Predict

'Cause we are dealing with ML, ain't we?

Our Motivating Example

Le'ts consider the problem of defining a predictive maintenance policy

We will rely on simplest possible formulation

- First, we estimate the Remaining Useful Life (RUL) of a component
- If it is too low, we trigger a maintenance operation

Overall, we stop when:

$$f(x, \omega) < \theta$$

Where f is a RUL estimator with input $\boldsymbol{x}$ and parameters $\boldsymbol{\omega}$

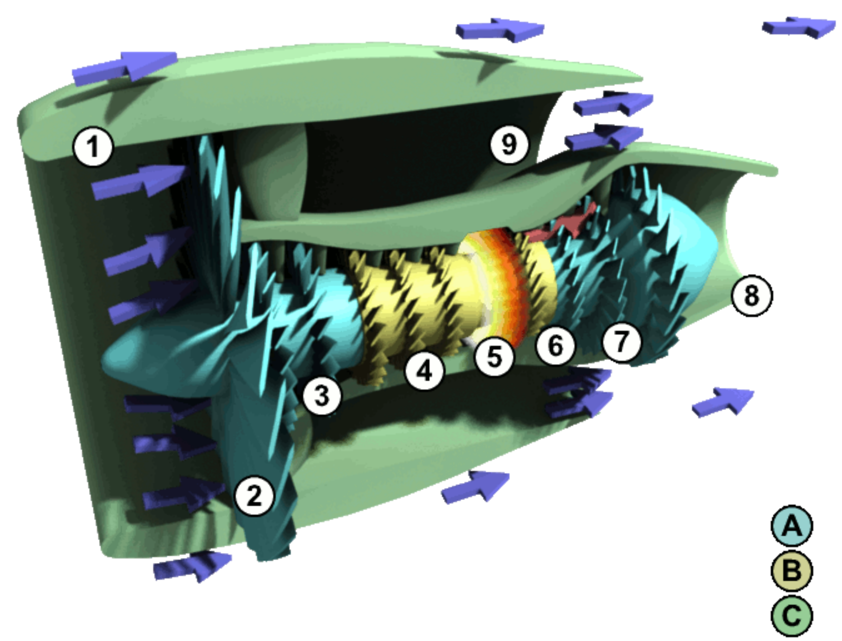
- Specifically, we will use a Neural Network

This will be our first motivating example

The Dataset

We will consider the NASA C-MAPSS dataset

- The Modular Aero-Propulsion System Simulation (MAPSS)
- ...Is a NASA-developed simulator for turbofan engines



- It comes with both a Military (MAPSS) and commercial versionn (C-MAPSS)
- ...Which differ only in the attributes of the considered engines

The Dataset

C-MAPPS was used to simulate a number of faults and defects

...With the goal to build a challenge dataset for the PHM08 conference

- The dataset contains both run-to-failure and truncated experiments
- Only the full experiments are suitable for testing a full policy

The full experiments include four data files:

Dataset	Operating conditions	Fault modes
FD001	1 (sea level)	HPC
FD002	6	HPC
FD003	1 (sea level)	HPC, fan
FD004	6	HPC, fan

- We will focus on the the toughest one, i.e. FD004

Inspecting the Data

Let's have a look at the row data

In [2]:
```
data_sv_dict = util.split_by_field(util.load_cmapss_data(data_folder), field='src')
data = data_sv_dict['train_FD004']
data.head()
```

Out[2]:

	src	machine	cycle	p1	p2	p3	s1	s2	s3	s4	...	s13	s14	s15	s16	s17
0	train_FD004	461	1	42.0049	0.8400	100.0	445.00	549.68	1343.43	1112.93	...	2387.99	8074.83	9.3335	0.02	330
1	train_FD004	461	2	20.0020	0.7002	100.0	491.19	606.07	1477.61	1237.50	...	2387.73	8046.13	9.1913	0.02	361
2	train_FD004	461	3	42.0038	0.8409	100.0	445.00	548.95	1343.12	1117.05	...	2387.97	8066.62	9.4007	0.02	329
3	train_FD004	461	4	42.0000	0.8400	100.0	445.00	548.70	1341.24	1118.03	...	2388.02	8076.05	9.3369	0.02	328
4	train_FD004	461	5	25.0063	0.6207	60.0	462.54	536.10	1255.23	1033.59	...	2028.08	7865.80	10.8366	0.02	305

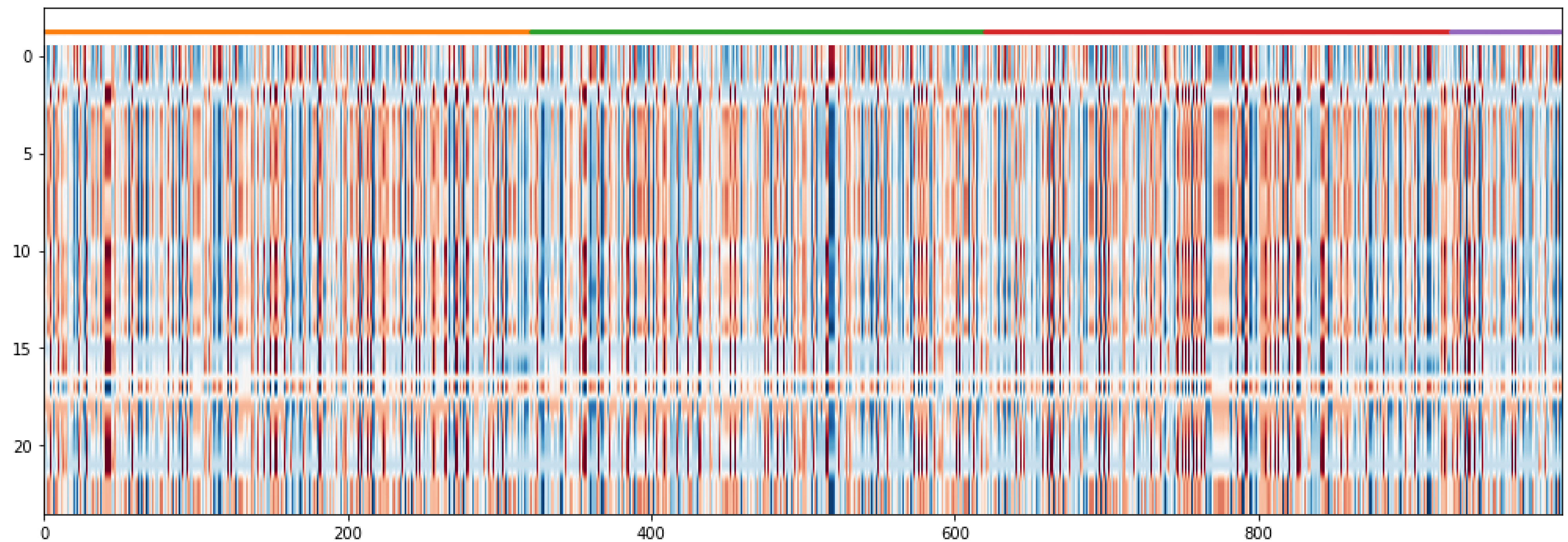
5 rows × 28 columns

- Columns "p1, p2, p3" refer to controlled parameters
- Columns "s1" to "s21" refer to sensor reading
- Binning has already been applied in the original dataset

Inspecting the Data

Let's have a look at a stretch of the dataset, in standardized form

In [3]:
```
dt_in = list(data.columns[3:-1])
tmp = data.iloc[:1000]
util.plot_df_heatmap(tmp[dt_in], labels=tmp['machine'], figsize=figsize)
```

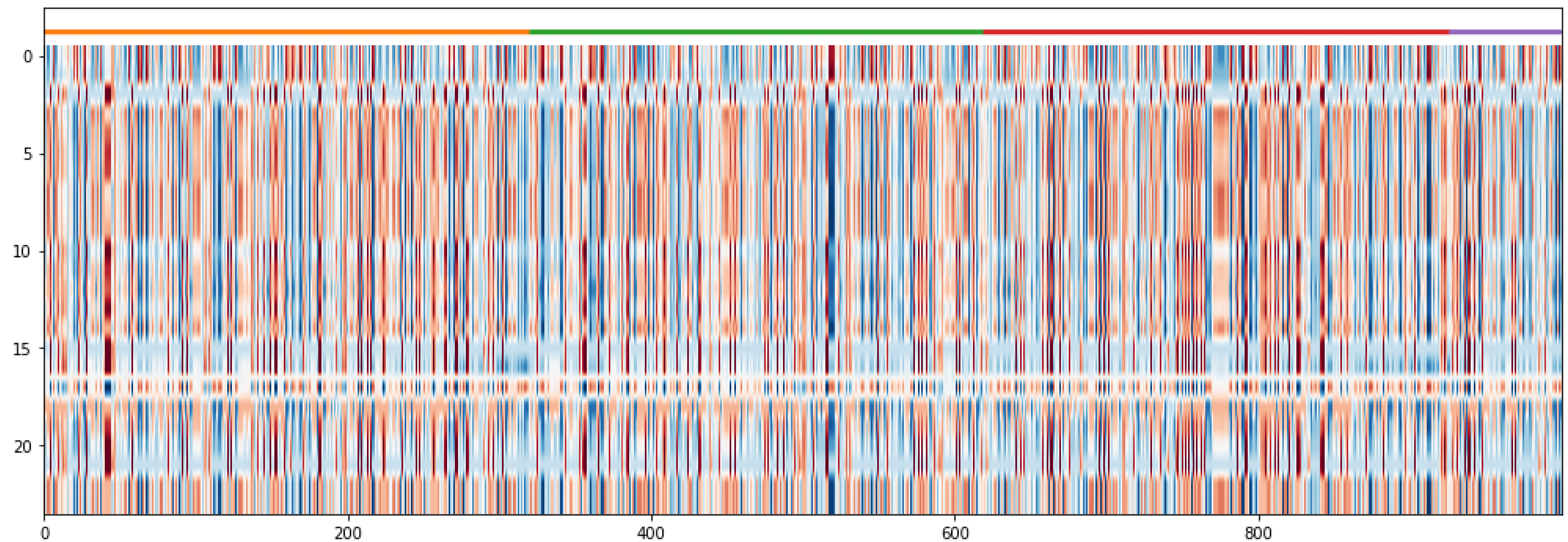


- Each color in the top row identifies a different run-to-failure experiment
- Other rows show values for all the columns (red = low, blue = high)

Inspecting the Data

Let's have a look at a stretch of the dataset, in standardized form

In [4]:
```
dt_in = list(data.columns[3:-1])
tmp = data.iloc[:1000]
util.plot_df_heatmap(tmp[dt_in], labels=tmp['machine'], figsize=figsize)
```



- There is a lot of variability within each experiment
- The sensor reading appear to oscillate very quickly (relatively speacking)

Training and Test Data

We now need to define our training and test data

We will split the available data by whole experiments, not individual examples:

- Some run-to-failure experiments will form the training set
- Others run-to-failure experiments will be used for testing

Let's check how many experiments (machines) we have:

```
In [5]: print(f'Number of machines: {len(data.machine.unique())}')
```

```
Number of machines: 249
```

- This is actually a very large number
- This high number is by far the less realistic aspect of the C-MAPSS dataset

Training and Test Data

Let's use 75% of the machine for training, the rest for testing

First, we partition the machine indexes:

In [6]:
```
tr_ratio = 0.75
np.random.seed(42)
machines = data.machine.unique()
np.random.shuffle(machines)

sep = int(tr_ratio * len(machines))
tr_mcn = machines[:sep]
ts_mcn = machines[sep:]
```

Then, we partition the dataset itself:

In [7]:
```
tr, ts = util.partition_by_machine(data, tr_mcn)
print(f'#Examples: {len(tr)} (traning), {len(ts)} (test)')
print(f'#Experiments: {len(tr["machine"].unique())} (traning), {len(ts["machine"].unique())} (te
```

```
#Examples: 45385 (traning), 15864 (test)
#Experiments: 186 (traning), 63 (test)
```

Standardization/Normalization

Now, we rescale the data via the `rescale_CMAPSS` function

- We will standardize all parameters and sensor inputs:
- ...And we normalize the RUL (it convenient to have it non-negative):

In [8]:
```
tr_s, ts_s, nparams = util.rescale_CMAPSS(tr, ts)
tr_s.describe()
```

Out[8]:

	machine	cycle	p1	p2	p3	s1	s2	s3
count	45385.000000	45385.000000	4.538500e+04	4.538500e+04	4.538500e+04	4.538500e+04	4.538500e+04	4.538500e+04
mean	582.490955	133.323896	2.894775e-16	1.302570e-16	1.178889e-16	4.664830e-15	2.522791e-15	1.727041e-15
std	71.283034	89.568561	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
min	461.000000	1.000000	-1.623164e+00	-1.838222e+00	-2.381839e+00	-1.055641e+00	-1.176507e+00	-1.646830e+00
25%	521.000000	62.000000	-9.461510e-01	-1.031405e+00	4.198344e-01	-1.055641e+00	-8.055879e-01	-6.341243e-01
50%	585.000000	123.000000	6.868497e-02	4.154560e-01	4.198344e-01	-3.917563e-01	-6.336530e-01	-4.718540e-01
75%	639.000000	189.000000	1.218855e+00	8.661917e-01	4.198344e-01	6.926385e-01	7.407549e-01	7.495521e-01
max	708.000000	543.000000	1.219524e+00	8.726308e-01	4.198344e-01	1.732749e+00	1.741030e+00	1.837978e+00

8 rows × 27 columns

Regression Model

We will use a feed-forward neural network (MLP) for regression

The code to build the model is in the `build_ml_model`:

```
# Build all layers
ll = [keras.Input(input_size)]
for h in hidden:
    ll.append(layers.Dense(h, activation='relu'))
ll.append(layers.Dense(output_size, activation=output_activation))
# Build the model
model = keras.Sequential(ll, name=name)
```

In [9]:
```
hidden = [32]
nn = util.build_ml_model(input_size=len(dt_in), output_size=1, hidden=hidden)
```

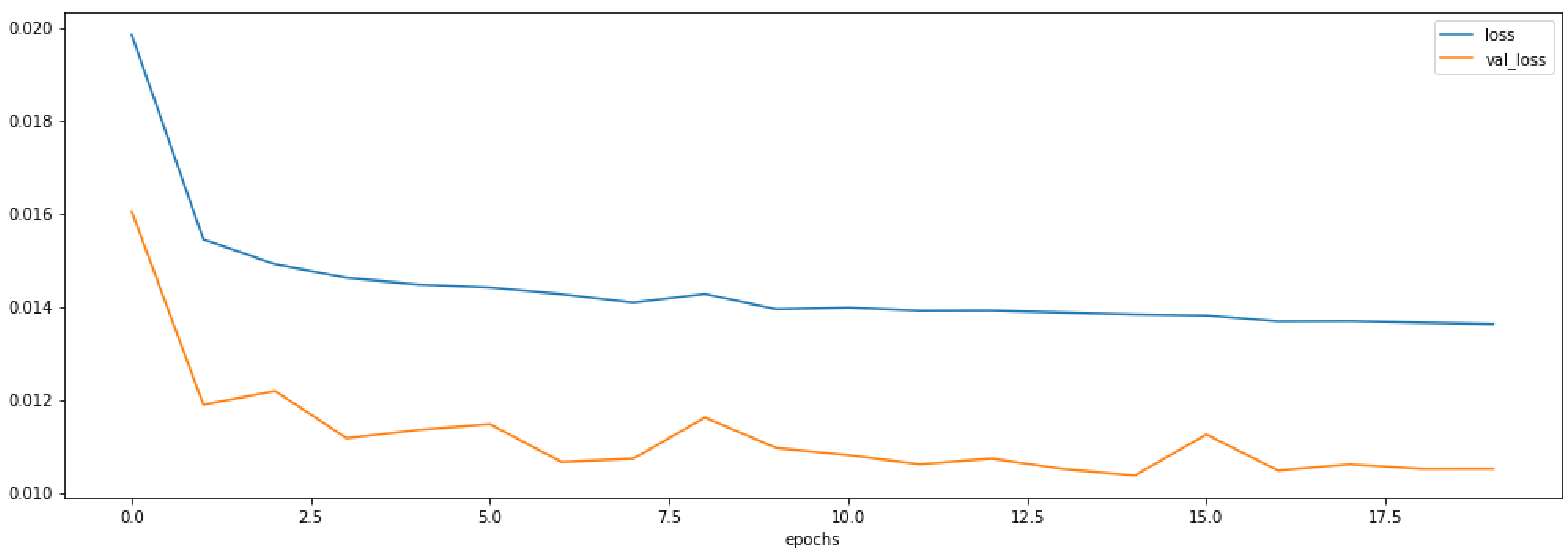
- The `hidden` argument is a list of sizes for the hidden layers
- By setting `hidden=[]` we would obtain a Linear Regression approach

Training

We can now train our model (using patience to control overfitting)

In [10]:
```
history = util.train_ml_model(nn, tr_s[dt_in], tr_s['rul'], epochs=20, validation_split=0.2)
nn.save('rul_regressor')
util.plot_training_history(history, figsize=figsize)
trl, vll = history.history["loss"][-1], np.min(history.history["val_loss"])
print(f'Final loss: {trl:.4f} (training), {vll:.4f} (validation)')
```

```
INFO:tensorflow:Assets written to: rul_regressor/assets
```

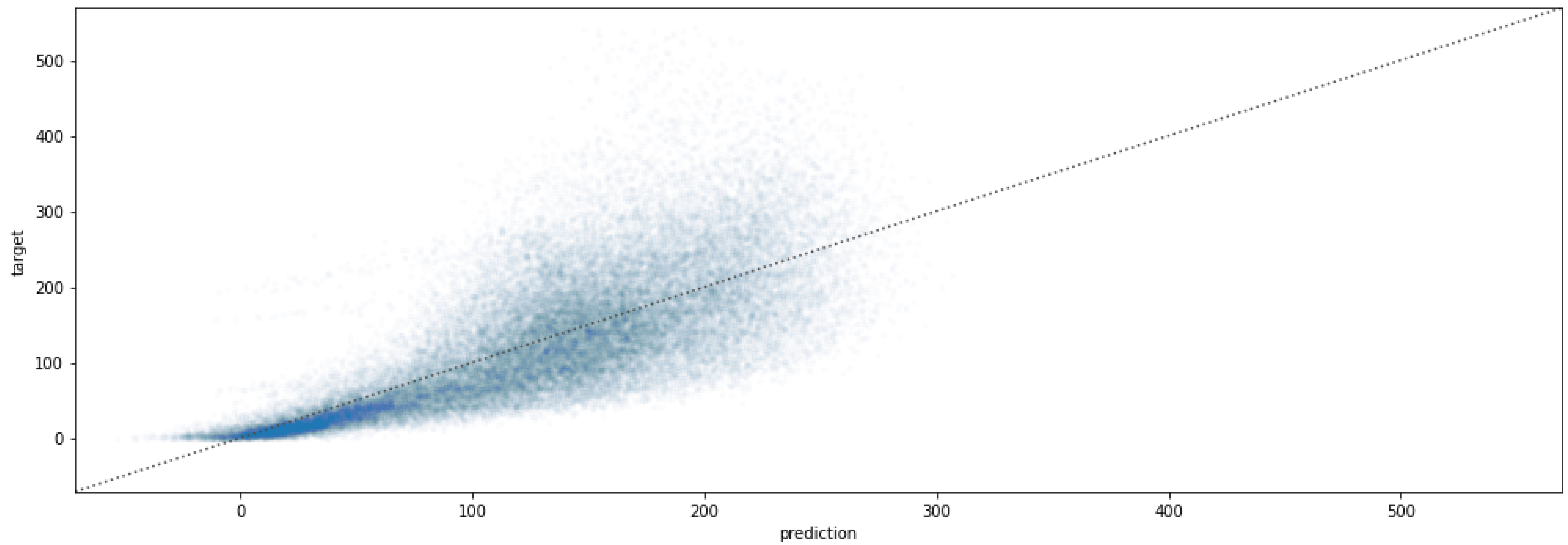



```
Model loss: 0.0138 (training) 0.0104 (validation)
Final loss: 0.0136 (training), 0.0104 (validation)
```

Predictions

...And finally check the prediction quality on the training set

In [11]:
```
tr_pred = nn.predict(tr_s[dt_in]).ravel() * nparams['trmaxrul']
util.plot_pred_scatter(tr_pred, tr['rul'], figsize=figsize)
print(f'R2 score: {r2_score(tr["rul"], tr_pred):.2}')
```




```
R2 score: 0.53
```


Predictions