From PFL to DFL

Eyes on the prize

Prediction and Optimization in the Wild

Real world problems typically rely on estimated parameters

E.g. travel times, demands, item weights/costs...

However, sometimes we have access to a bit more information

Prediction and Optimization in the Wild

Take traffic-dependent travel times as an example

If we know the time of the day we can probably estimate them better

Let's see how these problems are often addressed

Predict...

First, we train an estimator for the problem parameters:

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

- L is the single-example loss (typically a likelihood)
- h is the estimator, with parameter vector θ
- $P(X, Y)$ is the data distribution
- ...Which will typically be approximated via a sample (training set)

In our example:

- x would be the time of the day
- y would be a vector of travel times

...Then Optimize

Then, we solve the optimization problem with the estimated parameters

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

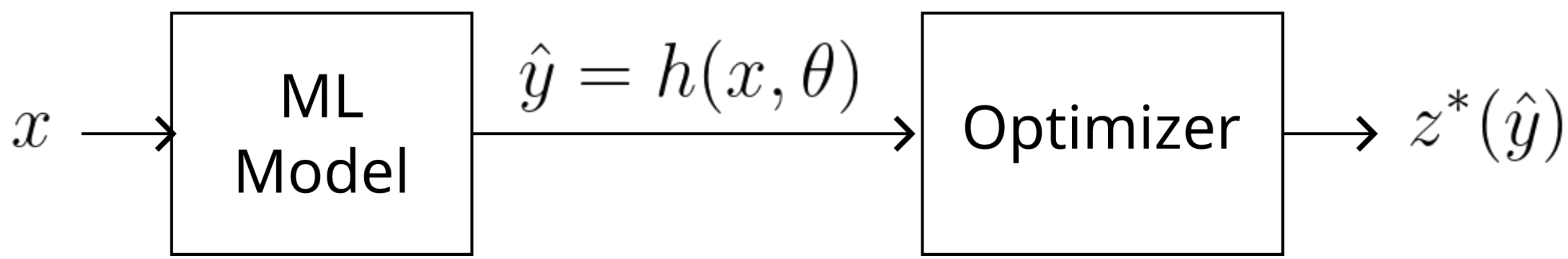
- z is the vector of variables of the optimization problem
- f is the cost function
- F is the feasible space
- In general, both c and F may depend on the estimated parameters

In our example

- z may represent routing decisions
- f may be the total travel time
- F may encode a deadline constraint

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:

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

Prediction Focused Learning

This two-stage approach used to have no name at all

These days, it is referred to as:

- Predict, then Optimization
- ...Ore Prediction Focused Learning

PFL has several favorable properties:

- It's easy to implement
- ...It has good scalability
- ...And it's asymptotically correct (perfect predictions result in minimum cost)

Application fields include logistics, planning, finance, etc.

However, the method has also a significant flaw

One that was only recently emphasized

A Toy Problem

Let's see this in action on a toy problem

Consider this two-variable optimization problem:

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

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 the following ML model with a scalar weight θ:

$$\hat{y}_0 = \theta^2 x$$
$$\hat{y}_1 = 0.5\theta$$

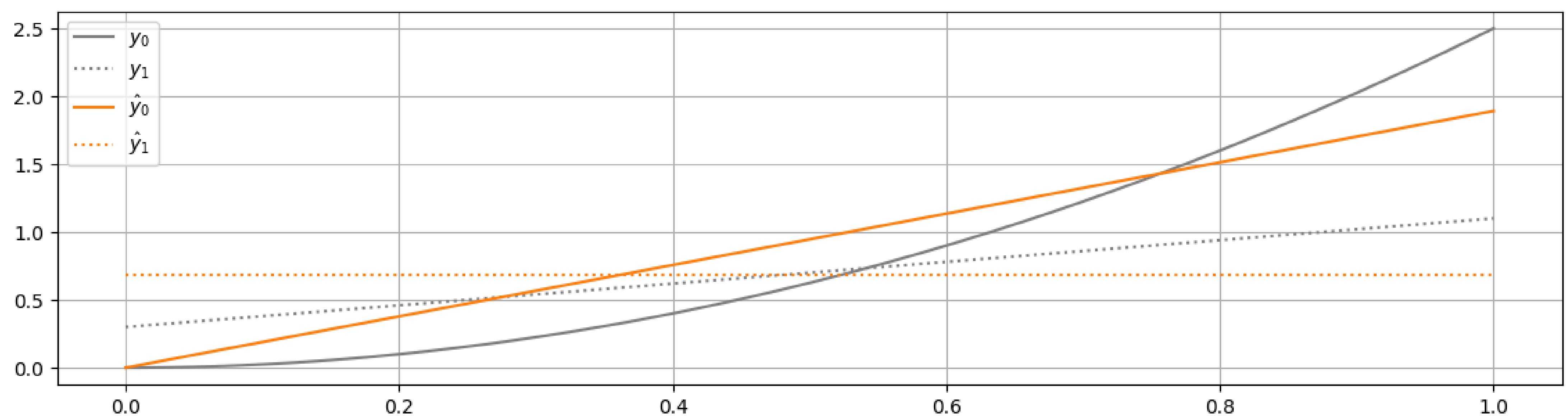
Our model cannot represent the true relation exactly

Spotting Trouble

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

```
In [182]: 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

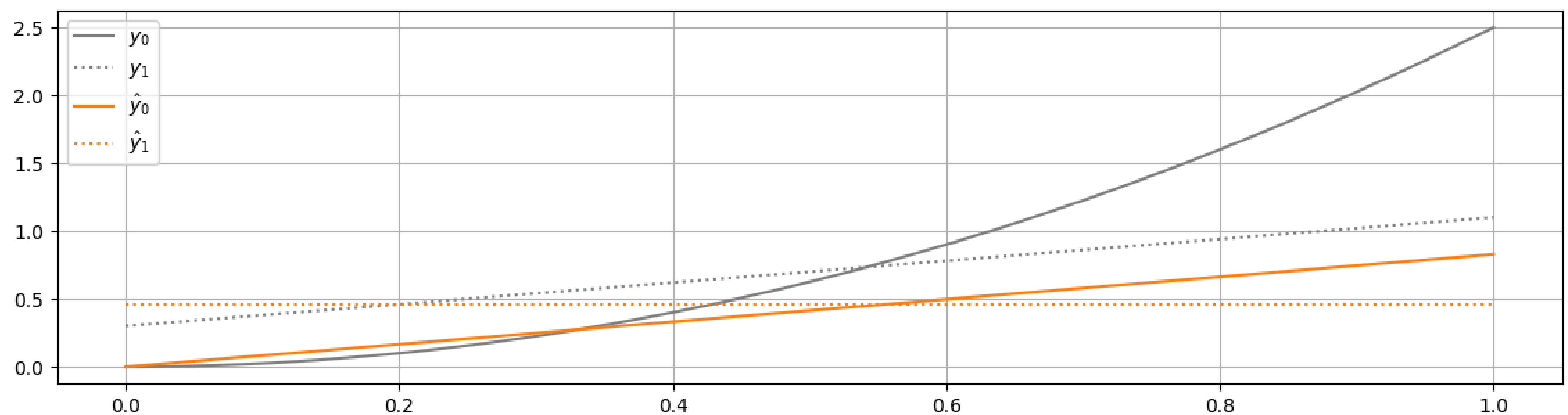
Hence, if we optimize based on our best predictions, we make a mistakes!

...But why is this happening?

Misaligned Objectives

We trained for maximum accuracy regardless of the decision cost!

```
In [183]: util.draw(w=0.91, figsize=figsize, model=1)
```



- However, if we focus on choosing θ to match the crossing point
- ...The same model can lead the optimizer to consistently to the correct choice

We just need to train for **minimanl decision cost**, which is key idea in **Decision Focused Learning**

Getting Started

We'll start withe setup considered in one of the DFL seminal papers

We consider an optimization problem in the form:

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

- z is the set of decisions
- F is the feasible space
- y is a cost vector

The y parameters cannot be masured

...But they depend on some observable x

- And both can be represented as random variables with a joint distribution:

$$X, Y \sim P(X, Y)$$

Getting Started

So, in practice we can estimate y via a ML model

$$\hat{y} = h(x; \theta)$$

...And at inference time we get our decisions by computing:

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

I.e. exactly as in Prediction Focused Learning

The key assumption is the use of a linear cost function

...And the lack of dependence of the constraints on y

- The constraint can otherwise be anything (including integrality)
- ...And thryb could also depend on the observable, i.e. $F \equiv F(x)$

The Main DFL Idea

The key difference between PFL and DFL is the training process

...Whic in DFL is done by minimizing a decision cost, i.e. by solving:

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

Where in our setting we have:

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

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

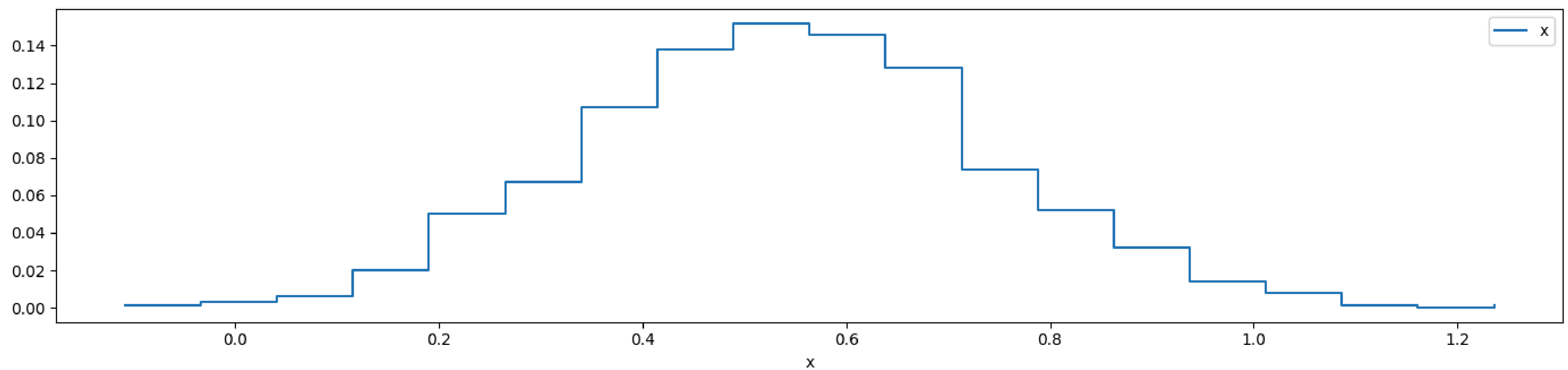
Intuitively, we want to loose as little as possible w.r.t. the best we could do

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 [184]:
```
x = util.normal_sample_(mean=0.54, std=0.2, size=1000)
util.plot_histogram(x, figsize=figsize, label='x')
```

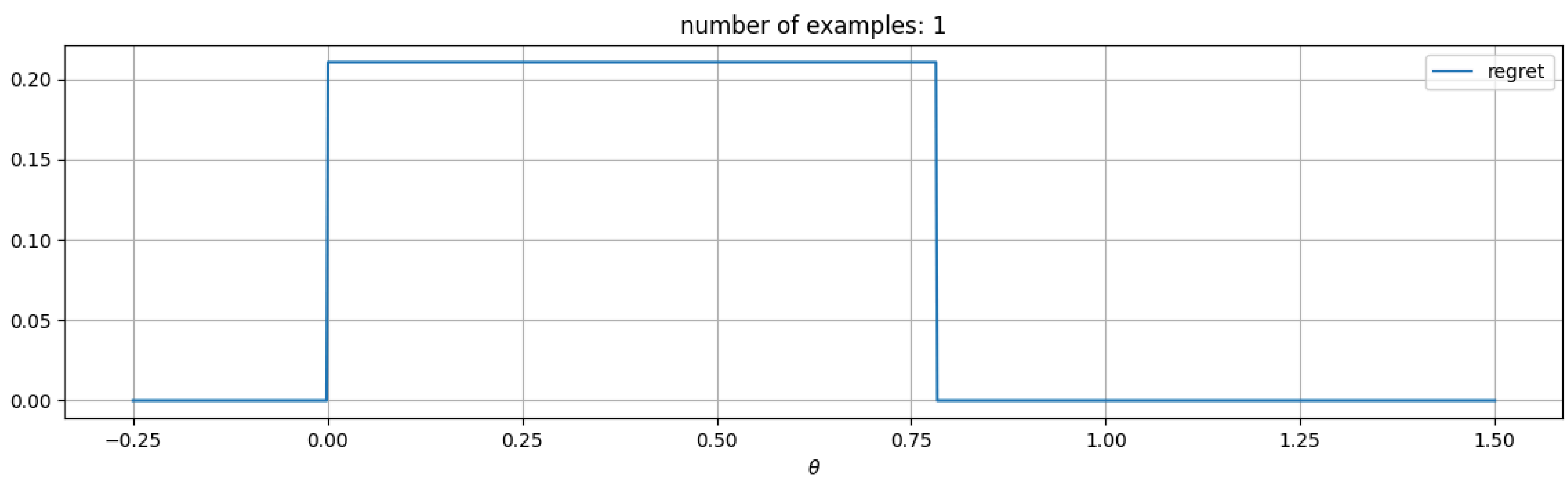


- Say we have access to a normally distributed collection of x values
- ...And to the corresponding true values y

Knowing Regret

This is how the regret looks like for a single example

```
In [185]: util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=1, figsize
```

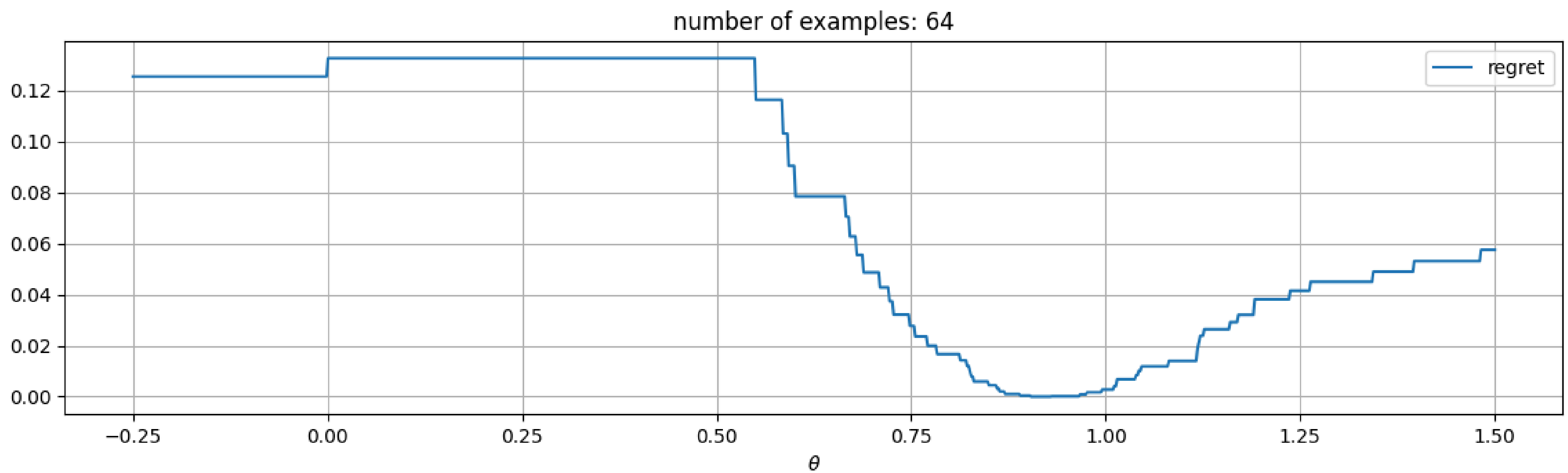


- If $f(x, \theta)$ leads to the correct decision, the regret is 0
- Otherwise we have some non-null value

Knowing Regret

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

In [186]:
```
util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=64, figsi
```



This function breaks havoc with gradient descent, for two main reasons

What's Wrong with Regret (1)

As a first issue, the loss is not inherently differentiable

Given:

$$\text{regret}(y, \hat{y}) = y^T z^*(\hat{y}) - y^T z^*(y) \qquad \text{with: } \hat{y} = h(x; \theta)$$

The derivative chain:

$$\frac{\partial\, \text{regret}}{\partial \theta} = \frac{\partial\, \text{regret}}{\partial z^*} \frac{\partial\, z^*}{\partial \hat{y}} \frac{\partial\, \hat{y}}{\partial \theta}$$

...Contains a term that is based on an $\mathbf{argmin}$ operator

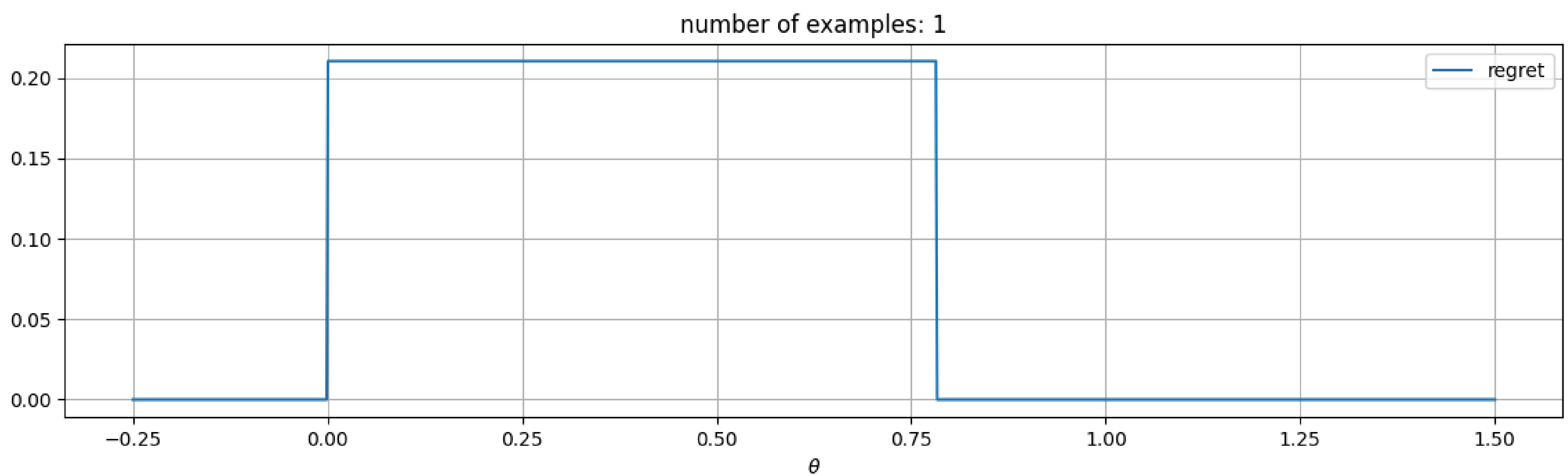
- For this reason, computing the derivative might be tricky
- ...And for some $\hat{y}$ values a derivative might not be defined at all

What's Wrong with Regret (2)

Second, when the derivative exists, it might be useless

E.g. for combinatorial and linear problems, regret will be piecewise constant

In [187]:
```
util.draw_loss_landscape(losses=[util.RegretLoss()], model=1, seed=42, batch_size=1, figsize
```



When the derivative is defined, it's value is 0

Self-Contrastive Loss

These issues have been addressed in multiple ways

Here we'll start with the idea of changing perspective

- In particular, any prediction vector $\hat{y}$ defines a cost function:

$$\hat{y}^T z$$

- ...Which will lead the solver toward the optimal solution:

$$z^*(\hat{y})$$

Self-Contrastive Loss

These issues have been addressed in multiple ways

Here we'll start with the idea of changing perspective

- In particular, any prediction vector $\hat{y}$ defines a cost function:

$$\hat{y}^T z$$

- ...Which will lead the solver toward the optimal solution:

$$z^*(\hat{y})$$

Given an example (x, y), for a good prediction vector $\hat{y}$

- The cost of the true optimal solution $z^*(y)$
- ...Should not be worse than the cost of the "estimated" optimal solution $z^*(\hat{y})$

Self-Contrastive Loss

Hence we can think of using as a surrogate loss the difference:

$$\hat{y}^T z^*(y) - \hat{y}^T z^*(\hat{y})$$

It represents "how wrong" the estimated cost function is w.r.t. the true one

- It contains a naturally differentiable term (i.e. $\hat{y}$)
- It is not constance, even when z^* is piecewise constant

The gradient represents the difference between the optimal solutions:

$$\nabla \left(\hat{y}^T z^*(y) - \hat{y}^T z^*(\hat{y}) \right) = z^*(y) - z^*(\hat{y})$$

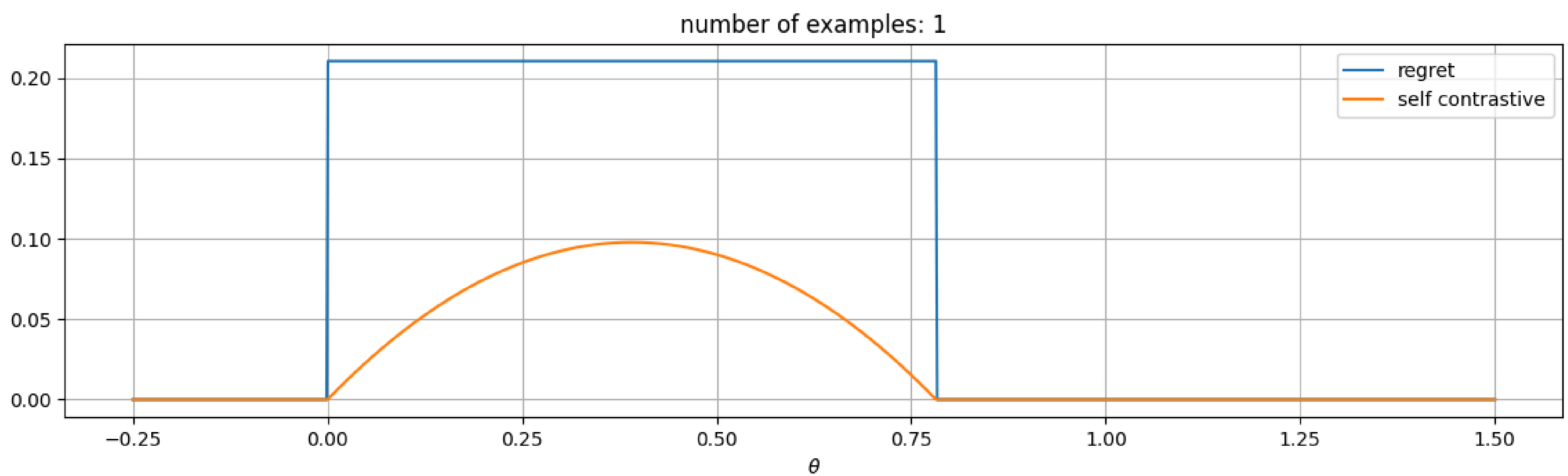
In the DFL literature, this is known as self-contrastive loss

Limitations of the Self-Contrastive Loss

However, the self-constrastive loss has some significant limitations

Here's how it looks for one example in our toy problem:

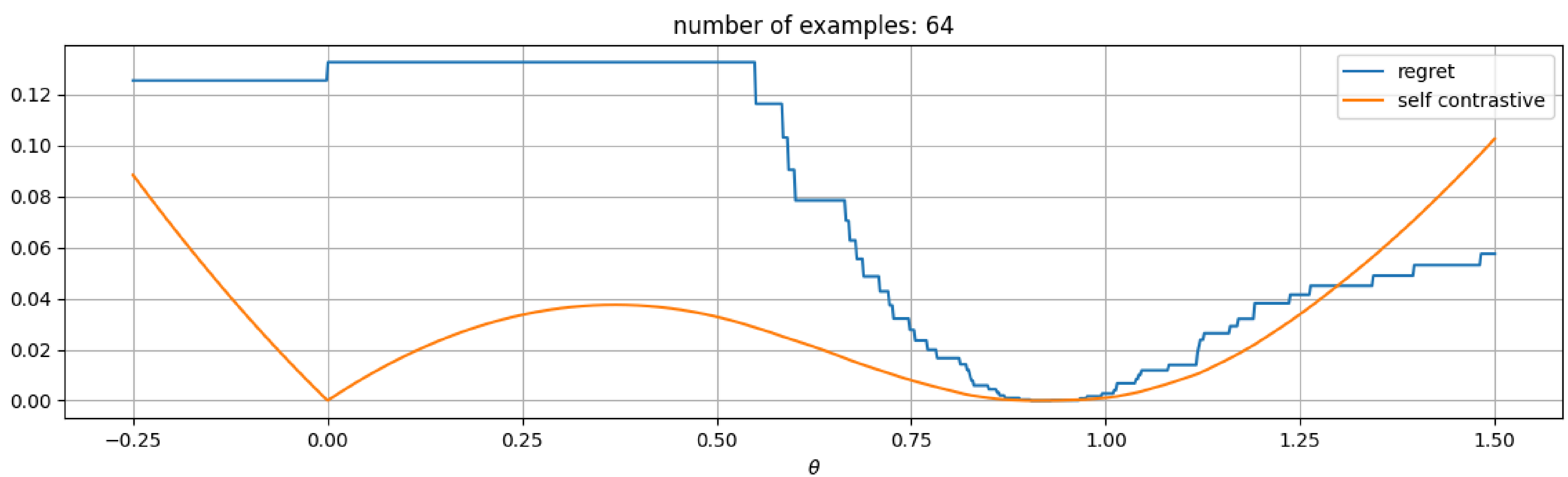
In [188]:
```
util.draw_loss_landscape(losses=[util.RegretLoss(), util.SelfContrastiveLoss()], model=1, se
```



Limitations of the Self-Contrastive Loss

Here's the plot for multiple examples

In [189]:
```
util.draw_loss_landscape(losses=[util.RegretLoss(), util.SelfContrastiveLoss()], model=1, se
```

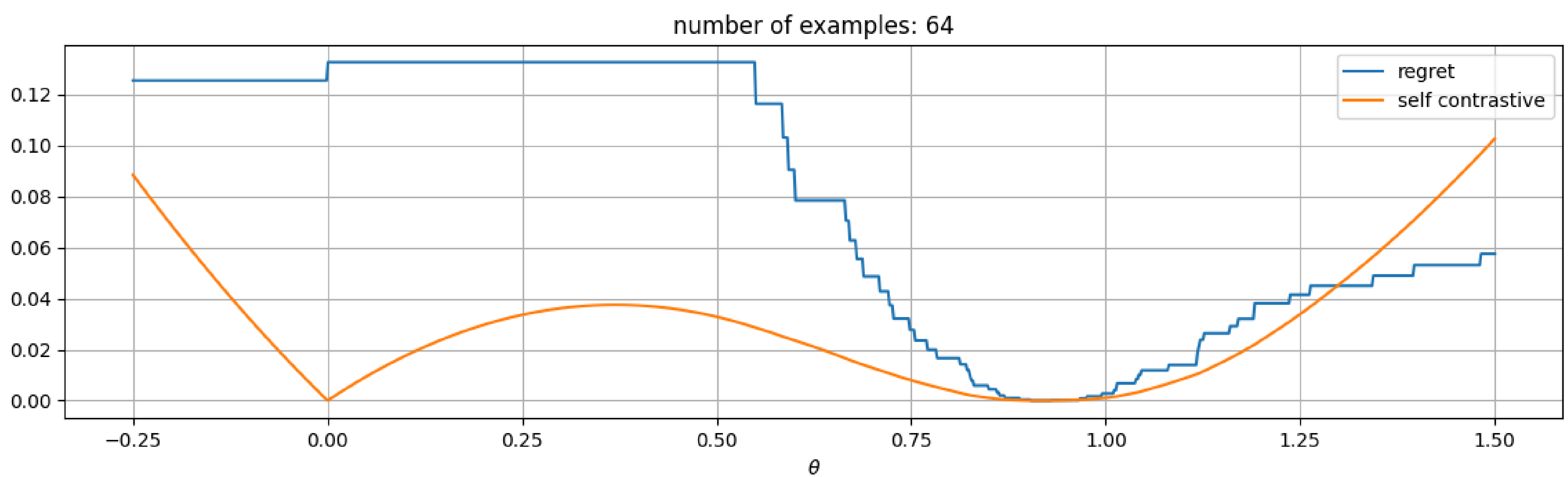


There's a problem here! **Can you see which one?**

Limitations of the Self-Contrastive Loss

There's a spurious minimum!

```
In [190]: util.draw_loss_landscape(losses=[util.RegretLoss(), util.SelfContrastiveLoss()], model=1, se
```



At training time, there's a chance of reaching the wrong minimum!

SPO+ Loss

A well-known DLF approach can be seen as solution for this issue

I.e. the SPO+ loss from [1], which can be defined as:

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

- The structure is the same as the self-constrastive loss
- ...But at training time we compute it w.r.t. a modified prediction vector

At inference time we behave as usual, i.e. we solve:

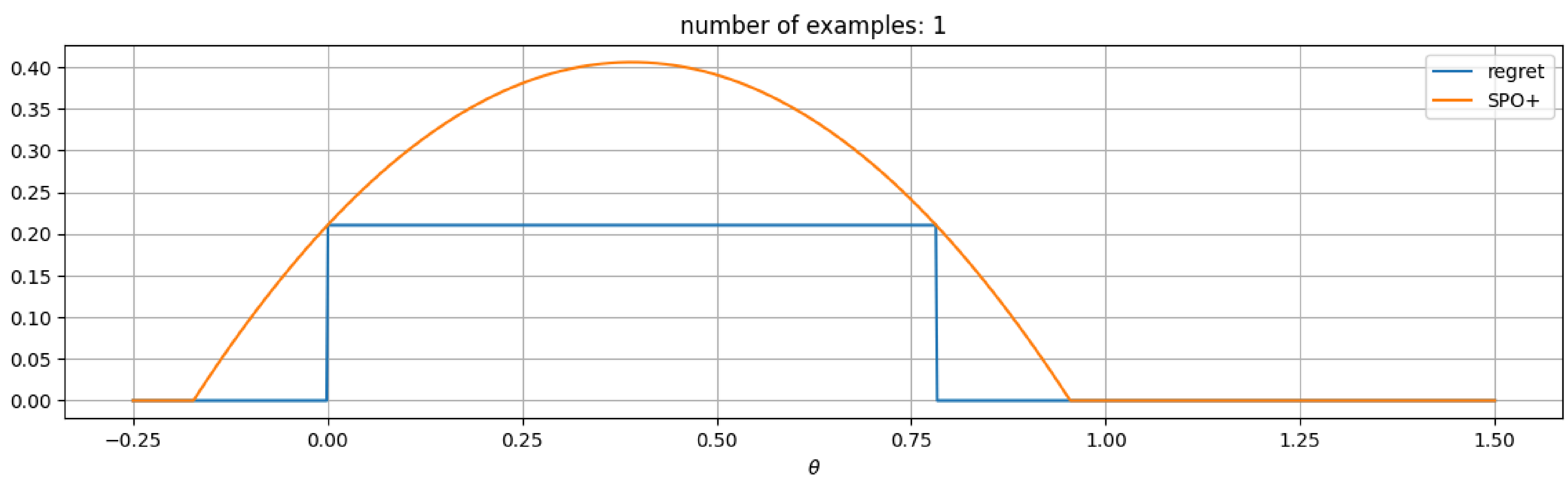
$$z^*(\hat{y}) \quad \text{with: } \hat{y} = h(x; \theta)$$

[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 [191]:
```
util.draw_loss_landscape(losses=[util.RegretLoss(), util.SPOPlusLoss()], model=1, seed=42, l
```

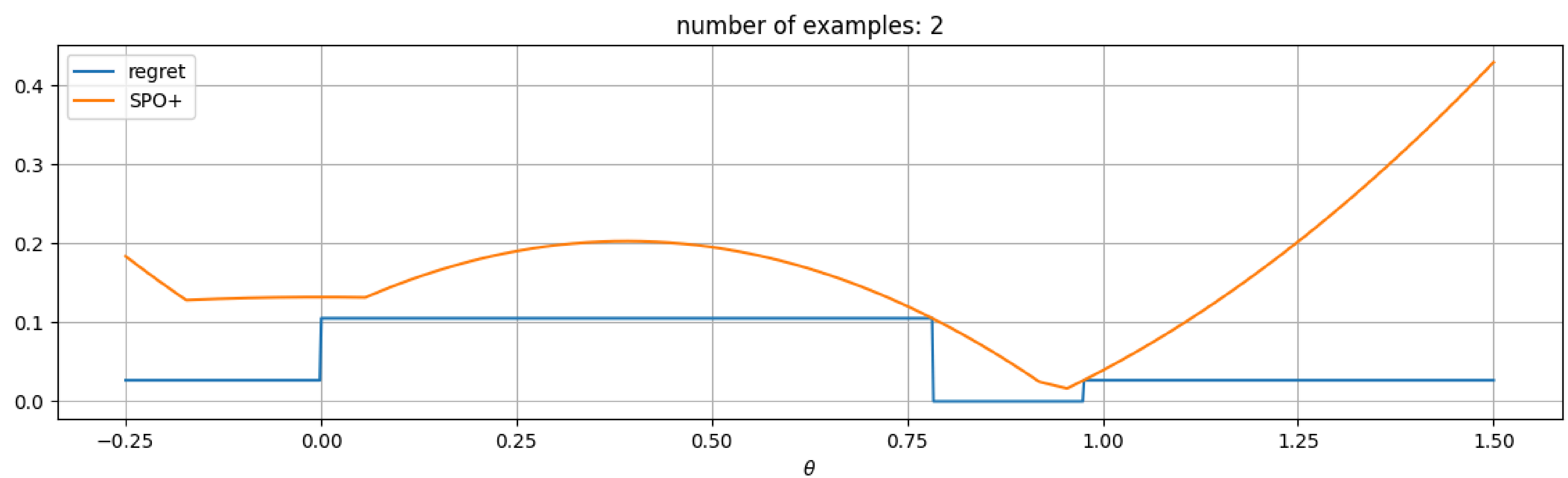


- Like in the self-constrastive case, there are two local minima

SPO+ Loss

This is the SPO+ loss for a two examples

In [192]:
```
util.draw_loss_landscape(losses=[util.RegretLoss(), util.SPOPlusLoss()], model=1, seed=42, 
```

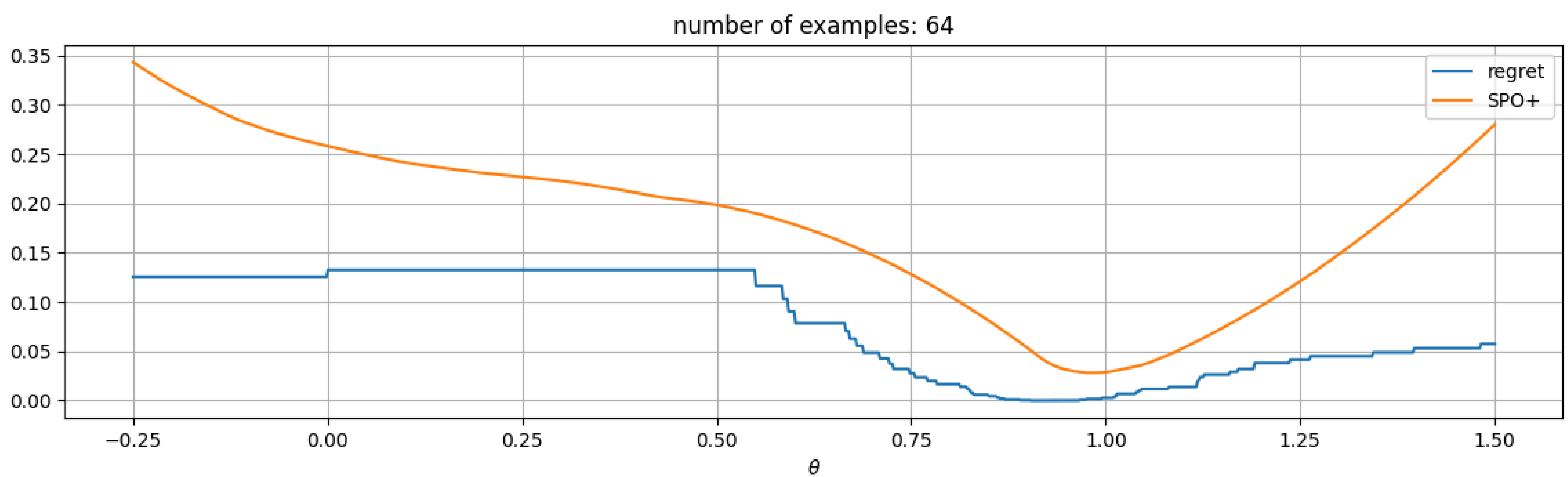


- The "good" local minima for both examples are roughly in the same place
- The "spurious" local minima fall in different position

SPO+ Loss

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

```
In [193]: util.draw_loss_landscape(losses=[util.RegretLoss(), util.SPOPlusLoss()], model=1, seed=42, 
```



- This effect is invaluable when training with gradient descent

Let's see the approach in action on a single problem

A (Sligthly) More Complex Example

We will an optimization problem in this form:

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

- 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

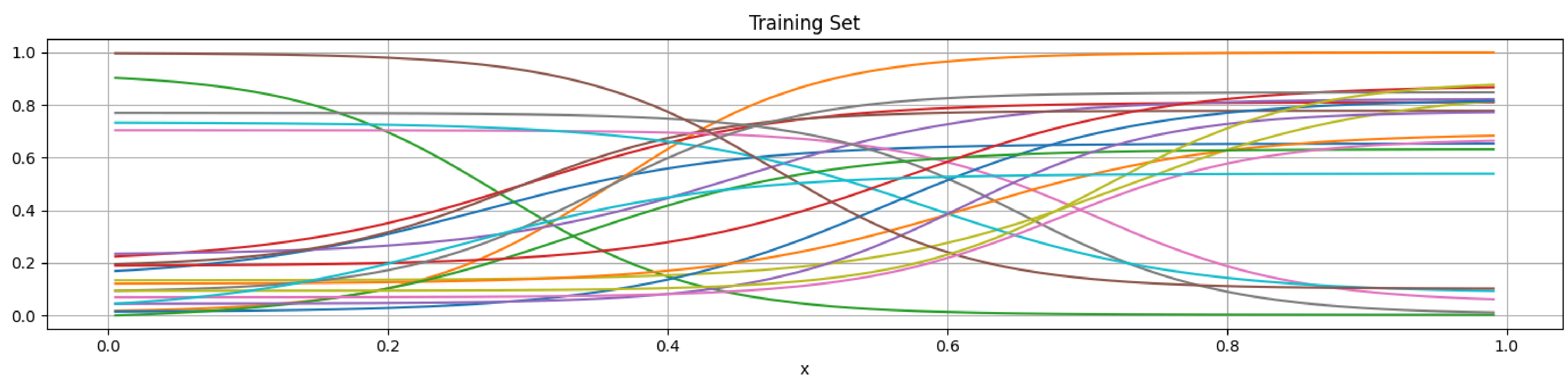
```
In [194]: nitems, rel_req, seed = 20, 0.5, 42
          prb = util.generate_problem(nitems=nitems, rel_req=rel_req, seed=seed)
          display(prb)
```

```
ProductionProblem(values=[1.14981605 1.38028572 1.29279758 1.23946339 1.06240746 1.0623978
1
 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 (Sligthly) More Complex Example

Next, we generate some training (and test) data

```
In [195]: data_tr = util.generate_costs(nsamples=350, nitems=nitems, seed=seed, noise_scale=0, noise_
          data_ts = util.generate_costs(nsamples=150, nitems=nitems, seed=seed, sampling_seed=seed+1,
          util.plot_df_cols(data_tr, figsize=figsize, title='Training Set')
```

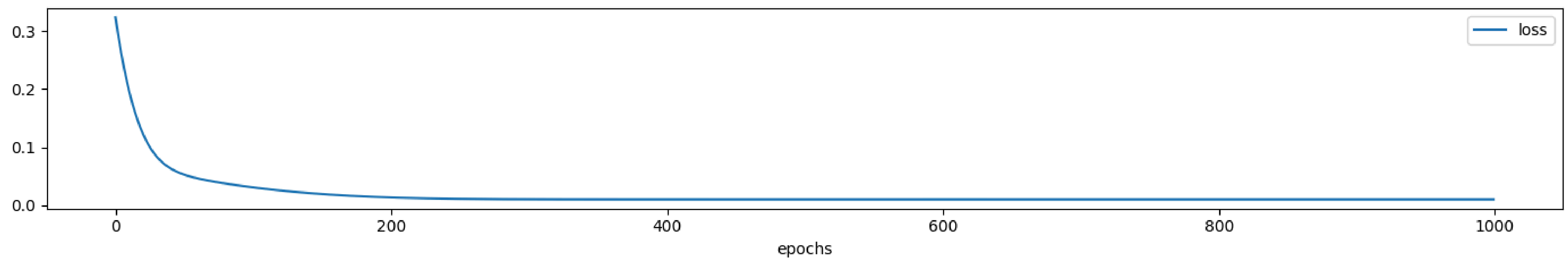


- We assume that costs can be estimated based on an scalar observable x
- The set of least expensive jobs changes considerably with x

Prediction Focused Approach

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

```
In [196]: 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=
          util.plot_training_history(history, figsize=figsize_narrow, print_scores=False, print_time=T
          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')
```



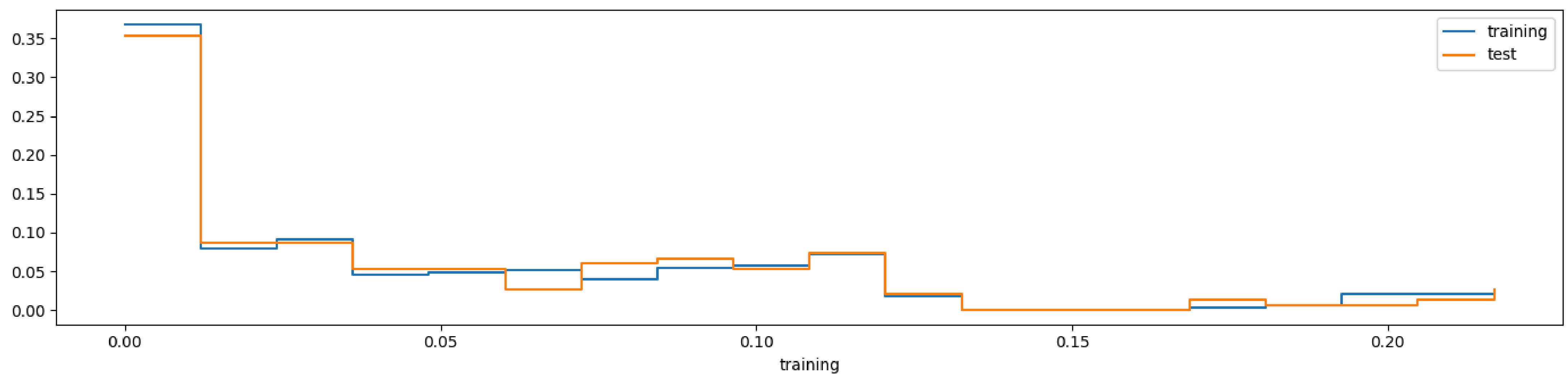

```
Training time: 7.2291 sec
R2: 0.86, MAE: 0.086, RMSE: 0.10 (training)
R2: 0.86, MAE: 0.087, RMSE: 0.10 (test)
```

- 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 [197]:
```
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
```



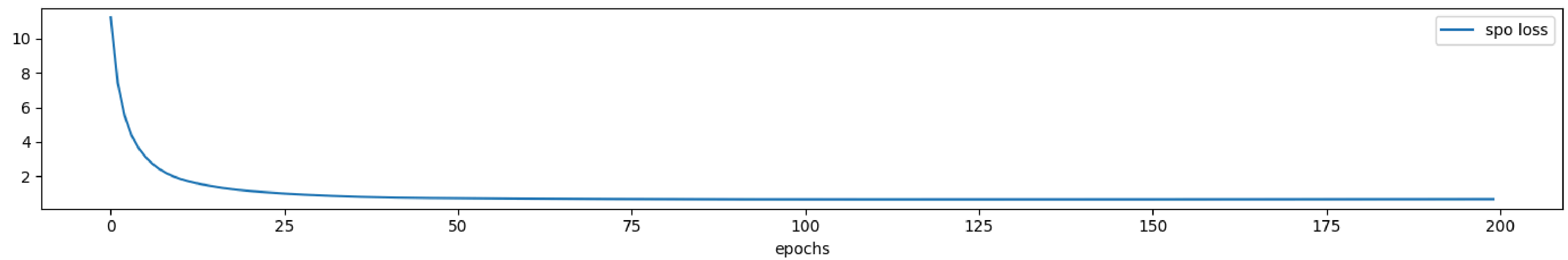

```
Mean: 0.053 (training), 0.053 (test)
```

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

A Decision Focused Learning Approach

Next, we train a DFL approach

```
In [198]: 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, verbo
          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')
```



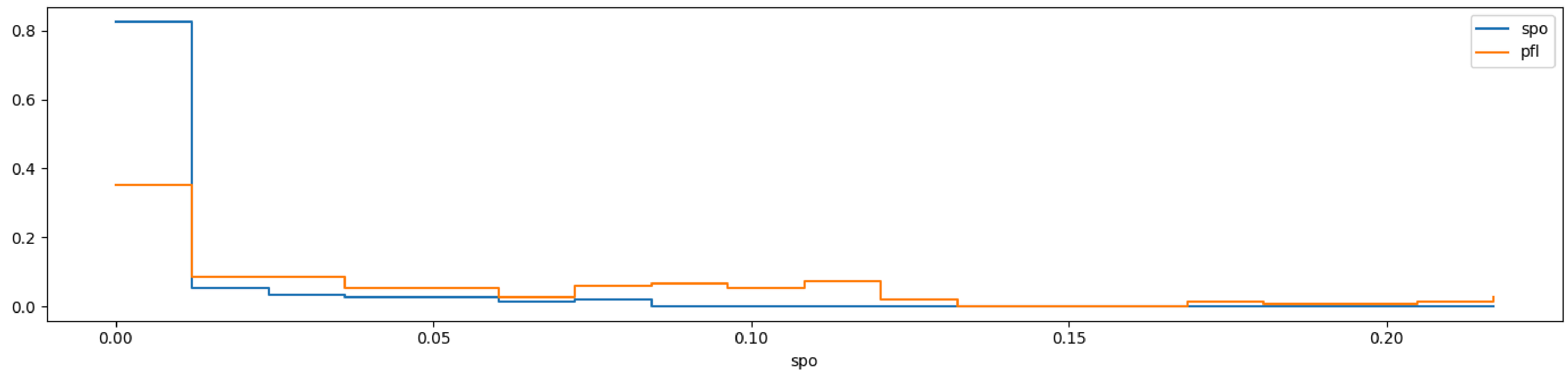

```
Training time: 112.7597 sec
R2: -0.50, MAE: 0.25, RMSE: 0.30 (training)
R2: -0.49, MAE: 0.25, RMSE: 0.30 (test)
```

In terms of accuracy, this is considerably worse

Comparing Regrets

But the regret is so much better!

```
In [199]: 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: 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).