COMP4680: Advanced Topics in Machine Learning
ASE: Total Variation Optimization

Yiran Mao

November 16, 2023

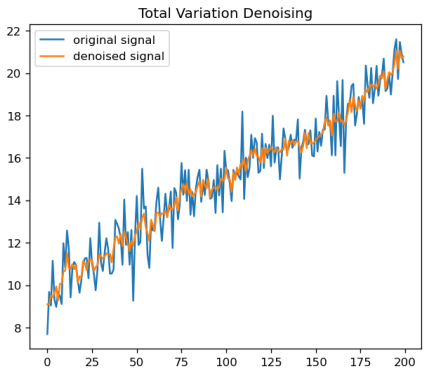
Abstract

This project involves building a autograd function for declarative total variation denoising node in PyTorch, verified the higher efficiency of this declarative node compared to the PyTorch built-in automatic differentiation.

1 Background

1.1 Total Variation Denoising

Total variation denoising is a method used to remove noise from a signal while preserving sharp transitions or rapid variations in the original signal. It is based on the total variation function, which assigns large values to rapidly varying signals. Total variation denoising can remove much of the noise while still preserving occasional rapid variations in the original signal. [2] The method involves minimizing a trade-off function: $f_\lambda(x, u) = \frac{1}{2}\|u - x\|_2^2 + \lambda \sum_{i=1}^{n-1} |u_{i+1} - u_i|$ where $x \in \mathbb{R}^n$ is input signal $u \in \mathbb{R}^n$ is denoised signal and $\lambda \in \mathbb{R}$ is balance factor. Specifically, total variation denoising is an unconstrained optimization problem:



$$\text{minimize}_u \quad \frac{1}{2}\|u - x\|_2^2 + \lambda \sum_{i=1}^{n-1} |u_{i+1} - u_i|$$

1.2 Declarative Nodes

An imperative node refers to a node within a neural network that explicitly defines the relationship between the input variable, denoted as x, and the output variable, denoted as y. On the other hand, a declarative node specifies the input-output relationship implicitly as the solution to an optimization problem. [3].

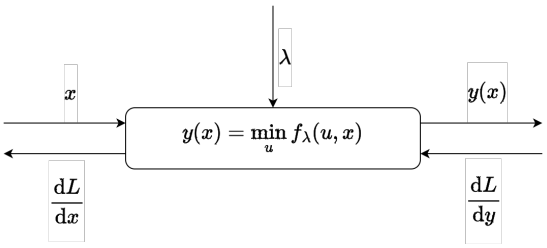


Figure 1: Total Variation Denoising Declarative Node

2 Implementation of Total Variation Denoising Node

2.1 Derivative Calculation

By chain rule, $\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}$, thus the derivative $\frac{\mathrm{d}y}{\mathrm{d}x}$ is crucial to implement a declarative node. Consider the total variation denoising problem:

$$\text{minimize}_u \quad f(x,u) = \frac{1}{2}\|u-x\|_2^2 + \lambda\sum_{i=1}^{n-1}|u_{i+1}-u_i|$$

We can convert this as an equality constrained problem with equality constrains h be constant zero function. Then by corollary, we have the following derivation:

$$\begin{aligned} A &= \tfrac{\partial h(x,u)}{\partial u} = \mathbf{0} \quad B = \tfrac{\partial^2 f(x,u)}{\partial x \partial u} - \textstyle\sum_{i=1}^{q} \nu_i \tfrac{\partial^2 h_i(x,u)}{\partial x \partial u} = \tfrac{\partial^2 f(x,u)}{\partial x \partial u} \\ C &= \tfrac{\partial h(x,u)}{\partial x} = \mathbf{0} \quad H = \tfrac{\partial^2 f(x,u)}{\partial u^2} - \textstyle\sum_{i=1}^{q} \nu_i \tfrac{\partial^2 h_i(x,u)}{\partial u^2} = \tfrac{\partial^2 f(x,u)}{\partial u^2} \end{aligned}$$

Therefore,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = H^{-1}A^T(AH^{-1}A^T)^{-1}(AH^{-1}B - C) - H^{-1}B = H^{-1}B$$

The first order partial derivative of $f(x,u)$ is given by

$$\frac{\partial f(x,u)}{\partial u_i} = \begin{cases} u_1 - x_1 + \lambda\,\mathbf{sign}(u_1 - u_2) & \text{if } i = 1 \\ u_n - x_n + \lambda\,\mathbf{sign}(u_n - u_{n-1}) & \text{if } i = n \\ u_i - x_i + \lambda\,(\mathbf{sign}(u_i - u_{i-1}) - \mathbf{sign}(u_{i+1} - u_i)) & \text{otherwise} \end{cases}$$

It follows that

$$B = \frac{\partial^2 f(x,u)}{\partial x \partial u} = \begin{bmatrix} -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & -1 \end{bmatrix} = -I$$

$$H = \frac{\partial^2 f(x,u)}{\partial u^2} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} = I$$

Thus $\frac{\mathrm{d}y}{\mathrm{d}x} = H^{-1}B = I$, and we can implement the backward pass as $\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y}$.

2.2 The PyTorch Implementation

The forward pass is executed through the utilization of gradient descent with backtracking line search[2], while the backward pass is implemented as the previous calculation.

```
class TotalVariationFcn(torch.autograd.Function):
    """PyTorch autograd function for total variation denoising."""

    @staticmethod
    def forward(ctx, x, lmd):
        with torch.no_grad():
            x_clone = x.detach()
            u = torch.nn.Parameter(x_clone, requires_grad=False)
            y = gradient_descent(u,
                lambda u: f(u, x_clone, lmd),
                lambda u: gradient(u, x_clone, lmd),
                line_search=True)
        return y

    @staticmethod
    def backward(ctx, dLdY):
        with torch.no_grad():
            dLdX = dLdY
        return dLdX, None
```

3 Analysis and Reflection

3.1 Bi-level Optimization Example Anaylsis

To analyze the performance, we construct the following bi-level optimization problem:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|y^\star - y^{\text{target}}\|_2^2 \\ \text{subject to} \quad & y^\star = \text{argmin}_u \left(\frac{1}{2} * \|u - x\|_2^2 + \lambda \sum_{i=1}^{n-1} |u_{i+1} - u_i| \right) \end{aligned}$$

An assessment of the efficiency of our TotalVariationFcn can be conducted by comparing the execution time required to solve this bi-level optimization problem using our TotalVariationFcn against that of the PyTorch built-in automatic differentiation. We call the former as explicit method, while the latter as automatic method. For a series of randomly generated y^{target} with exponentially increasing dimension(from 2 to 2^{17}), plot the graph of runtime versus the dimension of y^{target}. The result shows as follows:

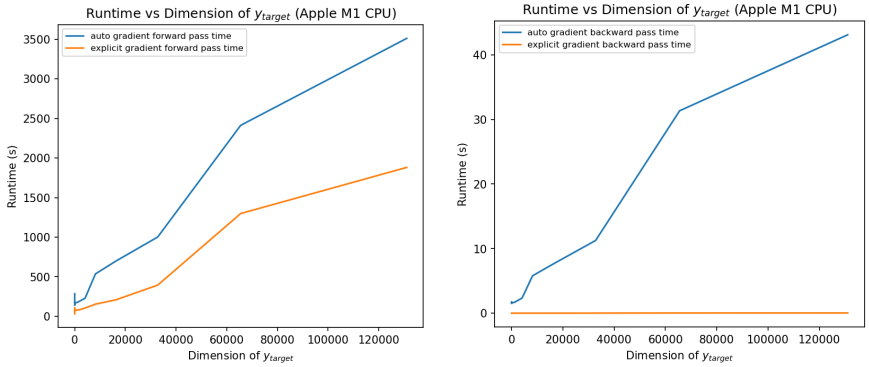


Figure 2: Plots of runtime versus y^{target} dimension

Based on the plot, it is evident that the explicit method significantly outperforms the automatic method in terms of total runtime, as anticipated. Specifically, the explicit method is approximately twice as fast as the automatic method during the forward pass and orders of magnitude faster during the backward pass. However, ideally the two forward pass runtime should be similar, while the difference between backward pass runtime of explicit method and automatic method can be even larger. Our findings are attributed to PyTorch performing automatic differentiation concurrently with

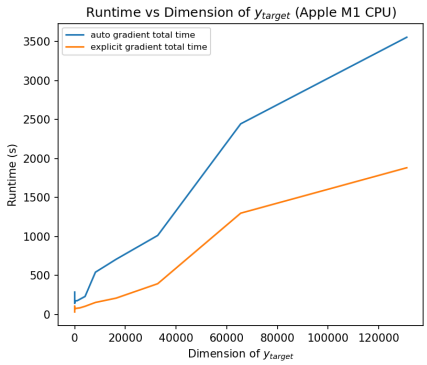


Figure 3: Total runtime versus y^{target} dimension

the forward pass, resulting in the inclusion of gradient calculations in the runtime of the forward pass. Regrettably, despite extensive research, it remains challenging to completely separate the forward pass and backward pass processes, making it difficult to precisely measure the runtime of each process. Nevertheless, it is clear that the explicit method indeed offers a significantly more efficient approach.

3.2 Reflection

The efficiency of the forward pass by using the manually implemented gradient descent method with line search is not ideal. The built-in PyTorch optimizer with stochastic gradient descent can enhance the speed of the forward pass. However, it is important to note that PyTorch has limitations in tracking the gradient of the forward pass automatically, as the optimizing step by the built-in optimizer is not considered "differentiable". So the built-in optimizer is not utilized in this scenario since automatic differentiation is required to set up comparison. Additionally, the execution of all operations is currently performed on the Apple M1 CPU, as there is a lack of cuda device, though the program has GPU support. Therefore, a performance comparison using GPU has not been conducted. However, it is expected that the explicit method will still be significantly faster, although the gap between may decrease.

4 Appendix

Theorem *Dinis Implicit Function Theorem*
Suppose $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be differentiable in a neighbourhood of (x, u) such that $f(x, u) = 0$, and let $\frac{\partial f(x,u)}{\partial u}$ be nonsingular. Then the solution mapping Y has a single-valued localisation y around x for u which is differentiable in a neighbourhood X of x with Jacobian satisfying

Corollary *Differentiating Equality Constrained Optimization Problems*
Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$. Let

$$y(x) \in \begin{array}{ll} \arg\min_{u \in \mathbb{R}^m} & f(x, u) \\ \text{subject to} & h(x, u) = 0 \end{array}$$

Assume that $y(x)$ exists, and f and h are twice differentiable in the neighbourhood of $(x, y(x))$, and that $\mathbf{rank}\left(\frac{\partial h(x,y)}{\partial y}\right) = q$. Then for H nonsingular:

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = H^{-1}A^T(AH^{-1}A^T)^{-1}(AH^{-1}B - C) - H^{-1}B$$

where

$$\begin{array}{ll} A = \frac{\partial h(x,y)}{\partial y} \in \mathbb{R}^{q \times m} & B = \frac{\partial^2 f(x,y)}{\partial x \partial y} - \sum_{i=1}^{q} \nu_i \frac{\partial^2 h_i(x,y)}{\partial x \partial y} \in \mathbb{R}^{m \times n} \\ C = \frac{\partial h(x,y)}{\partial x} \in \mathbb{R}^{q \times n} & H = \frac{\partial^2 f(x,y)}{\partial y^2} - \sum_{i=1}^{q} \nu_i \frac{\partial^2 h_i(x,y)}{\partial y^2} \in \mathbb{R}^{m \times m} \end{array}$$

Definition *Bi-level Optimization Problem*
Bi-level optimization problems refer to two related optimization tasks, each one is assigned to a decision level (i.e., upper and lower levels).[1][2] The evaluation of an upper level solution requires the evaluation of the lower level.

$$\begin{array}{ll} \text{minimize} & J(x, y) \\ \text{subject to} & y \in \operatorname{argmin}_u f(x, u) \end{array}$$

References

[1] Bilevel optimization, wikipedia, May 2023. Page Version ID: 1157661776.

[2] Stephen P. Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge Univ. Pr., 2011.

[3] Stephen Gould. Lecture notes on Differentiable Optimisation in Deep Learning. Technical report, 2023.