda \sum_{i=1..m} \sum_{\substack{j=i+1..m \\ c_i = c_j}} \Big(f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) - (j - i) \Big)^2$$

Our Regularizer

We can focus on contiguous pairs, i.e.

$$L(\hat{x}, \omega) + \lambda \sum_{\substack{i \prec j \\ c_i = c_j}} \left(f(\hat{x}_i, \omega) - f(\hat{x}_j, \omega) - (j - i) \right)^2$$

- Where $i \prec j$ iff j is the next sample for after i for a given machine
- This approach requires a linear (rather than quadratic) number of constraints

It can work with mini-batches

- In this case, $\prec$ will refer to contiguous samples in the same batch
- ...And of course for the same component

We will now see how to implement this approach

Removing RUL Values

We start by preparing a bit more the unsupervised data

- First, we remove the end of the unsupervised data sequences
- This simulate the fact that the machines are still operating

In [17]:
```
tru_s_by_m = cst.split_by_field(tru_s, 'machine')
np.random.seed(42)
for mcn, tmp in tru_s_by_m.items():
    cutoff = int(np.random.randint(10, 50, 1))
    tru_s_by_m[mcn] = tmp.iloc[:-cutoff]
tru_st = pd.concat(tru_s_by_m.values())
```

Then we assign an invalid value to the RUL for unsupervised data:

In [18]:
```
trsu_s = pd.concat((trs_s, tru_st))
trsu_s.loc[tru_st.index, 'rul'] = -1
```

We also buid a single dataset containing both supervised and unsupervised data

Generating Batches from the Same Machine

Our SBR approach requires to have sorted samples from the same machine

The easiest way to ensure we have enough of them is using a custom `DataGenerator`

```
class SMBatchGenerator(tf.keras.utils.Sequence):
    def __init__(self, data, in_cols, batch_size, seed=42): ...

    def __len__(self): ...

    def __getitem__(self, index): ...

    def on_epoch_end(self): ...

    def __build_batches(self): ...
```

Generating Batches from the Same Machine

The `__init__` method takes care of the initial setup

```python
def __init__(self, data, in_cols, batch_size, seed=42):
    super(SMBatchGenerator).__init__()
    self.data = data
    self.in_cols = in_cols
    self.dpm = split_by_field(data, 'machine')
    self.rng = np.random.default_rng(seed)
    self.batch_size = batch_size
    # Build the first sequence of batches
    self.__build_batches()
```

- We store some fields
- We split the data by machine
- We build a dedicated RNG
- ...And finally we call the custom-made `__build_batches` method

Generating Batches from the Same Machine

The `__build_batches` method prepares the batches for one full epoch

```python
def __build_batches(self):
    self.batches, self.machines = [], []
    mcns = list(self.dpm.keys()) # sort the machines at random
    self.rng.shuffle(mcns)
    for mcn in mcns: # Loop over all machines
        index = self.dpm[mcn].index # sample indexes for this machine
        padsize = self.batch_size - (len(index) % self.batch_size)
        padding = self.rng.choice(index, padsize) # pad the last batch
        idx = np.hstack((index, padding))
        self.rng.shuffle(idx) # shuffle sample indexes for this machine
        bt = idx.reshape(-1, self.batch_size) # split into batches
        bt = np.sort(bt, axis=1) # sort every batch individually
        self.batches.append(bt) # store
        self.machines.append(np.repeat([mcn], len(bt)))
    self.batches = np.vstack(self.batches) # concatenate
    self.machines = np.hstack(self.machines)
```

Generating Batches from the Same Machine

A few other functions become very simple at this point

```
def __len__(self):
    return len(self.batches)

def on_epoch_end(self):
    self.__build_batches()
```

- `__len__` return the number of batches in the collection
- `__getitem__` simply retrieves one batch from the collection
- We rebuild the batches every epoch

Generating Batches from the Same Machine

Most of the remaining work is done in the `__getiitem__` method:

```python
def __getitem__(self, index):
    idx = self.batches[index]
    x = self.data[self.in_cols].loc[idx].values
    y = self.data['rul'].loc[idx].values
    flags = (y != -1)
    info = np.vstack((y, flags, idx)).T
    return x, info
```

- We retrieve the sample indexes `idx` for the batch
- ...The the corresponding input and RUL values from `self.data`
- The RUL value is -1 for the unsupervised data: we flag the meaningful RULs
- ...We pack indexes, RUL values, and flags into a single `info` tensor

Custom Training Step

We then enforce the constraints by means of a custom training step

```
class CstRULRegressor(MLPRegressor):
    def __init__(self, input_shape, alpha, beta, hidden=[]): ...

    def train_step(self, data): ...

    @property
    def metrics(self): ...
```

- We subclass our `MLPRegressor`, so we share its model structure
- We also inherit its `call` method
- The custom training step is implemented in `train_step`
- The `metrics` property allows us to rely on keras metric tracking

Custom Training Step

In the `__init__` function:

```
def __init__(self, input_shape, alpha, beta, maxrul, hidden=[]):
    super(CstRULRegressor, self).__init__(input_shape, hidden)
    # Weights
    self.alpha = alpha
    self.beta = beta
    self.maxrul = maxrul
    # Loss trackers
    self.ls_tracker = keras.metrics.Mean(name='loss')
    self.mse_tracker = keras.metrics.Mean(name='mse')
    self.cst_tracker = keras.metrics.Mean(name='cst')
```

- `beta` is the regularizer weight, `alpha` is a weight for the loss function itself
    - `alpha=0,beta=1` corresponds to a fully unsupervised approach
- We also store the maximum RUL
- We build several "trackers" for the terms in our loss function

Custom Training Step

In the custom training step:

```python
def train_step(self, data):
    x, info = data
    y_true = info[:, 0:1]
    flags = info[:, 1:2]
    idx = info[:, 2:3]

    with tf.GradientTape() as tape:
        y_pred = self(x, training=True) # predictions
        mse = k.mean(flags * k.square(y_pred-y_true)) # MSE loss
        delta_pred = y_pred[1:] - y_pred[:-1] # pred. difference
        delta_rul = -(idx[1:] - idx[:-1]) /self.maxrul # index difference
        deltadiff = delta_pred - delta_rul # difference of differences
        cst = k.mean(k.square(deltadiff)) # regualization term
        loss = self.alpha * mse + self.beta * cst # loss
    ...
```

Custom Training Step

In the custom training step:

```
def train_step(self, data):
    ...
    tr_vars = self.trainable_variables
    grads = tape.gradient(loss, tr_vars) # gradient computation

    self.optimizer.apply_gradients(zip(grads, tr_vars)) # weight update

    self.ls_tracker.update_state(loss) # update the loss trackers
    self.mse_tracker.update_state(mse)
    self.cst_tracker.update_state(cst)
    return {'loss': self.ls_tracker.result(), # return loss statuses
            'mse': self.mse_tracker.result(),
            'cst': self.cst_tracker.result()}
```

- We then apply the (Stochastic) Gradient Descent step
- Then we update and retun the loss trackers

The SBR Approach: Fully Unsupervised Training

We can now train our SBR-based approach

We will make a first attempt with a pure unsupervised training:

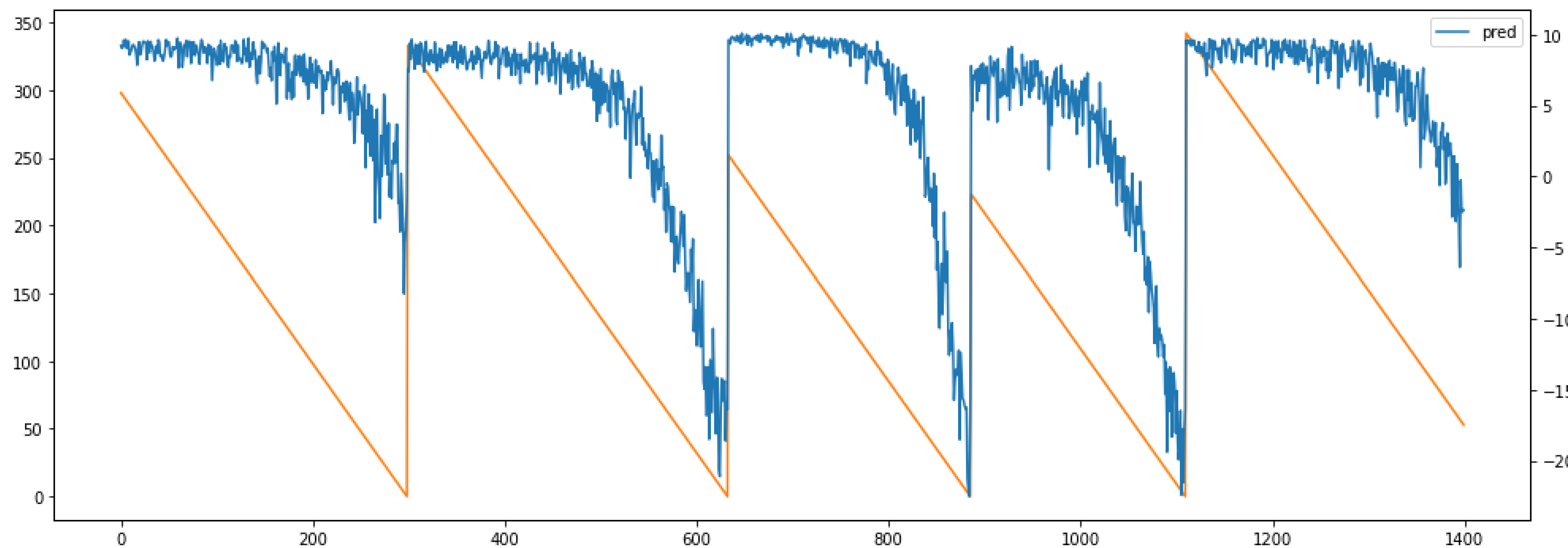
In [19]:
```
nn2 = cst.CstRULRegressor(input_shape=len(dt_in), alpha=0, beta=1, maxrul=trmaxrul, hidden=[32,
batch_gen = cst.CstBatchGenerator(trsu_s, dt_in, batch_size=32)
cb = [callbacks.EarlyStopping(monitor='loss', patience=10, restore_best_weights=True)]
nn2.compile(optimizer='Adam', run_eagerly=False)
history = nn2.fit(batch_gen, epochs=20, verbose=1, callbacks=cb)
```

```
Epoch 1/20
1350/1350 [==============================] - 5s 3ms/step - loss: 0.0011 - mse: 0.0055 - cst:
0.0011
Epoch 2/20
1350/1350 [==============================] - 5s 4ms/step - loss: 4.8427e-04 - mse: 0.0046 - cs
t: 4.8427e-04
Epoch 3/20
1350/1350 [==============================] - 5s 3ms/step - loss: 4.3203e-04 - mse: 0.0039 - cs
t: 4.3203e-04
Epoch 4/20
1350/1350 [==============================] - 5s 4ms/step - loss: 4.2454e-04 - mse: 0.0035 - cs
t: 4.2454e-04
Epoch 5/20
1350/1350 [==============================] - 5s 4ms/step - loss: 4.0726e-04 - mse: 0.0031 - cs
t: 4.0726e-04
Epoch 6/20
1350/1350 [==============================] - 5s 3ms/step - loss: 3.9672e-04 - mse: 0.0027 - cs
t: 3.9672e-04
```

Inspecting the Predictions

Then let's check the predictions for the test data

In [20]:
```
ts_pred = nn2.predict(ts_s[dt_in]).ravel() * trmaxrul
cst.plot_rul(ts_pred[:stop], ts["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```



- In many cases, we are already obtaining the trend we are familiar with!
- The scale is however completely off

The SBR Approach

Let's try again using both supervised and unsupervised data:

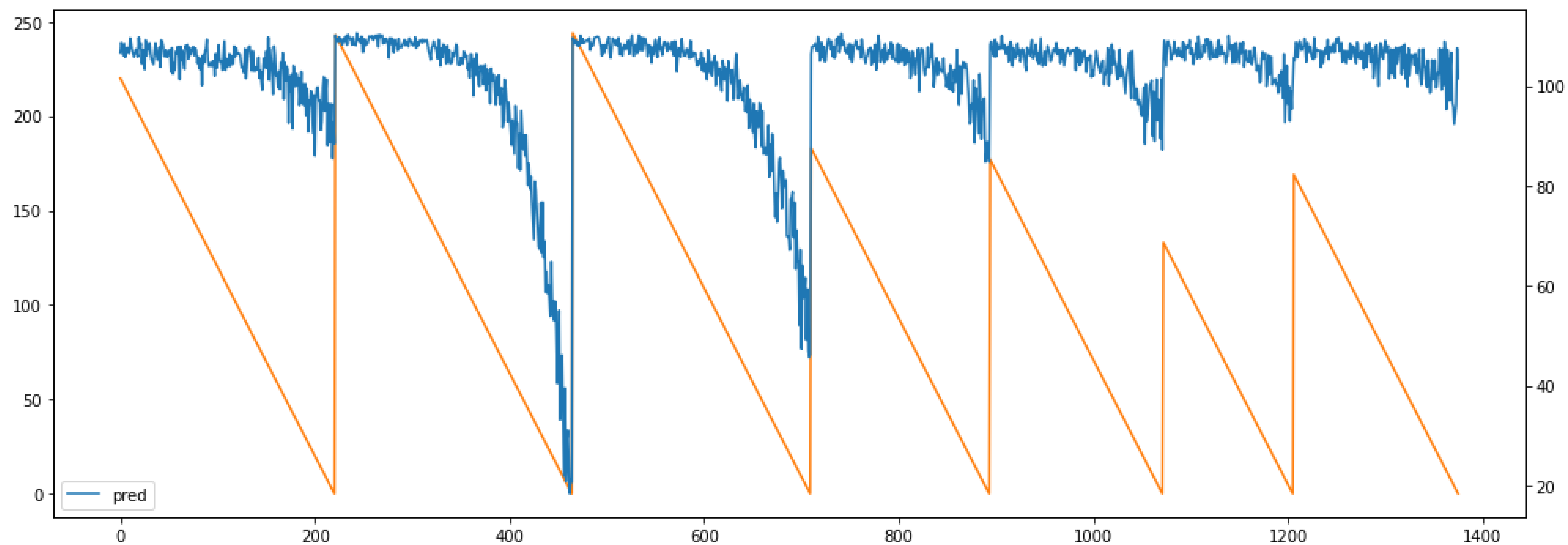
```
In [21]: nn2 = cst.CstRULRegressor(input_shape=len(dt_in), alpha=1, beta=5, maxrul=trmaxrul, hidden=[32,
         batch_gen = cst.CstBatchGenerator(trsu_s, dt_in, batch_size=32)
         cb = [callbacks.EarlyStopping(monitor='loss', patience=10, restore_best_weights=True)]
         nn2.compile(optimizer='Adam', run_eagerly=False)
         history = nn2.fit(batch_gen, epochs=20, verbose=1, callbacks=cb)
```

```
Epoch 1/20
1350/1350 [==============================] - 6s 4ms/step - loss: 0.0163 - mse: 5.1499e-04 - cs
t: 0.0032
Epoch 2/20
1350/1350 [==============================] - 4s 3ms/step - loss: 0.0028 - mse: 4.3059e-04 - cs
t: 4.6456e-04
Epoch 3/20
1350/1350 [==============================] - 5s 4ms/step - loss: 0.0027 - mse: 4.1129e-04 - cs
t: 4.6451e-04
Epoch 4/20
1350/1350 [==============================] - 5s 4ms/step - loss: 0.0027 - mse: 4.1235e-04 - cs
t: 4.6661e-04
Epoch 5/20
1350/1350 [==============================] - 5s 4ms/step - loss: 0.0027 - mse: 4.0606e-04 - cs
t: 4.5803e-04
Epoch 6/20
1350/1350 [==============================] - 5s 3ms/step - loss: 0.0025 - mse: 4.1048e-04 - cs
t: 4.2701e-04
Epoch 7/20
1350/1350 [==============================] - 4s 3ms/step - loss: 0.0024 - mse: 4.0041e-04 - cs
t: 4.0981e-04
```

Inspecting the Predictions

Let's have a look at the predictions on the supervised data

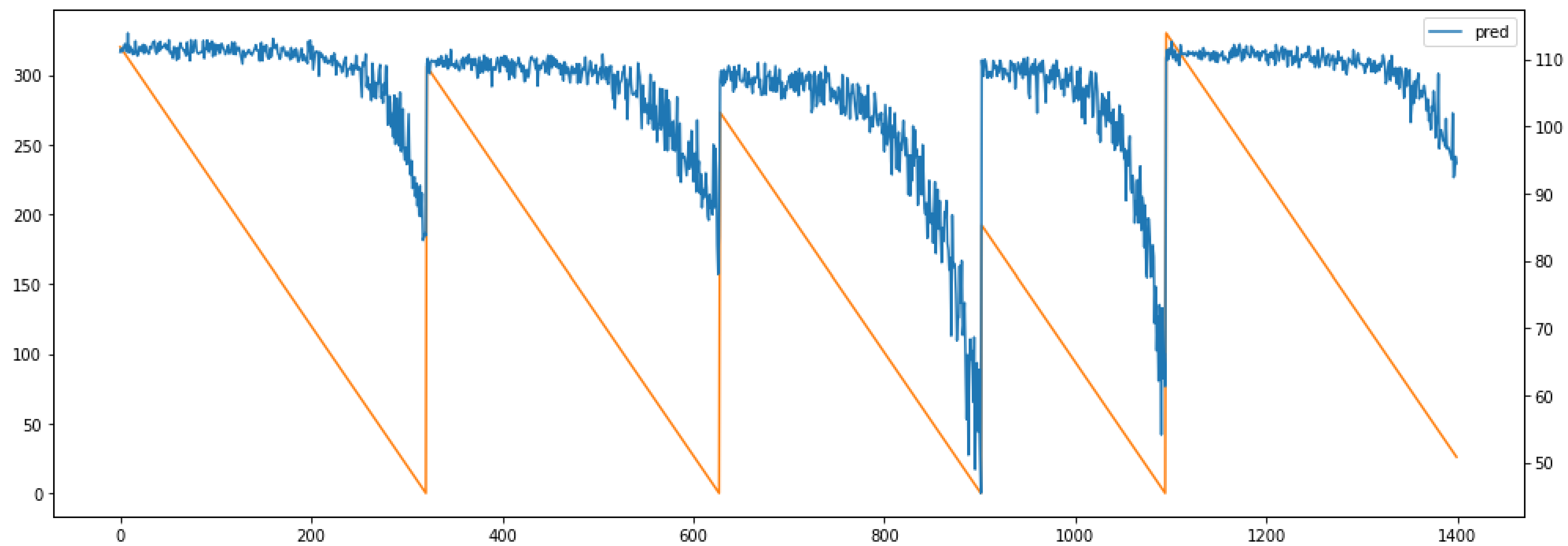
In [22]:
```
trs_pred = nn2.predict(trs_s[dt_in]).ravel() * trmaxrul
cst.plot_rul(trs_pred[:stop], trs["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```



Inspecting the Predictions

Then let's do the same for the unsupervised data

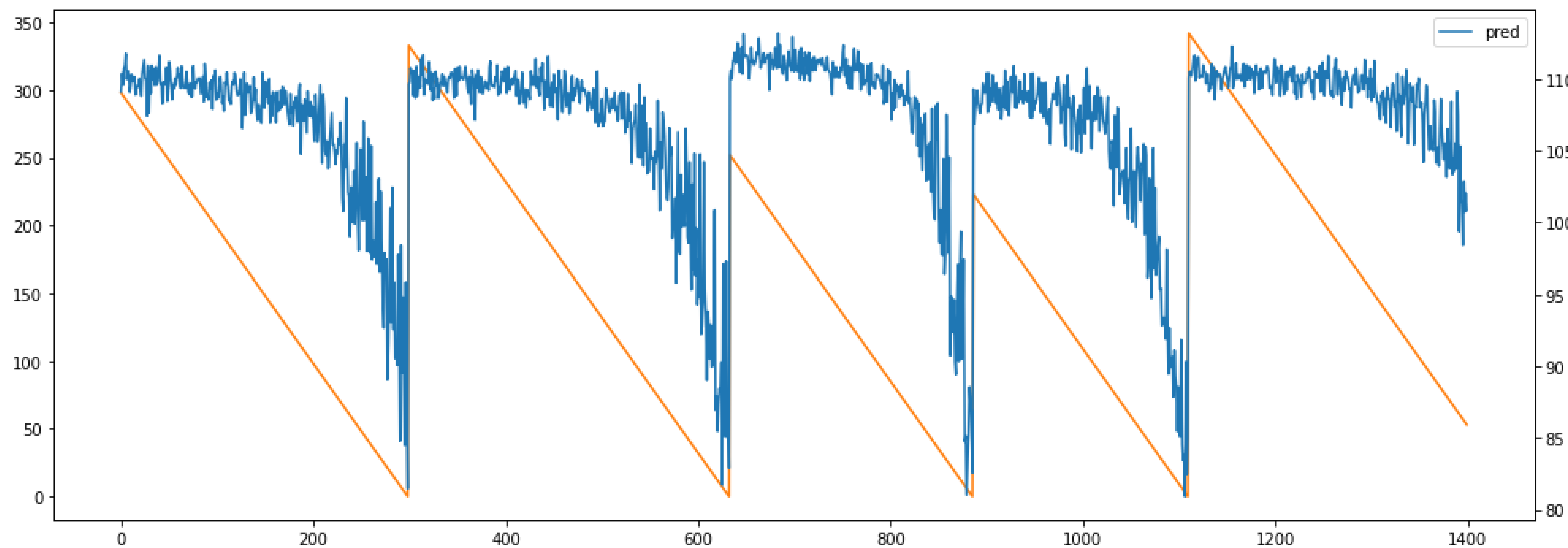
In [23]:
```
tru_pred = nn2.predict(tru_s[dt_in]).ravel() * trmaxrul
cst.plot_rul(tru_pred[:stop], tru["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```



Inspecting the Predictions

Then let's do the same for the test data

In [24]:
```
ts_pred = nn2.predict(ts_s[dt_in]).ravel() * trmaxrul
cst.plot_rul(ts_pred[:stop], ts["rul"].iloc[:stop], same_scale=False, figsize=figsize)
```



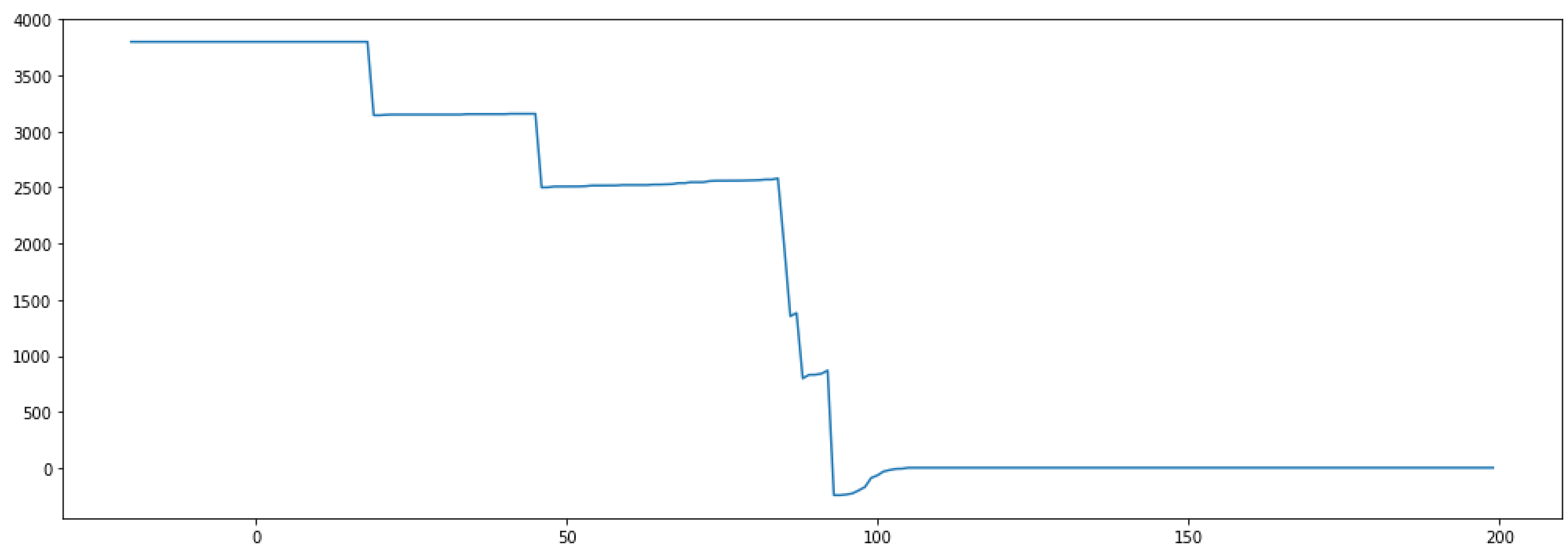
The behavior is more stable and consistent than before

Threshold Optimization and Cost Evaluation

We can now optimize the threshold (on the supervised data)

In [25]:
```
cmodel = cst.RULCostModel(maintenance_cost=maintenance_cost, safe_interval=safe_interval)
th_range = np.arange(-20, 200)
trs_thr = cst.opt_threshold_and_plot(trs_s['machine'].values, trs_pred, th_range, cmodel, figsiz
print(f'Optimal threshold for the training set: {trs_thr}')
```

```
Optimal threshold for the training set: 93
```



Threshold Optimization and Cost Evaluation

Finally, we can evaluate the SBR approach in terms of cost

In [26]:
```
trs_c, trs_f, trs_sl = cmodel.cost(trs_s['machine'].values, trs_pred, trs_thr, return_margin=Tru
tru_c, tru_f, tru_sl = cmodel.cost(tru_s['machine'].values, tru_pred, trs_thr, return_margin=Tru
ts_c, ts_f, ts_sl = cmodel.cost(ts['machine'].values, ts_pred, trs_thr, return_margin=True)
print(f'Cost: {trs_c} (supervised), {tru_c} (unsupervised), {ts_c} (test)')
```

```
Cost: -245 (supervised), -10993 (unsupervised), -4241 (test)
```

In [27]:
```
print(f'Avg. fails: {trs_f/len(trs_mcn):.2f} (supervised), {tru_f/len(tru_mcn):.2f} (unsupervise
print(f'Avg. slack: {trs_sl/len(trs_mcn):.2f} (supervised), {tru_sl/len(tru_mcn):.2f} (unsupervi
```

```
Avg. fails: 0.00 (supervised), 0.04 (unsupervised), 0.05 (test)
Avg. slack: 34.00 (supervised), 30.17 (unsupervised), 29.32 (test)
```

- The number of fails has decreased very significantly
- The slack is still contained

And we did this with just a handful of run-to-failure experiments

Considerations

Regularized approaches for knowledge injection are very versatile

They work as long as we have a good differentiable regularizer

- E.g. negative labels (here we assume a one-hot encoding for the output)
  - Constraint: $round(f_j(\hat{x}_i)) \neq 1$
  - A possbible regularizer: $f_j(\hat{x}_i)$
- E.g. subclass-class relations in multiclass classification
  - Constraint: $round(f_j(\hat{x}_i)) \Rightarrow round(f_k(\hat{x}_i))$ if j is a subclass of k
  - A possible regularizer: $\max\left(0, f_j(\hat{x}_i) - f_k(\hat{x}_i)\right)$
- E.g. logical formulas can be translated into regularizers by mean of fuzzy logic

Choosing the correct regularizer weight can be complicated

- Since the is still improving generalization, we could use a validation set
- However, if supervised data is scarce this may not be practical
- In general: an open research problem

Considerations

Domain knowledge is ubiquitous

- It is sometimes contrasted with deep learning
- ...But isn't it better to use both?

Differentiability may be an issue

- Some constraints are not naturally differentiable
- E.g. say we know that the (binary) classes are roughly balanced
  - The constraint: $\sum_{i=1}^{m} round(f(\hat{x}_i)) = m/2$
  - A possible regularizer: $\left(\sum_{i=1}^{m} f(\hat{x}_i) - m/2\right)^2$
- The penalty can be minimized by balancing the classes...
- ...But also by predicting 0.5 (complete uncertainty) for all examples!
- This is another open research issue