



# Rambling Away from Decision Focused Learning

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

Michele Lombardi <[michele.lombardi2@unibo.it](mailto:michele.lombardi2@unibo.it)>



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

Many thanks to:

**Senne Berden, Victor Bucarey, Allegra De Filippo, Michelangelo Diligenti, Tias Guns, Irfan Mahmudogullari, Jayanta Mandi, Michela Milano, Maxime Mulamba, Mattia Silvestri**



**This presentation is in fact a tutorial:**

[https://github.com/lompabo/acp\\_summer\\_school\\_2023](https://github.com/lompabo/acp_summer_school_2023)

All code is executable (but the SW engineering is awful)





## 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

**Say we are browsing wares in a 2nd hand shop**

We want to get the best deal for our budget

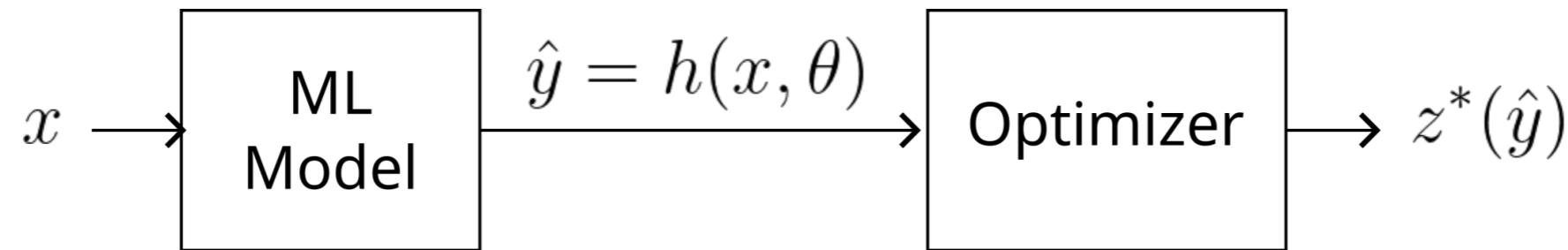


- We can look at the displayed items
- ....But we don't know their actual value



# 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  $\theta$ :

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

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



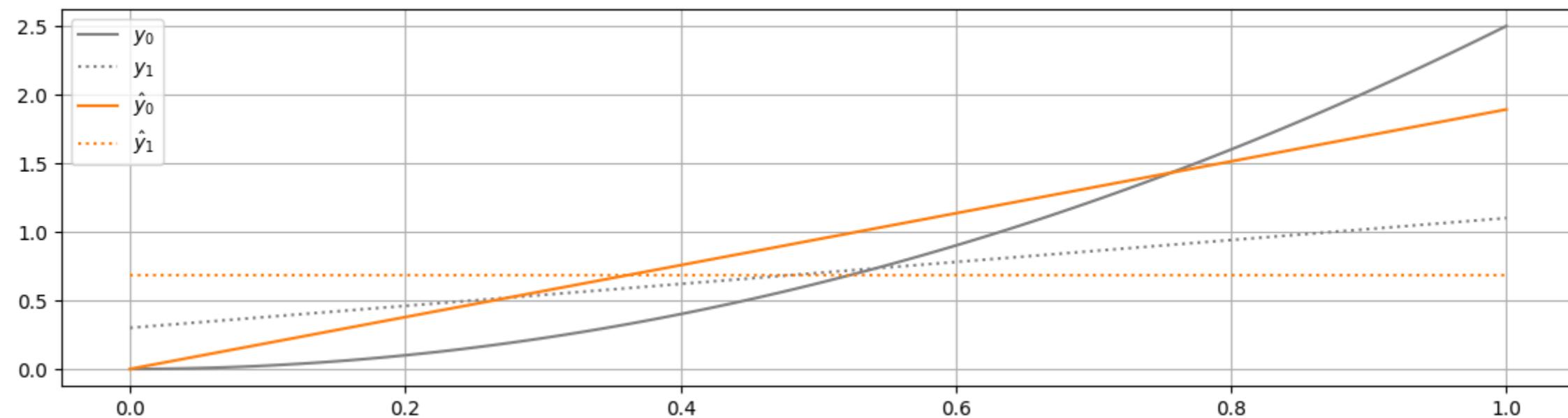
Our model cannot represent the true relation exactly

# Spotting Trouble

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

```
In [2]: util.draw(w=None, figsize=figsize, model=1)
```

Optimized theta: 1.375



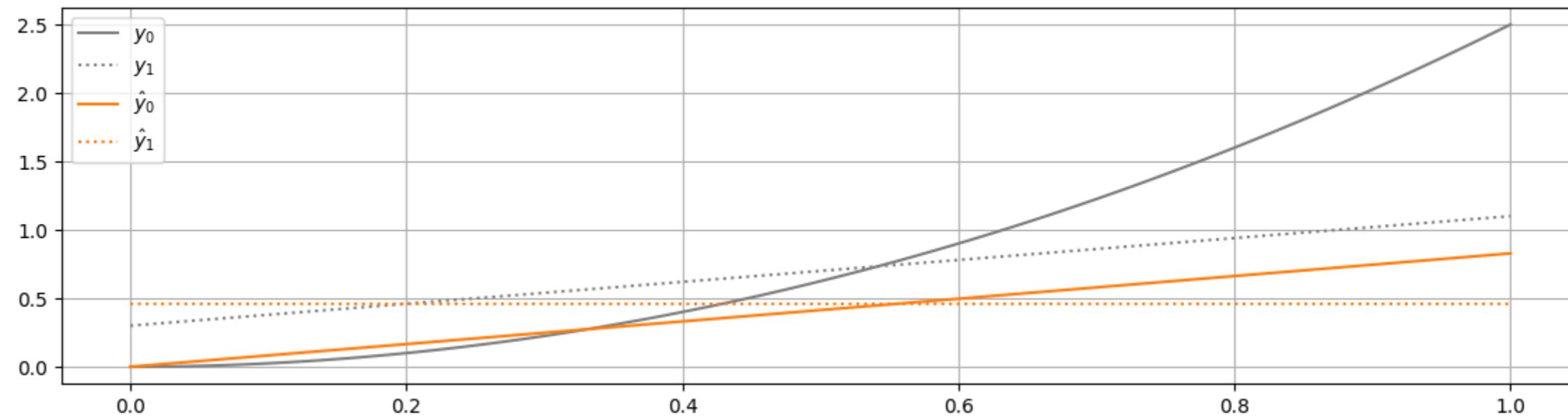
- The crossing point of the grey lines is where we should pick item 0 instead of 1
- The orange lines (trained model) miss it by a wide margin



# Not All is Lost

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

```
In [3]: util.draw(w=0.91, figsize=figsize, model=1)
```



- If we focus on choosing  $\theta$  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

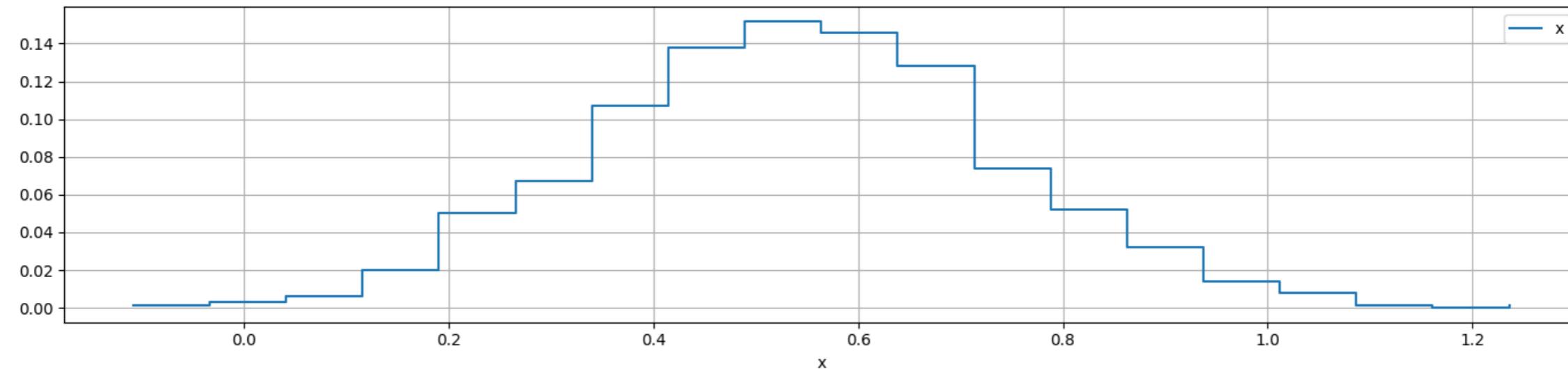
**One of the main challenges in DFL is dealing with this loss**



# Knowing Regret

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

```
In [4]: x = util.normal_sample_(mean=0.54, std=0.2, size=1000)  
util.plot_histogram(x, figsize=figsize, label='x')
```



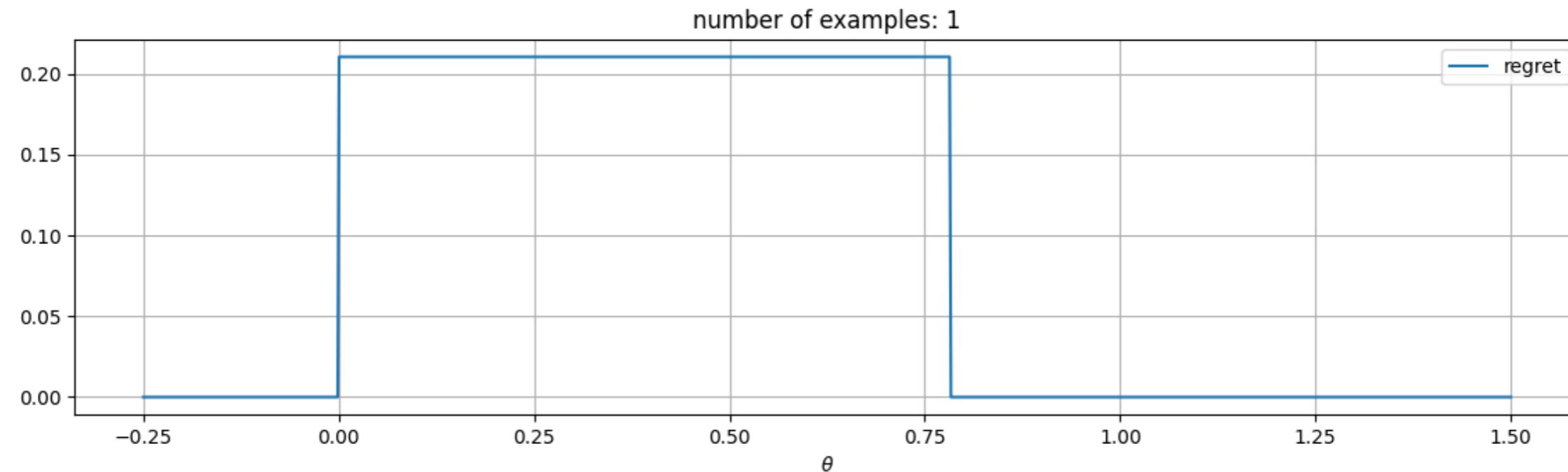
- Say we have access to a normally distributed collection of  $x$  values
- ...And to the corresponding true values  $y$



# Knowing Regret

This is how the regret looks like for a single example

```
In [5]: util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=1, figsize=figsize)
```



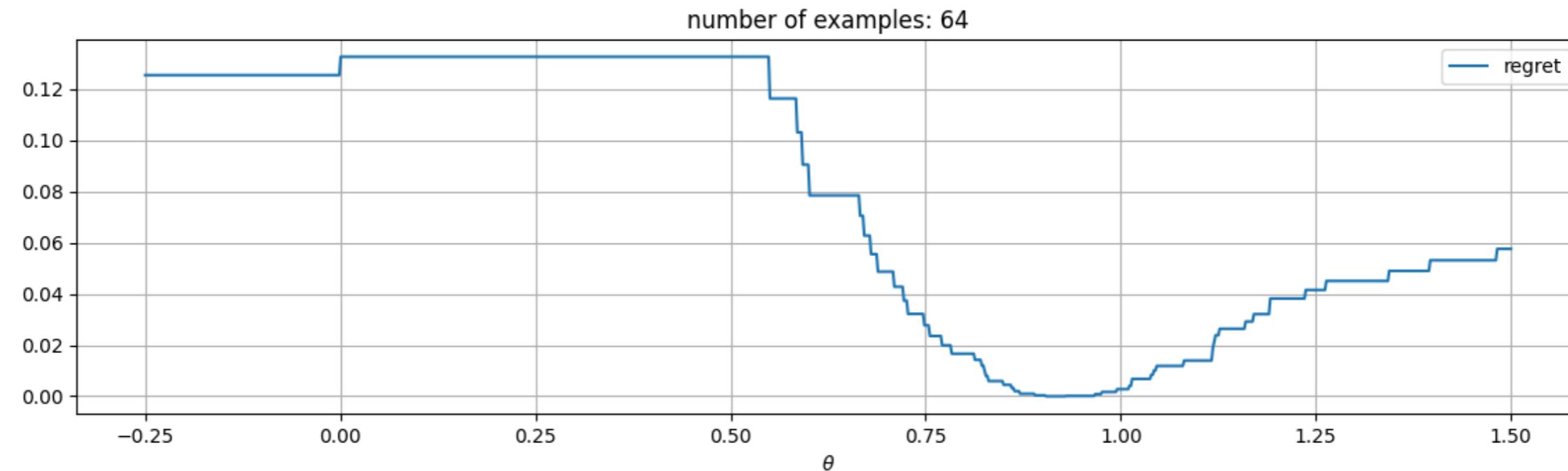
- If  $f(x, \theta)$  leads to the correct decision, the regret is 0
- Otherwise we have some non-null value



# Knowing Regret

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

```
In [6]: 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

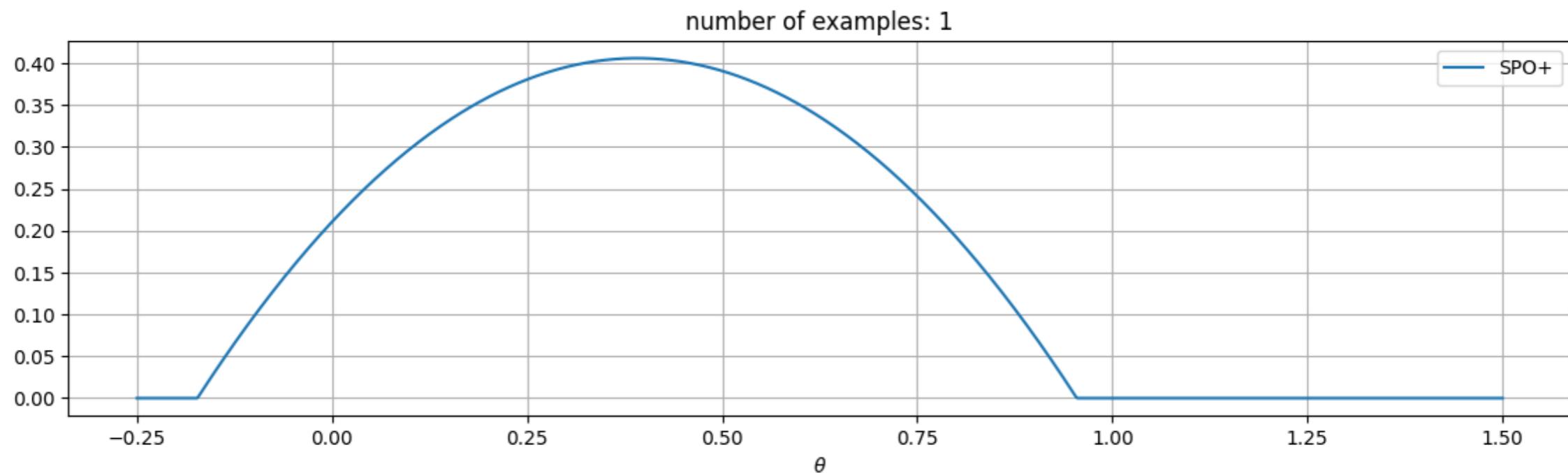
[1] Elmachtoub, Adam N., and Paul Grigas. "Smart ‘predict, then optimize’." *Management Science* 68.1 (2022): 9-26.



## 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)
```



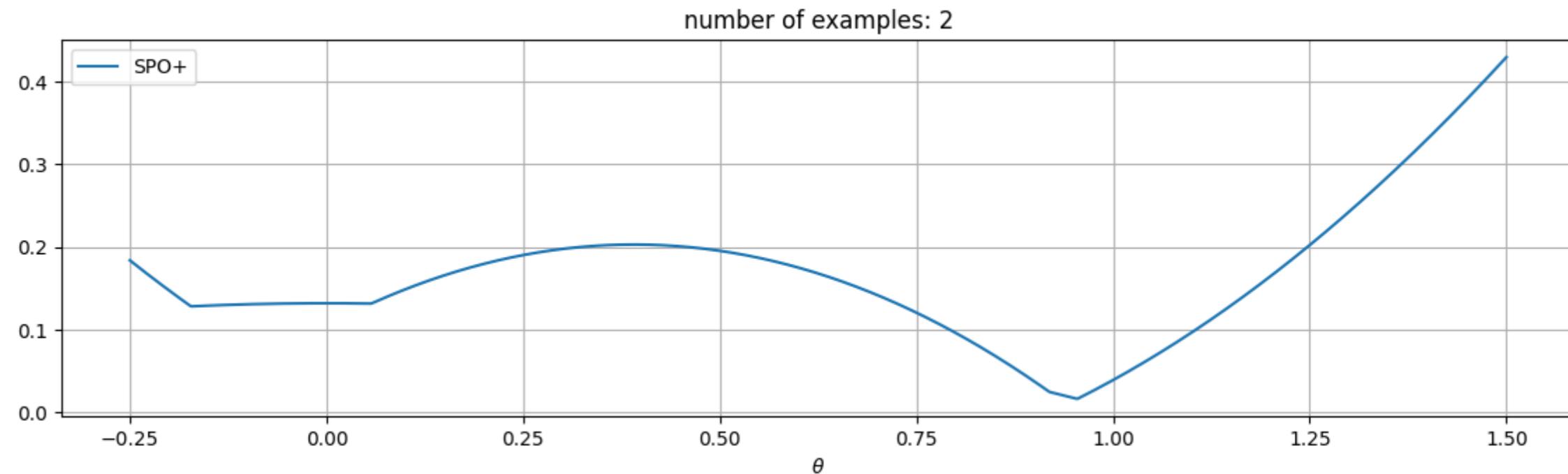
- As expected, there are two local minima



# 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)
```



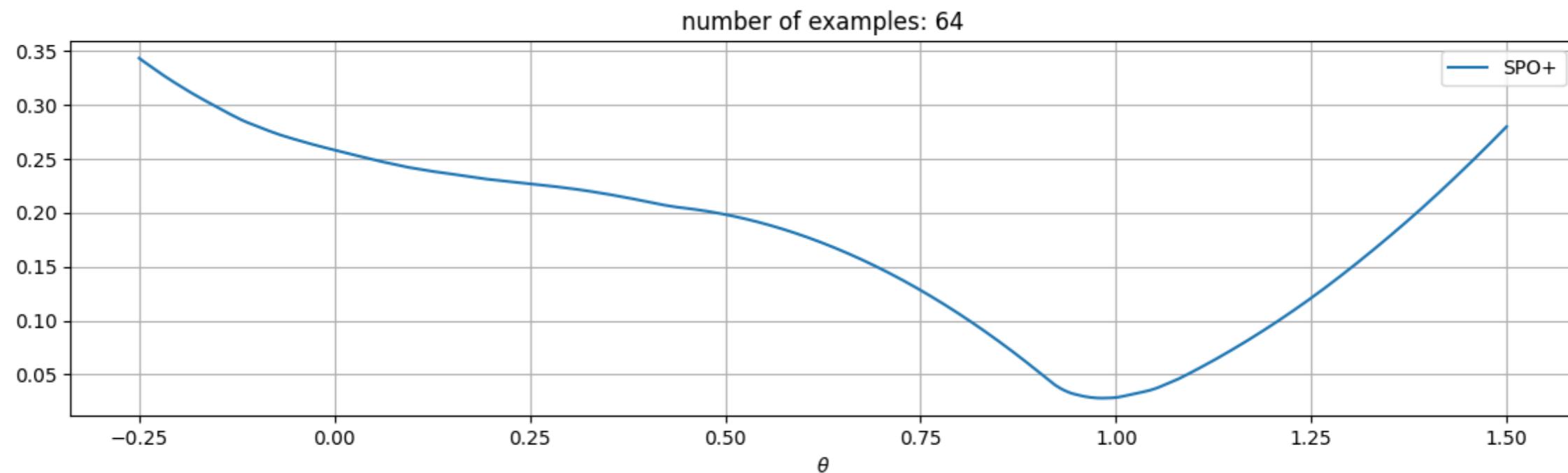
- The "good" local minima for both examples are roughly in the same place
- The "spurious" local minima fall in different position



## 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)
```

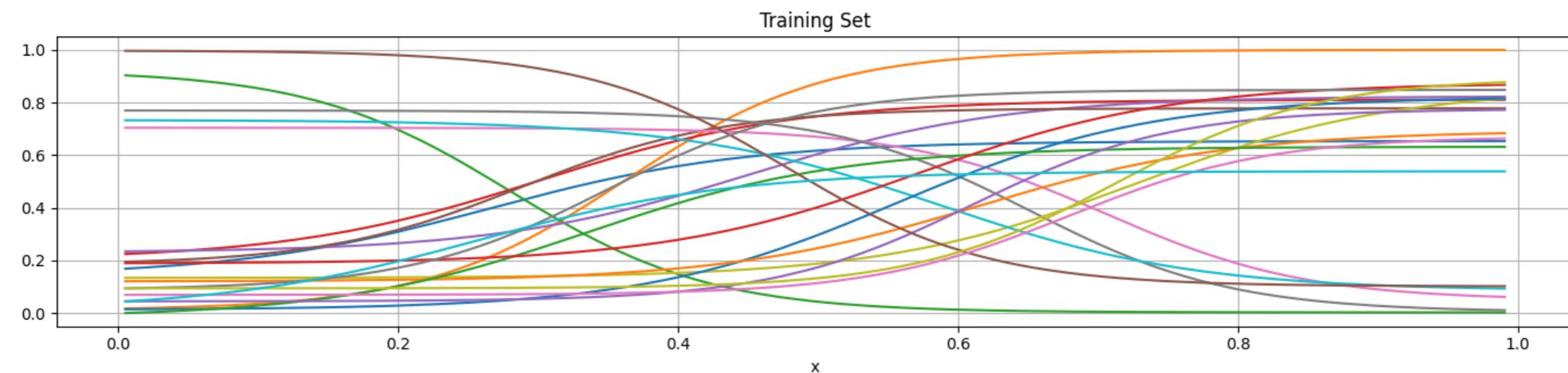
```
ProductionProblem(values=[1.14981605 1.38028572 1.29279758 1.23946339 1.06240746 1.06239781
 1.02323344 1.34647046 1.240446 1.28322903 1.0082338 1.38796394
 1.33297706 1.08493564 1.07272999 1.0733618 1.1216969 1.20990257
 1.17277801 1.11649166], requirement=11.830809153591138)
```



# 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  $x$



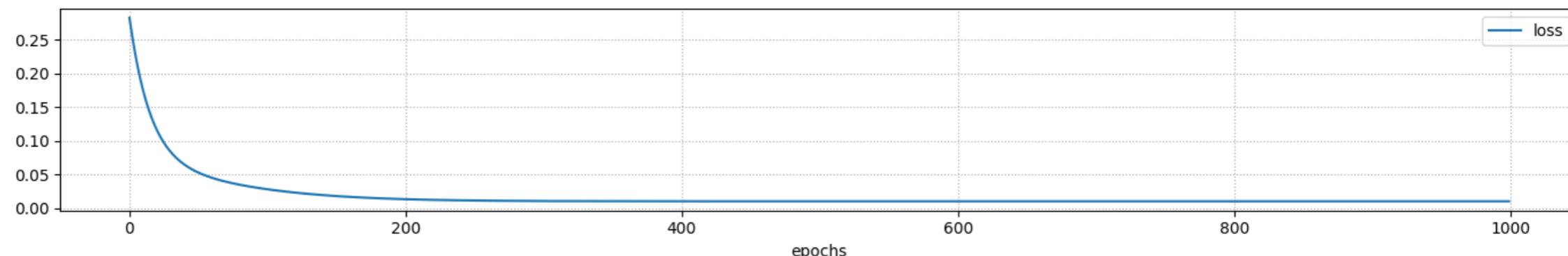
# Prediction Focused Approach

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)

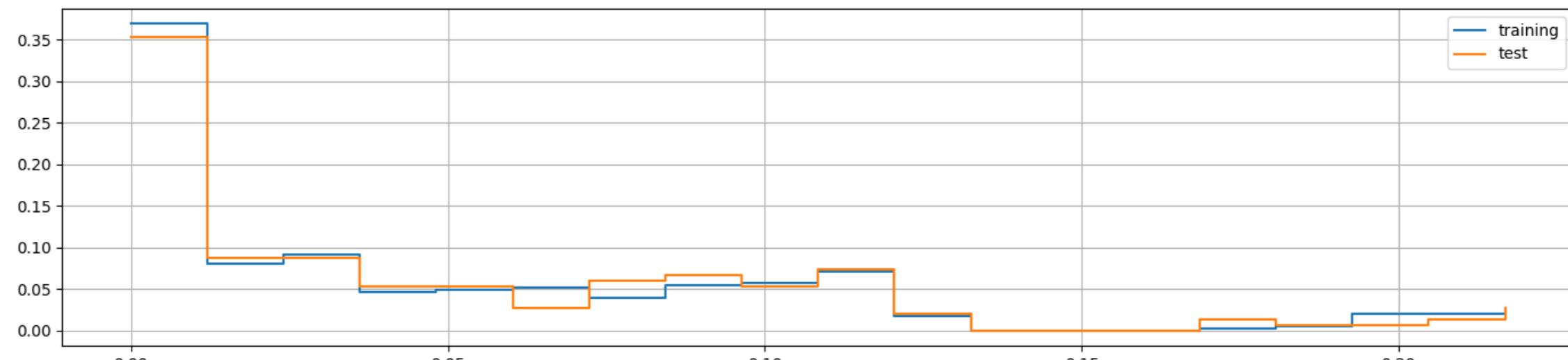


The ML model is just a linear regressor, but it is decently accurate

# 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%



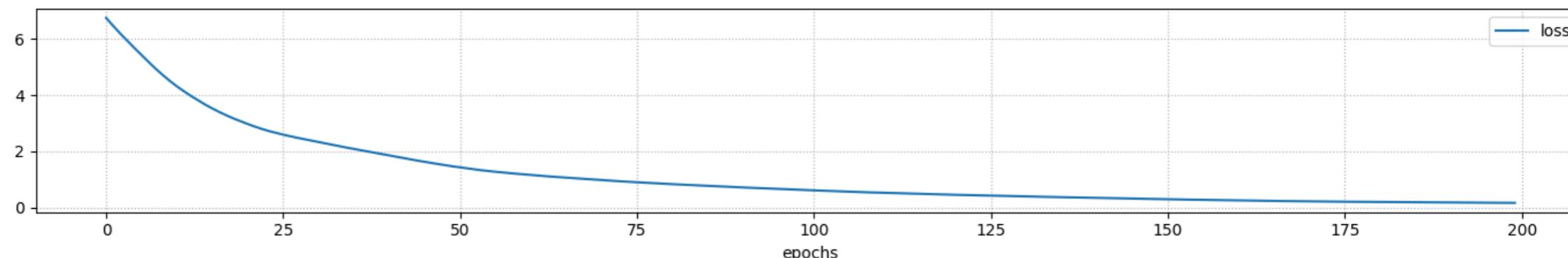
# A Decision Focused Learning Approach

Next, we train a DFL approach

```
In [27]: spo = util.build_dfl_ml_model(input_size=1, output_size=nitems, problem=prb, hidden=[], name='spo')
%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 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)

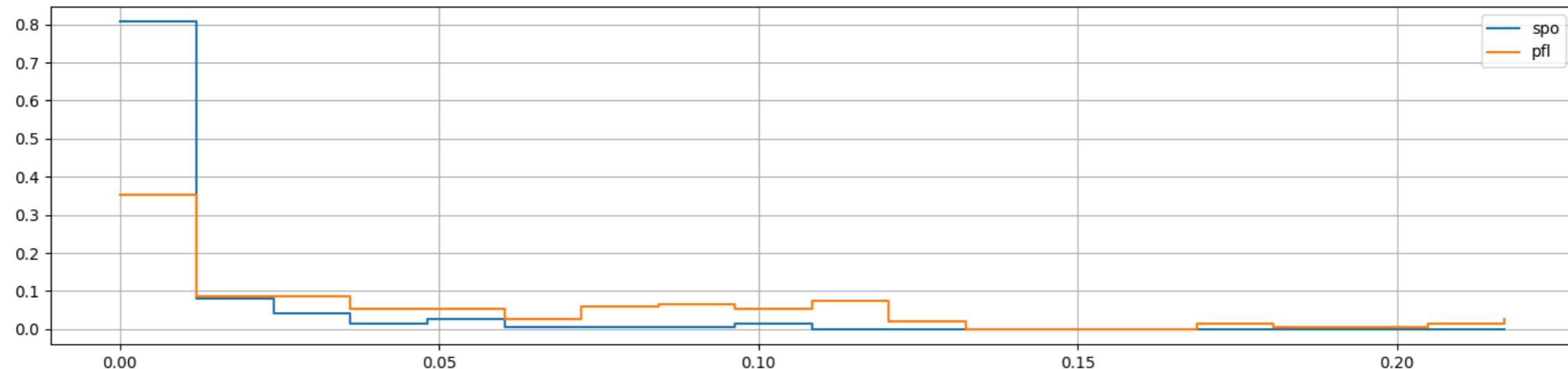


In terms of accuracy, this is considerably worse

# 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).

