Constrained ML via Lagrangian Approaches

Constraints as Penalties

Let's consider our general situation

We have a training problem:

$$\underset{\theta}{\operatorname{argmin}} \left\{ L(\hat{y}) \mid \hat{y} = f(x;\theta) \right\}$$

And constraints, that we'll view as an inequality on a vector function

$$g(\hat{y}) \leq 0$$

- Here $g(\hat{y}) = \{g_k(\hat{y})\}_{k=1}^{m}$

Constraints as Penalties

Let's consider our general situation

We have a training problem:

$$\underset{\theta}{\operatorname{argmin}} \left\{ L(\hat{y}) \mid \hat{y} = f(x;\theta) \right\}$$

And constraints, that we'll view as an inequality on a vector function

$$g(\hat{y}) \leq 0$$

- Here $g(\hat{y}) = \{g_k(\hat{y})\}_{k=1}^{m}$

We will explore the idea of turning the constraints into loss terms

- Doing tha will steer the model towards satisfying the constraints
- ...And can be thought of as a form of regularization

In fact, an early example of this approach is called Semantic Based Regularization

Lagrangian-like Loss

The basic theory is rooted in Lagrangian duality

...Where our constrained optimization problem would be turned into:

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

- Where $\boldsymbol{\lambda}$ is a vector of weigths $\in \mathbb{R}^m$, called Lagrangian multipliers

Lagrangian-like Loss

The basic theory is rooted in Lagrangian duality

...Where our constrained optimization problem would be turned into:

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

- Where $\boldsymbol{\lambda}$ is a vector of weigths $\in \mathbb{R}^m$, called Lagrangian multipliers

However, this formulation is not a good choice in our case

- There are a few reasons for this, non of them trivial
- Here we will focus just on the main one

A Stop-gain Mechanism

We are considering inequality constraints

$$g(\hat{y}) \leq 0$$

- Predictions with $g_k(\hat{y}) < 0$ are equivalent to those with $g_k(\hat{y}) = 0$
- ...But in a classical Lagrangian approach a slack translates to a reward

$$\mathcal{L}(\theta, \lambda) = L(\hat{y}) + \lambda^T \underbrace{g(\hat{y})}_{<0}$$

In classical Lagrangian theory, this is countered by changing the sign of λ_k

- However, this is sound only under specific assumptions (e.g. convexity)
- ...And it requires to optimize θ and λ simultaneously

Clipped Penalizers

However, there's a far easier alternative

We can just use non-linearity to remove the reward effect, e.g. by clipping:

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

- The maximum operator will neutralized any reward when $g_k(\hat{y}) < 0$
- ...Which is effectively equivalent to forcing λ_k to 0

Clipped Penalizers

However, there's a far easier alternative

We can just use non-linearity to remove the reward effect, e.g. by clipping:

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

- The maximum operator will neutralized any reward when $g_k(\hat{y}) < 0$
- ...Which is effectively equivalent to forcing λ_k to 0

With the new penalizer, for any $\lambda \geq 0$

- When all constraints are feasible, we preserve the original loss function
- When a constraint is infeasible, we introduce a penalty

This approach comes from penalty methods

Equality Constraints

Equality constraints allow for a simpler formulation

In principle, given an equality constraint:

$$g_k(\hat{y}) = 0$$

We can state it as two inequality constraints:

$$g_k(\hat{y}) \leq 0 \quad \text{and} \quad -g_k(\hat{y}) \leq 0$$

...And build two (weighted) violation terms:

$$\lambda'_k \max\left(0, g_k(\hat{y})\right) \quad \text{and} \quad \lambda''_k \max\left(0, -g_k(\hat{y})\right)$$

- With $\lambda'_k, \lambda''_k \geq 0$

Equality Constraints

Summing the two terms leads to a simplified formula

$$\lambda'_k \max\left(0, g_k(\hat{y})\right) + \lambda''_k \max\left(0, -g_k(\hat{y})\right) = \lambda_k |g_k(\hat{y})|$$

- Where $\lambda_k = \lambda'_k + \lambda''_k$ and there is no sign restriction

Equality Constraints

Summing the two terms leads to a simplified formula

$$\lambda'_k \max\left(0, g_k(\hat{y})\right) + \lambda''_k \max\left(0, -g_k(\hat{y})\right) = \lambda_k |g_k(\hat{y})|$$

- Where $\lambda_k = \lambda'_k + \lambda''_k$ and there is no sign restriction

Another common form relies on the square of the violation

...Meaning that we consider the loss:

$$L(\hat{y}) + \lambda^T g(\hat{y})^2$$

This form of penalizer is related to properties of the Normal distribution

- It is particularly well-suited for soft constraints
- ...But we won't discuss the connection in detail

Differentiability

It's worth talking about differentiability

- Lagrangian approaches for knowledge injection
- ...Are most common with differentiable constraints

...Even if differentiability is not strictly needed

Differentiability

It's worth talking about differentiability

- Lagrangian approaches for knowledge injection
- ...Are most common with differentiable constraints

...Even if differentiability is not strictly needed

Differentiability might be needed

...Depending on which training algorithms is used, e.g.:

- Gradient descent
- Gradient boosting
- ...

...Which means that we need differentiability when using Neural Networks

Calibrating the Multipliers

Any $\lambda \geq 0$ results in a sound behavior, but which value should we pick?

- Larger values will likely allow for some degree violation
- Under proper assumptions, larger values can guarantee satisfaction

The best strategy depends on our goal

If the system value stems from its accuracy

- Then the constraints are just a source of symbolic knowledge
- ...And we will typically work with soft constraints

If satisfying constraints is a value per-se

- Then the constraints are our main goal
- ...And we will typically want hard constraints

Calibrating λ for Maximal Accuracy

When the goal is improving accuracy

...We can just assess the quality of a λ vector by cross-validation

- Then λ can be thought of as any other ML hyperparameter
- ...And we can optmize it via grid search, Baysian optimization, etc.

In practice, however, this approach doest not always work

Calibrating λ for Maximal Accuracy

When the goal is improving accuracy

...We can just assess the quality of a λ vector by cross-validation

- Then λ can be thought of as any other ML hyperparameter
- ...And we can optmize it via grid search, Baysian optimization, etc.

In practice, however, this approach doest not always work

In most cases, knowledge injection is used when supervised data is scarce

...And in this situation cross-validation is not very reliable

- Then we λ can be calibrated via probabilistic considerations
- ...Or via heuristic considerations

Both approaches are not ideal, but at least they are applicable

Calibrating λ for Constraint Satisfaction

If our main goal is constraint satisfation

...Then we might think of just choosing a very large λ

- Intuitively, for sufficiently large λ values
- ...We should at least approach constraint satisfaction

In practice, this almost never a good idea

Overly large λ values may lead to numerical issues:

$$L(y, f(x, \theta)) + \lambda^T \max(0, g(f(x, \theta)))$$

- Any time we have a constraint violation
- ...The gradient may include a disproportionaly large term

Gradient Ascent to Control the Multipliers

A gentler approach consists in using gradient ascent for the multipliers

Let's consider our modified loss:

$$\mathcal{L}(\theta, \lambda) = L(y, f(x, \theta)) + \lambda^T \max(0, g(f(x, \theta)))$$

- This is actually differentiable in λ

The gradient is also a very simple expression:

$$\nabla_\lambda \mathcal{L}(\theta, \lambda) = \max(0, g(f(x, \theta)))$$

- For satisfied constraints, the partial derivative is 0
- For violated constraints, it is equal to the violation

The Dual Ascent Method

Therefore, we can solve our constrained ML problem

...By alternating gradient descent and ascent:

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

We might still reach impractical values for λ

...But the gentle updates can keep the gradient more stable

- At the beginning, SGD will be free to prioritize accuracy
- After some iterations, both θ and λ will be nearly (locally) optimal