



Rambling Away from Decision Focused Learning

A circuitous investigation of what DFL can do
if you keep pushing at its limits



What I'll present is the result of joint work!

Many thanks to: Senne Berden, Victor Bucarey, Allegra De Filippo, Michelangelo Diligenti, Tias Guns, Jayanta Mandi, Irfan Mahmutogullari, Michela Milano, Maxime Mulamba, Mattia Silvestri





01. Getting Started



Getting Started

As stated, our starting point is Decision Focused Learning

Specifically the SPO formulation, where we focus on problems in the form:

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

- z is the set of decisions (numeric or discrete)
- F is the feasible space
- y is a cost vector, which is not directly measureable

Rather than to y , we have access to an observable x

- Based on x , we can attempt to train a parametric estimator $h(x, \theta)$
- ...Using training examples $\{(x_i, y_i)\}_{i=1}^m$



A Possible Example

For example, we may have to deal with routing problem

We need to select the best path to reach our destination

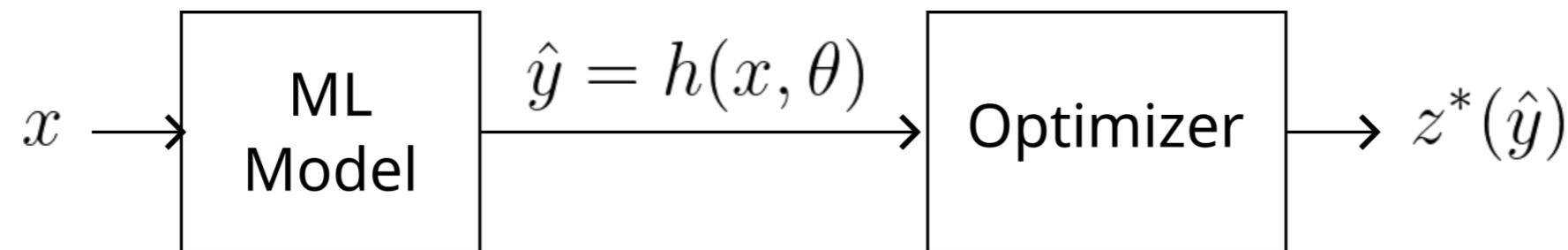


- We don't know the current state of the traffic
- But we can guess! E.g. based on the time, weather, etc.



Inference

This setup involves using the estimator and the optimizer in sequence



At inference time:

- We observe x
- We evaluate our estimator $h(x, \theta)$ to obtain y
- We solve the problem to obtain $z^*(y)$

Overall, the process consists in evaluating:

$$z^*(h(x, \theta))$$



A Two-phase Approach

We can use supervised learning for the estimator

Formally, we obtain an optimal parameter vector by solving:

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{(x,y) \sim P(X,Y)} [L(\hat{y}, y)] \mid \hat{y} = h(x, \theta) \}$$

- Where L is a suitable loss function (e.g. a squared error)
- We'll refer to this as a prediction-focused approach

However, using supervised learning is suboptimal

- A small mistake in terms of L
- ...May lead the optimizer to choosing a poor solution

The root of the issue is a misalignment between the cost metric at training and inference time



Spotting Trouble

Let's see this in action on a toy problem

Consider this two-variable optimization problem:

$$\operatorname{argmin}_z \{ y_0 z_0 + y_1 z_1 \mid z_0 + z_1 = 1 \}$$

Let's assume that the true relation between x (a scalar) and y is:

$$y_0 = 2.5x^2$$

$$y_1 = 0.3 + 0.8x$$

...But that we can only learn this model with a scalar weight θ :

$$\hat{y}_0 = \theta^2 x$$

$$\hat{y}_1 = 0.5\theta$$



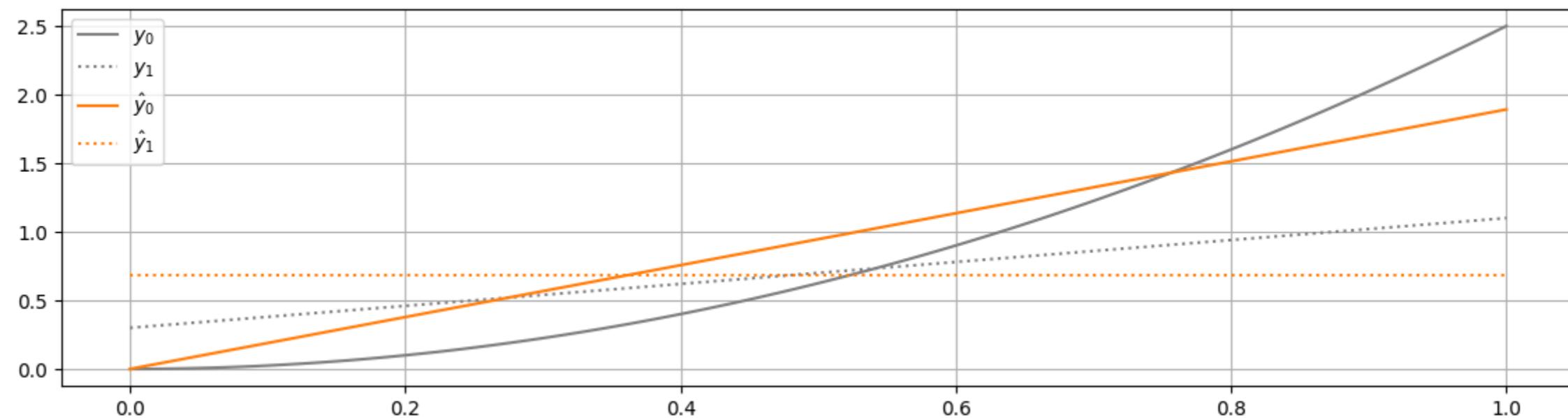
Our model cannot represent the true relation exactly

Spotting Trouble

This is what we get from supervised learning with uniformly distribute data:

```
In [15]: util.draw(w=None, figsize=figsize, model=1)
```

Optimized theta: 1.375



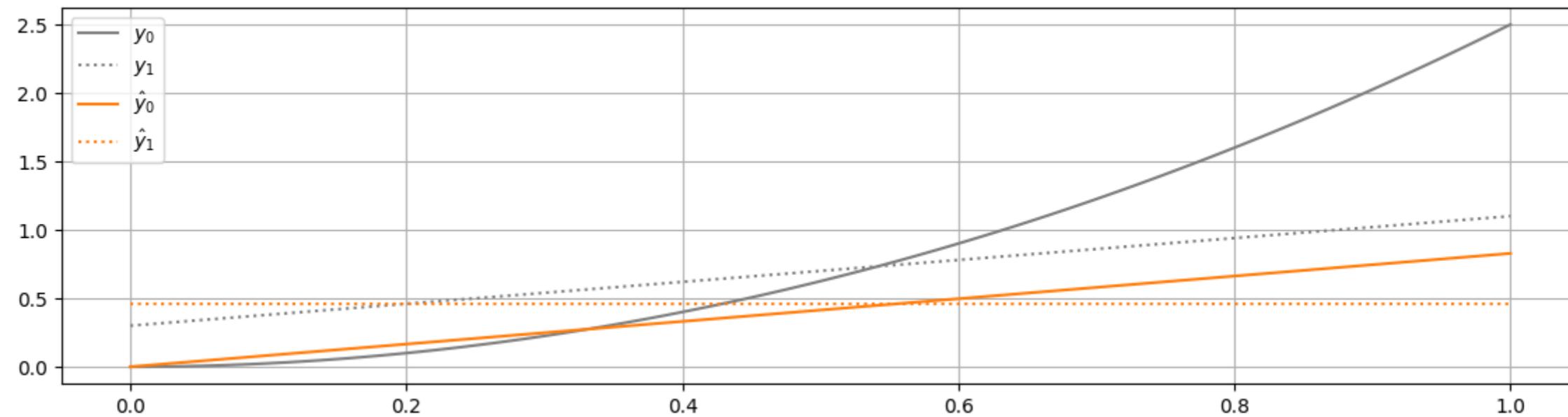
- The crossing point of the grey lines is where we should pick item 0 instead of 1
- The orange lines (trained model) miss it by a wide margin



Not All is Lost

However, we can sidestep the issue by **disregarding accuracy**

```
In [16]: util.draw(w=0.91, figsize=figsize, model=1)
```



- If we focus on choosing θ to match the crossing point
- ...We lead the optimizer to consistently making the correct choice



The Main DFL Idea

DFL attempts to achieve this by using a task-based loss at training time

There's some consensus on this "holy grail" training problem:

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{(x,y) \sim P(X,Y)} [\text{regret}(\hat{y}, y)] \mid \hat{y} = h(x, \theta) \}$$

Where in our setting we have:

$$\text{regret}(\hat{y}, y) = y^T z^*(\hat{y}) - y^T z^*(y)$$

- $z^*(\hat{y})$ is the best solution with the **estimated** costs
- $z^*(y)$ is the best solution with the **true** costs

Intuitively, we want to **lose as little as possible** w.r.t. the best we could do

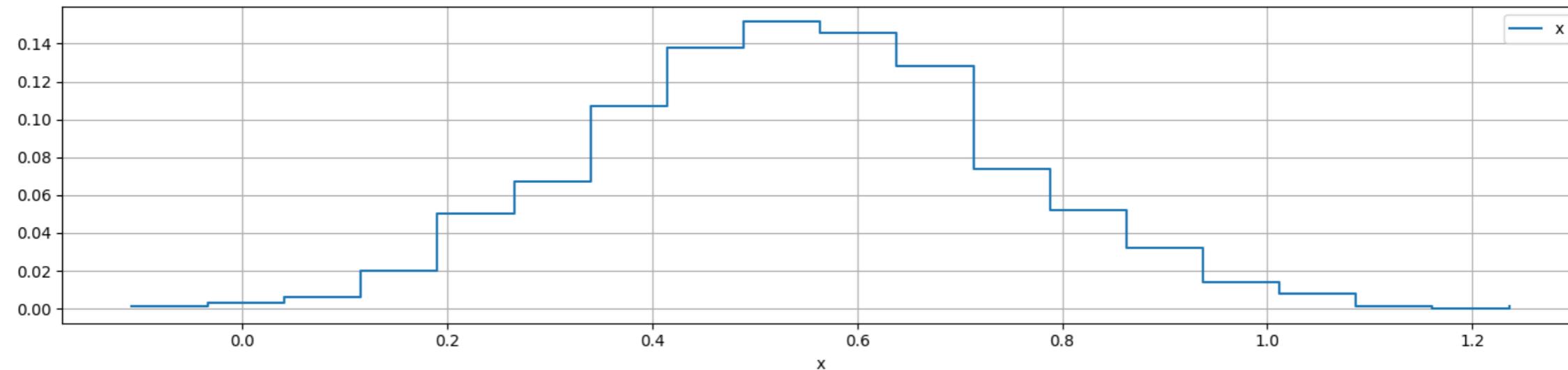
One of the main challenges in DFL is dealing with this loss



Knowing Regret

To see this, let's push our example a little further

```
In [17]: x = util.normal_sample_(mean=0.54, std=0.2, size=1000)  
util.plot_histogram(x, figsize=figsize, label='x')
```



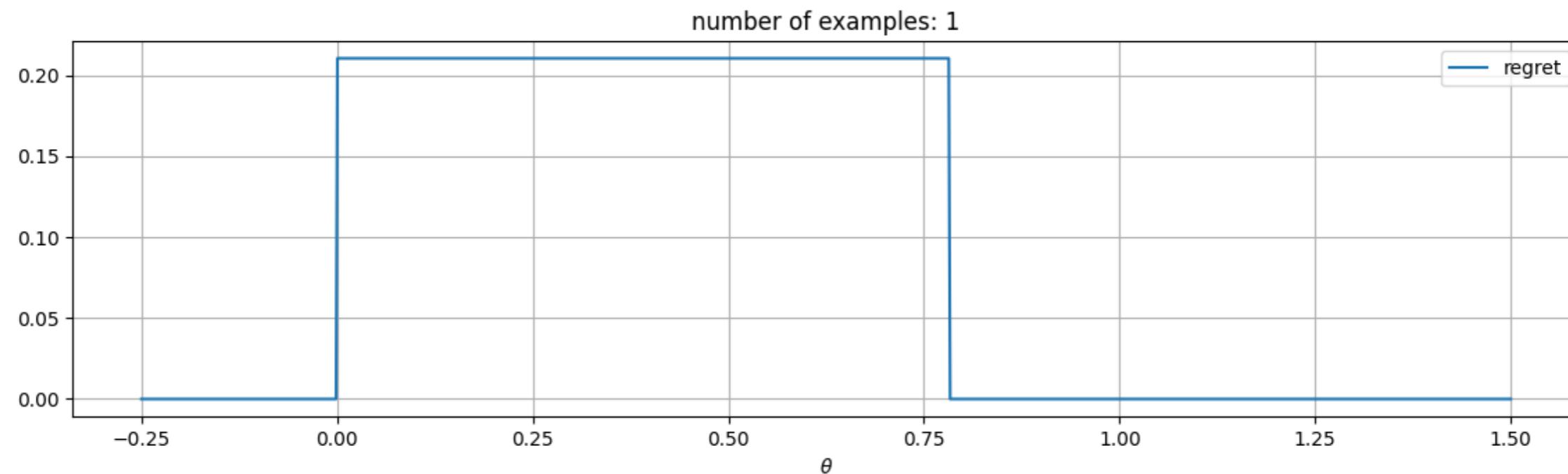
- Say we have access to a normally distributed collection of x values
- ...And to the corresponding true values y



Knowing Regret

This is how the regret looks like for a single example

```
In [18]: util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=1, figsize=figsize)
```



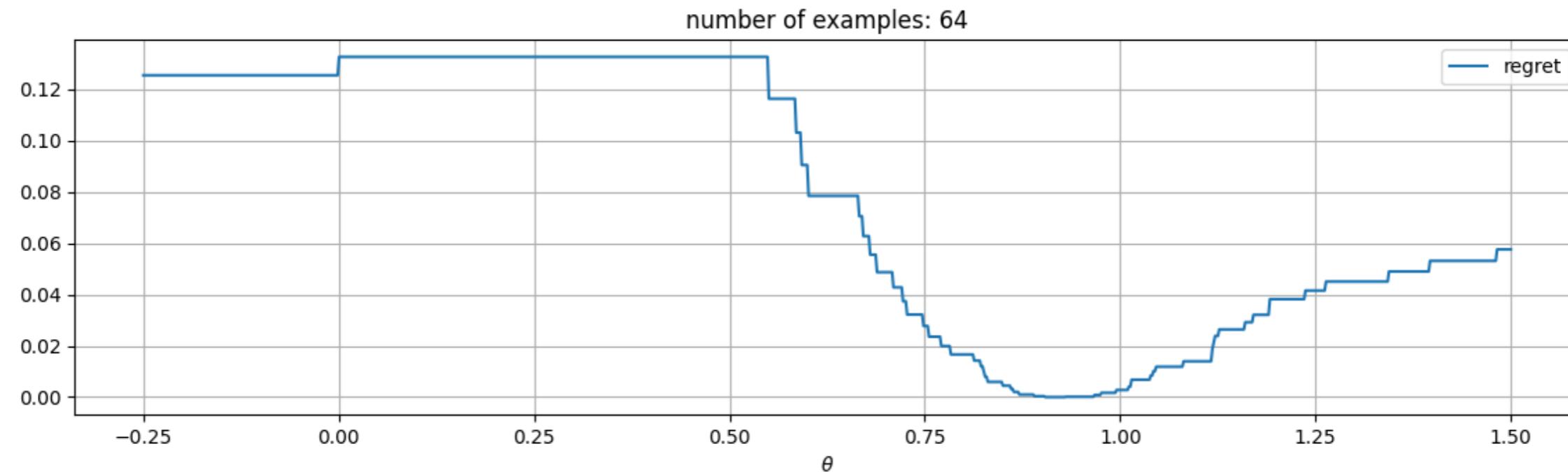
- If $f(x, \theta)$ leads to the correct decision, the regret is 0
- Otherwise we have some non-null value



Knowing Regret

...And this is the same for a larger sample

```
In [19]: util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=64, figsize=f)
```



- For linear problems and finite samples the regret function is **piecewise constant**
- ...Which makes a direct use of gradient descent impossible



SPO+ Loss

A lot of research in the DFL field is about addressing this problem

We will just recap the SPO+ loss from [1], which is (roughly) defined as:

$$\text{spo}^+(\hat{y}, y) = \hat{y}_{spo}^T z^*(y) - \hat{y}_{spo}^T z^*(\hat{y}_{spo}) \quad \text{with: } \hat{y}_{spo} = 2\hat{y} - y$$

There are two main ideas here:

The first it to see what happens with the predicted (not the true) costs

- We know $z^*(\hat{y}_{spo})$ is the optimal solution for \hat{y}_{spo}
- But we wish for $z^*(y)$ to be optimal instead
- Therefore if $\hat{y}_{spo}^T z^*(y) > \hat{y}_{spo}^T z^*(\hat{y}_{spo})$ we give a penalty

With this trick, a differentiable term (i.e. \hat{y}_{spo}) appears in the loss

[1] Elmachtoub, Adam N., and Paul Grigas. "Smart "predict, then optimize"." *Management Science* 68.1 (2022): 9-26.



SPO+ Loss

A lot of research in the DFL field is about addressing this problem

We will just recap the SPO+ loss from [1], which is (roughly) defined as:

$$\text{spo}^+(\hat{y}, y) = \hat{y}_{spo}^T z^*(y) - \hat{y}_{spo}^T z^*(\hat{y}_{spo}) \quad \text{with: } \hat{y}_{spo} = 2\hat{y} - y$$

There are two main ideas here:

The second is to avoid using the estimates y directly

- We rely instead on an altered cost vector, i.e. \hat{y}_{spo}
- Using \hat{y}_{spo} directly would result in a **local minimum** for $\hat{y} = 0$
- With \hat{y}_{spo} , the local minimum is in a location _that depends on \hat{y}

We'll try to visualize this phenomenon

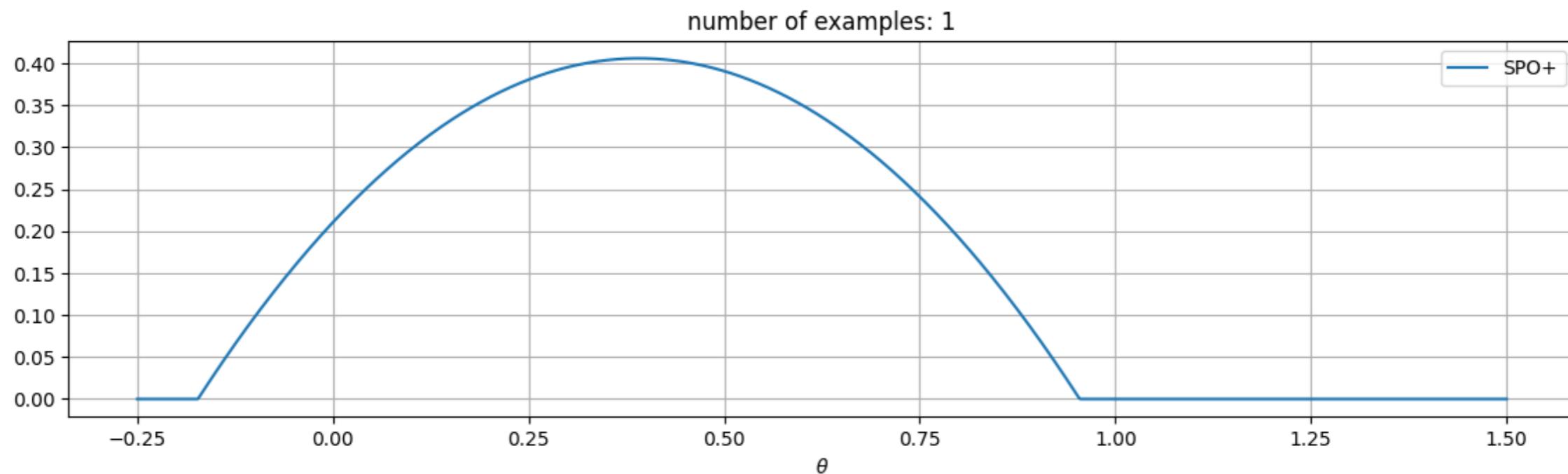
[1] Elmachtoub, Adam N., and Paul Grigas. "Smart ‘predict, then optimize’." *Management Science* 68.1 (2022): 9-26.



SPO+ Loss

This is the SPO+ loss for a single example on our toy problem

```
In [20]: util.draw_loss_landscape(losses=[util.SPOPlusLoss()], model=1, seed=42, batch_size=1, figsize=figsize)
```



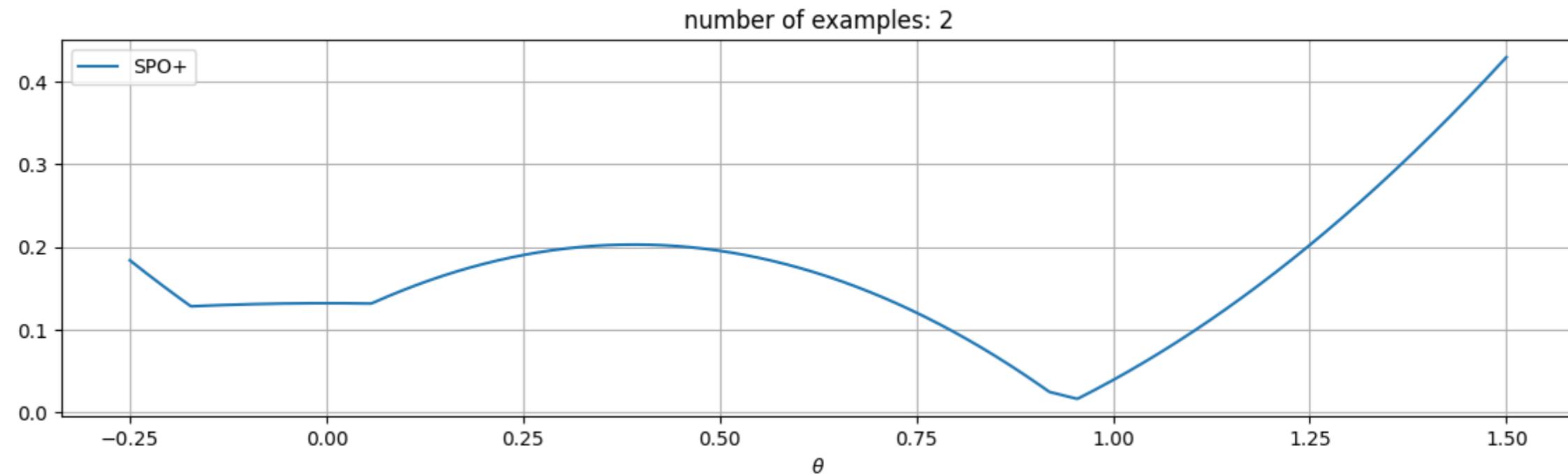
- As expected, there are two local minima



SPO+ Loss

This is the SPO+ loss for a two examples

```
In [21]: util.draw_loss_landscape(losses=[util.SPOPlusLoss()], model=1, seed=42, batch_size=2, figsize=f)
```



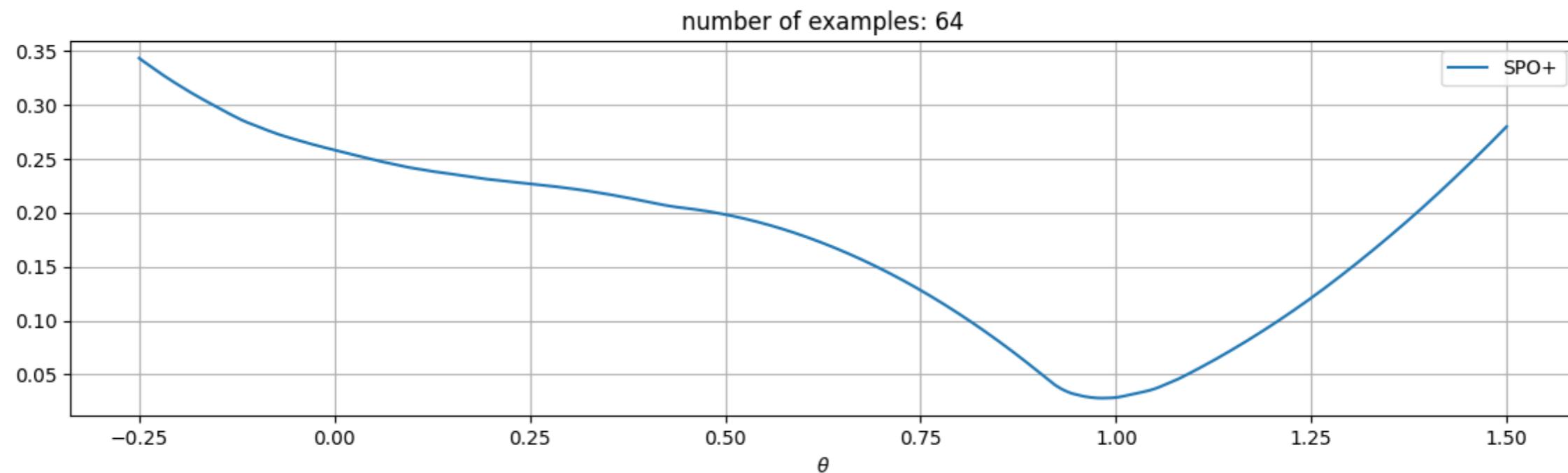
- The "good" local minima for both examples are roughly in the same place
- The "spurious" local minima fall in different position



SPO+ Loss

Over many example, the spurious local minima tend to cancel out

```
In [22]: util.draw_loss_landscape(losses=[util.SPOPlusLoss()], model=1, seed=42, batch_size=64, figsize=1)
```



- This effect is **invaluable** when training with gradient descent



A (Slightly) More Complex Example

Let's see the approach in action on a second example

We will consider this simple optimization problem:

$$z^*(y) = \operatorname{argmin}\{ y^T z \mid v^T z \geq r, z \in \{0, 1\}^n \}$$

- We need to decide which of a set of jobs to accept
- Accepting a job ($z_j = 1$) provides immediate value v_j
- The cost y_j of the job is not known
- ...But it can be estimated based on available data

```
In [23]: nitems, rel_req, seed = 20, 0.5, 42
prb = util.generate_problem(nitems=nitems, rel_req=rel_req, seed=seed)
display(prb)
```

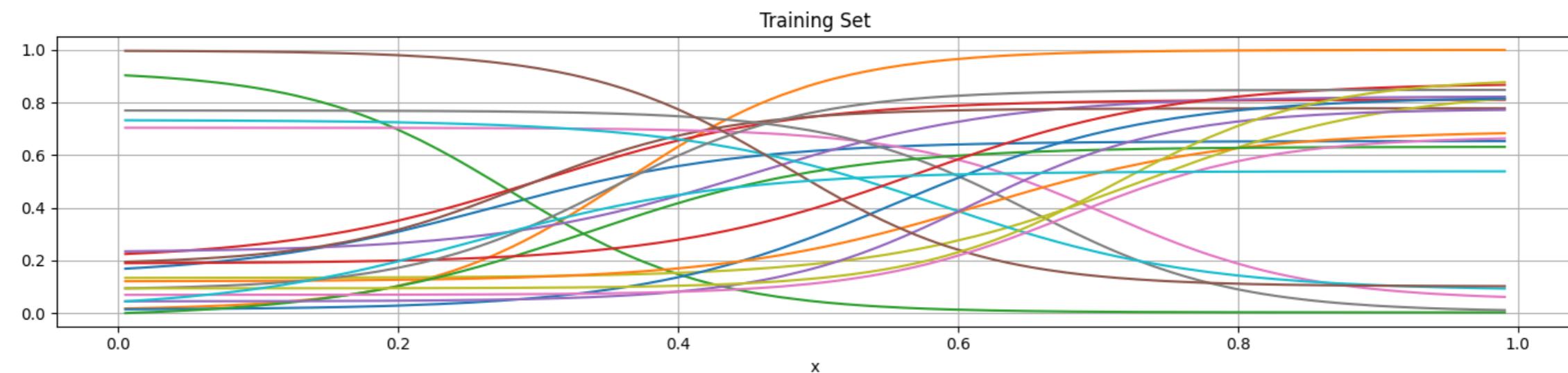
```
ProductionProblem(values=[1.14981605 1.38028572 1.29279758 1.23946339 1.06240746 1.06239781
 1.02323344 1.34647046 1.240446 1.28322903 1.0082338 1.38796394
 1.33297706 1.08493564 1.07272999 1.0733618 1.1216969 1.20990257
 1.17277801 1.11649166], requirement=11.830809153591138)
```



A (Slightly) More Complex Example

Next, we generate some training (and test) data

```
In [24]: data_tr = util.generate_costs(nsamples=350, nitems=nitems, seed=seed, noise_scale=0, noise_type='uniform', sampling_type='uniform', sampling_size=1, sampling_seed=seed)
data_ts = util.generate_costs(nsamples=150, nitems=nitems, seed=seed, sampling_seed=seed+1, noise_type='uniform', noise_scale=0.05, sampling_type='uniform', sampling_size=1, sampling_seed=seed+1)
util.plot_df_cols(data_tr, figsize=figsize, title='Training Set')
```



- We assume that costs can be estimated based on an scalar observable x
- The set of least expensive jobs changes considerably with x



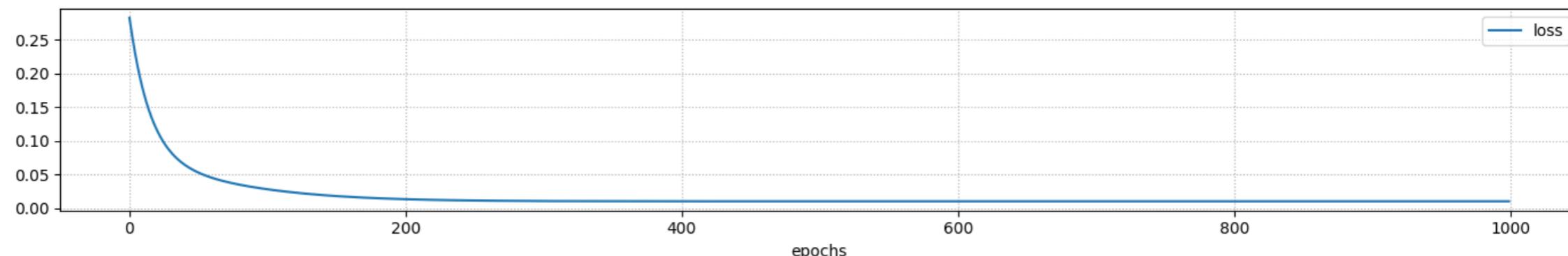
Prediction Focused Approach

As a baseline, we'll consider a basic prediction-focused approach

```
In [25]: pfl = util.build_nn_model(input_shape=1, output_shape=nitems, hidden=[], name='pfl_det', output_
%time history = util.train_nn_model(pfl, data_tr.index.values, data_tr.values, epochs=1000, loss_
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
util.print_ml_metrics(pfl, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(pfl, data_ts.index.values, data_ts.values, label='test')
```

CPU times: user 8.94 s, sys: 330 ms, total: 9.27 s

Wall time: 7.36 s



R2: 0.86, MAE: 0.086, RMSE: 0.10 (training)

R2: 0.86, MAE: 0.087, RMSE: 0.10 (test)

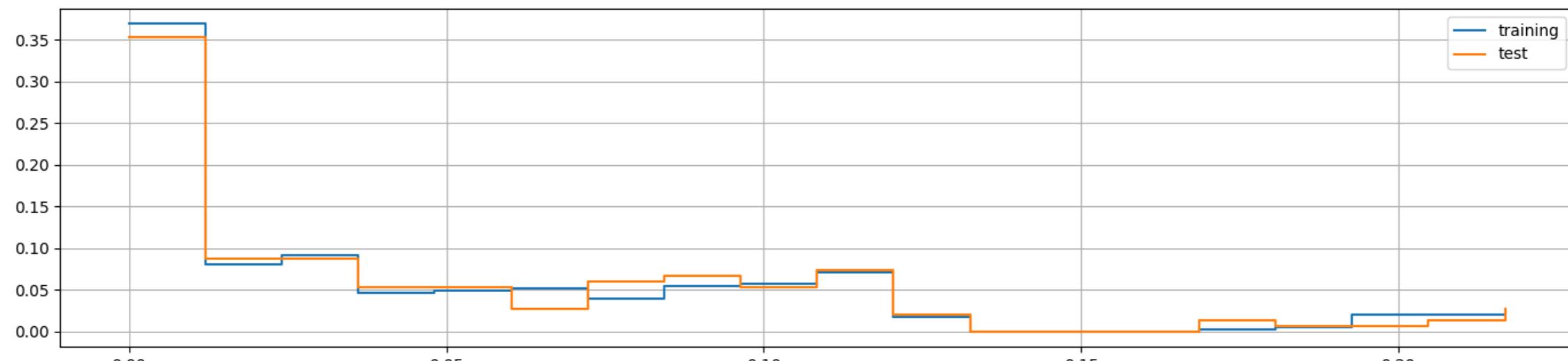


The ML model is just a linear regressor, but it is decently accurate

Prediction Focused Approach

...But our true evaluation should be in terms of regret

```
In [26]: 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', print_m
```



Mean: 0.052 (training), 0.053 (test)

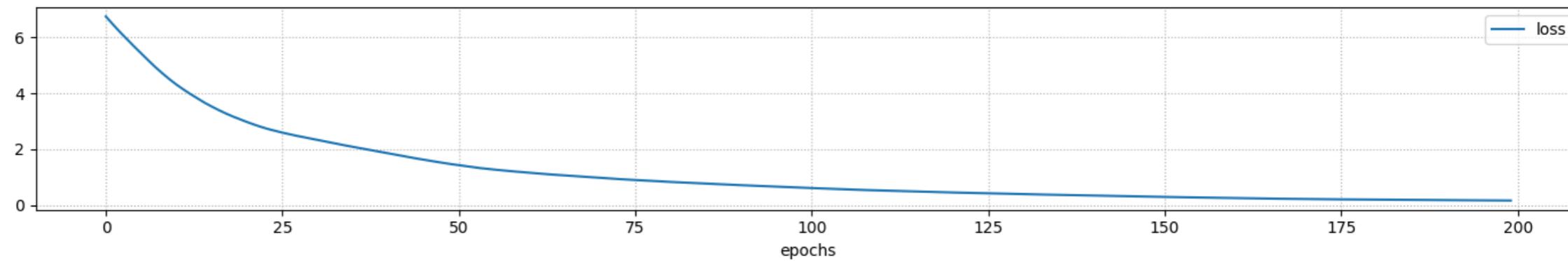
- In this case, the average relative regret is ~5%



A Decision Focused Learning Approach

```
In [27]: spo = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[], name='sp  
%time history = util.train_dfl_model(spo, data_tr.index.values, data_tr.values, epochs=200, verk  
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)  
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')
```

CPU times: user 4min 31s, sys: 20.3 s, total: 4min 51s
Wall time: 4min 51s



R2: -0.14, MAE: 0.22, RMSE: 0.27 (training)
R2: -0.14, MAE: 0.22, RMSE: 0.27 (test)

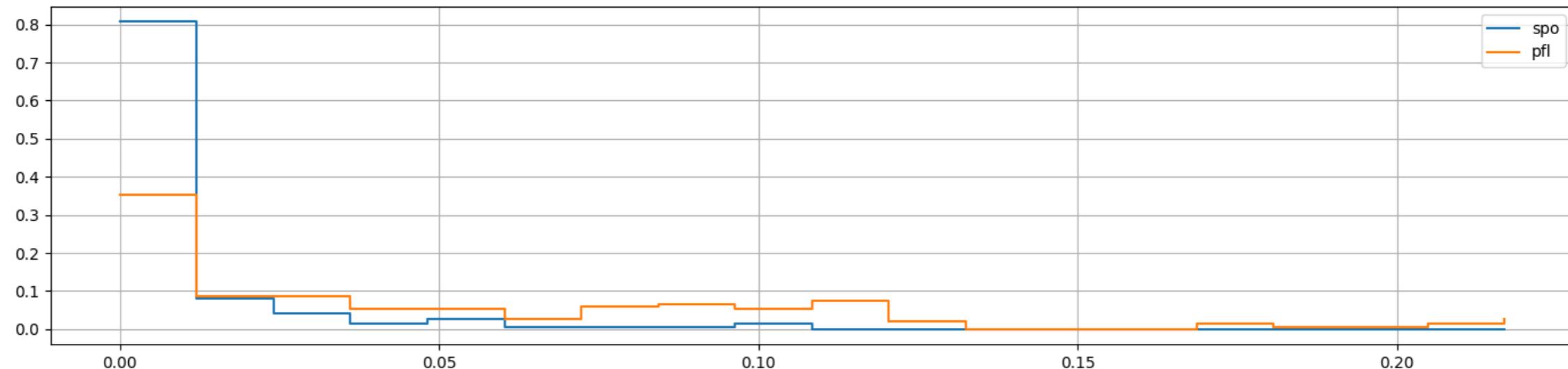
In terms of accuracy, this is considerably worse



Comparing Regrets

But the regret is so much better!

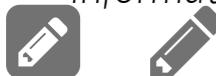
```
In [28]: r_ts_spo = util.compute_regret(prb, spo, data_ts.index.values, data_ts.values)
util.plot_histogram(r_ts_spo, figsize=figsize, label='spo', data2=r_ts, label2='pfl', print_mean=True)
```



Mean: 0.008 (spo), 0.053 (pfl)

This is the kind of result that attracted so much attention since [2]

[2] Donti, Priya, Brandon Amos, and J. Zico Kolter. "Task-based end-to-end model learning in stochastic optimization." *Advances in neural information processing systems* 30 (2017).





02. Picking a Direction



Let's Second-Guess Ourselves

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

In fact, it's easy to see that:

$$\begin{aligned}\mathbb{E}[\text{regret}(\hat{y}, y)] &\xrightarrow{\mathbb{E}[L(\hat{y}, y)] \rightarrow 0} 0\end{aligned}$$

Intuitively::

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



Let's Second-Guess Ourselves

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

In fact, it's easy to see that:

$$\begin{aligned}\mathbb{E}[\text{regret}(\hat{y}, y)] &\xrightarrow{\mathbb{E}[L(\hat{y}, y)] \rightarrow 0} 0\end{aligned}$$

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?

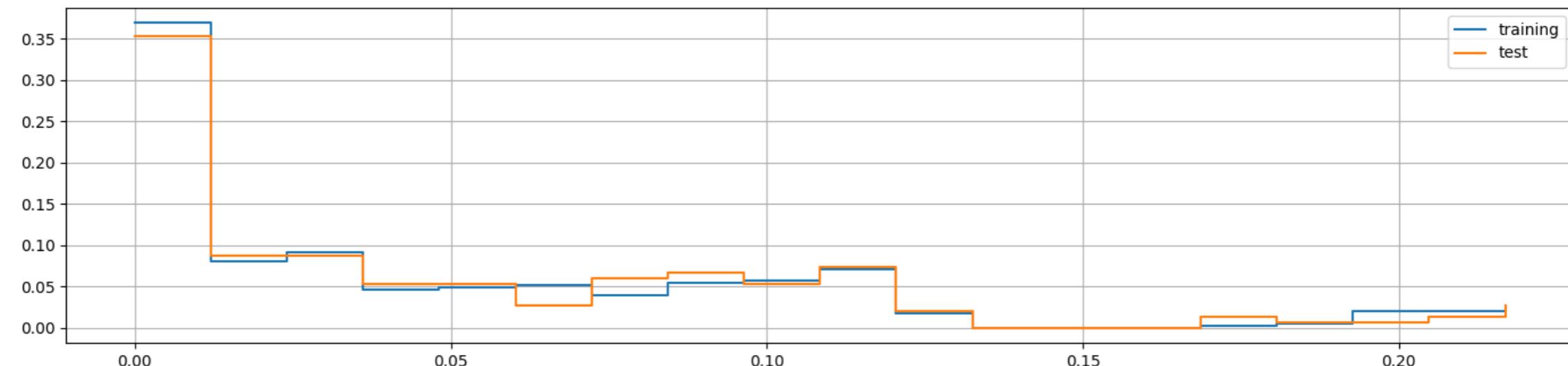
- We could get good predictions **and** good regret
- ...And training would be **much** faster



Our Baseline

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

```
In [2]: pfl = util.build_nn_model(input_shape=1, output_shape=nitems, hidden=[], name='pfl_det', output_
history = util.train_nn_model(pfl, data_tr.index.values, data_tr.values, epochs=1000, loss='mse'
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', print_me
```



Mean: 0.052 (training), 0.053 (test)



This will be our main baseline

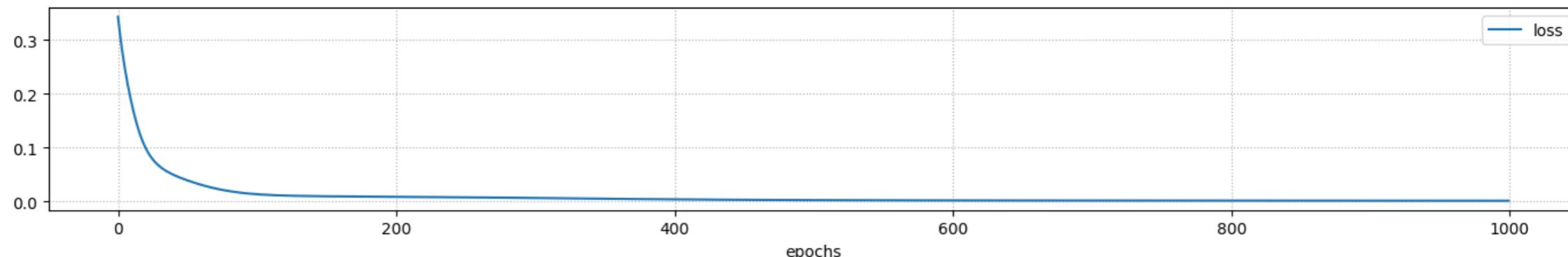
PFL Strikes Back

Let's try to use a non-linear model

```
In [3]: pfl_acc = util.build_nn_model(input_shape=1, output_shape=nitems, hidden=[8], name='pfl_det_acc')
%time history = util.train_nn_model(pfl_acc, data_tr.index.values, data_tr.values, epochs=1000,
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
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')
```

CPU times: user 10.3 s, sys: 367 ms, total: 10.7 s

Wall time: 7.95 s



R2: 0.99, MAE: 0.019, RMSE: 0.03 (training)

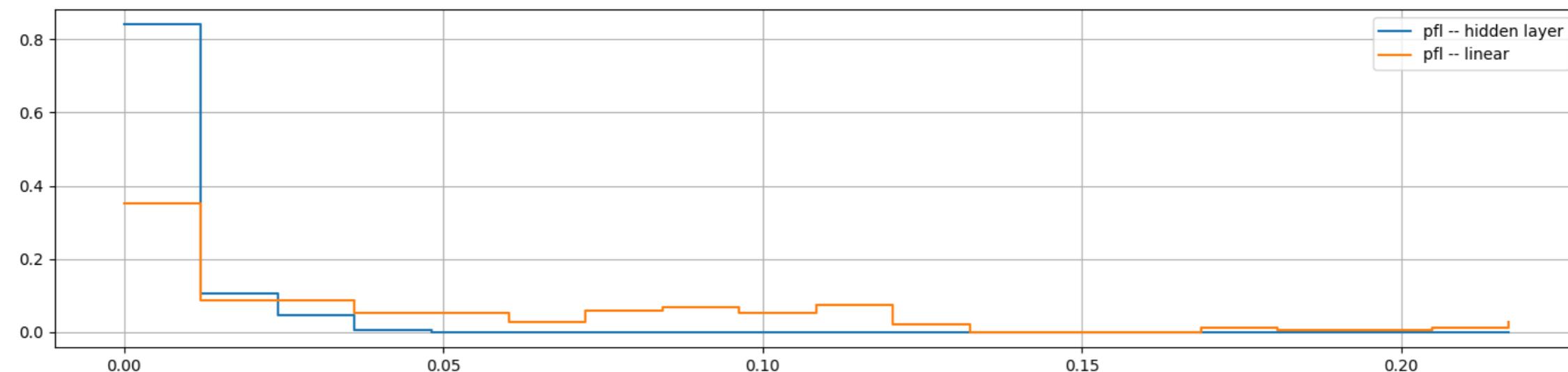
R2: 0.99, MAE: 0.019, RMSE: 0.03 (test)

More accurate, it is!

PFL Strikes Back

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

```
In [4]: 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, label2='
```



Mean: 0.005 (pfl -- hidden layer), 0.053 (pfl -- linear)

DFL might do better with the same model complexity, but we the return would be diminished



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



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 **problem relaxation**, as in [1]
- You can limit recomputation by caching past solutions, as in [2]
- You can warm start the DFL approach with the PFL weights

Let's see the last two tricks in deeper detail



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 **problem relaxation**, as in [1]
- You can limit recomputation by caching past solutions, as in [2]
- You can warm start the DFL approach with the PFL weights

Let's see the last two tricks in deeper detail

[1] Mandi, Jayanta, and Tias Guns. "Interior point solving for lp-based prediction+ optimisation." *Advances in Neural Information Processing Systems* 33 (2020): 7272-7282.

[2] Maxime Mulamba, Jayanta Mandi, Michelangelo Diligenti, Michele Lombardi, Victor Bucarey, Tias Guns: Contrastive Losses and Solution Caching for Predict-and-Optimize. *IJCAI 2021*: 2833-2840



Solution Cache and Warm Start

Solution caching is applicable if the feasible space is fixed

i.e. to problems in the form:

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

- 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 $z^*(\hat{y})$ 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 $\hat{z}^*(y) = \operatorname{argmin}_z \{ f(z) \mid z \in S \}$

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

Since accuracy is correlated with regret, this might accelerate convergence

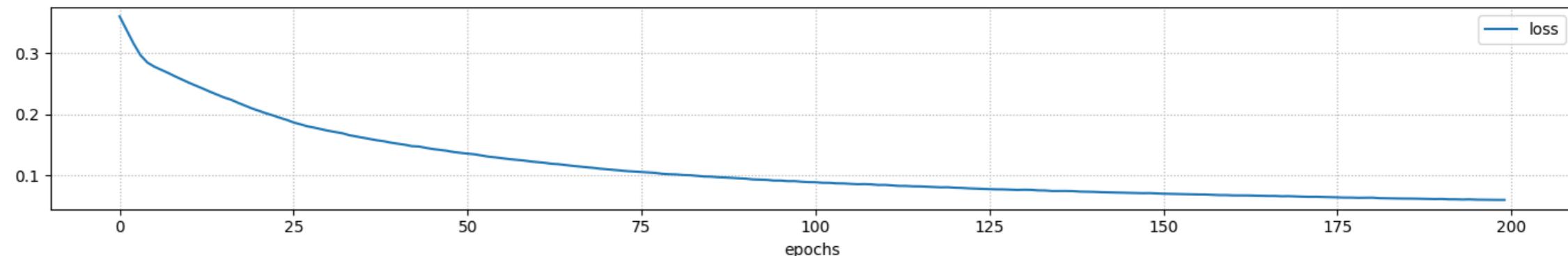


Speeding Up DFL

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

```
In [5]: spo = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[], name='sp')
%time history = util.train_dfl_model(spo, data_tr.index.values, data_tr.values, epochs=200, verbose=True)
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
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')
```

CPU times: user 50.8 s, sys: 4.91 s, total: 55.7 s
Wall time: 55.4 s



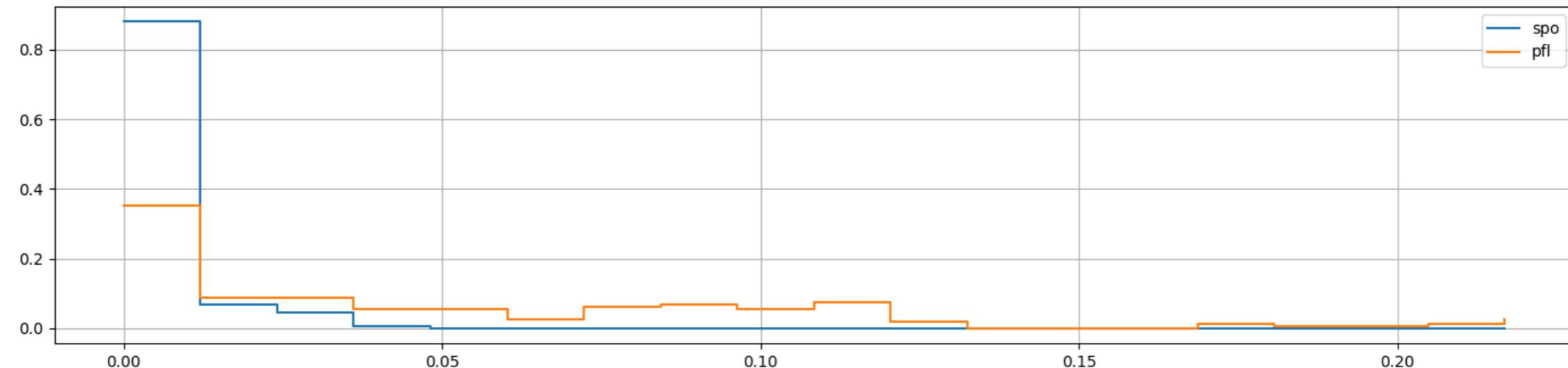
R2: 0.65, MAE: 0.12, RMSE: 0.16 (training)
R2: 0.65, MAE: 0.12, RMSE: 0.16 (test)

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

Speeding Up DFL

And the regret is even better!

```
In [6]: 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', data2=r_ts, label2='pfl', print_mean=True)
```



Mean: 0.004 (spo), 0.053 (pfl)

We are matching the more complex PFL model with a simple linear regressor



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 approach

- 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 approach

- 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?



Maximizing Results

There's a simple case where PFL cannot make perfect predictions



You just need to target a stochastic problem!

- E.g. you can usually tell the traffic situation based on (e.g.) time and weather
- ...But there still a lot of variability

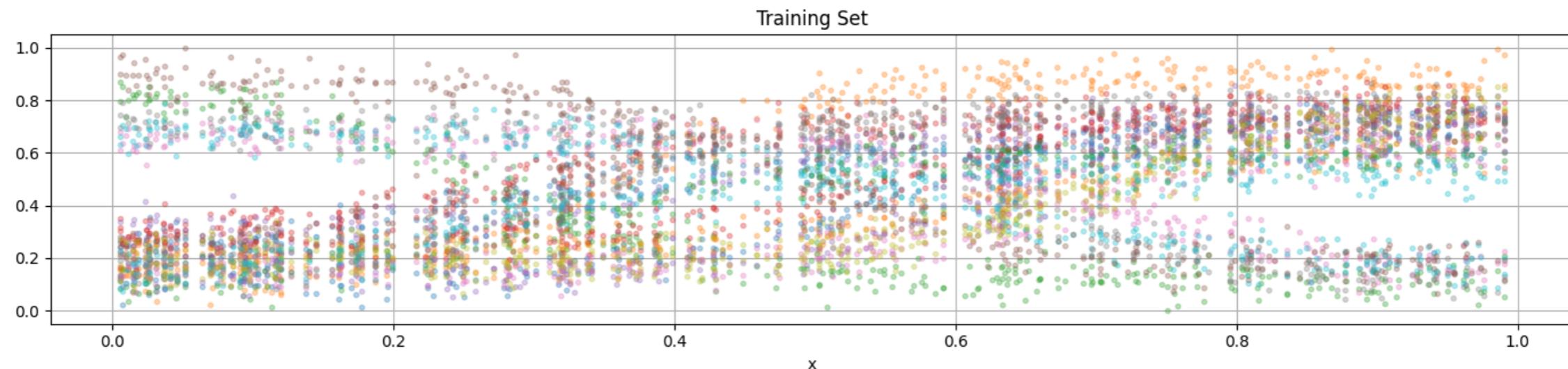


Maximizing Results

Formally, we need a stochastic process, i.e. a stochastic function

We can generate for a stochastic variant of our problem

```
In [8]: data_tr = util.generate_costs(nsamples=350, nitems=nitems, seed=seed, noise_scale=.15, noise_type='uniform')
util.plot_df_cols(data_tr, figsize=figsize, title='Training Set', scatter=True)
```



We treat both X and Y as random variables, with distribution $P(X, Y)$



Adjusting Goals

But with a stochastic process, what is our real objective?

For a given x , we can formalize it like this:

$$\operatorname{argmin}_z \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [y^T z] \mid z \in F \right\}$$

- Given a value for the observable x
- We want to find a single decision vector z
- Such that z is feasible
- ...And z minimized the **expected cost** over the distribution $P(Y \mid X = x)$

This is called a **one-stage stochastic optimization problem**



...And Keeping the Setup

Let's look again at the DFL training problem

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{(x,y) \sim P(X,Y)} [\text{regret}(\hat{y}, y)] \mid \hat{y} = h(x, \theta) \}$$

With:

$$\text{regret}(\hat{y}, y) = y^T z^*(\hat{y}) - y^T z^*(y)$$

Since $y^T z^*(y)$ is independent on θ , this is equivalent to:

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{(x,y) \sim P(X,Y)} [y^T z^*(\hat{y})] \mid \hat{y} = h(x, \theta) \}$$

Which can be rewritten as:

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{x \sim P(X), y \sim P(Y|X)} [y^T z^*(\hat{y})] \mid \hat{y} = h(x, \theta) \}$$



...And Keeping the Setup

Now, let's restrict to the case where x is fixed

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{y \sim P(Y|X=x)} [y^T z^*(\hat{y})] \mid \hat{y} = h(x, \theta) \}$$

Finally, by definition of $z^*(\cdot)$ we have:

$$\theta^* = \operatorname{argmin}_{\theta} \{ \mathbb{E}_{y \sim P(Y|X=x)} [y^T z^*(\hat{y})] \mid \hat{y} = h(x, \theta), z^*(\hat{y}) \in F \}$$

In other words:

- We are choosing θ
- So that $z^*(\hat{y})$ minimizes $\mathbb{E}_{y \sim P(Y|X=x)} [y^T z^*(\hat{y})]$

This is **almost identical** to one-stage stochastic optimization!



DFL For One-Stage Stochastic Optimization

This means that DFL can address these problems, with one restriction and two "superpowers":

The restriction is that we control z only through θ

- Therefore, depending on the chosen ML model architecture
- ...Obtaining some solutions might be impossible
- This issue can be sidestepped with a careful model choice

The first superpower is that we are not restricted to a single x value

- Given a new value for x , we just need to evaluate $h(x, \theta^*)$
- ...And then solve the usual optimization problem
- Many approaches do not deal with the estimation of the y distribution

For the second superpower, we need to investigate a bit more



Classical Solution Approach

What would be the classical solution approach?

Starting from:

$$\operatorname{argmin}_z \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [y^T z] \mid z \in F \right\}$$

We can use linearity to obtain:

$$\operatorname{argmin}_z \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [y]^T z \mid z \in F \right\}$$

- So, we would first need to estimate the expected costs
- ...Then we could solve a deterministic problem



Classical Solution Approach

What would be the classical solution approach?

Starting from:

$$\operatorname{argmin}_z \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [y^T z] \mid z \in F \right\}$$

We can use linearity to obtain:

$$\operatorname{argmin}_z \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [y]^T z \mid z \in F \right\}$$

- So, we would first need to estimate the expected costs
- ...Then we could solve a deterministic problem

But isn't this what PFL is doing?



Regression and Expectation

(Stochastic) Regression is often presented as learning an expectation

...But it's trickier than that

- Using an MSE loss is equivalent to trying to learn $\mathbb{E}_{y \sim P(Y|X=x)} [y]$
- ...But only assuming that $P(Y | X = x)$ is **Normally distributed**
- ...And that it has **the same variance everywhere**

It is possible to do the same under more general conditions

...But it is much more complex

- If we know the distribution type, we can use a neuro-probabilistic model
- Otherwise, we need a fully fledged contextual generative model

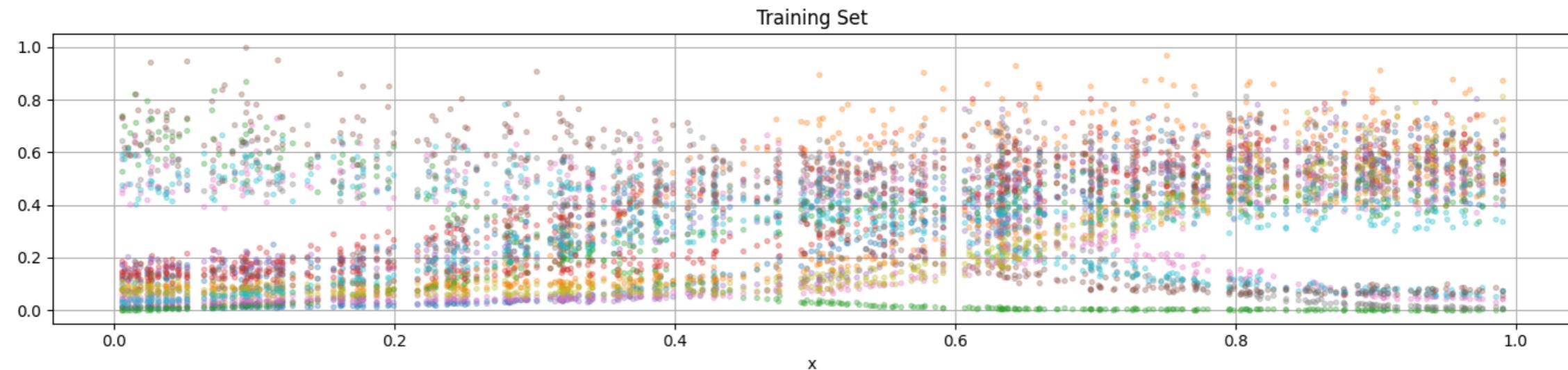
In DFL, we can address this problem with 0 added effort!



A Simple Stress Test

We can test this idea by generating a stochastic dataset

```
In [34]: data_tr = util.generate_costs(nsamples=350, nitems=nitems, seed=seed, noise_scale=.2, noise_type='uniform')
data_ts = util.generate_costs(nsamples=150, nitems=nitems, seed=seed, sampling_seed=seed+1, noise_type='uniform')
util.plot_df_cols(data_tr, figsize=figsize, title='Training Set', scatter=True)
```



...And scaling the variance with y (a very common setting in practice)



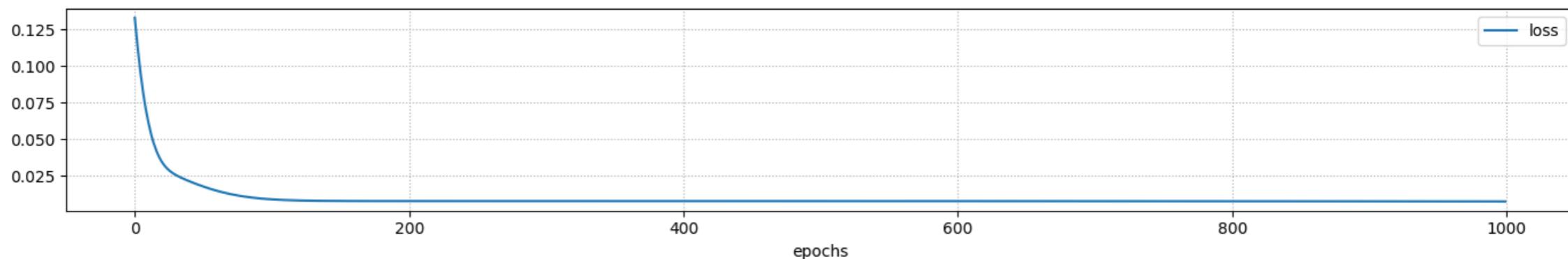
Training a PFL Approach

We will train again a non-linear prediction focused approach

```
In [36]: pfl_1s = util.build_nn_model(input_shape=1, output_shape=nitems, hidden=[8], name='pfl_1s', output_activation='softmax')
@time history = util.train_nn_model(pfl_1s, data_tr.index.values, data_tr.values, epochs=1000, batch_size=128)
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
util.print_ml_metrics(pfl_1s, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(pfl_1s, data_ts.index.values, data_ts.values, label='test')
```

CPU times: user 9.74 s, sys: 330 ms, total: 10.1 s

Wall time: 7.53 s



R2: 0.81, MAE: 0.068, RMSE: 0.09 (training)

R2: 0.82, MAE: 0.068, RMSE: 0.08 (test)

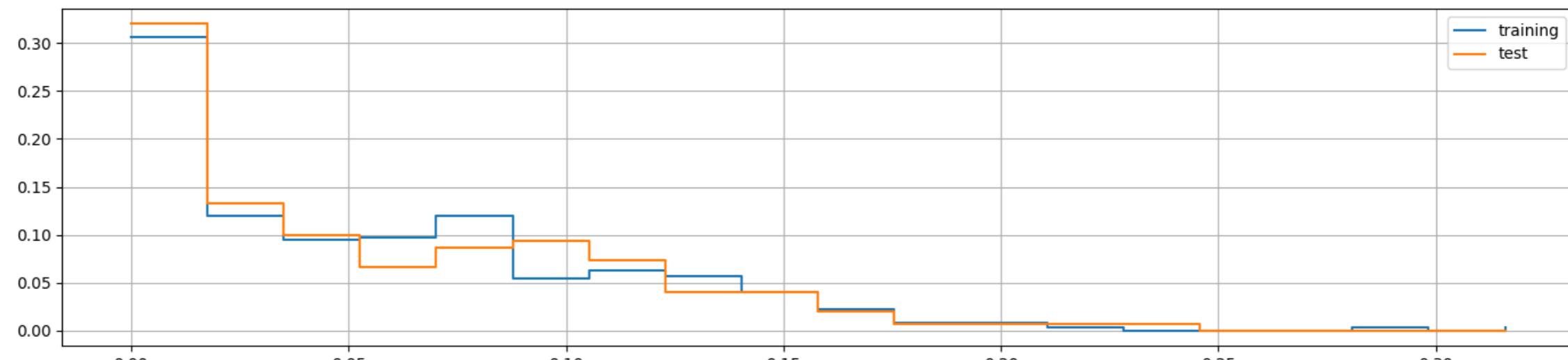


The accuracy is (inevitably) worse, but still pretty good

PFL Regret

Let's evaluate the regret of the PFL approach

```
In [37]: r_tr_1s = util.compute_regret(prb, pfl_1s, data_tr.index.values, data_tr.values)
r_ts_1s = util.compute_regret(prb, pfl_1s, data_ts.index.values, data_ts.values)
util.plot_histogram(r_tr_1s, figsize=figsize, label='training', data2=r_ts_1s, label2='test', pr
```



Mean: 0.059 (training), 0.057 (test)

The regret is has worsened, due to the effect of uncertainty

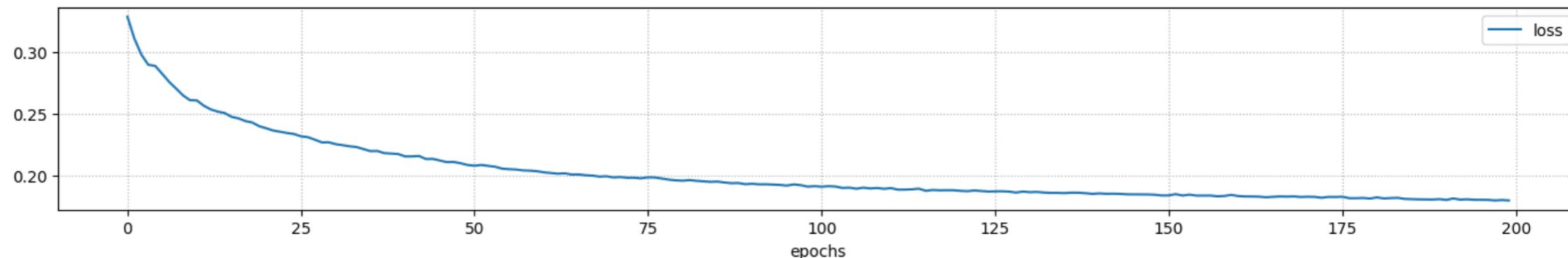


Training a DFL Approach

We also a DFL approach with the same non-linear model

```
In [39]: spo_1s = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[8], name='spo_1s')
%time history = util.train_dfl_model(spo_1s, data_tr.index.values, data_tr.values, epochs=200, verbose=False)
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
util.print_ml_metrics(spo_1s, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(spo_1s, data_ts.index.values, data_ts.values, label='test')
```

CPU times: user 2min 51s, sys: 11min 19s, total: 14min 11s
Wall time: 1min 23s



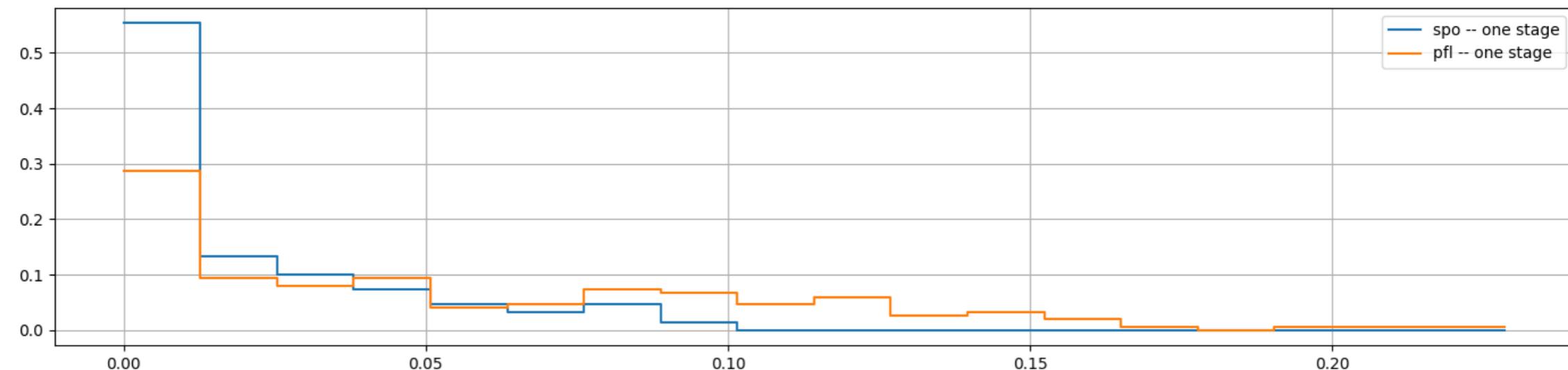
R2: 0.27, MAE: 0.12, RMSE: 0.19 (training)
R2: 0.27, MAE: 0.12, RMSE: 0.19 (test)



DFL Regret

Now we can compare the regret for both approaches

```
In [40]: r_ts_spo_1s = util.compute_regret(prb, spo_1s, data_ts.index.values, data_ts.values)
util.plot_histogram(r_ts_spo_1s, figsize=figsize, label='spo -- one stage', data2=r_ts_1s, label]
```



Mean: 0.020 (spo -- one stage), 0.057 (pfl -- one stage)

There is a significant gap again, since the PFL approach is operating on an incorrect semantic



Considerations

DFL can be thought of as a one-stage stochastic optimization approach

In this setting:

- In particular, using a more accurate PFL model might still have poor regret
- ...Unless we know a lot about the distribution
- ...or we use a very complex estimator
- Conversely, DFL has not such issues

The gap becomes wider in case of **non-linear cost functions**:

- In this case the expected cost would not be equivalent to a sum of expectations
- But a DFL approach would have no such issues
- ...Provided it could deal with non-linear functions





03. Breakng Off



Two-Stage Stochastic Optimization

If DFL targets one-stage stochastic optimization, could we do **two-stage**?



- For example, in **first stage** we decide what to pack in our suitcase
- ...During the trip, we may realize we have **forgotten something**
- ...And we need to spend money to buy the missing stuff



Two-Stage Stochastic Optimization

If DFL targets one-stage stochastic optimization, could we do two-stage?

Two-stage problems are among the most interesting in stochastic optimization

- They involve making a set of decisions now
- Then observing how uncertainty unfolds
- ...And making a second set of decisions

The former are called first-stage decisions, the latter recourse actions

Here's an example we will use for this topic

Say we need to secure a supply of resources

- First, we make contracts with primary suppliers to minimize costs
- If there are unexpected setbacks (e.g. insufficient yields)
- ...Then we can buy what we lack from another source, but at a higher cost



Two-Stage Stochastic Optimization

Let's define two-stage stochastic optimization problems (2s-SOP) formally:

$$\operatorname{argmin}_z \left\{ f(z) + \mathbb{E}_{y \sim P(Y|X=x)} \left[\min_{z''} r(z'', z, y) \right] \mid z \in F, z'' \in F''(z, y) \right\}$$

- Y represents the uncertain information
- z is the vector of **first stage decisions**
- F is the feasible space for the first stage
- z'' is the vector of **recourse actions**
- z'' is not fixed: it can change for every sampled y
- The set of feasible recourse actions $F''(z, y)$ also changes for every y
- f is the immediate cost function, r is the cost of the recourse actions



A Simple Example

We will consider this simple problem

...Which is based on our previous supply planning example:

$$\operatorname{argmin}_z c^T z + \mathbb{E}_{y \sim P(Y|X=x)} \left[\min_{z''} c'' z'' \right]$$

subject to: $y^T z + z'' \geq y_{min}$

$$z \in \{0, 1\}^n, z'' \in \mathbb{N}_0$$

- $z_j = 1$ iff we choose then h -th supply contract
- c_j is the cost of the j -th contract
- y_j is the yield of the j -th contract, which is uncertain
- y_{min} is the minimum total yield, which is known
- z'' is the number of units we buy at cost c'' to satisfy the yield requirement



Scenario Based Approach

Classical solution approaches for 2s-SOP are scenario based

We start by sampling a finite set of N values from $P(Y \mid X = x)$

$$\begin{aligned} & \operatorname{argmin}_z \min_{z''} c^T z + \frac{1}{N} c'' z''_k \\ & \text{subject to: } y^T z + z''_k \geq y_{min} \quad \forall k = 1..N \\ & z \in \{0, 1\}^n \\ & z''_k \in \mathbb{N}_0 \quad \forall k = 1..N \end{aligned}$$

Then we build different recourse action variables for each scenario

- ...We define the feasible sets via constraints
- ...And we use the Sample Average Approximation to estimate the expectation

The method is effective, but also computationally expensive



DFL for 2s-SOP

Could we do something similar with DFL?

As a recap, our DFL training problem is:

$$\theta^* = \operatorname{argmin}_{\theta} \left\{ \mathbb{E}_{(x,y) \sim P(X,Y)} [\text{regret}(\hat{y}, y)] \mid \hat{y} = h(x, \theta) \right\}$$

With:

$$\text{regret}(\hat{y}, y) = y^T z^*(\hat{y}) - y^T z^*(y)$$

And:

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



DFL for 2s-SOP

With the same transformations used in the one-stage case, we get:

$$\theta^* = \operatorname{argmin}_{\theta} \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [y^T z^*(\hat{y})] \mid \hat{y} = h(x, \theta), z^*(\hat{y}) \in F \right\}$$

Now, say we had a DLF approach that could deal with any function $g(z, y)$

- In this case y would be a vector of uncertain parameters (not necessarily costs)
- The function should compute the equivalent of $y^T z^*(\hat{y})$
- ...i.e. the true cost of the solution computed for the estimate costs

Under this conditions, at training time we could solve:

$$\theta^* = \operatorname{argmin}_{\theta} \left\{ \mathbb{E}_{y \sim P(Y|X=x)} [g(z^*(\hat{y}), y)] \mid \hat{y} = h(x, \theta), z^*(\hat{y}) \in F \right\}$$

It would still be DFL, just a bit more general



DFL for 2s-SOP

At this point, let's choose:

$$g(z, y) = \min_{z''} \{ f(z) + r(z'', z, y) \mid z'' \in F''(z, y) \}$$

- For a given solution z , $g(z, y)$ computes the best possible objective
- ...Assuming that the value of the parameters is y

By substituting in the training formulation we get:

$$\operatorname{argmin}_{\theta} f(z^*(\hat{y})) + \mathbb{E}_{y \sim P(Y|X=x)} \left[\min_{z''} r(z'', z^*(\hat{y}), y) \right]$$

subject to: $\hat{y} = h(x, \theta)$, $z^*(\hat{y}) \in F$, $z'' \in F''(z, y)$



..Which can definitely be used for 2s-SOP problems!

Grouding the Approach

We can ground the approach by relying on the scenario-based formulation

In our example problem, we compute $z^*(y)$ by solving:

$$\begin{aligned} z^*(y) = \operatorname{argmin}_z \min_{z''} & c^T z + c'' z''_k \\ \text{subject to: } & y^T z + z''_k \geq y_{min} \quad \forall k = 1..N \\ & z \in \{0, 1\}^n \\ & z''_k \in \mathbb{N}_0 \quad \forall k = 1..N \end{aligned}$$

And we define $g(z, y)$ as:

$$\begin{aligned} g(z, y) = \min_{z''} & c^T z + c'' z''_k \\ \text{subject to: } & y^T z + z''_k \geq y_{min} \quad \forall k = 1..N \\ & z''_k \in \mathbb{N}_0 \quad \forall k = 1..N \end{aligned}$$



Overview and Properties

Intuitively, the approach works as follows

- We observe x and we compute \hat{y}
 - We compute $z^*(\hat{y})$ by solving a scenario problem
 - We compute $g(z^*(\hat{y}), y)$ by solving a scenario problem with fixed z values
- ...And we end up minimizing the expected cost of the 2s-SOP

Compared to the classical approach, we have 1 restriction and 3 "superpowers"

- The restriction: we control z^* only through θ
- Superpower 1: we are not restricted to a single x
- Superpower 2: works with any distribution
- Superpower 3: at inference time, we always consider a single scenario



Scalable Two-stage Stochastic Optimization

The last advantage is massive

The weakest point of classical 2s-SOP approach is scalability

- Multiple scenarios are required to obtain good results
- ...But they also add more variables

With NP-hard problem, that solution time may grow exponentially

With this approach, the computational cost is all at training time

- It can even be lower, since you always deal with single scenarios
- There are alternatives, such as [1], where ML is used to estimate the recourse
- ...These have their own pros and cons

[1] Dumouchelle, Justin, et al. "Neur2sp: Neural two-stage stochastic programming." arXiv preprint arXiv:2205.12006 (2022).



The Elephant in the Room

So far, so good, but how to we make $g(z, y)$ differentiable?

There are a few alternatives, all with limitations:

- The approach from [1] handles parameters in the problem constraints
 - It is based on the idea of differencing the recourse action
 - ...But it is (mostly) restricted to 1D packing problems
- The approach from [2] can be used for 2s-SOP with a stretch
 - It based on idea of embedding a MILP solver in ML
 - ...But it's semantic does not fully align with 2s-SOP

Here, we will see different technique

[1] Hu, X., Lee, J. C. H., and Lee, J. H. M. *Predict+optimize for packing and covering lps with unknown parameters in constraints*. CoRR, abs/2209.03668, 2022. doi: 10.48550/arXiv.2209.03668.

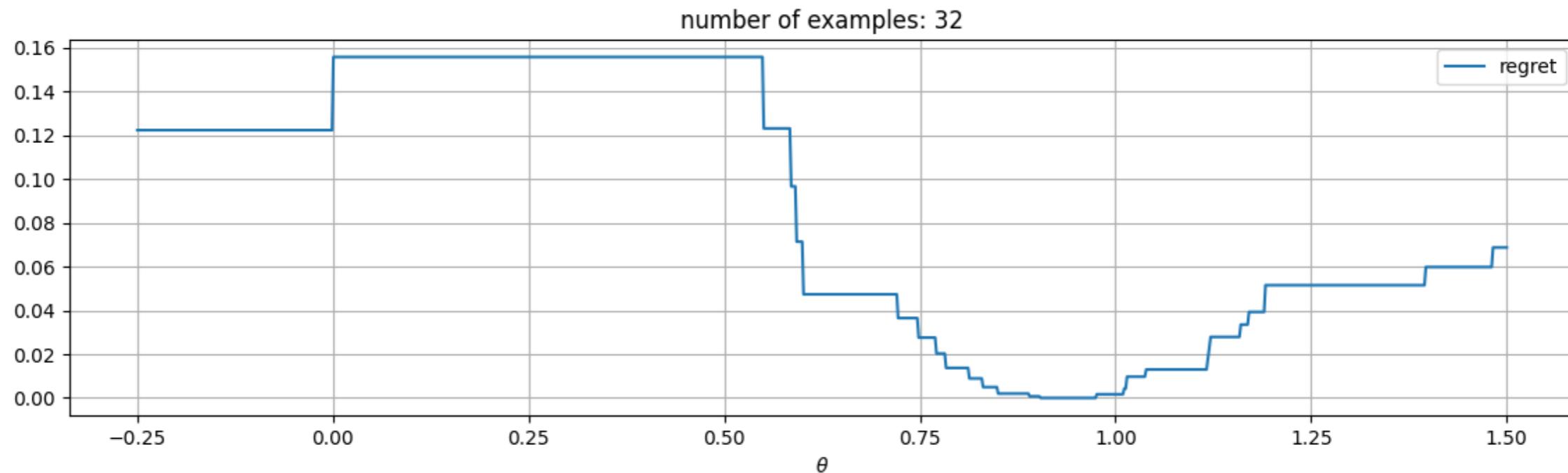
[2] Paulus, Anselm, et al. "Comboptnet: Fit the right np-hard problem by learning integer programming constraints." International Conference on Machine Learning. PMLR, 2021.



Looking Back at SPO

Let's look again at the regret loss for our original toy example

```
In [2]: util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=32, figsize=f)
```



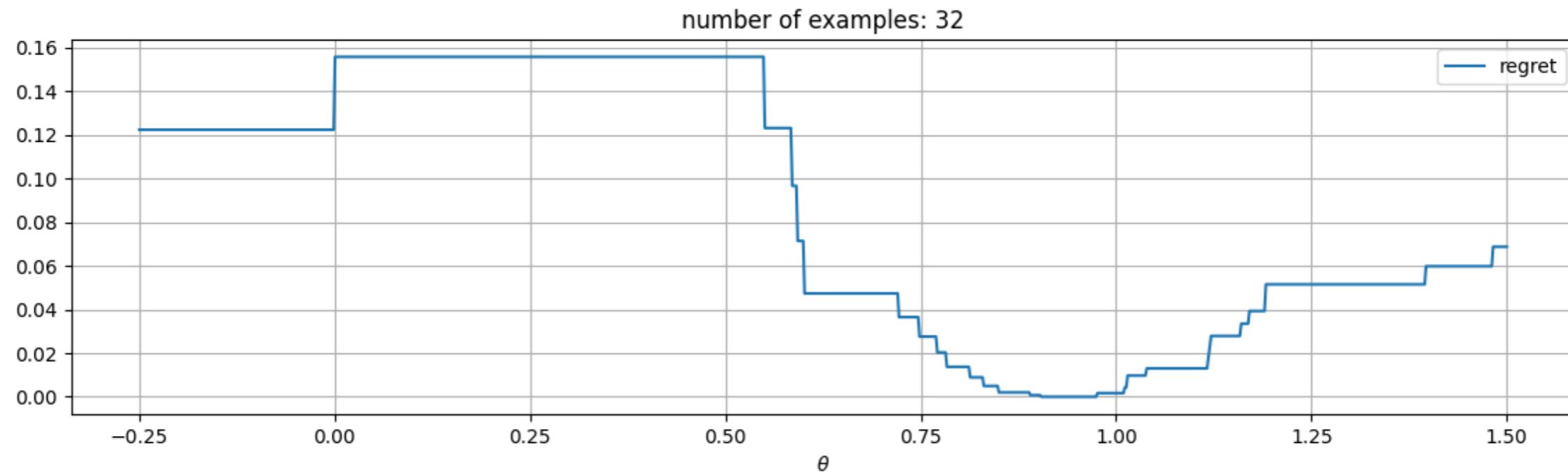
- It is non-differentiable at places, and flat almost everywhere
- Can we think of another way to address these issues?



Looking Back at SPO

If we could act on this function itself, a simple solution would be **smoothing**

```
In [3]: util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=32, figsize=f)
```



- We could think of computing a convolution with a Gaussian kernel
- It would be like applying a Gaussian filter to an image



Stochastic Smoothing

But how can we do it through an optimization problem?

A viable approach is using stochastic smoothing

- Rather than learning a point estimator $h(x, \theta)$
- We learn a **stochastic estimator** s.t. $\hat{y} \sim \mathcal{N}(h(x, \theta), \sigma)$

Intuitively:

- We still use a point estimator, but to predict **a vector of means**
- Then we sample \hat{y} from a normal distribution having the specified mean
- ...And a fixed standard deviation

We end up smoothing over \hat{y} rather than over θ

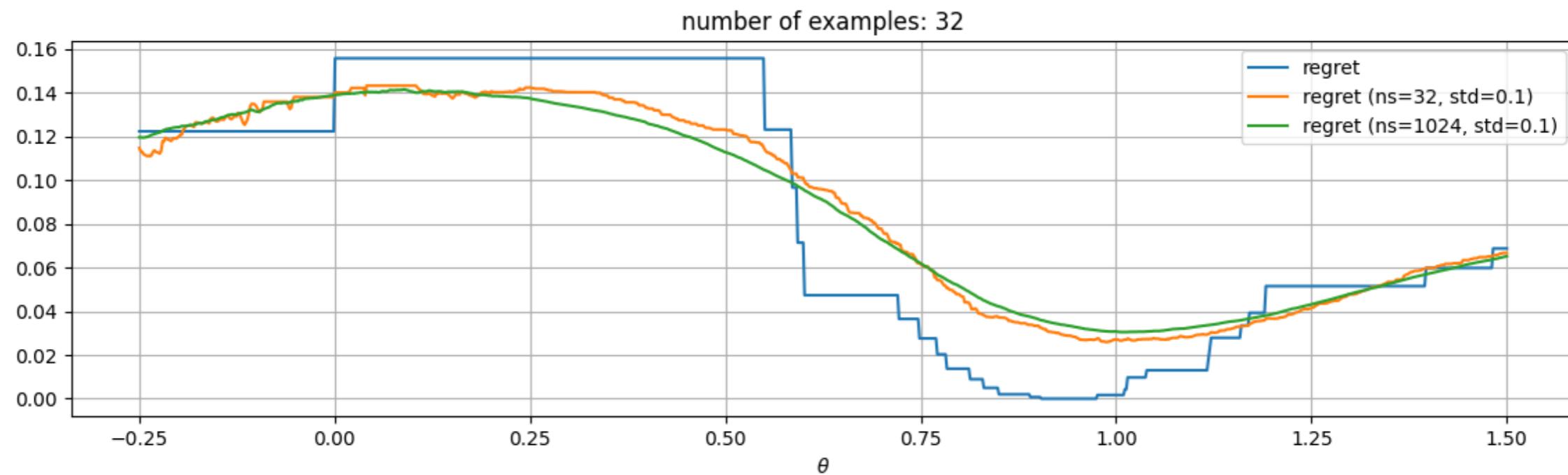
But it's very close to what we wanted to do!



Stochastic Smoothing

Let's see how it works on our toy example

```
In [4]: util.draw_loss_landscape(losses=[util.RegretLoss(), util.RegretLoss(smoothing_samples=32, smooth
```



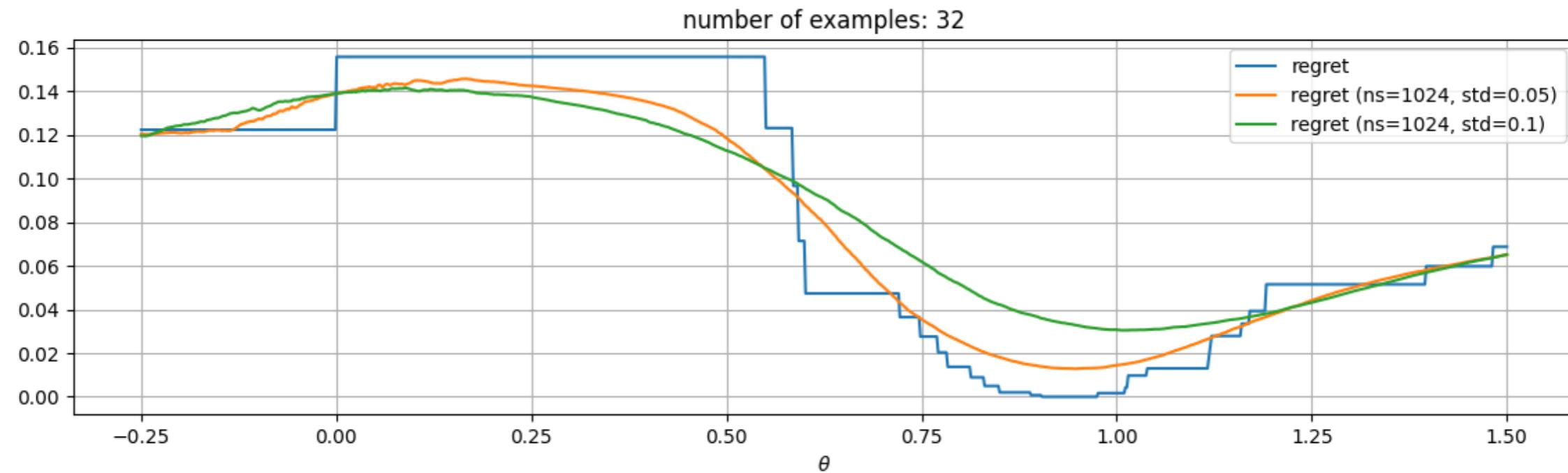
- It's a stochastic approach, some noise is to be expected
- Using **more samples** leads to better smoothing



Stochastic Smoothing

We can control the smoothing level by adjusting σ

```
In [5]: util.draw_loss_landscape(losses=[util.RegretLoss(), util.RegretLoss(smoothing_samples=1024, smoo
```



- Larger σ value remove flat sections better
- ...But also cause a shift in the position of the optimum



Score Function Gradient Estimation

How does that help us?

Normally, the DFL loss looks like this:

$$L_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y)} [\text{regret}(\hat{y}, y)]$$

When we apply stochastic smoothing, it turns into:

$$\tilde{L}_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y), \hat{y} \sim \mathcal{N}(h(x,\theta))} [\text{regret}(\hat{y}, y)]$$

The expectation is now computed on x , y , and \hat{y}

- We can use a sample average to handle the expectation on x and y
- ...But if we do it on \hat{y} we are left with nothing differentiable



Score Function Gradient Estimation

So we expand the last expectation on \hat{y} :

$$\tilde{L}_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y)} \left[\int_{\hat{y}} \text{regret}(\hat{y}, y) p(\hat{y}, \theta) d\hat{y} \right]$$

- $\text{regret}(\hat{y}, y)$ cannot be differentiated, since \hat{y} is a fixed sample in this setup
- However, the probability $p(\hat{y}, \theta)$ can! It's just a Normal PDF

Now, we just need a way to handle the integral

We do it by **focusing on the gradient**

Due to linearity of expectation and integration, this is given by:

$$\nabla \tilde{L}_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y)} \left[\int_{\hat{y}} \text{regret}(\hat{y}, y) \nabla_{\theta} p(\hat{y}, \theta) d\hat{y} \right]$$



Score Function Gradient Estimation

Let's consider again the expression we have obtained

$$\nabla \tilde{L}_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y)} \left[\int_{\hat{y}} \text{regret}(\hat{y}, y) \nabla_{\theta} p(\hat{y}, \theta) d\hat{y} \right]$$

By taking advantage of the fact that $\log'(f(x)) = 1/x f'(x)$, we can rewrite it as:

$$\nabla \tilde{L}_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y)} \left[\int_{\hat{y}} \text{regret}(\hat{y}, y) p(\hat{y}, \theta) \nabla_{\theta} \log p(\hat{y}, \theta) d\hat{y} \right]$$

Now, the integral is again an expectation, so we have:

$$\nabla \tilde{L}_{DFL}(\theta) = \mathbb{E}_{(x,y) \sim P(X,Y), \hat{y} \sim \mathcal{N}(h(x,\theta), \sigma)} [\text{regret}(\hat{y}, y) \nabla_{\theta} \log p(\hat{y}, \theta)]$$



Score Function Gradient Estimation

Finally, we can use a sample average to approximate both expectations:

$$\nabla \tilde{L}_{DFL}(\theta) \simeq \frac{1}{m} \sum_{i=1}^m \frac{1}{N} \sum_{k=1}^N \text{regret}(\hat{y}, y) \nabla_\theta \log p(\hat{y}, \theta)$$

- For every training example we sample \hat{y} from the stochastic estimator
- We compute $\text{regret}(\hat{y}, y)$ as usual
- ...And we obtain a gradient since $p(\hat{y}, \theta)$ is easily differentiable in θ

We can trick a tensor engine into doing the calculation by using this loss:

$$\tilde{L}_{DFL}(\theta) \simeq \frac{1}{m} \sum_{i=1}^m \frac{1}{N} \sum_{k=1}^N \text{regret}(\hat{y}, y) \log p(\hat{y}, \theta)$$



Score Function Gradient Estimation

This approach is also known as **Score Function Gradient Estimation (SFGE)**

- It is a known approach (see e.g. [3]), but it has seen limited use in DFL
- We applied it to 2s-SOP in [4] (accepted, not yet published)

It works with any function, not just regret

...And in practice it can be improved by standardizing the gradient terms:

$$\nabla \tilde{L}_{DFL}(\theta) \simeq \frac{1}{m} \sum_{i=1}^m \frac{1}{N} \sum_{k=1}^N \frac{g(\hat{y}, y) - \text{mean}(g(\hat{y}, y))}{\text{std}(g(\hat{y}, y))} \nabla \log p(\hat{y}, \theta)$$

- Standardization helps in particular with small numbers of samples

[3] Berthet, Quentin, et al. "Learning with differentiable perturbed optimizers." *Advances in neural information processing systems* 33 (2020): 9508-9519.

[4] Silvestri, Mattia et al. "Score Function Gradient Estimation to Widen the Applicability of Decision-focused Learning", *Differentiable Almost Everywhere workshop at ICML 2023*



A Practical Example

We test this on our supply planning problem

We start by generating a dataset of **contract values** (the costs are fixed)

```
In [21]: seed, nitems = 42, 20  
data_tr = util.generate_costs(nsamples=350, nitems=nitems, seed=seed, noise_scale=.2, noise_type='uniform')  
util.plot_df_cols(data_tr, figsize=figsize, title='Training Set', scatter=True)
```



The distribution is the same we used for the one-stage problem



A Practical Example

Then we generate the remaining problem parameters

```
In [22]: # Generate the problem
rel_req = 0.6
rel_buffer_cost = 10
prb = util.generate_2s_problem(nitems, requirement=rel_req * data_tr.mean().sum(), rel_buffer_cost=rel_buffer_cost)
prb
```

```
Out[22]: ProductionProblem2Stage(costs=[1.14981605 1.38028572 1.29279758 1.23946339 1.06240746 1.062397
81
1.02323344 1.34647046 1.240446 1.28322903 1.0082338 1.38796394
1.33297706 1.08493564 1.07272999 1.0733618 1.1216969 1.20990257
1.17277801 1.11649166], requirement=3.8862101169088654, buffer_cost=11.830809153591137)
```

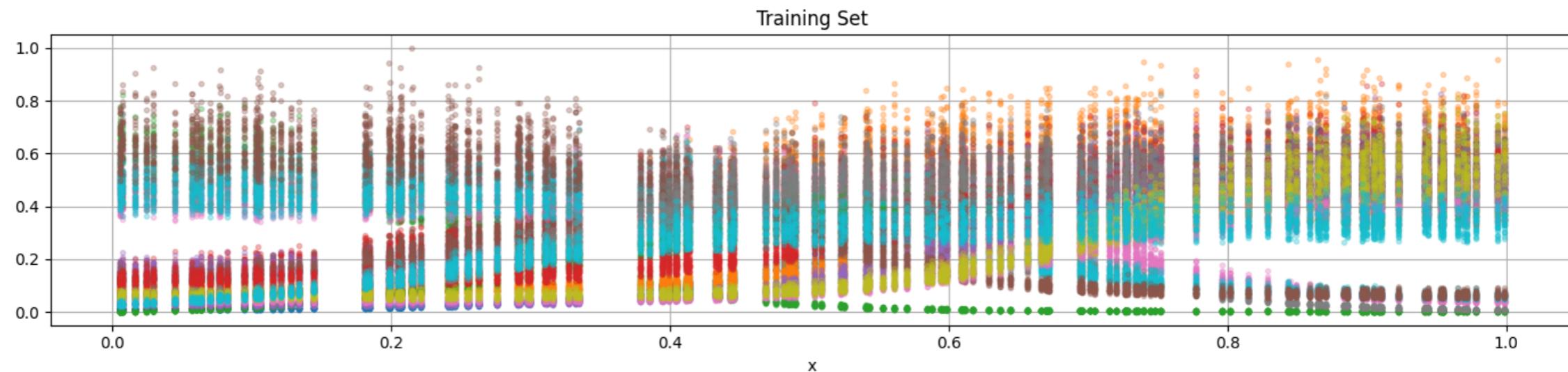
- The minimum value if 60% of the sum of average values on the training data
- Buying in the second stage is 10 times more expensive then the average cost



A Practical Example

For testing, we generate multiple samples per instance

```
In [23]: data_ts = util.generate_costs(nsamples=150, nitems=nitems, seed=seed, sampling_seed=seed+1, noise=noise)
util.plot_df_cols(data_ts, figsize=figsize, title='Training Set', scatter=True)
```



By doing this, we get a more reliable evaluation of uncertainty



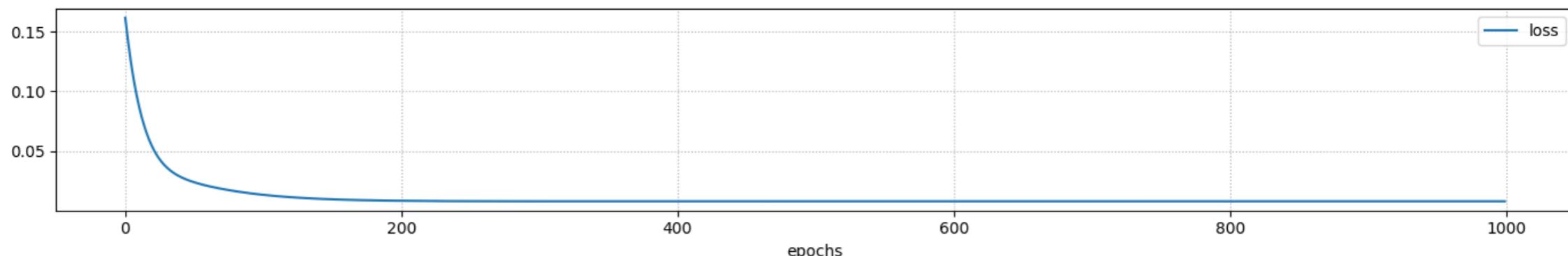
A PFL Approach

We start by training a prediction focused approach

```
In [24]: pfl_2s = util.build_nn_model(input_shape=1, output_shape=nitems, hidden=[], name='pfl_2s', output_activation='softmax')
@time history = util.train_nn_model(pfl_2s, data_tr.index.values, data_tr.values, epochs=1000, verbose=False)
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
util.print_ml_metrics(pfl_2s, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(pfl_2s, data_ts.index.values, data_ts.values, label='training')
```

CPU times: user 8.21 s, sys: 352 ms, total: 8.56 s

Wall time: 6.83 s



R2: 0.80, MAE: 0.071, RMSE: 0.09 (training)

R2: 0.75, MAE: 0.072, RMSE: 0.09 (training)



This is as fast as the DFL approach, and can be used for warm-starting

Evaluating Two-Stage Approaches

Two-state stochastic approaches can be evaluated in two ways

We can compare them with **the best we could do**

- The cost difference is the proper regret
- Its computation requires solving a 2s-SOP with high accuracy
- ...Making it a very computationally expensive metric

We can compare them with the expected cost of **a clairvoyant approach**

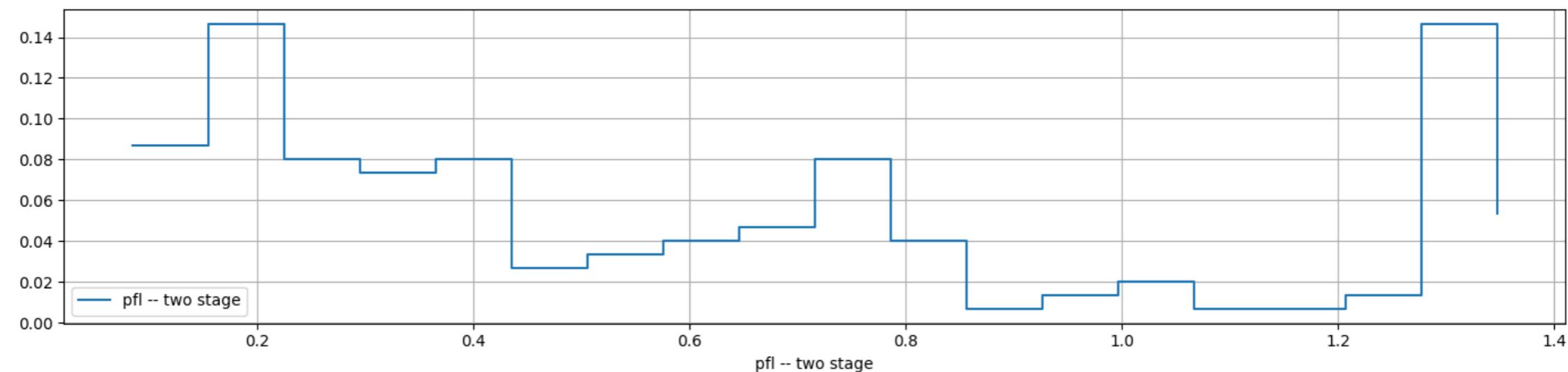
- The cost difference is called Expected Value of Perfect Information
- ...Or sometimes Post-hoc regret
- Its computation requires solving a 2s-SOP **with just a single scenario**
- ...So it's much faster, but only provide an upper bound on true regret



Evaluating the PFL Approach

Let's check the EVPF/Post-hoc regret for the PFL Approach

```
In [25]: pfl_2s_evpf = util.compute_evpf_2s(prb, pfl_2s, data_ts, tlim=10)
util.plot_histogram(pfl_2s_evpf, figsize=figsize, label='pfl -- two stage', print_mean=True)
```



Mean: 0.634 (pfl -- two stage)

This will be our baseline



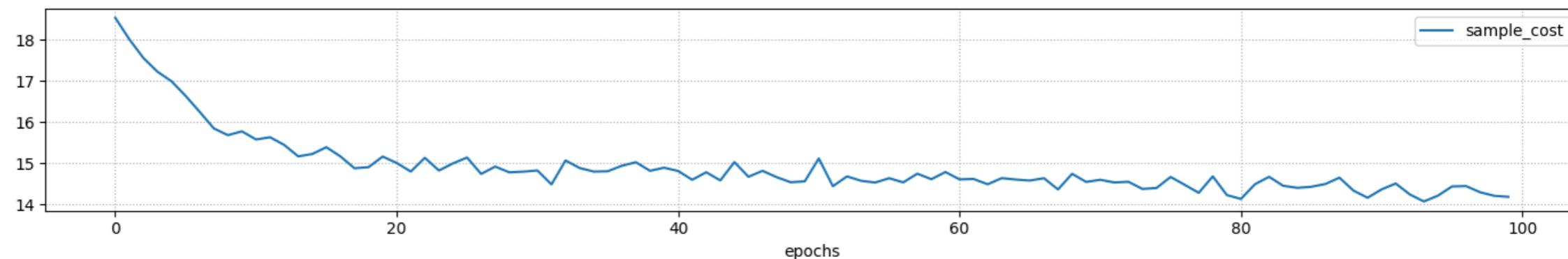
Training a DFL Approach

We traing a DFL with warm starting, but no solution cache

...Since the feasible space for the recourse actions is not fixed

```
In [30]: sfge_2s = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[], name='sample_cost')
@time history = util.train_dfl_model(sfge_2s, data_tr.index.values, data_tr.values, epochs=100,
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False, excluded_r
util.print_ml_metrics(sfge_2s, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(sfge_2s, data_ts.index.values, data_ts.values, label='test')
```

CPU times: user 4min 41s, sys: 34.8 s, total: 5min 16s
Wall time: 5min 16s



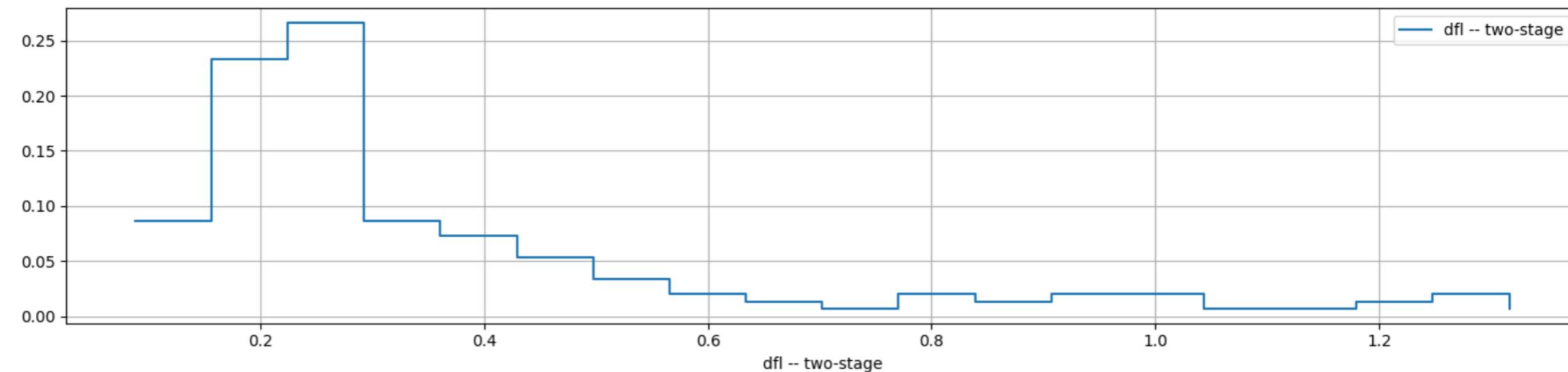
R2: 0.61, MAE: 0.095, RMSE: 0.12 (training)
R2: 0.66, MAE: 0.081, RMSE: 0.10 (test)



Evaluating the DFL Approach

We can now inspect the EVPF/Post-hoc regret for the DLF approach, as well

```
In [31]: sfge_2s_evpf = util.compute_evpf_2s(prb, sfge_2s, data_ts, tlim=10)
util.plot_histogram(sfge_2s_evpf, figsize=figsize, label='dfl -- two-stage', print_mean=True)
```



Mean: 0.382 (dfl -- two-stage)



A More In-depth Comparison

A more extensive experimentation will be found in [4]

The method has been tested on:

- Some "normal" DFL benchmarks
- Several two-stage stochastic problems

The baselines are represented by:

- Specialize methods (e.g. SPO, the one from [1]), when applicable
- A neuro-probabilistic model + a scenario based approach

Specialized method tend to work better

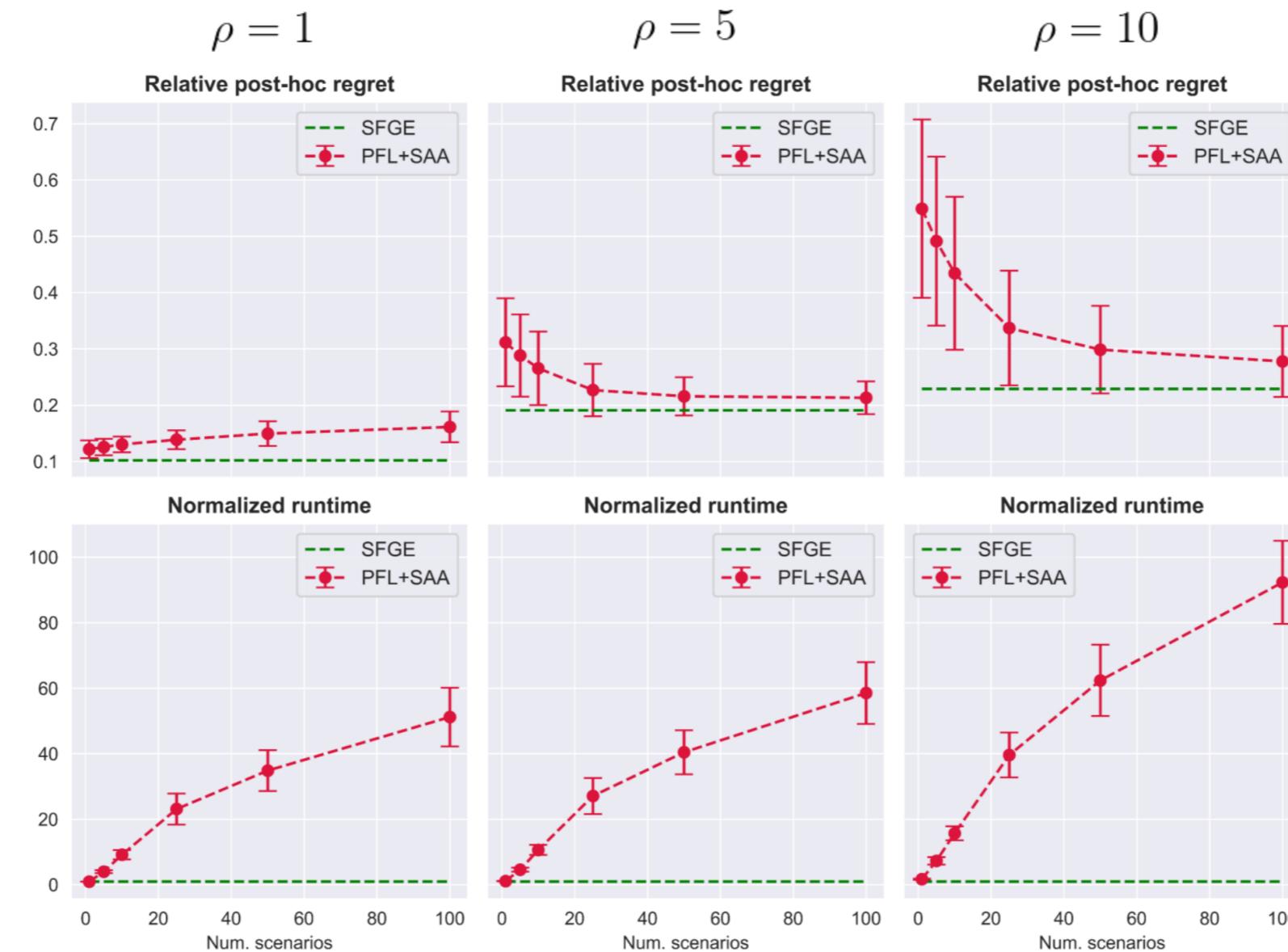
- ...But SFGE is much more versatile
- The best results are obtained on 2s-SOPs



A More In-depth Comparison

This is how the approach fares again the scenario based method

...On a problem somewhat similar to our supply planning one





03. Off-beat Path



Two-Stage Stochastic Optimization

Let's consider this variant of our example problem

$$z^*(u) = \operatorname{argmin} \left\{ \mathbb{E}_{y \sim P(Y|X=x)} \left[\sum_{j=1}^n \sin(2\pi u_j z_j) \right] \mid v^T z \geq r, z \in [0, 1]^n \right\}$$

It's similar to the one-stage variant, except that:

- The decision variables are continuous
- The cost function is non-linear
- We use u to denote the uncertain parameters

It's just another one-stage stochastic optimization problem,
but the use of sin terms makes it much more challenging



SFGE Enables Objective Decoupling

We will see that it can **also** be addressed via DFL

The point is that the SFGE approach has a nice property:

- The function $g(\hat{y}, y)$ used as a loss term
- ...And the cost we use to compute $z^*(y)$

...Can be **completely distinct**

We can put this to our advantage

- In particular, we can use an ML model
- ...To guide a low-complexity problem
- ...So that we get a solution for tougher one

...And since we are using DFL we also get a contextual approach (we react to x)



Target Problem

Let's try to come up with a formalization

Say we want to target an optimization problem in the form:

$$\operatorname{argmin}_z \left\{ \mathbb{E}_{u \sim P(Y|X=x)} [f(z, u, x)] \mid z \in F(x) \right\}$$

Where:

- x is an observable, z is a vector of decisions
- u is a vector of parameters
- $f(z, u, x)$ is the cost function (which can depend on the observable)
- $F(x)$ is the feasible space (which can depend on the observable)
- A training sample $\{(x_i, u_i)\}_{i=1}^m$ from the distribution $P(X, U)$

This is a (slightly) generalized version of the problem class targeted by DFL



A DFL Approach

In principle we can apply "normal" DFL to this problem

First, we define:

$$z^*(u, x) = \operatorname{argmin}_z \{ f(z, u, x) \mid z \in F(x) \}$$

Then, at training time we solve:

$$\theta^* = \operatorname{argmin}_{\theta} \left\{ \mathbb{E}_{(x,u) \sim P(X,U)} [f(z^*(\hat{u}, x), u, x) - f(z^*(u, x), u, x)] \mid \hat{y} = h \right.$$

In practice, if $f(z, u, x)$ is not linear like in our current example

...Then doing it would **not be easy at all**

- We'd need to use a non-linear solver
- ...And the computational cost would be much higher



Another DFL Approach

But we can cheat! Since SFGF enables distinct costs

...We can compute z^* through a **surrogate problem**:

$$z^*(y, x) = \operatorname{argmin}_z \{ \tilde{f}(z, y, x) \mid z \in \tilde{F}(y, x) \}$$

- z is the same decision vector as before
- ...But y is a set of created ad-hoc for the surrogate
- We'll call them **virtual parameters**, because they may have real counterpart

Then:

- $\tilde{f}(z, y, x)$ is a surrogate cost function
- $\tilde{F}(y, x)$ is a surrogate feasible space

For the solution to be valid we need to have $z \in \tilde{F}(y, x) \Rightarrow z \in F(x)$



Another DFL Approach

At training time, we solve:

$$\theta^* = \operatorname{argmin}_{\theta} \left\{ \mathbb{E}_{(x,u) \sim P(X,U)} [f(z^*(y, x), u, x)] \mid y = h(x, \theta) \right\}$$

Intuitively:

- We observe x and we estimate a virtual parameter vector y
- We obtain a decision vector $z^*(y, x)$ through the surrogate problem
- Then we evaluate the cost via the true cost function $f(z, u)$

There is a distinction between the virtual parameter y for $z^*(y, x)$

...And the parameters u for $f(z, u, x)$ are distinct

- For this reason, there is no ground truth for y
- ...Which prevents us from using a regret loss



Motivation

The appeal here is that the surrogate problem can be easier to solve

In our example, instead of using:

$$z^*(u, x) = \operatorname{argmin} \left\{ \sum_{j=1}^n \sin(2\pi u_j z_j) \mid v^T z \geq r, z \in [0, 1]^n \right\}$$

We could use instead the following surrogate:

$$z^*(y, x) = \operatorname{argmin} \left\{ y^T z \mid v^T z \geq r, z \in [0, 1]^n \right\}$$

The surrogate is an LP, so it's very fast to solve

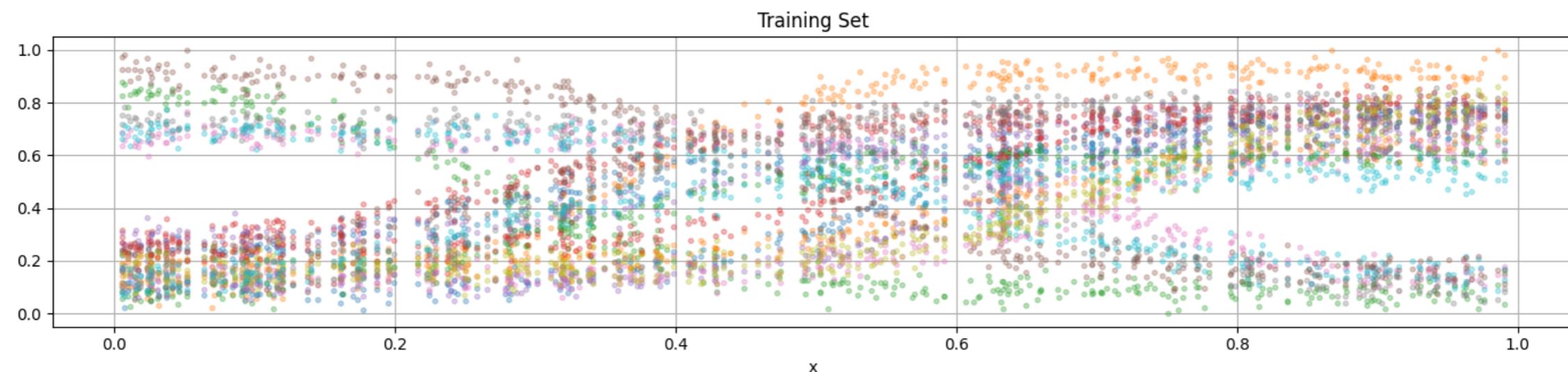
- Together with the ML estimator, it can still lead to high-quality solutions
- As long as the surrogate is sufficiently well aligned with the true problem



Benchmark Data

Let's try a proof-of-concept experiment

```
In [3]: seed, nitems = 42, 20  
data_tr = util.generate_costs(nsamples=350, nitems=nitems, seed=seed, noise_scale=.1, noise_type='uniform')  
util.plot_df_cols(data_tr, figsize=figsize, title='Training Set', scatter=True)
```



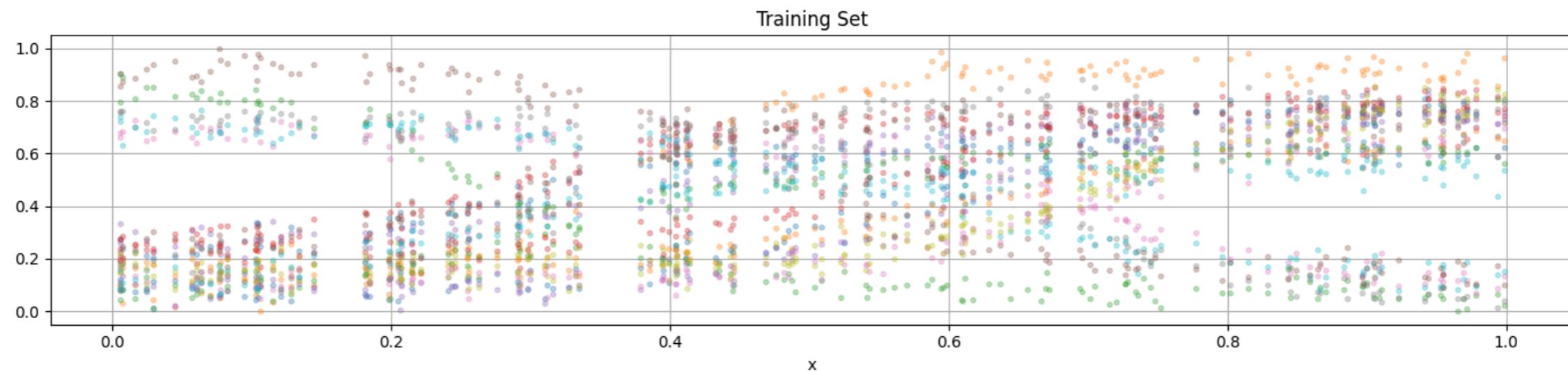
- We generate data for the u parameter as in all previous variants
- We keep the distribution simple, since we want to stress non-linearity



Benchmark Data

Then we generate the remaining problem data and a test set

```
In [4]: rel_req = 0.6
prb = util.generate_problem(nitems=nitems, rel_req=rel_req, seed=seed, surrogate=True)
data_ts = util.generate_costs(nsamples=150, nitems=nitems, seed=seed, sampling_seed=seed+1, noise=noise)
util.plot_df_cols(data_ts, figsize=figsize, title='Training Set', scatter=True)
```



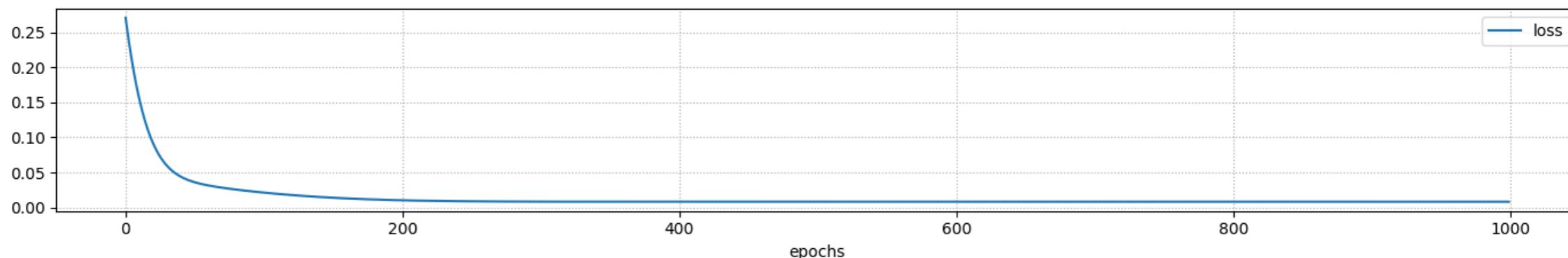
A Baseline

We'll use again a PFL approach as a baseline

Note this is **not a particularly good choice**, but it's difficult to find an alternative

```
In [5]: pfl_nl = util.build_nn_model(input_shape=1, output_shape=nitems, hidden=[], name='pfl_nl', output_activation='softmax')
@time history = util.train_nn_model(pfl_nl, data_tr.index.values, data_tr.values, epochs=1000, verbose=False)
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False)
util.print_ml_metrics(pfl_nl, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(pfl_nl, data_ts.index.values, data_ts.values, label='training')
```

CPU times: user 8.61 s, sys: 256 ms, total: 8.87 s
Wall time: 7.16 s



R2: 0.84, MAE: 0.076, RMSE: 0.09 (training)
R2: 0.84, MAE: 0.079, RMSE: 0.09 (training)

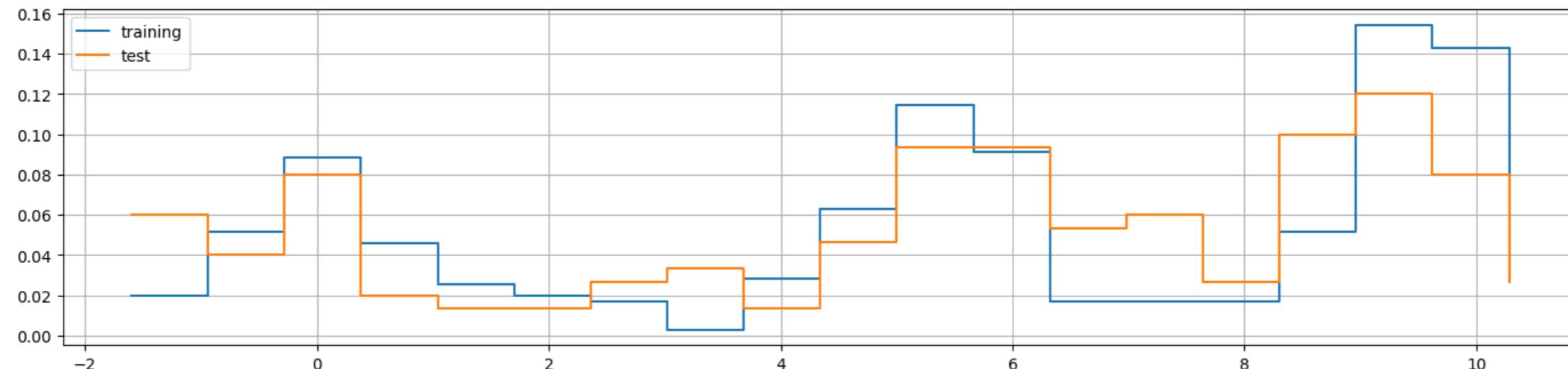


PFL Approach Evaluation

We'll also evaluate the results in terms of cost, not relative regret

Again, the reason is that this problem is not so easy to solve

```
In [6]: tc_tr_nl = util.compute_regret_surrogate(prb, pfl_nl, data_tr, tlim=10, cost_only=True)
tc_ts_nl = util.compute_regret_surrogate(prb, pfl_nl, data_ts, tlim=10, cost_only=True)
util.plot_histogram(tc_tr_nl, figsize=figsize, label='training', data2=tc_ts_nl, label2='test',
```



Mean: 5.691 (training), 5.497 (test)



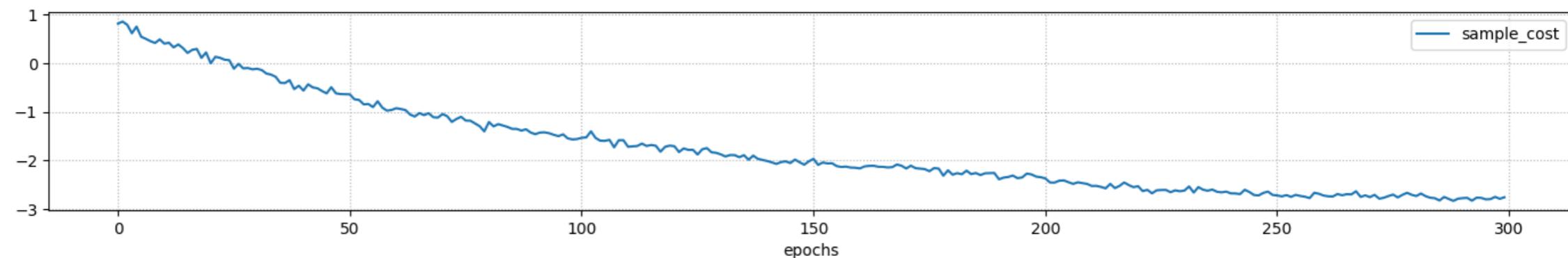
Alternative DFL Approach

We can now try out alternative DFL approach

In this case, warm starting may not be a good idea

```
In [7]: sfge_sg = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[], name='sample_cost')
@time history = util.train_dfl_model(sfge_sg, data_tr.index.values, data_tr.values, epochs=300,
util.plot_training_history(history, figsize=figsize_narrow, print_final_scores=False, excluded_r
util.print_ml_metrics(sfge_sg, data_tr.index.values, data_tr.values, label='training')
util.print_ml_metrics(sfge_sg, data_ts.index.values, data_ts.values, label='test')
```

CPU times: user 3min 35s, sys: 3.43 s, total: 3min 38s
Wall time: 3min 38s



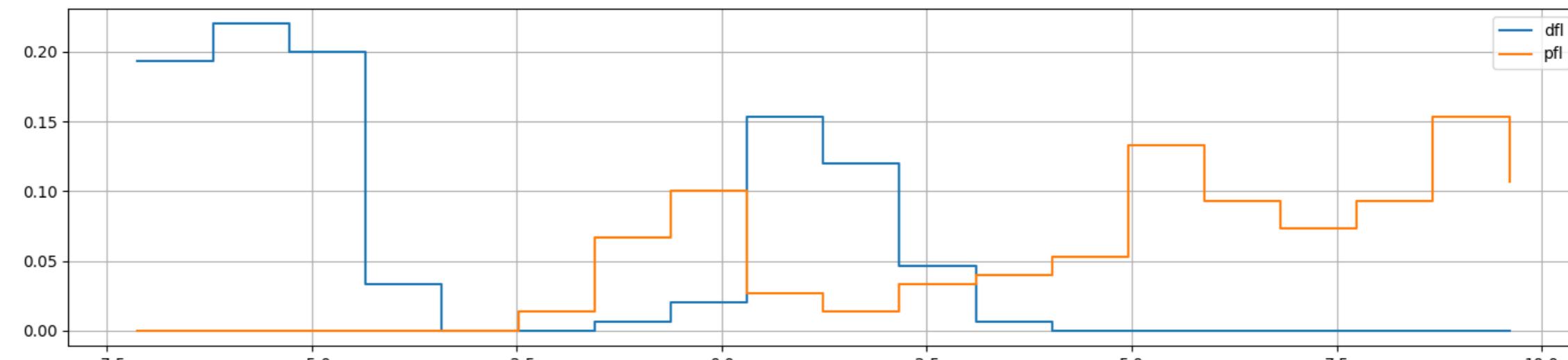
R2: -6.83, MAE: 0.5, RMSE: 0.61 (training)
R2: -6.54, MAE: 0.5, RMSE: 0.61 (test)



DFL Approach Evaluation

...And we can compare the two cost distributions on the training data

```
In [8]: tc_tr_sg = util.compute_regret_surrogate(prb, sfge_sg, data_tr, tlim=10, cost_only=True)
tc_ts_sg = util.compute_regret_surrogate(prb, sfge_sg, data_ts, tlim=10, cost_only=True)
util.plot_histogram(tc_ts_sg, figsize=figsize, label='dfl', data2=tc_ts_nl, label2='pfl', print_
```



Mean: -3.178 (dfl), 5.497 (pfl)

The difference is very noticeable



Considerations

By using DFL + a surrogate we can "partition" the problem complexity

- We can simplify some elements that the solver has trouble addressing
- ...And dump them partially on the ML model
- ...Or we can do the opposite (e.g. hard constraints in ML based decision making)
- It can work without an observable (no x)
- It could be used for black-box optimization

Some caveats:

- This is not a well investigate approach: treat is as a proof-of-concept
- Banning special case, the method works as a heuristic
- ...And finding a good surrogate can be quite difficult





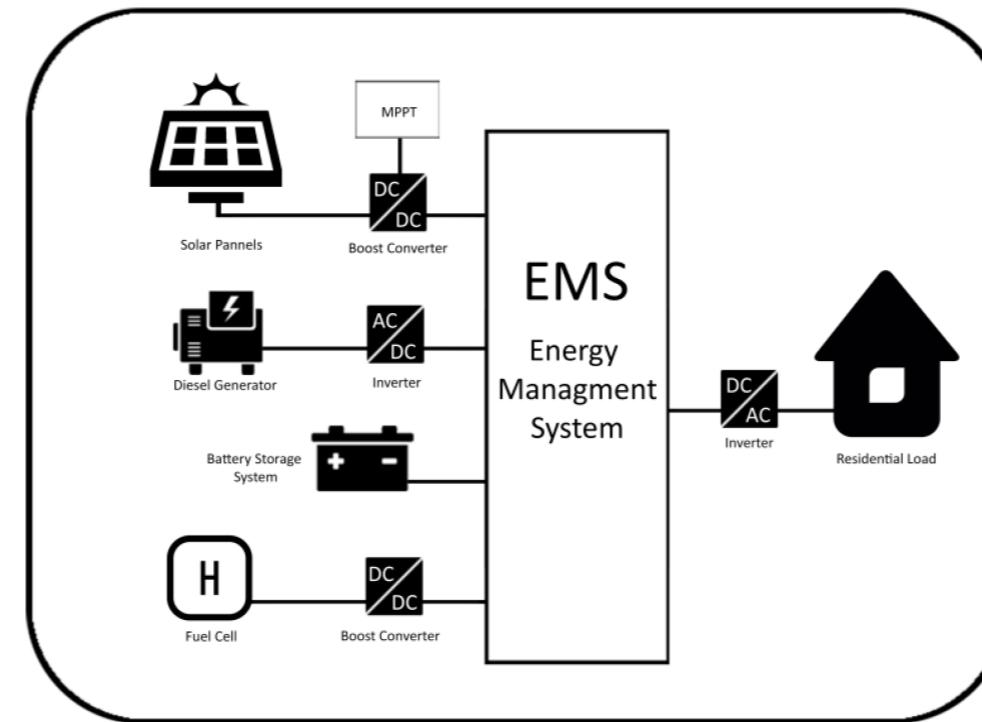
05. Last Leg of the Journey



Multi-Stage Stochastic Optimization

What if we have a **sequence** of decision stages?

Consider for example and Energy Management System:



- We need to make some decisions (using a generator, buying from the grid...)
- ...Then observe how uncertainty unfolds
- ...Based on that, we make another round of decisions and so on



Multi-Stage Stochastic Optimization

We will also assume that there are **non-trivial constraints**

- This setup is called multi-stage stochastic optimization
- ...Or also online stochastic optimization, or sequential decision making

There are a few possible solution approaches

One approach consist in using scenarios, again

- ...But since there are many stages, the decisions variables branch out
- A solution is called a policy tree, which is **very** expensive to compute

A second approach consists in using anticipatory algorithms

- We iteratively solve an optimization problem with a bit of look-ahead
- Several examples can be found in [1]

[1] Hentenryck, Pascal Van, and Russell Bent. *Online stochastic combinatorial optimization*. The MIT Press, 2006.



Formalization

Formally, this setup is well captured by a constrained Markov Decision Process (MDP)

In particular, we will consider a **constrained** $\langle X, Z, P^0, P, f, F \rangle$ be an MDP with:

- A set of possible (observable) states X
- A set of possible decisions Z
- A distribution $P^0(X)$ for the initial state
- A distribution $P(X | X, Z)$ for the possible state transitions
- A cost function $f(z, x, x^+)$
- A feasible space $F(x)$ which depends on the state

Some comments:

- The next state depends on the current state and decisions
-  The cost depends on the current state and decisions, and on the next state

Formalization

Within this framework, we can formalize a multi-stage problem

Our goal is to define a solution policy π^* from a set of candidates Π s.t.:

$$\pi^* = \operatorname{argmin}_{\pi \in \Pi} \mathbb{E}_{x^0 \sim P^0, x^{t+1} \sim P(X|x^t, z^t)} \left[\sum_{t=1}^{eoh} f(z^t, x^t, x^{t+1}) \right]$$

subject to: $z^t = \pi(x^t)$

$z^t \in F(x^t)$

This is very complex problem:

- We are not searching for a fixed solution, but for a policy
- The decisions can be anything (including discrete and combinatorial)
- ...They affect the state at the next stage (endogenous uncertainty)
- ...And they should be feasible according to hard constraints



Solution Approach Wanted

Normally, with an MDP we may turn to Reinforcement Learning

...But in this case there are a couple of difficulties:

- Handling constraints (hard ones in particular) in RL is challenging
- Handling combinatorial decisions in RL is **very** challenging

Let's recap our situation

- Classical approaches from stochastic optimization have poor scalability
- RL approaches have poor support for constraints and combinatorial spaces

Can we use DFL in this scenario?

[1] Garcia, Javier, and Fernando Fernández. "A comprehensive survey on safe reinforcement learning." *Journal of Machine Learning Research* 16.1 (2015): 1437-1480.

DFL and RL (UNIFY)

Indeed we can, and at this point it's not even that difficult

The trick is simply to decompose the policy π , leading to:

$$\theta^* = \operatorname{argmin}_{\theta} \mathbb{E}_{x^0 \sim P^0, x^{t+1} \sim P(X|x^t, z^t)} \left[\sum_{t=1}^{eoh} f(z^t, x^t, x^{t+1}) \right]$$

$$\text{subject to: } z^t = z^*(y^t, x^t)$$

$$y^t = h(x^t, \theta)$$

Intuitively:

- We use a ML model to output a set of virtual parameters y
- ...Then we compute z^k by solving a constrained optimization problem
- The ML model take care of uncertainty
- The optimization problem take care of the constraints



DFL and RL (UNIFY)

We use the generalized, surrogate-based approach to compute z^*

In particular, we have:

$$z^*(y, x) = \operatorname{argmin}_z \{ \tilde{f}(z, y, x) \mid z \in \tilde{F}(y, x) \}$$

- Depending on our choice for the virtual parameters
- We will need to craft the surrogate cost \tilde{f} and feasible space \tilde{F}
- The original constraints are satisfied as long as $z \in F(y, x) \Rightarrow z \in F(x)$

The surrogate terms can usually be designed by tweaking a bit f and F

The overall idea is that the ML model guides the optimizer,
exactly as in normal DFL



DFL and RL (UNIFY)

For training, we can rely on a simple reformulation

In particular, we define a new **unconstrained** MDP $\langle X, \Theta, P^0, P, g \rangle$ such that:

- The set of states is the same as before
- The set of states is the set Θ of possible training parameters
- The state transition distributions are the same as before
- The cost function is defined as:

$$g(y, x, x^+) = f(z^*(y), x, x^+)$$

Intuitively, we treat the solver as part of the environment

This new MDP can be addressed by **any RL learning approach**

so we can benefit from recent advances in such field



DFL and RL (UNIFY)

This setup is the most general we have seen so far

It can be used to address a wide number of problem types

- Optimization with parameters that need to be estimated
- One-stage stochastic programming
- Two-state stochastic programming
- Sequential decision making with constraints
- In principle, also black-box optimization and parameter tuning
- ...Though it probably would not a good fit for such cases

You can find it described in [1], under the name **UNIFY**

[2] Silvestri, Mattia, et al. "UNIFY: a Unified Policy Designing Framework for Solving Constrained Optimization Problems with Machine Learning." arXiv preprint arXiv:2210.14030 (2022).



An Example

Let's consider the Energy Management System example in detail

Every 15 minutes, we need to adjust power flow to/from a set of nodes

- Nodes can be generators, demand points, or the grid
- One special node represents a storage system

The decisions z^t at time t include:

- A vector of power flows z_{nodes}^t to/from the main nodes
- A power flow $z_{storage}^t$ to/from the storage system

The state x^t at time t is given by:

- The power x_{power}^t generated by some nodes (e.g. PV plants)
- The demand x_{demand}^t for some nodes (e.g. production sites or housing)
- The storage charge level $x_{storage}^t$



An Example

The transition distribution P is defined by:

- A distribution P_{power} of the yield of renewable energy generators
- A distribution P_{demand} of the demand
- The deterministic transition $x_{storage}^{t+1} = x_{storage}^t + \eta z_{storage}$

The feasible space $F(x^t)$ is defined via:

- Flow capacity constraints: $l \leq z^t \leq t$
- Flow balance constraints: $1^T z + x_{power} - x_{demand} = 0$
- Storage capacity constraints $0 \leq x_{storage}^t + \eta z_{storage} \leq C$

The cost $f(z^t, x^t, x^{t+1})$ is given by:

$$f(z^t, x^t, x^{t+1}) = c^T z_{nodes}$$

- There is no cost associate to demands, renewable generators, and the storage

The Optimization Problem

We can compute $z^*(y, x)$ by solving the following LP

$$\begin{aligned} & \operatorname{argmin}_z c^T z_{\text{nodes}} + yz_{\text{storage}} \\ \text{subject to: } & l \leq z^t \leq t \\ & 1^T z + x_{\text{power}} - x_{\text{demand}} = 0 \\ & 0 \leq x_{\text{storage}}^t + \eta z_{\text{storage}} \leq C \end{aligned}$$

The main alteration is that a virtual cost is associated to the storage system

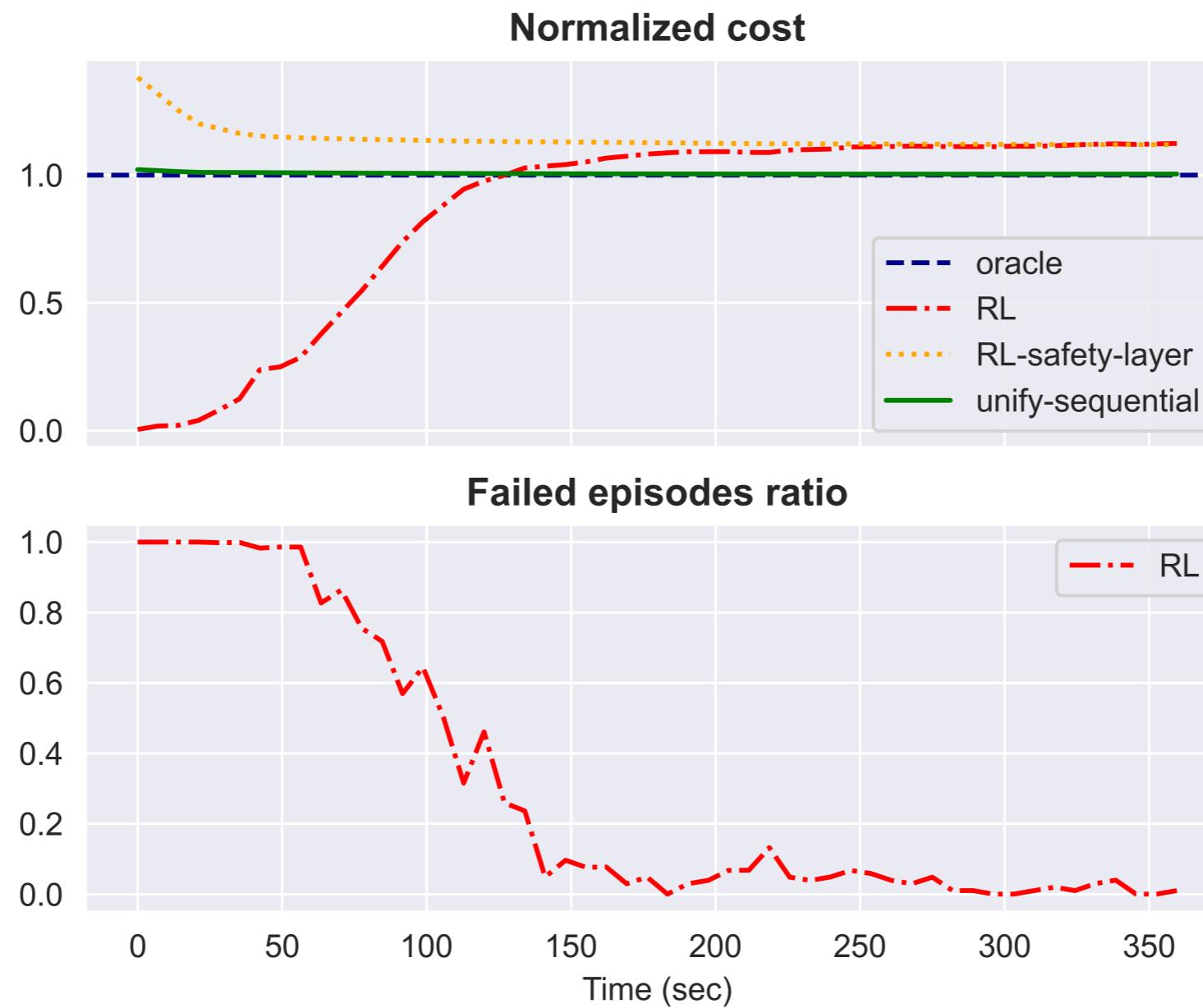
- If $y > 0$, the solve will tend to charge the storage
- If $y < 0$, the solve will tend to draw power from the storage
- ...So that the ML model can alter the decisions

Without the virtual cost, the storage system would never be charged



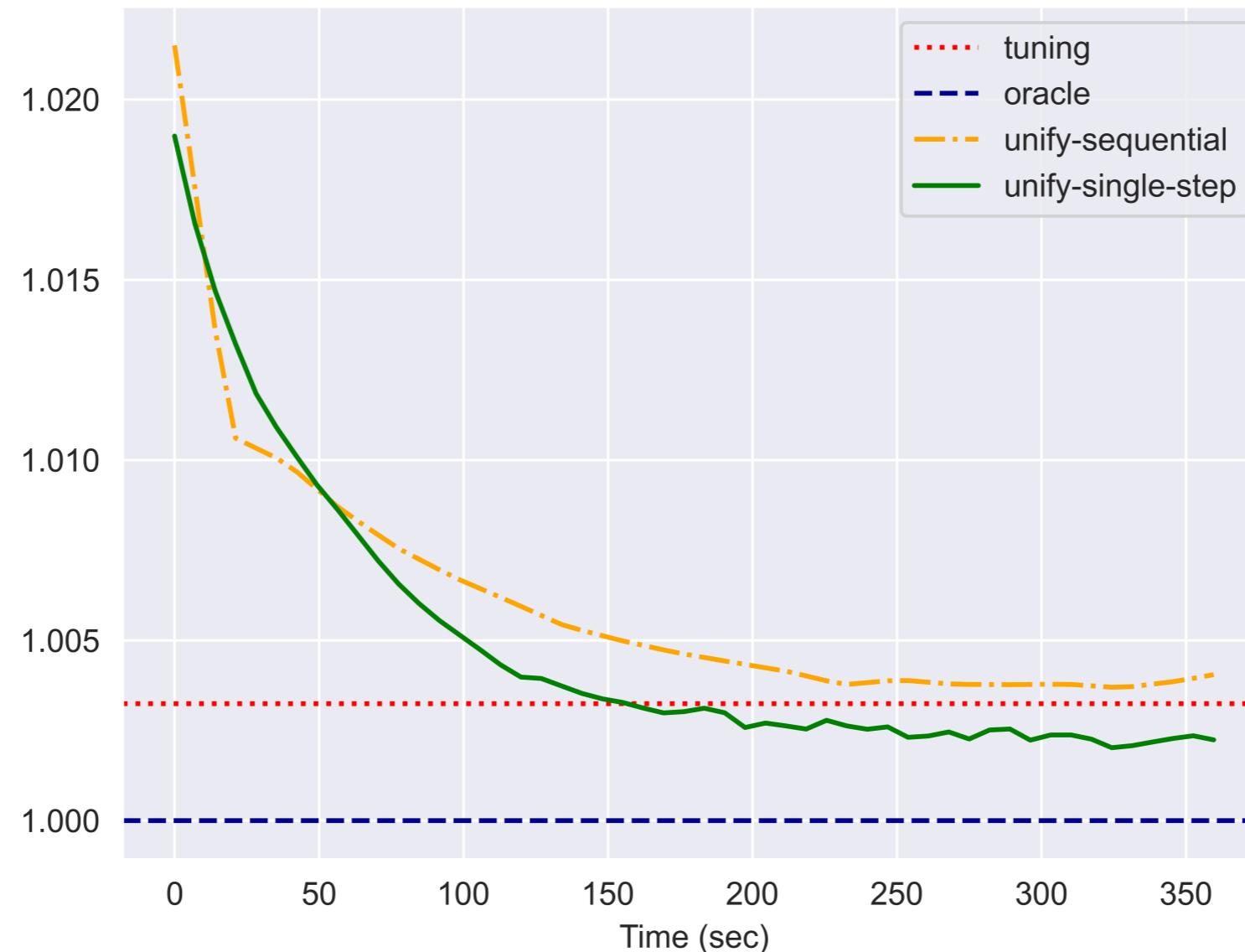
Some Results

Here's a comparison with some constrained RL methods



Some Results

And here's a comparison with a specialized stochastic optimization approach



Some Final Thoughts

If you retain one idea from our ramble, makes sure it is this:

DFL can be used for **way more than one purpose!**

You just need to stretch it a little bit ;-)

Where next?

- We can reap what we haven't sowed! Let's test more RL algos (spoiler: started)
- Scalability is still a big issue
- We need more (and more realistic) applications
- ...



Thanks for your patience! Any question?

