Lagrangian Approaches for Constrained ML

There's Lagrangian and Lagrangian

Constrained Machine Learning

We have some constraints, in the general form:

$$g[\hat{Y}] \leq 0 \qquad \text{with: } \hat{Y} = f(X;\theta),\, x \sim P(X)$$

- The model input X is represented as a random variable with distribution $P(X)$
- The output $\hat{Y}$ is a deterministic function of the input X
- $g[\cdot]$ represents a (possibly vector) function defined over the distribution of Y

Constrained Machine Learning

We have some constraints, in the general form:

$$g[\hat{Y}] \leq 0 \qquad \text{with: } \hat{Y} = f(X;\theta),\, x \sim P(X)$$

- The model input X is represented as a random variable with distribution $P(X)$
- The output $\hat{Y}$ is a deterministic function of the input X
- $g[\cdot]$ represents a (possibly vector) function defined over the distribution of Y

The formulation above is very general, but also challenging to deal with

For this reason, we often take a Monte Carlo approximation:

$$g(\hat{y}) \leq 0 \qquad \text{with: } \hat{y} = f(x;\theta) \text{ for a sample: } \{x_i\}_{i=1}^{m}$$

- Under this assumption, g becomes a regular function

Constrained Training Process

Our goal is training a model that satisfies the constraints

We can frame the problem as:

$$\underset{\theta}{\operatorname{argmin}}\{\, L(\hat{y}) \mid \hat{y} = f(x;\theta), g(\hat{y}) \leq 0\}$$

- As a main trick, we use the training data to for the loss function
- ...But also to approximate the constraint function

Constrained Training Process

Our goal is training a model that satisfies the constraints

We can frame the problem as:

$$\underset{\theta}{\operatorname{argmin}}\{ L(\hat{y}) \mid \hat{y} = f(x;\theta), g(\hat{y}) \leq 0\}$$

- As a main trick, we use the training data to for the loss function
- ...But also to approximate the constraint function

The basic ingredients are the same as in knowledge injection

- However, in knowledge injection the constraints are meant to improve accuracy
- Here, feasibility is more important than being accurate

During the evaluation, feasibility and accuracy should be checked separately

Constrained Training Process

Our goal is training a model that satisfies the constraints

We can frame the problem as:

$$\underset{\theta}{\operatorname{argmin}}\{ L(\hat{y}) \mid \hat{y} = f(x;\theta), g(\hat{y}) \leq 0\}$$

- As a main trick, we use the training data to for the loss function
- ...But also to approximate the constraint function

The constraints are enforced during the training process

- This is not the only possible constrained ML formulation!
- ...But it is one of the more popular ones

Sometimes, it is said that the constriants are "distilled in the model"

Constrained Training Process

Our goal is training a model that satisfies the constraints

We can frame the problem as:

$$\underset{\theta}{\operatorname{argmin}}\{ L(\hat{y}) \mid \hat{y} = f(x;\theta), g(\hat{y}) \leq 0\}$$

- As a main trick, we use the training data to for the loss function
- ...But also to approximate the constraint function

Handling relational constraints in this formulation is possible

- This is the case since we can access all training data
- There are however some limitations and/or technical difficulties

We'll encouter some of these later in the lecture

Constrained Training Process

Our goal is training a model that satisfies the constraints

We can frame the problem as:

$$\underset{\theta}{\operatorname{argmin}}\{ L(\hat{y}) \mid \hat{y} = f(x;\theta), g(\hat{y}) \leq 0\}$$

- As a main trick, we use the training data to for the loss function
- ...But also to approximate the constraint function

The constraints are enforced on the training data

- Hence, we can get guarantees at best on the training data
- The behavior on unseen examples depends on how the model generalizes

So, be on the watch out for overfitting!

Constrained Training Process

Our goal is training a model that satisfies the constraints

We can frame the problem as:

$$\underset{\theta}{\operatorname{argmin}}\{ L(\hat{y}) \mid \hat{y} = f(x;\theta), g(\hat{y}) \leq 0\}$$

- As a main trick, we use the training data to for the loss function
- ...But also to approximate the constraint function

Constraint satisfaction usually comes with decreased accuracy

- One model might have better accuracy, but worse constraint satisfaction
- This is especially true on unseen examples (test data)

There are some (lucky) exceptions where feasibility and accuracy are correlated

Lagrangian Approaches for Constrained ML

Lagrangian approaches can be used to tackle constraint ML

...And in fact, they are a very popular approach

- This is partly due to the fact that they can work with Neural Networks
- For this reason, they are often associated with differentiability
- ...Even if they do not strictly require it

Lagrangian Approaches for Constrained ML

Lagrangian approaches can be used to tackle constraint ML

...And in fact, they are a very popular approach

- This is partly due to the fact that they can work with Neural Networks
- For this reason, they are often associated with differentiability
- ...Even if they do not strictly require it

The formulation is similar to the one we used for knowledge injection

We start by building a Lagrangian-like loss function:

$$\mathcal{L}(\hat{y}, \lambda) = L(\hat{y}) + \lambda^T h(\max(0, g(\hat{y})))$$

- Where $\max(0, g(\hat{y}))$ is the level of constraint violation
- h is a penalizer, which might not be always necessary
- $\lambda \geq 0$ is vector of weights

Lagrangian Approaches for Constrained ML

Lagrangian approaches can be used to tackle constraint ML

...And in fact, they are a very popular approach

- This is partly due to the fact that they can work with Neural Networks
- For this reason, they are often associated with differentiability
- ...Even if they do not strictly require it

Lagrangian Approaches for Constrained ML

Lagrangian approaches can be used to tackle constraint ML

...And in fact, they are a very popular approach

- This is partly due to the fact that they can work with Neural Networks
- For this reason, they are often associated with differentiability
- ...Even if they do not strictly require it

The formulation is similar to the one we used for knowledge injection

We start by building a Lagrangian-like loss function:

$$\mathcal{L}(\hat{y}, \lambda) = L(\hat{y}) + \lambda^T h(\max(0, g(\hat{y})))$$

- Where $\max(0, g(\hat{y}))$ is the level of constraint violation
- h is a (monotone) penalizer function
- $\lambda \geq 0$ is vector of weights

Lagrangian Lower Bound

Under robust assumptions, these approaches have a nice property

For any $\lambda \geq 0$, we have that:

$$\min_{\hat{y}} L(\hat{y}) + \lambda^T h(\max(0, g(\hat{y}))) \leq \min_{\hat{y}} \{ L(\hat{y}) \mid g(\hat{y}) \leq 0 \}$$

- The minimum of the Lagrangian function
- ...Is always a lower bound for the loss of the constrained problem

This is due to the use of clipping (the $\max$ operator)

- In the feasible region, we always have the original loss
- ...But by entering the infeasible region, we might get even better loss values

(Almost) the only requirement for this property is that L has a finite lower bound

Choosing the Multipliers

Our prior observation suggests a simple rule for choosing λ

Namely, we could just make all multipliers very large

- The intution is that any $\lambda \geq 0$ will lead to a lower bound
- ...And that increasing a λ_k component can only increase the bound

Therefore, we should get the best bound by just raising λ

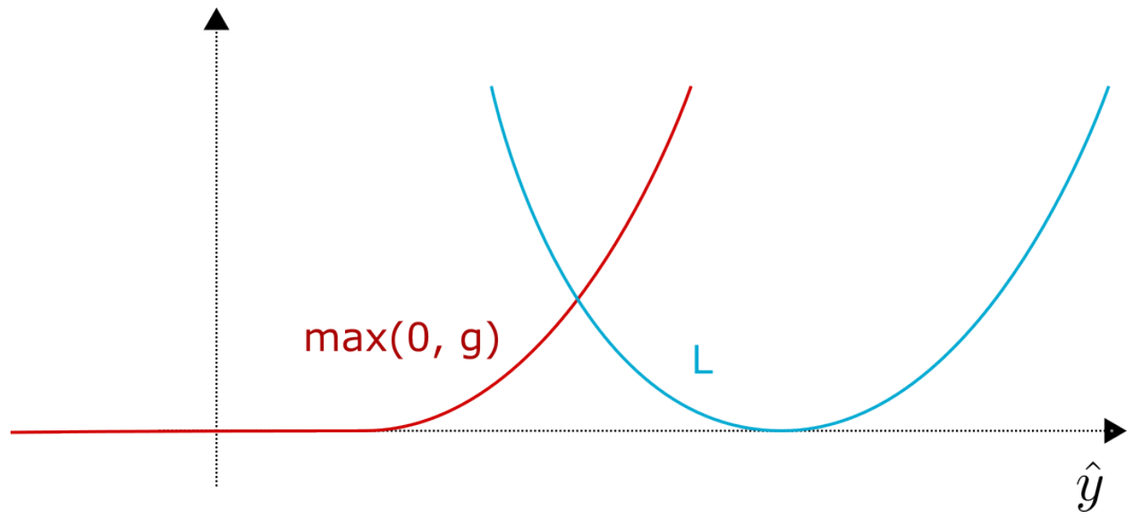
In practice, there's a couple of issue with this approach

- First, in some cases, feasibility might require $\lambda_k \rightarrow \infty, \forall k$
- Second, using large λ_k can lead to numerical instability

While the second issue is easy to understand, the first deserves a better discussion

Battle of the Gradients

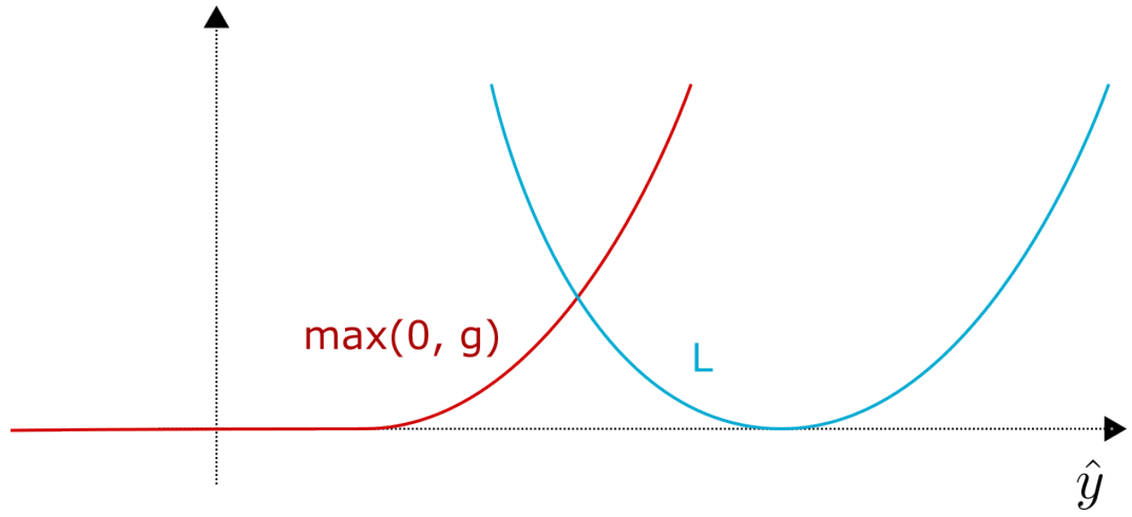
Consider the following situation



- There's a single prediction to make (i.e $\hat{y}$ is scalar)
- The loss function L is quadratic (e.g. Mean Squared Error)
- The constraint violation function $\max(0, g)$ is quadratic in the infeasible region
- By increasing λ, we make the constraint more relevant in the Lagrangian loss

Battle of the Gradients

Consider the following situation



- However, g has a vanishing gradient when the constraint is tight
- Hence, for a given λ increase, the closer we get to satisfaction
- ...The smaller will be the reduction of constraint violation
- Reaching feasibility will require $\lambda \rightarrow \infty$

We can prevent this situation by relying on two techniques

Technique 1: Avoid Vanishing Gradients

First, we should avoid penalizers with vanishing gradients

Consider the general form of a penalizer:

$$h(\max(0, g(\hat{y}))$$

Start by checking ∇g when $g(\hat{y}) = 0$:

- If the gradient is non-null, you are fine and h is unneeded
- If the gradient is null, you need to get creative
- ...For example, you could consider adding a linear "regularization" term
- In general, viable solutions will depend on the specific constraint

Technique 1: Avoid Vanishing Gradients

First, we should avoid penalizers with vanishing gradients

Consider the general form of a penalizer:

$$h(\max(0, g(\hat{y}))$$

Start by checking ∇g when $g(\hat{y}) = 0$:

- If the gradient is non-null, you are fine and h is unneeded
- If the gradient is null, you need to get creative
- ...For example, you could consider adding a linear "regularization" term
- In general, viable solutions will depend on the specific constraint

Also, never used a square penalizer ($\max(0, g(\hat{y})^2$)!

- In truth, sometimes it makes sense (e.g. in Augmented Lagrangian formulations)
- ...But you must have a good idea of what you are doing for that to work

Technique 2: Dual Ascent

Since every $\lambda \geq 0$ provides a valid lower bound

...We build for a constrained training problem a bi-level formulation:

$$\underset{\lambda \geq 0}{\operatorname{argmax}} \underset{\theta}{\operatorname{argmin}} \{ L(\hat{y}) + \lambda^T h(\max(0, g(\hat{y}))) \mid \hat{y} = f(x; \theta) \}$$

- The inner **argmin** operator minimizes the Lagrangian loss
- The outer **argmax** operator seeks for the largest lower bound

Technique 2: Dual Ascent

Since every $\lambda \geq 0$ provides a valid lower bound

...We build for a constrained training problem a bi-level formulation:

$$\underset{\lambda \geq 0}{\operatorname{argmax}} \underset{\theta}{\operatorname{argmin}} \{ L(\hat{y}) + \lambda^T h(\max(0, g(\hat{y}))) \mid \hat{y} = f(x; \theta) \}$$

- The inner **argmin** operator minimizes the Lagrangian loss
- The outer **argmax** operator seeks for the largest lower bound

Solving the formulation requires joint optimization of θ and λ

- Otherwise, given a θ with non-zero violation
- The Lagrangian loss can be made to diverge by arbitrarily increasing λ

Technique 2: Dual Ascent

Since every $\lambda \geq 0$ provides a valid lower bound

...We build for a constrained training problem a bi-level formulation:

$$\underset{\lambda \geq 0}{\operatorname{argmax}} \underset{\theta}{\operatorname{argmin}} \{ L(\hat{y}) + \lambda^T h(\max(0, g(\hat{y}))) \mid \hat{y} = f(x; \theta) \}$$

- The inner **argmin** operator minimizes the Lagrangian loss
- The outer **argmax** operator seeks for the largest lower bound

Solving the formulation requires joint optimization of θ and λ

- Otherwise, given a θ with non-zero violation
- The Lagrangian loss can be made to diverge by arbitrarily increasing λ

However, joint optimization of θ and λ may not be so hard

Technique 2: Dual Ascent

A viable approach is given by the dual ascent method

Dual ascent is a gradient based-algorithm that interleaves two steps

- In the primal step, we make a gradient descent update to θ

$$\theta^{t+1} = \theta^{(t)} - \eta_{t,\theta} \nabla_\theta \mathcal{L}(f(x;\theta), \lambda)$$

- In the dual step, we make a gradient ascent update to λ

$$\lambda^{t+1} = \lambda^{(t)} + \eta_{t,\lambda} \nabla_\lambda \mathcal{L}(f(x;\theta), \lambda)$$

It's useful to see what form the ∇_λ term gets:

$$\nabla_\lambda \mathcal{L}(\hat{y}, \lambda) = \max(0, h(g(\hat{y})))$$

- The gradient over λ is the vector of penalizer values

Technique 2: Dual Ascent

Hence, here's an intuition of dual ascent in Lagrangian constrained ML

In the primal step:

- We focus on minimizing accuracy
- The penalizers attempt to prevent the model from becoming infeasible

In the duald step:

- In case a constraint is violated
- ...We increase the corresponding Lagrangian multiplier

The multipliers are never decreased

- Therefore, if initialized from a feasible value (e.g. $\lambda = 0$)
- ...Then they will stay feasible (i.e. $\lambda \geq 0$) throughout the process

An early use of this approach in constrained ML comes from this paper

Equality Constraints

Equality constraints (i.e. $g(\hat{y}) = 0$) deserve special attention

In theory, they can be associated to a simplified term in the form:

$$\lambda^T |g(\hat{y})| \qquad \text{with: } \lambda \geq 0$$

- The terms is obtained by a reduction from a double-inequality formulation

Equality Constraints

Equality constraints (i.e. $g(\hat{y}) = 0$) deserve special attention

In theory, they can be associated to a simplified term in the form:

$$\lambda^T |g(\hat{y})| \qquad \text{with: } \lambda \geq 0$$

- The terms is obtained by a reduction from a double-inequality formulation

In practice, it is not a good idea

- Equality constraint satisfaction in a training setting is very hard to achieve
- ...Meaning that the constraint will always be a bit violated
- ...And dual ascent will constantly increase the multipliers

In the worst case, this can lead to multiplier divergence

Equality Constraints

Equality constraints (i.e. $g(\hat{y}) = 0$) deserve special attention

In theory, they can be associated to a simplified term in the form:

$$\lambda^T |g(\hat{y})| \qquad \text{with: } \lambda \geq 0$$

- The terms is obtained by a reduction from a double-inequality formulation

In practice, it is not a good idea

- Equality constraint satisfaction in a training setting is very hard to achieve
- ...Meaning that the constraint will always be a bit violated
- ...And dual ascent will constantly increase the multipliers

In the worst case, this can lead to multiplier divergence

There are (at least) two ways to deal with this issue

Equality Constraints

First, we can acknowledge that some violation is inevitable

...Which allows turning the original constraint into:

$$|g(\hat{y})| \leq \varepsilon$$

From which we can derive a penalizer in the form:

$$\lambda^T \max(0, |g(\hat{y})| - \varepsilon)$$

For a sufficiently large ε the penalizer will be well-behaved

- In theory, we still have the problem of finding a good value for $\boldsymbol{\varepsilon}$
- In practice, we can usually find one based on the semantic of our constraint
- ...And/or the errors made by an unconstrained model

Equality Constraints

Alternatively, we can undo some of the adjustments made for inequalities

In particular, we can switch to a classic signed Lagrangian term:

$$\lambda^T g(\hat{y})$$

- With an unconstrained λ

The dual step will still try to maximize the Lagrangian loss

- If $g_k(\hat{y}) > 0$, λ_k will be increased as usual
- ...But if $g_k(\hat{y}) < 0$, then λ_k will be decreased instead

When close to satisfaction, positive and negative changes will balance out

Equality Constraints

Alternatively, we can undo some of the adjustments made for inequalities

In particular, we can switch to a classic signed Lagrangian term:

$$\lambda^T g(\hat{y})$$

- With an unconstrained λ

The dual step will still try to maximize the Lagrangian loss

- If $g_k(\hat{y}) > 0$, λ_k will be increased as usual
- ...But if $g_k(\hat{y}) < 0$, then λ_k will be decreased instead

When close to satisfaction, positive and negative changes will balance out

However, be careful of numeric stability!

If the dual step is too fast or too slow, convergence may suffer

Now, let's see this approach in action

Fixed-Multiplier Approach

We'll start by building an approach with fixed λ

...Which will serve as a baseline

```
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

Fixed-Multiplier Approach

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

```
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
```

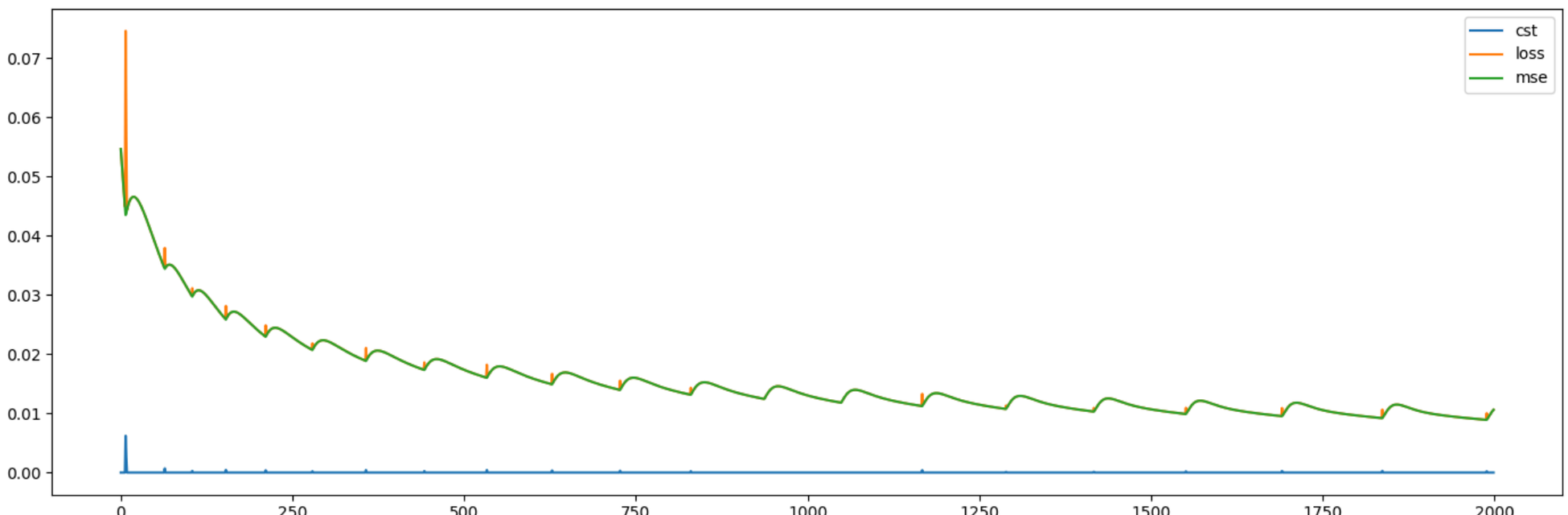
- The main loss is defined when calling `compile`

Training the Fixed-Multiplier Approach

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 [8]:
```
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=
util.plot_training_history(history, figsize=figsize)
```



Fixed-Multiplier Approach Evaluation

Let's check both the prediction quality and the DIDI

In [10]:
```
tr_pred2 = nn2.predict(tr[attributes], verbose=0)
r2_tr2 = r2_score(tr[target], tr_pred2)
ts_pred2 = nn2.predict(ts[attributes], verbose=0)
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: 0.30 (training), 0.20 (test)
DIDI: 0.08 (training), 0.04 (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

Dual Ascent Approach

Now, let's switch to the dual ascent approach

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

    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
- ...Which we'll manage through a separate optimizer

Dual Ascent Approach

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

```
def __custom_loss(self, x, y_true, sign=1):
    y_pred = self.base_pred(x, training=True) # obtain the predictions
    mse = self.compute_loss(x, 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`)

Dual Ascent Approach

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

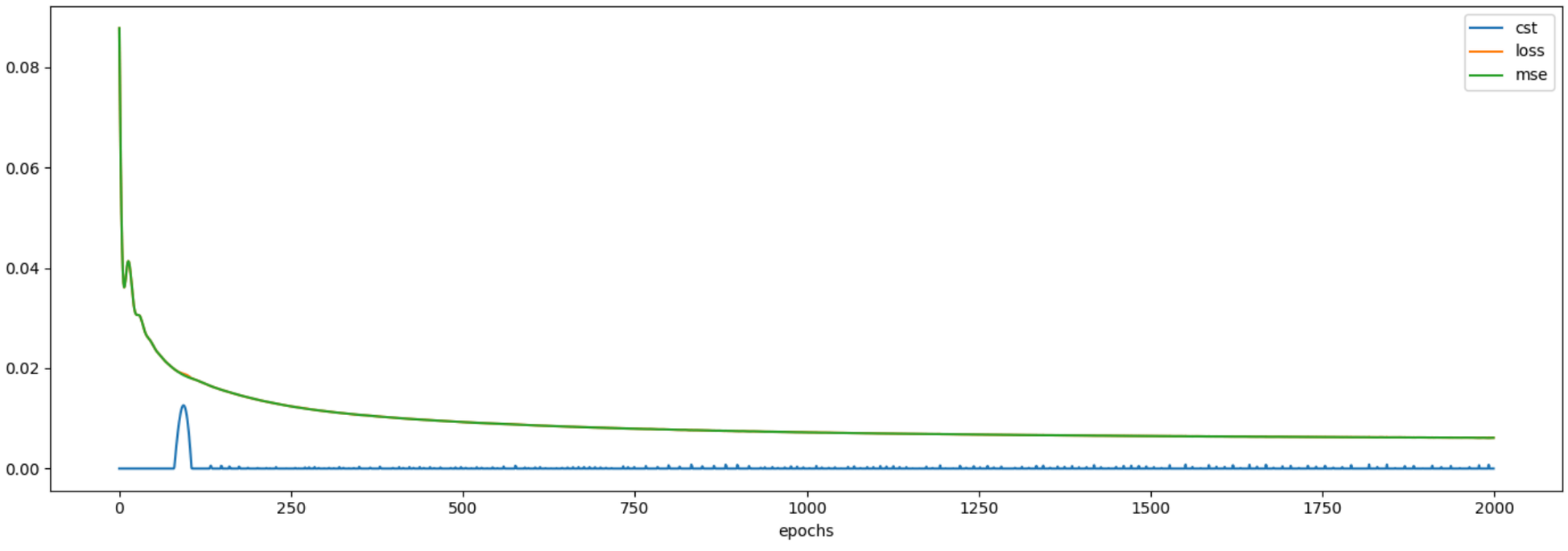
```
    def train_step(self, data):
        x, y_true = data # unpacking
        with tf.GradientTape() as tape: # first loss (minimization)
            loss, mse, cst = self.__custom_loss(x, y_true, sign=1)
        grads = tape.gradient(loss, wgt_vars) # adjust the network weights
        self.optimizer.apply_gradients(zip(grads, 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.dual_optimizer.apply_gradients(zip([grads], [self.alpha]))
```

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

Training the Dual Ascent Approach

The new approach leads fewer oscillations at training time

In [34]:
```
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=2(
util.plot_training_history(history, figsize=figsize)
```




```
Model loss: 0.0061 (training)
```

Lagrangian Dual Evaluation

Let's check the new results

In [36]:
```
tr_pred3 = nn3.predict(tr[attributes], verbose=0)
r2_tr3 = r2_score(tr[target], tr_pred3)
ts_pred3 = nn3.predict(ts[attributes], verbose=0)
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: 0.60 (training), 0.54 (test)
DIDI: 0.13 (training), 0.13 (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!