Practical 1: Smooth and Non-smooth Methods for Regression

1. Introduction

This page contains the instruction for the first practical.

The code template requires some dependencies to be installed and provides some useful utilities. Please read the code setup for more information before going forward with this practical. All the code will be run using the `main.py` file and should be run from its directory.

By the end of this lab, you will be able to express different regression problems and use an appropriate algorithm to solve it.

This lab is split in three parts, first we present the regression problem that we consider here. Then we consider the ridge regression problem which is a simple smooth problem. In the third part, we extend this problem to more complex objective of the lasso problem. Finally a bonus section considers an even less smooth objective.

2. Ridge Regression Problem

The optimization problem considered here is a regression problem. Formally, we have a training set of N data samples $(x_i, y_i)_{1 \leq i \leq N}$. Each sample $x_i \in \mathbb{R}^d$ has a corresponding ground truth label $y_i \in \mathbb{R}$. (Note: in the code, d is denoted by $n\_features$).

In this practical, we consider a linear prediction model. This means that given a set of weights $w \in \mathbb{R}^{d \times 1}$ and an input x_i, the prediction of our model is given by $x_i^\top w$.

To evaluate the quality of our model prediction, we use a loss function. In this practical, this loss function is the mean squared error (except for the last bonus part).

Finally, to make sure that the problem is well defined (it has a single solution) and that it generalize to similar unseen data, we need a regularization function, called reg below. In this practical, this regularization function will be either the l1 or l2 norm.

The optimization problem that we want to solve here is thus given as:

$$\min_w \frac{1}{N} \sum_i loss(w, x_i, y_i) + \frac{\mu}{2} reg(w)$$

Here μ is a hyper parameter that we use to control the balance between the loss term and the regularization term. We will come back to it later.

3. Datasets

As usual in empirical risk minimization, we use three "splits" of the available data. The training set is used to fit the model parameters w. The validation set is used to tune the hyper-parameters (e.g. the regularization coefficient μ, the type of regularization etc.). Finally, the test set is used to assess the final performance of the model once all hyper-parameters have been set (it should not be used for any algorithmic choice or choice of hyper-parameter!).

For this practical, two standard datasets are provided to you. The first one (Boston Housing Data) is small enough so that all the algorithms run quickly on it. The second one (California Housing Data) is larger scale to be able to see how well the algorithms scale for larger datasets.

3.1. Boston Housing Data

This dataset contains data about the house prices in the suburbs of Boston in the 80s. The goal of the regression is to predict the price of a given house given some attributes such as for example crime rate in the neighborhood, proportion of shops, number of rooms, distance to employment centres or pupil to teacher ratio in town. There is a total of 13 such attributes that we call features in this practical. The number that should be predicted is the price in thousands of $.

For this practical, we split the dataset to get 256 training samples, 50 validation samples and 200 test samples. This is the default dataset used when no `--dataset` argument is specified.

3.2. California Housing Data

For this dataset, the goal is to predict the median price house in the different districts in California in the 90s. This dataset is significantly bigger as it contains $20,640$ samples in total. These are split in 50% train, 25% validation and 25% test. For the need of this practical, the number of features for each sample is $80,000$. You can use this dataset by specifying the `--dataset california` option in the command line.

WARNING: The memory requirement for this dataset is large especially for methods with bad asymptotic complexity. Don't use it if you have something really important running on your computer.

4. Ridge Regression

In this first part we consider the Ridge Regression problem. The loss is the mean square error between the prediction and the ground truth and the regularization

is the squared norm of the weights. We can thus write the problem as:

$$\min_w obj(w) = \frac{1}{N}\sum_i ||x_i^\top w - y_i||^2 + \frac{\mu}{2}||w||^2$$

Question 1: Is this function convex, strongly convex? Why?

Question 2: Is this function smooth? Which algorithm from class can you use to solve it? Can you think of another way to solve this problem?

4.1. Closed-Form Solution

The ridge regression problem can be solved much more efficiently than what is done in class.

Question 3: What is the equation satisfied by the gradient of *obj* at the optimal point w^*? Write this gradient, the equation it must verify and explain how to solve it.

Task 1: Implement the `oracle` and `task_error` methods for the `Ridge_ClosedForm` Objective in `objective/ridge.py`. You can test this with `python run_test.py TestObj_Ridge_ClosedForm`.

Task 2: Implement the `ClosedForm` Optimizer that simply stores the optimal solution into `variables.w`. You can test this with `python run_test.py TestOpt_ClosedForm`.

Task 3: Run this optimization with `python main.py --obj ridge --opt closed-form`. How precise are the predictions on the test set?

Question 4: How fast is this solver? What is the drawback of this method? Hint: What are the time and memory complexity of the methods.

(bonus) Task 4: Use the previously derived gradients and implement the `Ridge_Gradient` Objective. After doing the next section, use the `GD` Optimizer and compare the speed with the closed form version running `python main.py --obj ridge --opt gd --epoch 100`. In which case should this optimization method be used instead of the closed form?

5. Lasso Regression

It is sometimes interesting to obtain sparse weights for a regression problem, for instance to obtain a model with small memory footprint or to explain which input features can be ignored. For such a use case, we can use Lasso regression,

which uses an l1 norm for the regularization term and the mean squared error for the loss function:

$$\min_w obj(w) = \frac{1}{N}\sum_i ||x_i^\top w - y_i||^2 + \frac{\mu}{2}|w|$$

> **Question 5:** Is this function convex, is it smooth? Which algorithm from class can be used to solve this?

5.1. Non-Smooth Optimization

We optimize this objective with subgradient descent. See the `How to compute gradients` section of the `code_setup` document for how to implement your (sub)gradients.

> **Task 5:** Implement the `oracle` and `task_error` methods method for the `Lasso_subGradient` Objective in `objective/lasso.py`. You can test your implementation with `python run_test.py TestObj_Lasso_subGradient`.

For simplicity in this practical, both gradient descent and subgradient descent are implemented in the same `Optimizer`. They both accept gradient or subgradient from the objective in the `dw` oracle info and the difference lies in the learning rates they should use. For this, we provide two arguments in the `main.py`. First the `--init-lr` argument can be used to fix the initial learning rate of the method. Then the `--fix_lr` argument can be given to keep this learning rate fixed (for the case of gradient descent) and if not supplied, the $\frac{1}{\sqrt{it}}$ rescaling is used (for the case of subgradient descent).

> **Task 6:** Implement the `GD` optimizer in `optim/gd.py`. Make sure that your implementation returns the right result for any `--init-lr` (that you can access with `self.hparams.init_lr`) given and if `--fix_lr` (that you can access with `self.hparams.fix_lr`) is given as well. You can test this by running `python run_test.py TestOpt_GD`.
>
> **Task 7:** Run this optimization with `python main.py --obj lasso --opt gd --epoch 10`. How precise are the predictions on the test set?
>
> **Question 6:** How fast is the algorithm converging? What is the drawback of this algorithm?
>
> *Hint: try and speed up convergence by changing the initial step size.*

5.2. Smoothed Objective Optimization

To speed up convergence, we are going to smooth the Lasso objective. To do so, we need to smooth the absolute value of the l1 norm in the regularization term. We write this l1 norm as $max(w, -w)$ and so the goal here is to smooth this max operation.

> **Question 7:** How would you smooth this *max* operation? We refer to this new smoothed version as $smooth\_max(w)$.

If you don't have a good answer for Question 7. A good smoothing function usually takes a "temperature" parameter `temp` and is such that, when `temp` goes to zero, the smoothed function is the same as the original function. We thus have a set of smoothed function for which the tradeoff between smoothness and difference to the original function can easily be adjusted. For the $max_i(a_i)$ function, one such function is $temp * log\big[\sum_i exp(\frac{a_i}{temp})\big]$. Now that we have a smooth function we can optimize it using gradient descent.

> **Task 8:** Implement the `oracle` and `task_error` methods method for the `SmoothLasso_Gradient` Objective in `objective/lasso.py`. You can test your implementation for the default *smooth_max* function using `python run_test.py TestObj_SmoothedLasso_Gradient`
>
> **Task 9:** Run `python run_test.py TestObj_SmoothedLasso_Gradient_lowtemp`. What is the difference with the test above? What is the reason for this test to fail (if it does) with your implementation? Fix your implementation to make sure this test passes?
>
> *Hint: how big is exp(1e3) and what is the precision of a single precision floating point number?*
>
> **Question 8:** Run this optimization with `python main.py --obj smooth-lasso --opt gd --epoch 10 --temp 1`. How fast is this method? What is the effect of changing the `temp` parameter?

6. Bonus Section: L1 Loss Function

For some application, we not only want l1 regularization but also a loss function based on l1 distance. This can be useful when we want to the relative importance of outliers for instance.

> **(bonus) Task 10:** Create a new file in `objective/` and implement the new `SmoothL1Loss_Gradient` Objective into it. Change the `objective/__init__.py` file to handle a `smoothl1` objective type. Change the `cli.py` file and add the new `smoothl1` objective to the allowed objectives.

(bonus) Question 9: Compare this objective to the Ridge and Lasso. How does it behave? What is the final error?

7. Bonus Section: Autograd Engine

For this practical, all the gradients (and subgradients) computation have been done by hand. This means that you have to write down by hand all the gradient computations and then implement them.

This has the advantage of being the simplest to implement and the fastest implementation (if you reduced the formula properly on paper). However, the drawback of this method is that writing down these gradients on paper can be error-prone and time-consuming, especially for complex functions.

Another technique to get these gradients is to use an autograd engine. In this case, you simply code the computation of your output and then ask the autograd engine to give your the gradient at the current point.

This has the advantage of being simpler to implement. Since you need to code the output computation in any case, you get the gradients with no extra derivations. The drawback of this method is that it is more involved code-wise (requires an autograd library) and the gradient computation can be slower than if implemented with a custom formula.

In the context of this practical, the library that we use for all the Tensors, `torch` includes an autograd engine. In this library, for any Tensor for which you require gradients, you need to create it with the argument `requires_grad=True`. You can then perform any operation on this Tensor. Once you have the final loss value (a Tensor containing a single element usually), you can call `.backward()` on it for the autograd engine to compute all the gradients. After this, you can simply access the gradients of all the Tensors you created with `requires_grad=True` by accessing their `.grad` attribute. A simple example:

```
import torch

# If x contains a single element
# Perform f(x) = x^2 + 2*x
def f(x):
    return x**2 + 2*x

# We have g(x0) = df/dx(x0) = 2*x0 + 2
# Evaluate g(1)
x = torch.tensor([1.], requires_grad=True)
output = f(x)
output.backward()
print(x.grad) # g(1) = 4
```

```
# Evaluate g(5)
x = torch.tensor([5.], requires_grad=True)
output = f(x)
output.backward()
print(x.grad) # g(5) = 12

# If x is a matrix
# Perform f(x) = mean(x^T x) + l2norm(x)
def f(x):
    return torch.mm(x.t(), x).mean() + x.norm()

# Evaluate g of a random matrix x of size 5x10
x = torch.rand(5, 10, requires_grad=True)
output = f(x)
output.backward()
print(x.grad) # 5x10 gradient matrix
```

In the case of this practical, this can be done by creating the variables with `requires_grad=True` and then calling backward on the objective after you computed it.

> **(bonus) Task 11:** Re-implement the above methods using the autograd engine. How much simpler is the code?