Lagrangian Approaches for Constrained ML

Machine Learning and Hard Constraints

Sometimes, you want a ML model to respect hard constraints

This happens for example:

- When there are physical laws that you know to be true
- In case of safety concerns
- When ethical aspects are involved

In principle, we could still use our $\max$-based Lagrangian approach:

$$\text{argmin}_{\omega} \left\{ L(y) + \lambda^T \max(0, g(y)) \right\} \text{ with: } y = f(\hat{x}; \omega)$$

Where g is the constrained quantity

- Intuitively, for a large enough λ...
- ...It should be possible to reach approximate satisfaction on the training set

A Case Study: Fairness in ML Models

As a case study, say we want to estimate the risk of violent crimes

- This is obviously a very ethically sensitive (and questionable) task
- Our model may easily end up discriminating some social groups

Loading and Preparing the Dataset

We will start by loading the "crime" UCI dataset

We will use a pre-processed version:

In [3]:
```
data = util.load_communities_data(data_folder)
attributes = data.columns[3:-1]
target = data.columns[-1]
data.head()
```

Out[3]:

	communityname	state	fold	pop	race	pct12-21	pct12-29	pct16-24	pct65up	pctUrban	...	pctForeignBorn	pctBornStateRe
1008	EastLampetertownship	PA	5	11999	0	0.1203	0.2544	0.1208	0.1302	0.5776	...	0.0288	0.8132
1271	EastProvidencecity	RI	6	50380	0	0.1171	0.2459	0.1159	0.1660	1.0000	...	0.1474	0.6561
1936	Betheltown	CT	9	17541	0	0.1356	0.2507	0.1138	0.0804	0.8514	...	0.0853	0.4878
1601	Crowleycity	LA	8	13983	0	0.1506	0.2587	0.1234	0.1302	0.0000	...	0.0029	0.9314
293	Pawtucketcity	RI	2	72644	0	0.1230	0.2725	0.1276	0.1464	1.0000	...	0.1771	0.6363

5 rows × 101 columns

The target is "violentPerPop" (number of violent offenders per 100K people)

Loading and Preparing the Dataset

We prepare for normalizing all numeric attributes

- The only categorical input is "race" (0 = primarily "white", 1 = primarily "black")
- Incidentally, "race" is a natural focus to check for discrimination

We define the train-test divide and we identify the numerical inputs

In [4]:
```
tr_frac = 0.8 # 80% data for training
tr_sep = int(len(data) * tr_frac)
nf = [a for a in attributes if a != 'race'] + [target]
```

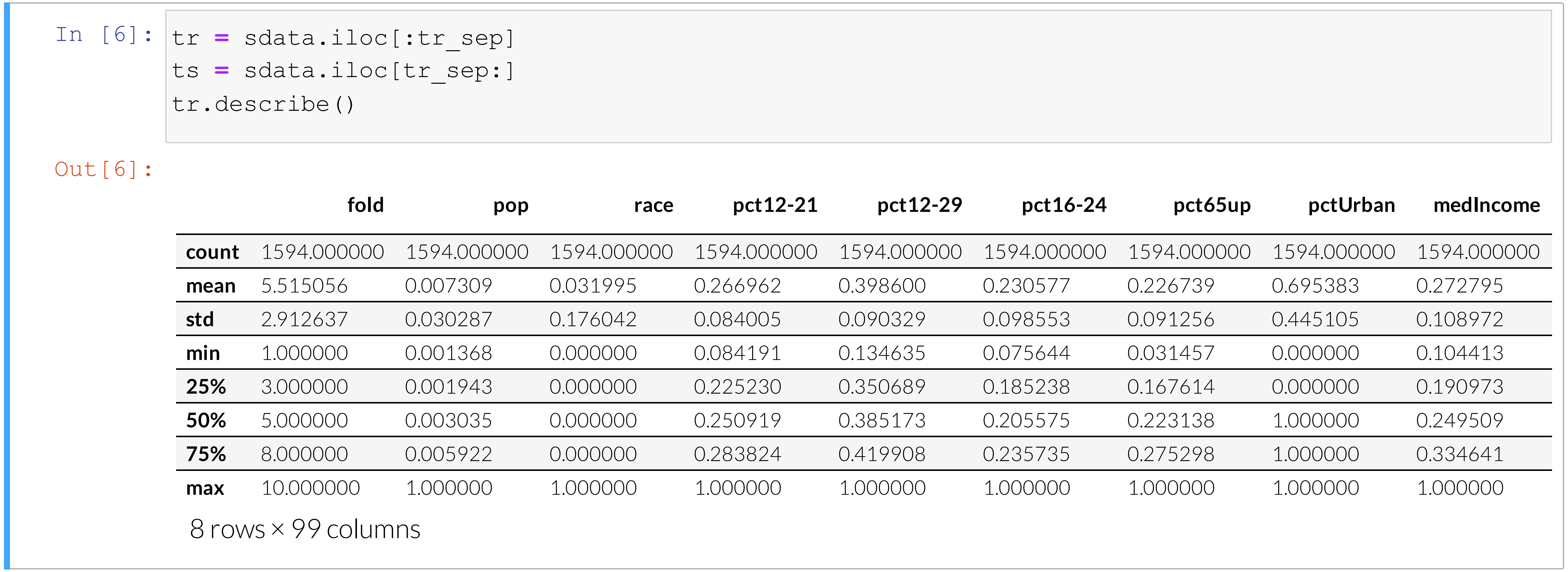
We normalize the data and convert to float32 (to make TensorFlow happier)

In [5]:
```
tmp = data.iloc[:tr_sep]
scale = tmp[nf].max()
sdata = data.copy()
sdata[nf] /= scale[nf]

sdata[attributes] = sdata[attributes].astype(np.float32)
sdata[target] = sdata[target].astype(np.float32)
```

Loading and Preparing the Dataset

Finally we can separate the training and test set



```
In [6]: tr = sdata.iloc[:tr_sep]
        ts = sdata.iloc[tr_sep:]
        tr.describe()
```

Out[6]:

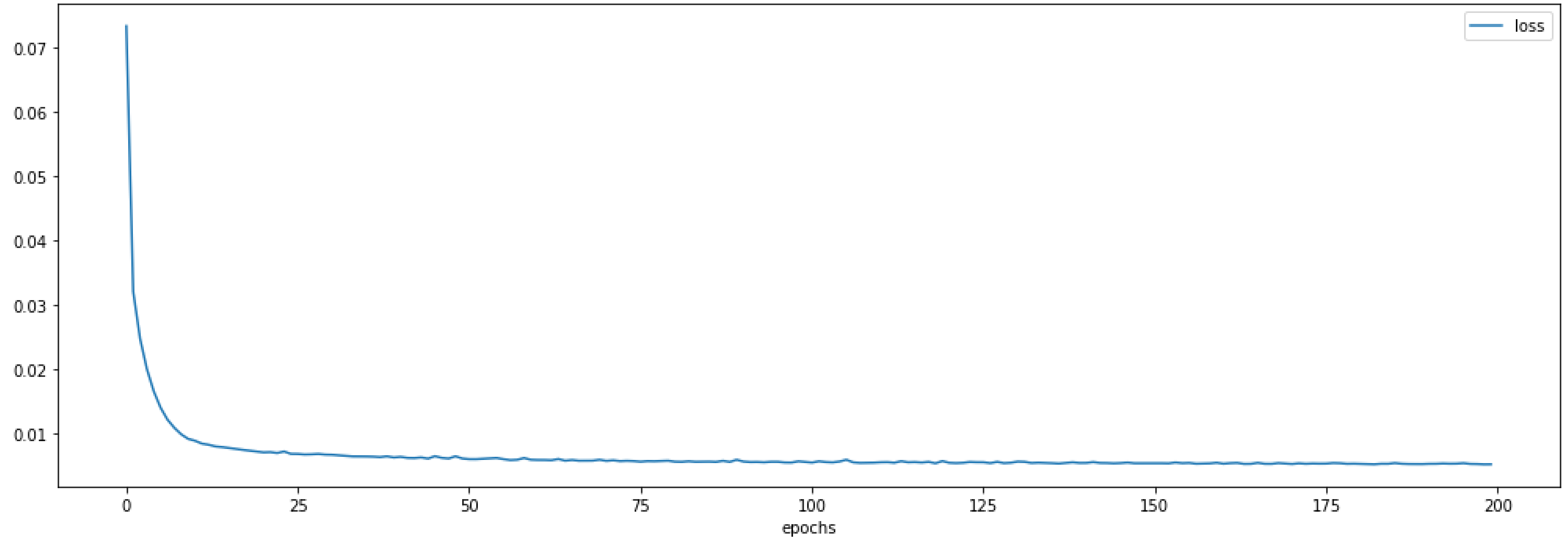
	fold	pop	race	pct12-21	pct12-29	pct16-24	pct65up	pctUrban	medIncome
count	1594.000000	1594.000000	1594.000000	1594.000000	1594.000000	1594.000000	1594.000000	1594.000000	1594.000000
mean	5.515056	0.007309	0.031995	0.266962	0.398600	0.230577	0.226739	0.695383	0.272795
std	2.912637	0.030287	0.176042	0.084005	0.090329	0.098553	0.091256	0.445105	0.108972
min	1.000000	0.001368	0.000000	0.084191	0.134635	0.075644	0.031457	0.000000	0.104413
25%	3.000000	0.001943	0.000000	0.225230	0.350689	0.185238	0.167614	0.000000	0.190973
50%	5.000000	0.003035	0.000000	0.250919	0.385173	0.205575	0.223138	1.000000	0.249509
75%	8.000000	0.005922	0.000000	0.283824	0.419908	0.235735	0.275298	1.000000	0.334641
max	10.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

8 rows × 99 columns

Baseline

Let's establish a baseline by tackling the task via Linear Regression

```
In [8]: nn = util.build_ml_model(input_size=len(attributes), output_size=1, hidden=[])
        history = util.train_ml_model(nn, tr[attributes], tr[target], validation_split=0.,
                                      epochs=200)
        util.plot_training_history(history, figsize=figsize)
```



```
Model loss: 0.0052 (training)
```

Baseline Evaluation

...And let's check the results

```
In [9]: tr_pred = nn.predict(tr[attributes])
        r2_tr = r2_score(tr[target], tr_pred)

        ts_pred = nn.predict(ts[attributes])
        r2_ts = r2_score(ts[target], ts_pred)

        print(f'R2 score: {r2_tr:.2f} (training), {r2_ts:.2f} (test)')
```

```
R2 score: 0.64 (training), 0.59 (test)
```

- They are not super (definitely not PreCrime level), but not alwful either
- Some improvements (not much) can be obtained with a Deeper model

We will keep Linear Regression as a baseline

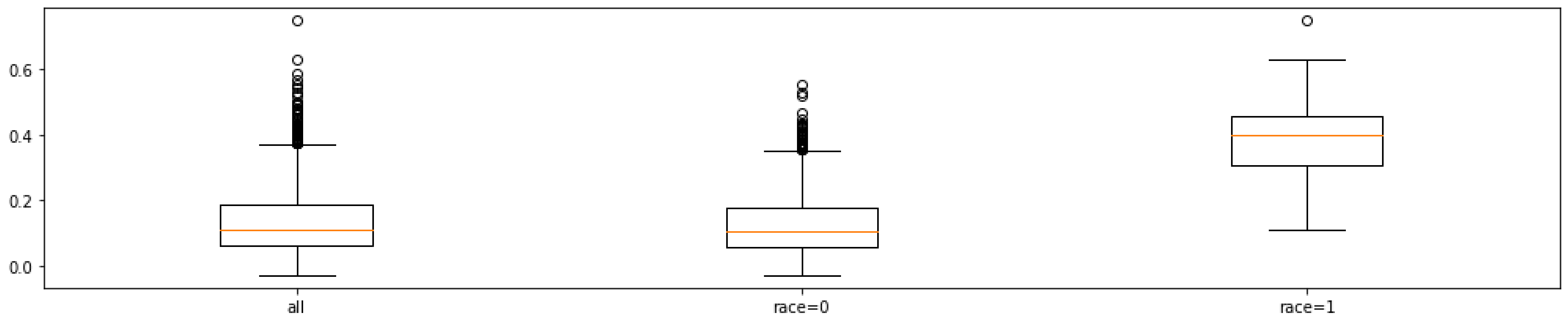
Discrimination Indexes

Discrimination can be linked to disparate treatment

- "race" may not be even among the input attributes
- ...And yet it may be taken into account implicitly (i.e. via correlates)

But we can check whether the model treats differently different groups:

```
In [10]: protected = {'race': (0, 1)}
         util.plot_pred_by_protected(tr, tr_pred, protected, figsize=(figsize[0], 0.6*figsize[1]))
```



Indeed, our model has a significant degree of discrimination

Discrimination Indexes

A number of discrimination indexes attempt to measure discrimination

- Whether ethics itself can be measured is highly debatable!
- ...But even if imperfect, this currently the best we can do

We will use the Disparate Impact Discrimination Index

- Given a set of categorical protected attribute (indexes) J_p
- ...The regression for of the regression form of the index (DIDI_r) is given by:

$$\sum_{j \in J_p} \sum_{v \in D_j} \left| \frac{1}{m} \sum_{i=1}^{m} y_i - \frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i \right|$$

- Where D_j is the domain of attribute j
- ...And $I_{j,v}$ is the set of example such that attribute j has value v

DIDI

Let's make some intuitive sense of the DIDI_r formula

$$\sum_{j \in J_p} \sum_{v \in D_j} \left| \frac{1}{m} \sum_{i=1}^{m} y_i - \frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i \right|$$

- $\frac{1}{m} \sum_{i=1}^{m} y_i$ is just the average predicted value
- The protected attribute defines social groups
- $\frac{1}{|I_{j,v}|} \sum_{i \in I_{j,v}} y_i$ is the average prediction for a social group

We penalize deviations from the global average

- Obviously this is not necessarily the best definition, but it is something
- In general, different tasks will call for different discrimination indexes

...And don't forget the whole "can we actually measure ethics" issue ;-)

DIDI

We can compute the DIDI via the following function

```
def DIDI_r(data, pred, protected):
    res, avg = 0, np.mean(pred)
    for aname, dom in protected.items():
        for val in dom:
            mask = (data[aname] == val)
            res += abs(avg - np.mean(pred[mask]))
    return res
```

- `protected` contains the protected attribute names with their domain

For our original Linear Regression model, we get

```
In [11]: tr_DIDI = util.DIDI_r(tr, tr_pred, protected)
         ts_DIDI = util.DIDI_r(ts, ts_pred, protected)
         print(f'DIDI: {tr_DIDI:.2f} (training), {ts_DIDI:.2f} (test)')
```

```
DIDI: 0.26 (training), 0.27 (test)
```

Fairness Constraints

Discrimination indexes can be used to state fairness constraints

For example, we may require:

$$\text{DIDI}_r(y) \leq \theta$$

If the chosen index is differentiable...

...Then we may try to inject the constraint in a NN via a semantic regularizer

- For example, we may use a loss function in the form:

$$L(y, \hat{y}) + \lambda \max(0, \text{DIDI}_r(y) - \theta)$$

For non-differentiable indexes (e.g. those found in classification), we can:

- Use a differentiable approximation (with some care!)
- Use an approach that does not require differentiability, e.g. this or that

Fairness as a Semantic Regularizer

We can once again use a custom Keras model

```python
class CstDIDIRegressor(keras.Model):
    def __init__(self, base_pred, attributes, protected, alpha, thr): ...

    def train_step(self, data): ...

    @property
    def metrics(self): ...
```

The full code can be found in the support module

- We subclass `keras.Model` and we provide a custom training step
- `alpha` is the regularizer weight
- `thr` is the DIDI threshold

In this case, we do not need a custom batch generator

Fairness as a Semantic Regularizer

The main logic is in the first half of the `train_step` method:

```python
def train_step(self, data):
    x, y_true = data # unpacking the mini-batch
    with tf.GradientTape() as tape:
        y_pred = self.based_pred(x, training=True) # obtain predictions
        mse = self.compiled_loss(y_true, y_pred) # base loss (kept external)
        ymean = k.mean(y_pred) # avg prediction
        didi = 0 # DIDI computation
        for aidx, dom in self.protected.items():
            for val in dom:
                mask = (x[:, aidx] == val)
                didi += k.abs(ymean - k.mean(y_pred[mask]))
        cst = k.maximum(0.0, didi - self.thr) # Regularizer
        loss = mse + self.alpha * cst
```

- The main loss is defined when calling `compile`

Training the Constrained Model

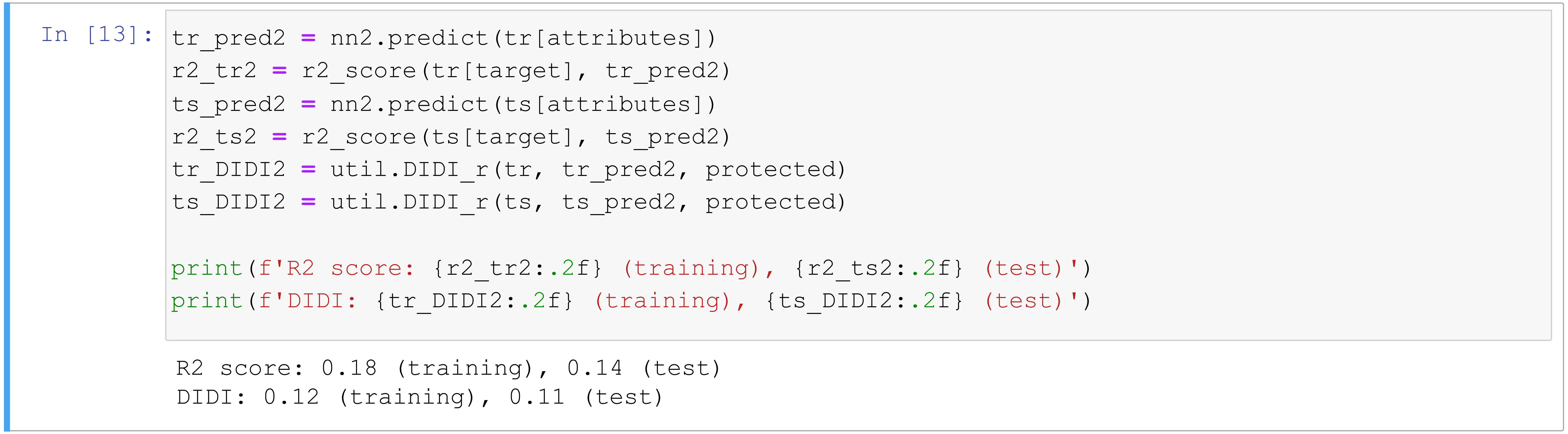
Let's try and train the model, trying to roughly halve the DIDI

- Important: it will be a good idea to need to keep all examples in every batch
- Mini-batches can be used, but make constraint satisfaction (more) stochastic

In [12]:
```
didi_thr = 0.13
base_pred = util.build_ml_model(input_size=len(attributes), output_size=1, hidden=[])
nn2 = util.CstDIDIModel(base_pred, attributes, protected, alpha=5, thr=didi_thr)
history = util.train_ml_model(nn2, tr[attributes], tr[target], validation_split=0., epochs=2000,
util.plot_training_history(history, figsize=figsize)
```

Constrained Model Evaluation

Let's check both the prediction quality and the DIDI



In [13]:
```python
tr_pred2 = nn2.predict(tr[attributes])
r2_tr2 = r2_score(tr[target], tr_pred2)
ts_pred2 = nn2.predict(ts[attributes])
r2_ts2 = r2_score(ts[target], ts_pred2)
tr_DIDI2 = util.DIDI_r(tr, tr_pred2, protected)
ts_DIDI2 = util.DIDI_r(ts, ts_pred2, protected)

print(f'R2 score: {r2_tr2:.2f} (training), {r2_ts2:.2f} (test)')
print(f'DIDI: {tr_DIDI2:.2f} (training), {ts_DIDI2:.2f} (test)')
```

```
R2 score: 0.18 (training), 0.14 (test)
DIDI: 0.12 (training), 0.11 (test)
```

The constraint is satisfied with some slack, leading to reduced performance

- A large λ (what we have here) slows down training
- ...But a small λ may lead to significant constraint violation

Lagrangian Dual Framework

Penalty Method

We can think of increasing λ gradually

...Which leads to the classical penalty method

- $\lambda^{(0)} = 1$
- $\omega^{(0)} = \arg\min_\omega \left\{ L(y) + \lambda^{(0)T} \max(0, g(y)) \right\}$ with: $y = f(\hat{x}; \omega)$
- For $k = 1..n$
  - If $g(y) \leq 0$, stop
  - Otherwise $\lambda^{(k)} = r\lambda^{(k)}$, with $r \in (1, \infty)$
  - $\omega^{(k)} = \arg\min_\omega \left\{ L(y) + \lambda^{(k)T} \max(0, g(y)) \right\}$ with: $y = f(\hat{x}; \omega)$

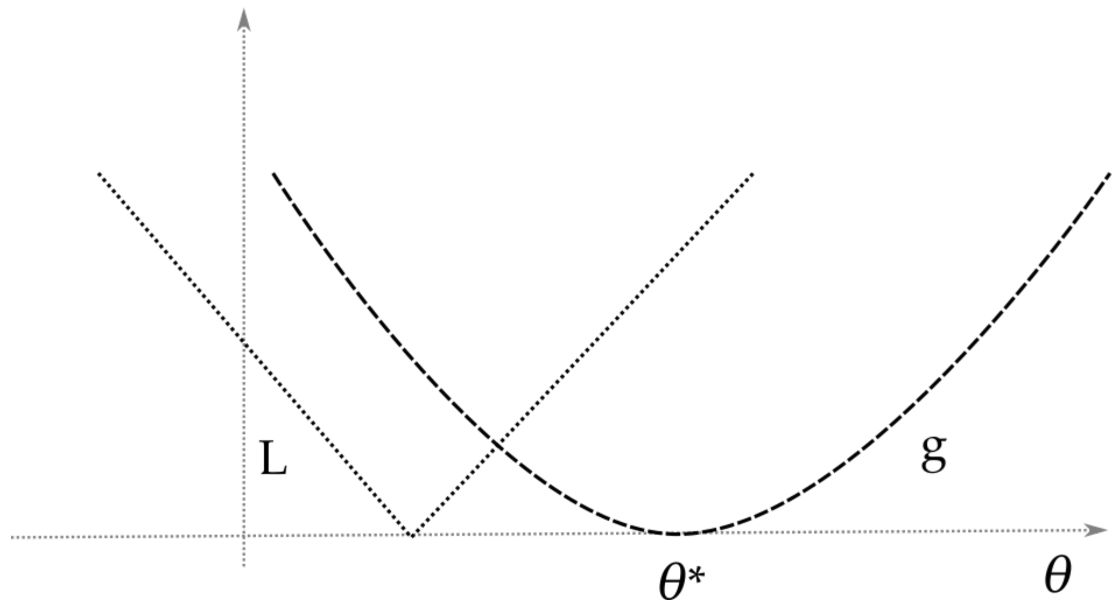
This is a simple, but flexible approach for constrained (numeric) optimization

- It works even with non differentiable constraints
- ...If your training engine can handle them, that is

Drawbacks of the Penalty Method

The penalty method can be quite effective, but it has some drawbacks:

1. You need an optimal solver
   - Without one, you only get approximate results (no guarantees)
2. The $\max$ (or square) is important
   - Without that, penalties turn into rewards for satisfied constraints
3. You may need arbitrarily large weights (and there is no way around this)



Lagrangian Dual Approach

A nice compromise is provided by the Lagrangian Dual approach

We start from the fact that solving:

$$\min_{\omega} \left\{ L(y) + \lambda^T \max(0, g(y)) \right\} \text{ with: } y = f(\hat{x}; \omega)$$

...Always provides a lower bound on the true constrained optimum

- The reason is that on the original feasible space, all penalty terms are 0
- ...And therefore the minimum cannot be worse than the original one

Therefore, it makes sense to pick λ so as to maximize this bound

$$\text{argmax}_{\lambda} \min_{\omega} \left\{ L(y) + \lambda^T \max(0, g(y)) \right\} \text{ with: } y = f(\hat{x}; \omega)$$

Solving this problem given the best possible lower bound

Lagrangian Dual Approach

Now, let's look more carefully at our problem:

$$\text{argmax}_\lambda \min_\omega \mathcal{L}(\lambda, \omega)$$

$$\text{where: } \mathcal{L}(\lambda, \omega) = L(y) + \lambda^T \max(0, g(y))$$

$$\text{with: } y = f(\hat{x}; \omega)$$

This is a bi-level optimization problem

It can be proved that it is concave in λ

- Therefore, it can be solved via sub-gradient descent
- ...Even for non-differentiable L, g, and f

By doing so, we increase λ only when and where it is needed

- It is a strong mitigation for the issues of the penalty method

Lagrangian Dual Approach

If additionally $\mathcal{L}(\lambda, \omega)$ is differentiable in ω

...Then we can solve the problem via alternate gradient descent/ascent:

- $\lambda^{(0)} = 0$
- $\omega^{(0)} = \arg\min_\theta \mathcal{L}(\lambda^{(0)}, \omega)$
- For $k = 1..n$ (or until convergence):
  - Obtain $\lambda^{(k)}$ via an ascent step with sub-gradient $\nabla_\lambda \mathcal{L}(\lambda, \omega^{(k-1)})$
  - Obtain $\omega^{(k)}$ via a descent step with sub-gradient $\nabla_\omega \mathcal{L}(\lambda^{(k)}, \omega)$

The approach is easy to implement in tensorflow/PyTorch

We just need to use two optimization steps

- It works since small changes to λ
- ...Usually require small changes to ω
- Hence, we can maintain the two vectors approximately optimal

Implementing the Lagrangian Dual Approach

We will implement the Lagrangian dual approach via another custom model

```
class LagDualDIDIRegressor(MLPRegressor):
    def __init__(self, base_pred, attributes, protected, thr):
        super(LagDualDIDIRegressor, self).__init__()
        self.alpha = tf.Variable(0., name='alpha')
        ...

    def __custom_loss(self, x, y_true, sign=1): ...

    def train_step(self, data): ...

    def metrics(self): ...
```

- We no longer pass a fixed `alpha` weight/multiplier
- Instead we use a trainable variable

Implementing the Lagrangian Dual Approach

In the `__custom_loss` method we compute the Lagrangian/regularized loss

```python
def __custom_loss(self, x, y_true, sign=1):
    y_pred = self.base_pred(x, training=True) # obtain the predictions
    mse = self.compiled_loss(y_true, y_pred) # main loss
    ymean = tf.math.reduce_mean(y_pred) # average prediction
    didi = 0 # DIDI computation
    for aidx, dom in self.protected.items():
        for val in dom:
            mask = (x[:, aidx] == val)
            didi += tf.math.abs(ymean - tf.math.reduce_mean(y_pred[mask]))
    cst = tf.math.maximum(0.0, didi - self.thr) # regularizer
    loss = mse + self.alpha * cst
    return sign*loss, mse, cst
```

- The code is the same as before
- ...Except that we can flip the loss sign via a function argument (i.e. `sign`)

Implementing the Lagrangian Dual Approach

In the training method, we make two distinct gradient steps:

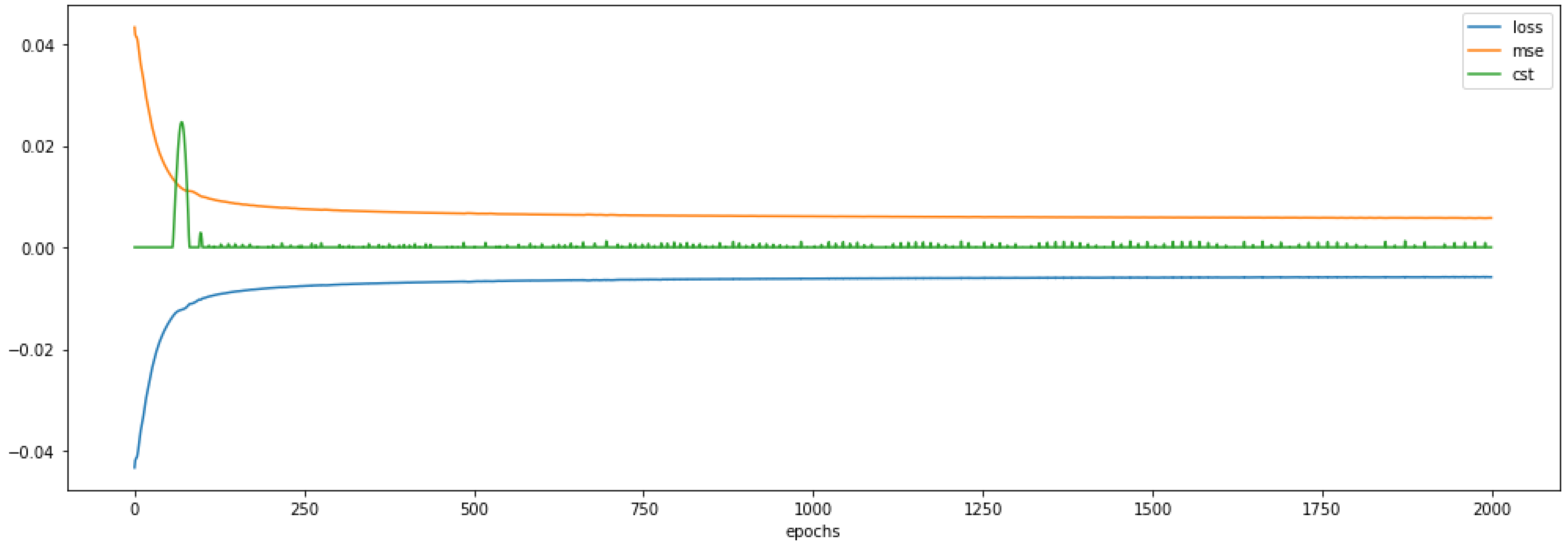
```python
def train_step(self, data):
    x, y_true = data # unpacking
    with tf.GradientTape() as tape: # first loss (minimization)
        loss, mse, cst = self.__custom_loss(x, y_true, sign=1)
    tr_vars = self.trainable_variables
    wgt_vars = tr_vars[:-1] # network weights
    mul_vars = tr_vars[-1:] # multiplier
    grads = tape.gradient(loss, wgt_vars) # adjust the network weights
    self.optimizer.apply_gradients(zip(grads, wgt_vars))
    with tf.GradientTape() as tape: # second loss (maximization)
        loss, mse, cst = self.__custom_loss(x, y_true, sign=-1)
    grads = tape.gradient(loss, mul_vars) # adjust lambda
    self.optimizer.apply_gradients(zip(grads, mul_vars))
```

- In principle, we could even have used two distinct optimizers
- That would allow to keep (e.g.) separate momentum vectors

Training the Lagrangian Dual Approach

The new approach leads less oscillations at training time

```
In [14]: base_pred = util.build_ml_model(input_size=len(attributes), output_size=1, hidden=[])
         nn3 = util.LagDualDIDIModel(base_pred, attributes, protected, thr=didi_thr)
         history = util.train_ml_model(nn3, tr[attributes], tr[target], validation_split=0.,
                                       epochs=2000, batch_size=len(tr))
         util.plot_training_history(history, figsize=figsize)
```



```
Model loss: -0.0058 (training)
```

Lagrangian Dual Evaluation

Let's check the new results

In [15]:
```python
tr_pred3 = nn3.predict(tr[attributes])
r2_tr3 = r2_score(tr[target], tr_pred3)
ts_pred3 = nn3.predict(ts[attributes])
r2_ts3 = r2_score(ts[target], ts_pred3)
tr_DIDI3 = util.DIDI_r(tr, tr_pred3, protected)
ts_DIDI3 = util.DIDI_r(ts, ts_pred3, protected)


print(f'R2 score: {r2_tr3:.2f} (training), {r2_ts3:.2f} (test)')
print(f'DIDI: {tr_DIDI3:.2f} (training), {ts_DIDI3:.2f} (test)')
```

```
R2 score: 0.62 (training), 0.55 (test)
DIDI: 0.12 (training), 0.14 (test)
```

- The DIDI has the desired value (on the test set, this is only roughly true)
- ...And the prediction quality is much higher than before!