Applicability of DFL

Let's Second-Guess Ourselvers

However, let's not discount the prediction-focused approach yet

In fact, it's easy to see that:

$$\mathbb{E}[\text{regret}(y, \hat{y})] \xrightarrow[\mathbb{E}[L(y,\hat{y})]\to 0]{} 0$$

Intuitively:

- The more accurate we can be, the lower the regret
- Eventually, perfect predictions will result in 0 regret

Let's Second-Guess Ourselvers

However, let's not discount the prediction-focused approach yet

In fact, it's easy to see that:

$$\mathbb{E}[\text{regret}(y, \hat{y})] \xrightarrow[\mathbb{E}[L(y,\hat{y})]\to 0]{} 0$$

Intuitively:

- The more accurate we can be, the lower the regret
- Eventually, perfect predictions will result in 0 regret

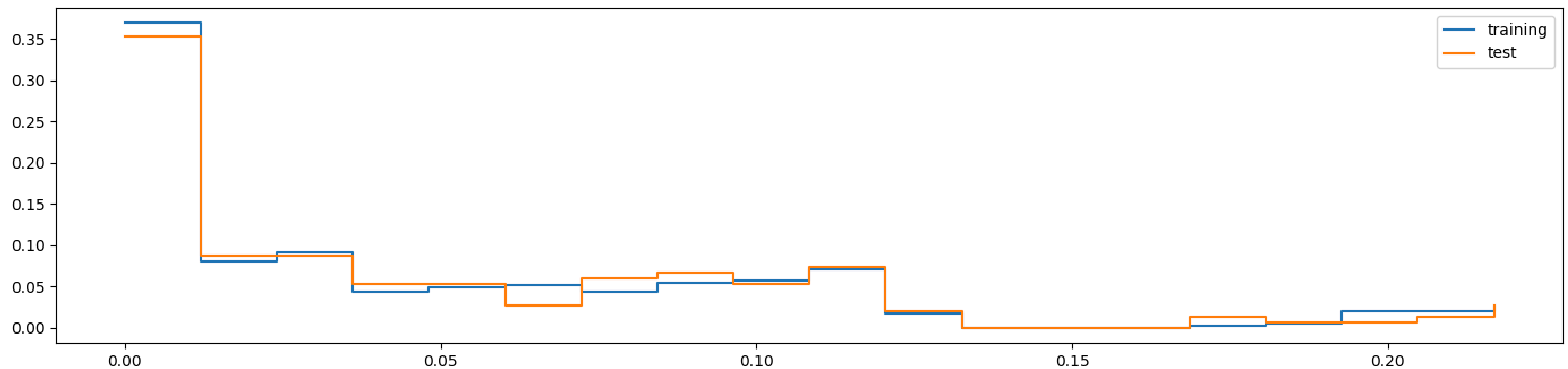
But then... What if we make our model bigger?

- We could get good predictions and good regret
- ...And training would be much faster

Our Baseline

Let's check again the results for our PFL linear regressor

In [39]:
```
pfl = util.build_ml_model(input_size=1, output_size=nitems, hidden=[], name='pfl_det', outpu
history = util.train_ml_model(pfl, data_tr.index.values, data_tr.values, epochs=1000, loss=
r_tr = util.compute_regret(prb, pfl, data_tr.index.values, data_tr.values)
r_ts = util.compute_regret(prb, pfl, data_ts.index.values, data_ts.values)
util.plot_histogram(r_tr, figsize=figsize, label='training', data2=r_ts, label2='test', pri
```



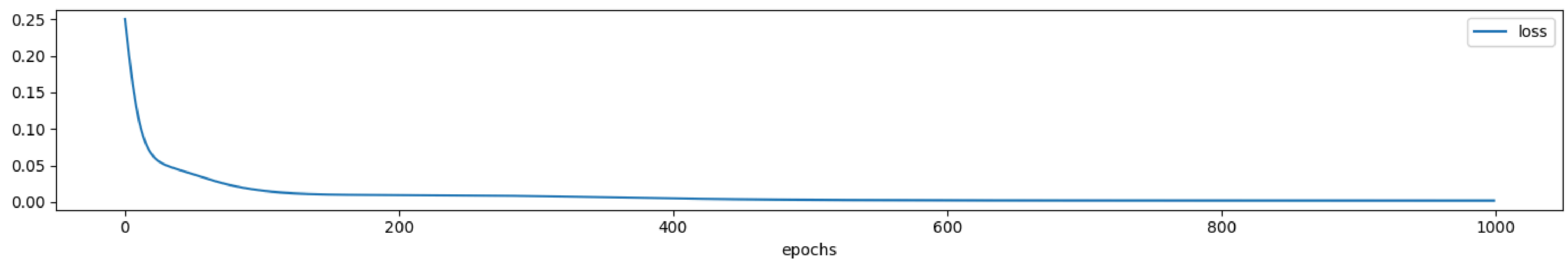

```
Mean: 0.052 (training), 0.052 (test)
```

This will be our main baseline

PFL Strikes Back

Let's try to use a non-linear model

```
In [40]: pfl_acc = util.build_ml_model(input_size=1, output_size=nitems, hidden=[8], name='pfl_det_ac
         history = util.train_ml_model(pfl_acc, data_tr.index.values, data_tr.values, epochs=1000, lo
         util.plot_training_history(history, figsize=figsize_narrow, print_scores=False, print_time=T
         util.print_ml_metrics(pfl_acc, data_tr.index.values, data_tr.values, label='training')
         util.print_ml_metrics(pfl_acc, data_ts.index.values, data_ts.values, label='test')
```



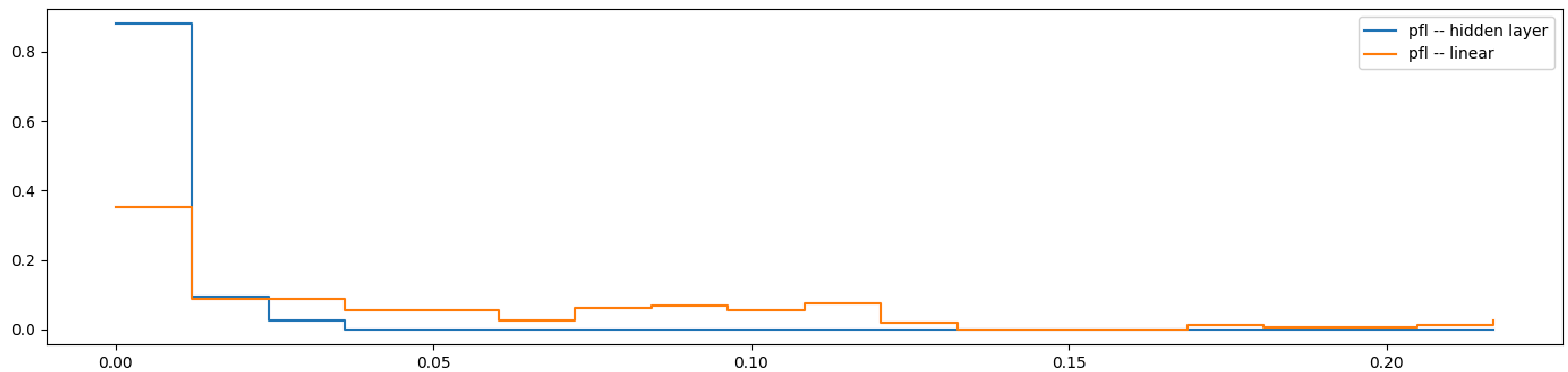

```
Training time: 12.5487 sec
R2: 0.98, MAE: 0.03, RMSE: 0.04 (training)
R2: 0.98, MAE: 0.03, RMSE: 0.04 (test)
```

More accurate, it is!

PFL Strikes Back

...And the improvement in terms of regret is remarkable

In [41]:
```
r_ts_acc = util.compute_regret(prb, pfl_acc, data_ts.index.values, data_ts.values)
util.plot_histogram(r_ts_acc, figsize=figsize, label='pfl -- hidden layer', data2=r_ts, labe
```

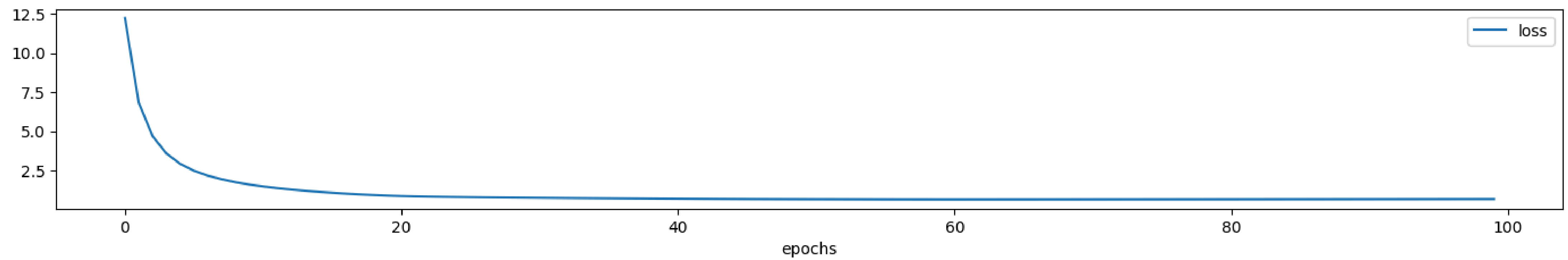



```
Mean: 0.004 (pfl -- hidden layer), 0.052 (pfl -- linear)
```

PFL Strikes Back

Let's what DFL could do with the same model coplexity

In [44]:
```
spo_nonlinear = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidde
history = util.train_dfl_model(spo_nonlinear, data_tr.index.values, data_tr.values, epochs=1
util.plot_training_history(history, figsize=figsize_narrow, print_scores=False, print_time=T
util.print_ml_metrics(spo_nonlinear, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(spo_nonlinear, data_ts.index.values, data_ts.values, label='test')
```



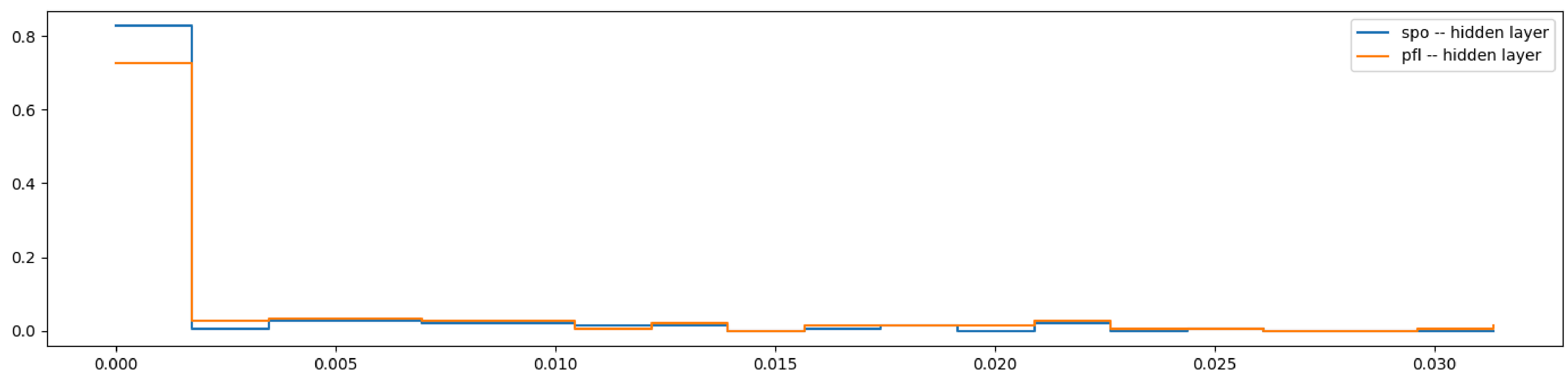

```
Training time: 52.3219 sec
R2: -0.29, MAE: 0.23, RMSE: 0.28 (training)
R2: -0.29, MAE: 0.23, RMSE: 0.28 (test)
```

PFL Strikes Back

The gap between the approach is basically closed

In [45]:

```
r_ts_spo_nonlinear = util.compute_regret(prb, spo_nonlinear, data_ts.index.values, data_ts.\
fig = util.plot_histogram(r_ts_spo_nonlinear, figsize=figsize, label='spo -- hidden layer',
fig.savefig('pfl_dfl_shallow.pdf')
```




```
Mean: 0.002 (spo -- hidden layer), 0.004 (pfl -- hidden layer)
```

Evening the Field

Can't we do anything about it?

- DFL predictions will always be off (more or less)
- ...But there are ways to make the approach faster

For example:

- You can use a relaxation, e.g. the LP relaxation of a MILP
- You can limit recomputation by caching past solutions
- You can warm start the DFL approach with the PFL weights

Let's see the last two tricks in deeper detail

Warm Starting and Solution Caching

Warm starting simple consists in using the PFL weights to initialize θ

Since accuracy is correlated with regret, this might accelerate convergence

Warm Starting and Solution Caching

Warm starting simple consists in using the PFL weights to initialize θ

Since accuracy is correlated with regret, this might accelerate convergence

Solution caching is applicable if the feasible space is fixed

I.e. to problems in the form:

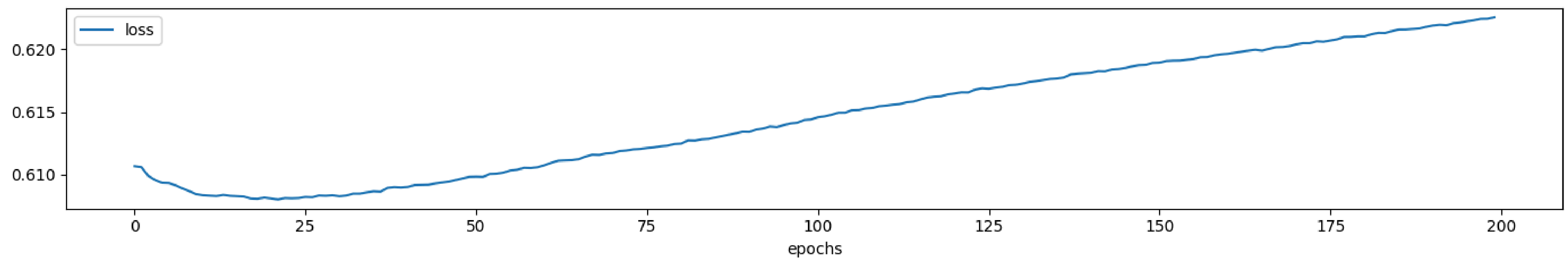
$$z^*(y) = \text{argmin}_z \{ f(z) \mid z \in F \}$$

- During training, we maintain a solution cache S
- Initially, we populate S with the true optimal solutions $z^*(y_i)$ for all examples
- Before computing $z^*(\hat{y})$ for the current prediction we flip a coin
- With probability p, we run the computation (and store any new solution in S)
- With probability $1 - p$, we solve instead $\hat{z}^*(\hat{y}) = \text{argmin}_z \{ f(z) \mid z \in S \}$

Speeding Up DFL

Let's use DFL with linear regression, a warm start, and a solution cache

In [46]:
```
spo = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[], name
history = util.train_dfl_model(spo, data_tr.index.values, data_tr.values, epochs=200, verbos
util.plot_training_history(history, figsize=figsize_narrow, print_scores=False, print_time=
util.print_ml_metrics(spo, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(spo, data_ts.index.values, data_ts.values, label='test')
```



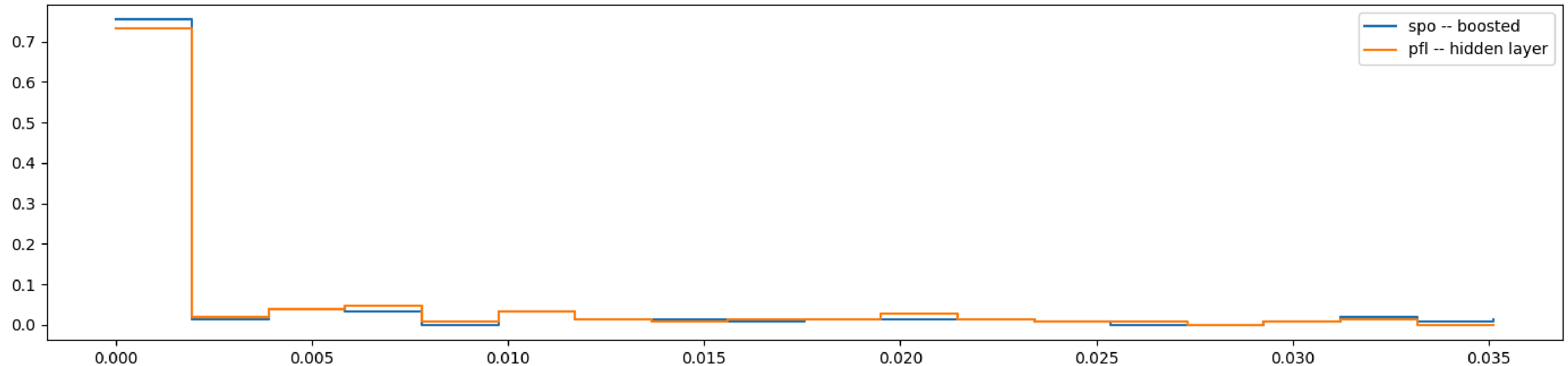

```
Training time: 22.7765 sec
R2: 0.64, MAE: 0.12, RMSE: 0.16 (training)
R2: 0.64, MAE: 0.12, RMSE: 0.16 (test)
```

The training time is still large, but much lower than our earlier DFL attempt

Speeding Up DFL

And the regret is even better than before!

In [49]:
```
r_ts_spo = util.compute_regret(prb, spo, data_ts.index.values, data_ts.values)
util.plot_histogram(r_ts_spo, figsize=figsize, label='spo -- boosted', data2=r_ts_acc, labe
```




```
Mean: 0.004 (spo -- boosted), 0.004 (pfl -- hidden layer)
```

- We are matching the more complex PFL model with a simple linear regressor
- ...And the training time is much better than before

Reflecting on What we Have

Therefore, DFL gives us at least two benefits

First, it can lead to lower regret compared to a prediction-focused appraoch

- As the models become more complex we have diminishing returns
- ...But for some applications every little bit counts

Second, it may allow using simpler ML models

- Simple models are faster to evaluate
- ...But more importantly they are easier to explain
- E.g. we can easily perform feature importance analysis

Reflecting on What we Have

Therefore, DFL gives us at least two benefits

First, it can lead to lower regret compared to a prediction-focused appraach

- As the models become more complex we have diminishing returns
- ...But for some applications every little bit counts

Second, it may allow using simpler ML models

- Simple models are faster to evaluate
- ...But more importantly they are easier to explain
- E.g. we can easily perform feature importance analysis

Intuitively, DFL works best where PFL has estimation issues

Can we exploit this fact to maximize our advantage?