SciML - Gradients and Optimization

Mark Asch - IMU/VLP/CSU

2023

Program

1. Optimization:
   (a) Gradients and Optimization.
   (b) Automatic Differentiation: Computational Graphs, Backpropagation and Adjoints.
   (c) Optimization and AD with PyTorch.

2. Physics constrained learning (PCL).

3. Physics Induced Neural Networks (PINN).

4. Operator-based learning.

OPTIMIZATION, GRADIENTS

Recall: Optimization

- Optimization is at the very heart of:

  $\Rightarrow$ Machine Learning
  $\Rightarrow$ Inverse Problems (including Data Assimilation)
  $\Rightarrow$ Digital Twins

Definition 1. In an optimization problem, we seek the minimum of a cost function, usually an error function describing the mismatch between model output and observations/data/measurements.

The general, unconstrained optimization problem is

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}), \tag{1}$$

or find $\mathbf{x}_*$ that satisfies

$$\mathbf{x}_* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}), \tag{2}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth function.

Recall: Gradients

- Necessary and sufficient conditions for finding a (local) minumum, rely on the existence of gradients that indicate the best direction for seeking the minimum—we just need to "slide" downhill . . .

Theorem 1 (Taylor's). *If* $f\colon \mathbb{R}^n \to \mathbb{R}$ *is continuously differentiable and if* $\mathbf{p} \in \mathbb{R}^n$*, then*

$$f(\mathbf{x}+\mathbf{p}) = f(\mathbf{x}) + \nabla f(\mathbf{x}+t\mathbf{p})^{\mathrm{T}}\mathbf{p}\,, \quad t \in (0,1),$$

where the gradient vector

$$\nabla f(\mathbf{x}) \doteq (\partial f(\mathbf{x})/\partial x_1, \ldots, \partial f(\mathbf{x})/\partial x_n)^{\mathrm{T}}\,.$$

Moreover, if $f \in C^2$ *(meaning that* f *now has two continuous derivatives), then*

$$\nabla f(\mathbf{x}+\mathbf{p}) = \nabla f(\mathbf{x}) + \int_0^1 \nabla^2 f(\mathbf{x}+t\mathbf{p})\mathbf{p}\,\mathrm{d}t$$

and

$$f(\mathbf{x}+\mathbf{p}) = f(\mathbf{x})+\nabla f(\mathbf{x}+t\mathbf{p})^{\mathrm{T}}\mathbf{p}+\frac{1}{2}\mathbf{p}^{\mathrm{T}}\nabla^2 f(\mathbf{x}+t\mathbf{p})\mathbf{p}$$

for some $t \in (0,1)$, *where the Hessian matrix*

$$\nabla^2 f(\mathbf{x}) \doteq \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}.$$

- We can now state the necessary condition for the existence of a minimum.

Theorem 2 (First-Order Necessary Condition). *If* $\mathbf{x}_*$ *is a local minimizer of* f, *and if* f *is differentiable in a neighborhood of the point* $\mathbf{x}_*$, *then*

$$\nabla f(\mathbf{x}_*) = 0. \tag{3}$$

- Please see [1] for ALL the details.

- But, as said above, the disappearance of the gradient is unfortunately not sufficient to guarantee that $\mathbf{x}_*$ is a minimizer.

  $\Rightarrow$ It only implies that f is stationary at $\mathbf{x}_*$ to first order, meaning that f is insensitive to small changes, or perturbations of $\mathbf{x}_*$.

  $\Rightarrow$ But we can (and do) use this necessary condition to solve the system of n algebraic equations in n unknowns defined by (3).

  $\Rightarrow$ Then we have to sort out and decide which of the candidate points are indeed minimizers.

  $\Rightarrow$ This requires second-order information that can be obtained from the Hessian, if and when it is available. If this is the case, then we can state a sufficient condition for $\mathbf{x}_*$ to be a local minimizer.

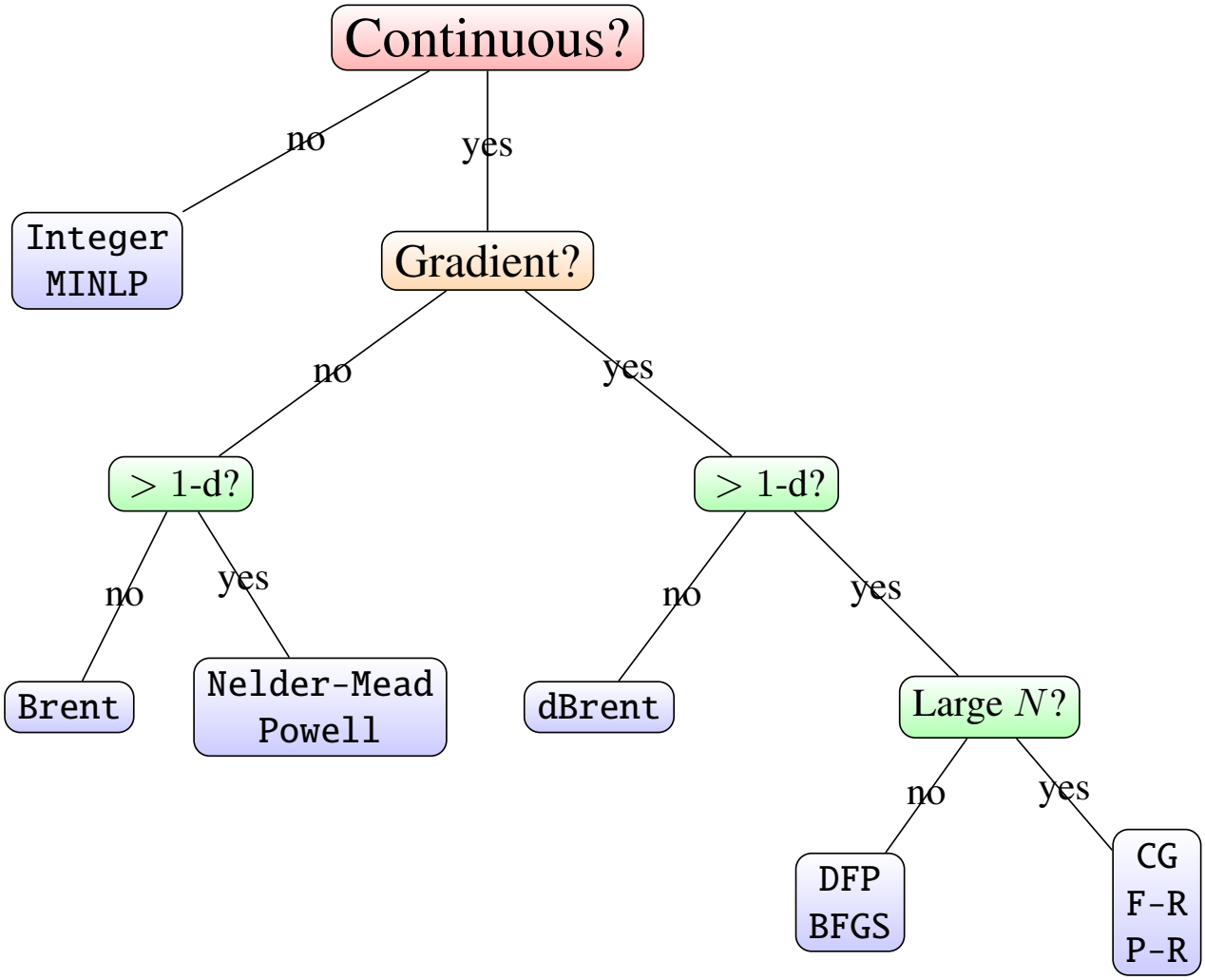
Theorem 3 (Second-Order Sufficient Condition). *Suppose for a point $\mathbf{x}_*$ the function f has first- and second-order derivatives. Suppose also that the gradient of f is zero and that the Hessian is positive definite at $\mathbf{x}_*$. Then $\mathbf{x}_*$ is a (strict) local minimum of f.*

- The only thing that is left is an eventual condition for a

global minimizer. Apart from the definition given above, there is only one case in which we can be sure that a local minimizer is indeed global.

Theorem 4 (Global Minimizer). *If f is a convex function, then any local minimizer $\mathbf{x}_*$ of f is a global minimizer. Moreover, if f is differentiable, then any stationary point $\mathbf{x}_*$ of f is a global minimizer.*

(Continuous, Unconstrained) Optimization Tree



Credit: [2].

Practical Optimization for SciML

- SciML problems will usually fall into the category "Large N", since

  $\Rightarrow$ the ML part usually implies a large number of weights
  $\Rightarrow$ the inversion/assimilation process, implies a large number of iterations

- Lower-order methods, based on gradient descent (which is NEVER used in classical CSE) are used here:

  $\Rightarrow$ SGD (stochastic gradient descent) and its variants, notably ADAM
  $\Rightarrow$ LBFGS (quasi-Newton) when higher order is needed and feasible

OPTIMIZATION ALGORITHMS

Recall: Gradient Descent

- Most minimization algorithms have the following basic structure:

```
while x^(k) is not a minimum
    calculate the step direction p^(k)
        with ||p^(k)|| = 1
    calculate the step size α^(k)
    update x^(k+1) = x^(k) + α^(k) p^(k)
    k = k + 1
 end.
```

- Gradient-based methods will use the slope, or gradient of the function as the choice for the direction, and then attempt to descend this slope to a lower-valued point.

- Steepest descent uses the fact that any (differentiable) function f decreases most rapidly in the direction of $-\nabla f$, at least locally.

- Then we need to solve a 1D minimization problem to find the optimal step size (to prevent overshooting).

- **Steepest Gradient Descent Algorithm:**

If $f: \mathbb{R}^n \to \mathbb{R}$ is a differentiable function and $\mathbf{x}_0 \in \mathbb{R}^n$ is an initial guess, then the following SDG algorithm computes $\mathbf{x}_* = \text{argmin}_{\mathbf{x}} f(\mathbf{x})$.

1: $k = 0$, initial guess $\mathbf{x}_0$ ▷ Initialization
2: **while** not converged **do** ▷ Convergence criteria
3: $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$ ▷ Steepest descent direction
4: Compute α_k that minimizes $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$
5: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ ▷ Update solution
6: $k = k + 1$
7: **end while**
8: $\mathbf{x}_* = \mathbf{x}_k$ ▷ Output the converged solution

Newton and Quasi-Newton Methods

- If we have access to the Hessian, in addition to the gradient, then we can formulate an extremely rapidly converging algorithm, Newton's method.

  $\Rightarrow$ However, this method is rarely used in oractise---see explanations below---and is replaced by quasi-Newton methods that use some type of approximation of the Hessian.

- Newton's method supplies a local quadratic approximation to an objective function, which is good because we can readily compute the minimum of a quadratic function.

- Take a truncated Taylor series expansion of the function f and expand in the neighborhood of a point $\mathbf{x}$ up to order two.

  $\Rightarrow$ Suppose that

$$f : \mathbb{R}^n \to \mathbb{R}$$

and that f has two continuous derivatives. Then we can expand

$$f(\mathbf{x}_k+\mathbf{p}) \approx f_k+\mathbf{p}^{\mathrm{T}}\nabla f_k+\frac{1}{2}\mathbf{p}^{\mathrm{T}}\nabla^2 f_k\mathbf{p} \doteq m_k(\mathbf{p}),$$

where $f_k = f(\mathbf{x}_k)$, ∇f is the gradient of f with i-th element $g_i = \frac{\partial f}{\partial x_i}$ and $\nabla^2 f$ is its Hessian, whose ij-th element is $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$

$\Rightarrow$ To minimize m_k with respect to $\mathbf{p}$, we take its gradient and impose the necessary condition, $\nabla_{\mathbf{p}} m_k(\mathbf{p}) = 0$, to obtain

$$0 + \nabla f_k + \nabla^2 f_k \mathbf{p} = 0,$$

from which we deduce the Newton direction

$$\mathbf{p}^{\mathrm{N}} = -\left(\nabla^2 f_k\right)^{-1} \nabla f_k.$$

$\Rightarrow$ The iteration step becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H^{-1}(\mathbf{x}_k) g(\mathbf{x}_k).$$

- The method is summarized in this very simple Newton algorithm:

$\mathbf{x} = \mathbf{x}_0$
`for` $k = 0, 1, 2, ...$
    `solve` $\nabla^2 f(\mathbf{x}_k)\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$
`end`

- The convergence is local, but quadratic, which is extremely rapid. However, Newton's method has three major inconveniences. The method can be:

1. Unreliable due to the (very) local convergence and the high sensitivity to the initial guess.

2. Expensive due to the denseness of the Hessian matrix, especially for large n.

3. Complicated since the Hessian is invariably difficult to compute.

Quasi-Newton: BFGS and L-BFGS

- Quasi-Newton methods have been developed to overcome some of the shortcomings of Newton's method, while still trying to use some kind of second-order information.

  $\Rightarrow$ The advantage will be a gain in convergence rate.

- These methods all use a Newton-like update of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k B_k^{-1} \nabla f(\mathbf{x}_k),$$

  where

  $\Rightarrow$ $\alpha_k > 0$ is the usual linesearch parameter
  $\Rightarrow$ B_k is some approximation to the Hessian matrix.

- This can be written, in general, as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k,$$

where

$$\mathbf{d}_k = -\alpha_k B_k^{-1} \nabla f(\mathbf{x}_k) \tag{4}$$

is the descent direction.

- Note that:

  ⇒ When $B_k = I$, we have the gradient descent method, and when $\alpha_k = 1$, steepest decent.
  ⇒ When $B_k = \nabla^2 f(\mathbf{x}_k)$, $\alpha_k = 1$, we have Newton's method.
  ⇒ When B_k is an approximation of the Hessian, we obtain a quasi-Newton method.

- We now consider the construction of appropriate approximation expressions for B_k. There are two commonly used formulations,

  ⇒ the Broyden-Fletcher-Goldfarb-Shanno (BFGS) and
  ⇒ the Davidon-Fletcher-Powell (DFP) methods.

- Both are what are called low-rank updates,

$$B_{k+1} = B_k + B_k^{\mathrm{u}} \,,$$

that iteratively build up an approximation B_k of the Hessian, starting (usually) from the initial guess $B_0 = I$, the steepest gradient descent direction, and using an update matrix of the form

$$B_k^{\mathrm{u}} = a\mathbf{u}\mathbf{u}^{\mathrm{T}} + b\mathbf{v}\mathbf{v}^{\mathrm{T}},$$

where a and b are scalars, and $\mathbf{u}$ and $\mathbf{v}$ are vectors that satisfy a secant condition.

- It can then be rigorously proven that the resulting B_k is indeed a good, symmetric, positive-definite matrix that ensures that $\mathbf{d}_k$ is a descent direction.

- The BFGS update is given by

$$B_{k+1} = B_k - \frac{B_k \mathbf{s}_k \mathbf{s}_k^{\mathrm{T}} B_k}{\mathbf{s}_k^{\mathrm{T}} B_k \mathbf{s}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^{\mathrm{T}}}{\mathbf{y}_k^T \mathbf{s}_k}, \tag{5}$$

where we have defined

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k, \qquad \mathbf{y}_k = \nabla f_{k+1} - \nabla f_k.$$

- The DFP update is given by

$$B_{k+1} = (I - \gamma_k \mathbf{y}_k \mathbf{s}_k^{\mathrm{T}}) B_k (I - \gamma_k \mathbf{s}_k \mathbf{y}_k^{\mathrm{T}}) + \gamma_k \mathbf{y}_k \mathbf{y}_k^{\mathrm{T}}, \tag{6}$$

where

$$\gamma_k = \frac{1}{\mathbf{y}_k^{\mathrm{T}} \mathbf{s}_k}.$$

- For the implementations, it is better to use directly the inverse,

$$H_k = B_k^{-1},$$

since it enables a direct computation of the search direction with a simple matrix-vector multiplication.

$\Rightarrow$ The Sherman-Morrison-Woodbury formula gives

$$H_{k+1}^{\mathrm{BFGS}} = (I - \gamma_k \mathbf{s}_k \mathbf{y}_k^{\mathrm{T}}) H_k (I - \gamma_k \mathbf{y}_k \mathbf{s}_k^{\mathrm{T}}) + \gamma_k \mathbf{s}_k \mathbf{s}_k^{\mathrm{T}}, \tag{7}$$

and

$$H_{k+1}^{\mathrm{DFP}} = H_k - \frac{H_k \mathbf{y}_k \mathbf{y}_k^{\mathrm{T}} H_k}{\mathbf{y}_k^{\mathrm{T}} H_k \mathbf{y}_k} + \frac{\mathbf{s}_k \mathbf{s}_k^{\mathrm{T}}}{\mathbf{y}_k^{\mathrm{T}} \mathbf{s}_k}, \tag{8}$$

for the BFGS and DFP methods respectively.

- **Algorithm:** BFGS and DFP Quasi-Newton with Line-search

If $f : \mathbb{R}^n \to \mathbb{R}$ is a differentiable function and $\mathbf{x}_0 \in \mathbb{R}^n$ is an initial guess, then the following Quasi-Newton algorithm computes $\mathbf{x}_* = \text{argmin}_{\mathbf{x}} f(\mathbf{x})$.

1: $k = 0$, initial guess $\mathbf{x}_0$, $f_0 = f(\mathbf{x}_0)$, $\nabla f_0 = \nabla f(\mathbf{x}_0)$, $H_0 = I$ ▷ Initialize
2: $\mathbf{p}_0 = -H_0 \nabla f_0$ ▷ Initial direction is steepest descent
3: **while** not converged **do** ▷ Convergence criteria
4: Compute α_k to minimize $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$ ▷ Linesearch
5: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$ ▷ Update solution point
6: Evaluate $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, $\mathbf{y}_k = \nabla f_{k+1} - \nabla f_k$
7: Compute Hessian approx. H_{k+1} ▷ BFGS or DFP
8: $\mathbf{p}_{k+1} = -H_{k+1} \nabla f_{k+1}$ ▷ Next search direction
9: $k = k + 1$
10: **end while**
11: $\mathbf{x}_* = \mathbf{x}_k$ ▷ Output converged solution

- **Linesearch:** As for the Conjugate Gradient method, Brent's method is recommended. An alternative is to use the *strong Wolfe conditions*, that are implemented in Python's `scipy.optimize.line_search`,

and Octave/MATLAB function `fminunc`. This approach is a combination of bracketing, followed by local polynomial—quadratic or cubic—interpolation. Note that this requires two constants, c_1 and c_2, that are hardwired into the routines.

- **Convergence:** The quasi-Newton methods have superlinear convergence, better than Conjugate Gradient (linear), but not as good as pure Newton (quadratic).

- **Initialization:** The initial guess for the Hessain, H_0, can be obtained by a finite-difference approximation at $\mathbf{x}_0$, but the identity matrix is most often used. The initial guess based on the identity can be modified to rescale the variables. Anyway, rescaling is highly recommended before using a quasi-Newton method—see below.

- **Choice of Method:** Note that the Hessian updates, (7) or (8), require matrix-matrix multiplications that make the overall complexity of order n^2. This is why, for large n, conjugate gradient is recommended. As far as the choice between the BFGS and DFP updates

goes, one clearly sees that they are duals (equivalents), but DFP involves divisions in which the denominator can become small and hinder convergence. In addition, BFGS is self-correcting in the sense that bad Hessian approximations that slow down the convergence are automatically corrected after a few iterations. This supposes that a good linesearch algorithm is used. For all these reasons, BFGS is the most widely-used approach.

- **Limited-Memory Quasi-Newton Methods:** For very large-scale optimization problems that can often have $\mathcal{O}(10^6)$ to $\mathcal{O}(10^9)$ variables—we need to reduce as far as possible both the computation time and the memory requirements of the optimization codes. The Hessian approximations seen above are usually dense and can imply excessive cost. For this reason, limited-memory variants are often used, in particular the L-BFGS method.

OPTIMIZATION - STOCHASTIC GRADIENT

Stochastic Gradient Method

- Optimization for large-scale machine learning is almost exclusively done today by stochastic methods.

- These methods choose a random point, or collection of points, and iteratively perform a low-dimensional, gradient-based minimization of the cost function, which is usually the expectation of a suitably defined loss function, over this small collection—a recent, comprehensive review can be found in [3].

- The stochastic gradient (SG) method is an special implementation of gradient descent, already seen above.

  $\Rightarrow$ In machine learning, and in particular neural nets—see ML lectures—it has become the most widely-used method to compute the optimal weights, $\mathbf{w}$, of the neural net that minimize a given loss function $L(\mathbf{w})$ by solving $\nabla_{\mathbf{w}} L = 0$.

SG for ML

- In the machine learning context, suppose we have a training set of N data points with their labels

$$\{y_1, \ldots, y_N\}.$$

  $\Rightarrow$ Seek a point x where the cost function—usually a mismatch function—-attains its minimal value.

  $\Rightarrow$ Gradient descent uses all samples at once,

$$x_{k+1} = x_k - \alpha_k \frac{1}{N} \sum_{i=1}^{N} \nabla f(x_k; y_i),$$

  $\Rightarrow$ Stochastic gradient (descent) uses just **one sample** at a time,

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k; y_{i_k}), \tag{9}$$

  where i_k is a randomly sampled index from the set $\{1, 2, \ldots, N\}$.

⇒ In-between the two extremes, we have mini-batch SGD,

$$x_{k+1} = x_k - \alpha_k \frac{1}{|B_k|} \sum_{i \in B_k} \nabla f(x_k; y_{i_k}),$$

where the batch size, the number of elements in the set B_k, is $b \ll N$.

⇒ It is usually recommended to take $b = 32$.

Stochastic Gradient: theory

- Taking expectation of the SG formula (9) with constant α, we find

$$\begin{aligned}\mathrm{E}\left[x_{k+1}\right] &= \mathrm{E}\left[x_k\right] - \alpha \mathrm{E}\left[f\left(x_k; y_{i_k}\right)\right] \\ &= \mathrm{E}\left[x_k\right] - \alpha \frac{1}{N} \sum_{i=1}^{N} \nabla f(x_k; y_i)\end{aligned}$$

  where we have used the definition of mathematical expectation

- Since we sample from a uniform distribution,

$$i_k \sim \mathcal{U}\{1, 2, \ldots, N\},$$

  we have a constant probability of $1/N$ for every choice of the random index.

  $\Rightarrow$ Thus, we obtain an unbiased estimator of the true gradient from just a single point.

$\Rightarrow$ It can then be shown, under some reasonable conditions [1] on the convexity, that the SG method converges in expectation—or "on average"—to the true point of minimal argument, with a sublinear rate of convergence.

$\Rightarrow$ This convergence is **independent of N,** hence its attractiveness for big data problems.

Stochastic Gradient: justification

- We can get a fairly good estimate of the gradient by looking at just a few sample points.

- For complicated functions, evaluating precise gradients using large datasets is often a waste of time, since the algorithm will have to recompute the gradient again anyway at the next step.

- It is often a better use of computer time to have a rough, noisy estimate and to move rapidly through parameter space.

  $\Rightarrow$ This is the rationale behind SG.

- As a result, SG is often less prone to getting stuck in shallow local minima, because it adds a certain amount of "noise" to the minimization process, just as is done in global optimization methods, such as

  $\Rightarrow$ Genetic Algorithms

$\Rightarrow$ Simulated Annealing.

- Consequently SG has become the most widely-used approach in the machine learning community for fitting models such as neural networks and deep belief networks with non-convex objectives.

Stochastic Gradient: algorithm

The algorithm is particularly simple:

Stochastic Gradient with Fixed Stepsize
Set $x^0 = 0,\ \alpha > 0$
for $k = 0, 1, \ldots, K-1$
    sample uniformly $j \in \{1, \ldots, n\}$
    update $x^{k+1} = x^k - \alpha \nabla f_j(x^k)$
output x^K

- This basic algorithm has undergone many improvements recently. Some of these are discussed below.

Stochastic Gradient: convergence

- Unlike gradient descent, which by definition reduces the function value at each iteration, stochastic gradient does not guarantee a decrease at each step, and the function value can indeed increase—see the loss function curves in Example below.

- That is why it is more exact to name the method "stochastic gradient", and not "stochastic gradient descent."

- However, by making a Taylor expansion of f and assuming reasonable Lipschitz bounds on the gradient ∇f, we can conclude that the method

  $\Rightarrow$ converges in expectation [1],
  $\Rightarrow$ at a rate that depends on α and
  $\Rightarrow$ on the empirical variance of ∇f.

Stochastic Gradient: paractical algorithms

- As mentioned in the previous section, stochastic gradient is a topic of active research, and there are numerous improvements of the basic algorithm.

- These improvements mainly concern the choice of the learning rate, or step size, and the use of previous search directions.

- The main stochastic gradient algorithms in use today will now be very briefly presented.

SGDM Stochastic Gradient Descent with Momentum, also known as the "heavy ball" method, since it is like pushing a ball down a hill, exploiting its previous momentum, uses an additional term to compute the

next step size that takes into account the recent history,

$$x_{k+1} = x_k - \eta z_k,$$
$$z_k = \nabla f(x_k) + \beta z_{k-1},$$

where η is the learning rate and β is the decay factor. Unrolling this recurrence, we can see how the past history is taken into account:

$$\begin{aligned}
x_1 &= x_0 - \eta z_0, \quad z_0 = \nabla f(x_0), \\
x_2 &= x_1 - \eta z_1, \quad z_1 = \nabla f(x_1) + \beta \nabla f(x_0), \\
x_3 &= x_2 - \eta z_2, \quad z_2 = \nabla f(x_2) + \beta \left(\nabla f(x_1) + \beta \nabla f(x_0)\right), \\
&\vdots \qquad\qquad \vdots \\
x_k &= x_{k-1} - \eta z_{k-1}, \quad z_{k-1} = \nabla f_{k-1} + \beta \left(\nabla f_{k-2} + \beta \nabla f_{k-3} + \cdots + \beta^{k-2} \nabla f_0\right),
\end{aligned}$$

where $\nabla f_k = \nabla f(x_k)$. This update, usually performed with $\beta = 0.9$, is remarkably robust and can give very good performance.

ADAGRAD Adaptive Gradient uses a varying learning rate, η, that increases when a particular weight is rarely updated.

RMSprop Root Mean Square Propagation corrects ADAGRAD with a discount factor to avoid updates that are too small.

ADAM Adaptive Moment adds an exponentially decreasing average to the RMSprop learning rate. It behaves like a heavy ball with friction, and relies on two factors, β_1 and β_2, that weight the estimates of the first and second moments of the gradient, respectively. They have default values of

$$\beta_1 = 0.9, \quad \beta_2 = 0.999.$$

Others Many recent improvements have been made to the above methods. Please consult https://ruder.io/optimizing-gradient-descent/ and follow the various blog updates as well as those on arXiv.

OPTIMIZATION and ML

Optimization for ML

- Most machine learning algorithms (see ML lectures) come down to optimizing a cost function that can be expressed as an average over the training examples.

- The *loss function* measures how well (or how badly) the learning system performs on each example—it is thus a measure of the error due to the approximation that has been learned from the data.

- The *cost function*, sometimes called the *empirical risk function*, is then the average of the loss function values over all training examples, possibly augmented with some control or regularization terms.

- As just discussed, *stochastic gradient methods* update the learning system on the basis of the loss function measured for a *single* example. We saw that this approach works because the averaged effect of these updates is the same.

⇒ Although the convergence is much more noisy, the elimination of the constant n in the computing cost can be a huge advantage for large-scale problems.

- Thus, the majority of machine learning methods are expressed as optimization problems of the form

$$\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f(x_i), y_i\right),$$

where

⇒ f is the *model*, belonging to a space of possible models, $\mathcal{F}$.

⇒ $\mathcal{L}$ is a *loss function*, such as found in linear regression, support vector machines, neural networks, k-means, etc.

⇒ (x_i, y_i), $i = 1, \ldots, n$, are the *training samples*—input, output pairs—over which the optimal value of f will be learned.

- We can then define the *empirical risk* as

$$R_n(f) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x_i), y_i),$$

  where

  $\Rightarrow$ $\mathcal{L}(f(x), y)$ is the loss function that measures the cost (error) of predicting $f(x)$ if the actual value is y.

- The optimization problem is then to find

$$f_* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \, R_n(f).$$

- The function f will usually depend on parameters, $\boldsymbol{\theta}$, or weights, $\mathbf{w}$. In this case the minimization will be, for a fixed type of f over $\boldsymbol{\theta}$ or $\mathbf{w}$.

Optimization for ML - loss functions

The loss functions commonly used are:

- Quadratic, or L2 loss,

$$\mathcal{L}(y, \hat{y}) = \|y - \hat{y}\|^2,$$

where $\hat{y} = f(x)$ is the approximation of y and $R(f)$ is called the *mean-squared error* (MSE),

- Absolute, or L1 loss,

$$\mathcal{L}(y, \hat{y}) = |y - \hat{y}|$$

and $R(f)$ is called the *mean absolute error* (MAE),

- Relative, or logistic loss,

$$\begin{aligned}\mathcal{L}(y, \hat{y}) &= \max\left[\frac{\hat{y}}{y} - 1, \frac{y}{\hat{y}} - 1\right] \\ &= \exp\left(|\log \hat{y} - \log y|\right) - 1.\end{aligned}$$

- Cross-entropy, or logistic loss, used for binary outcomes,

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}).$$

- These will be described in the relevant machine learning methods of the lectures on ML

Examples

- comparison between steepest and stochastic gradient methods—see `01GDvsSGD.ipynb`

- many more examples will be presented when we study PyTorch (see following lectures):

  $\Rightarrow$ function approximation
  $\Rightarrow$ linear regression
  $\Rightarrow$ comparison between LBFGS and ADAM

OPTIMIZATION - PENALIZED and CONSTRAINED

Penalized and Constrained Optimization

- Traditional ML deals with target functions and tasks that are NOT linked to physical principles, eg. conservation laws, symmetries, etc.

- Scientific ML seeks solutions that respect these principles, and they must somehow be included in the target function

- In order to include physics and phyics (see Advanced Course) and physical principles in an objective function, there are 2 approaches:

  1. penalizations
  2. constraints

- The general cost function will then have the form

$$J(\mathbf{w}) = J_0(\mathbf{w}) + \alpha R(\mathbf{w}) + \ddot{\lambda}^{\mathrm{T}} C(\mathbf{w}),$$

where

⇒ $\mathbf{w}$ is the vector of optimization parameters (weights, for example in ML)
⇒ J_0 is the ML loss function term
⇒ α is a penalization coefficient
⇒ λ is a (vector) Lagrange multiplier

Penalized and Constrained Optimization - Differences

- **Constrained optimization:** the goal is to find the maximum or minimum of a function while satisfying certain constraints.

  $\Rightarrow$ Constraints define the allowable region in which the optimization process takes place.

  $\Rightarrow$ The problem can be mathematically formulated as:

$$(\mathrm{CO}) \begin{cases} \text{minimize} & f(\mathbf{x}) \\ \text{subject to} & g_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \ldots, m, \\ \text{and} & h_j(\mathbf{x}) = 0, \quad j = 1, 2, \ldots, p. \end{cases}$$

  where

  $\rightarrow$ $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is the function to minimize/maximize

  $\rightarrow$ $g_i(\mathbf{x})$ are the inequality constraints

  $\rightarrow$ $j(\mathbf{x})$ are the equality constraints

- **Penalized optimization:** involves adding a penalty

term to the objective function to encourage certain properties in the solution.

⇒ Mathematically:

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) + \alpha g(\mathbf{x}), \tag{10}$$

where

→ $\alpha > 0$ is a constant penalty coefficient

→ $g(\mathbf{x})$ is a positive-valued function

⇒ often used in machine learning and statistical modeling to prevent overfitting and control the complexity of the model

⇒ L1 and L2 regularization are common examples

⇒ also used in PINN (physics informed neural networks—see PINN Lecture) to enforce respect of (P)DEs and initial or boundary conditions

- **Conclusions**:

⇒ Both constrained optimization and penalized optimization are powerful techniques with different applications.

$\Rightarrow$ Constrained optimization deals with explicit constraints on variables, while

$\Rightarrow$ penalized optimization introduces penalties to achieve certain desired properties in the solution.

Lagrange Multipliers for Constrained Optimization

- Lagrange multipliers are a mathematical technique used to solve constrained optimization problems

  $\Rightarrow$ by incorporating the constraints directly into the objective function

  $\Rightarrow$ through the introduction of Lagrange multipliers.

- This method transforms the constrained optimization problem into an unconstrained optimization problem, which can then be solved using standard optimization techniques.

- **Idea**: construct a new function called the Lagrangian by adding the product of the Lagrange multipliers and the constraints to the original objective function.

  $\Rightarrow$ The Lagrange multipliers act as weights that balance the impact of the constraints on the objective.

Lagrange Multipliers - Procedure

1. **Formulate the Lagrangian**:

   (a) Given the original optimization problem:

$$(\mathrm{CO}) \begin{cases} \text{maximize or minimize} & f(\mathbf{x}) \\ \text{subject to} & g_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \ldots \\ \text{and} & h_j(\mathbf{x}) = 0, \quad j = 1, 2, .. \end{cases}$$

   (b) The Lagrangian is defined as:

$$L(x, \lambda, \mu) = f(x) + \sum(\lambda_i * g_i(x)) + \sum(\mu_j * h_j(x))$$

   where λ_i and μ_j are the Lagrange multipliers associated with the inequality and equality constraints, respectively.

2. **Solve the Unconstrained Problem**:

(a) Find x and Lagrange multipliers λ, μ that maximize or minimize the Lagrangian:

$$\text{maximize or minimize } L(x, \lambda, \mu)$$

(b) The values of x, λ, and μ that satisfy the KKT (Karush-Kuhn-Tucker) conditions are the solutions to the original constrained optimization problem.

Lagrange Multipliers - KKT Conditions

- The Karush-Kuhn-Tucker (KKT) conditions are a set of necessary conditions that must be satisfied by the optimal solution of a constrained optimization problem.
- These conditions extend the Lagrange multiplier method to handle both inequality and equality constraints.
- The KKT conditions ensure that the solution is not only optimal with respect to the objective function, but also satisfies the given constraints.

1. **Stationarity Condition**:

$$\nabla f(x_*) + \sum(\lambda_i \nabla g_i(x_*)) + \sum(\mu_j \nabla h_j(x_*)) = 0$$

2. **Primal Feasibility**:

$$g_i(x_*) \leq 0,$$

for $i = 1, 2, ..., m$ and

$$h_j(x_*) = 0,$$

for $j = 1, 2, ..., p$

3. **Dual Feasibility**:

$$\lambda_i \geq 0,$$

for $i = 1, 2, ..., m$

4. **Complementary Slackness**:

$$\lambda_i g_i(x) = 0,$$

for $i = 1, 2, ..., m$. This implies that either $\lambda_i = 0$ or $g_i(x) = 0$.

- These conditions collectively ensure that

$\Rightarrow$ the optimal solution x_* satisfies the constraints while

$\Rightarrow$ also making the objective function as large (in the case of maximization)
$\Rightarrow$ or as small (in the case of minimization) as possible.

- **Notes**:

1. The first condition is the gradient of the Lagrangian (objective function plus constraints weighted by Lagrange multipliers) being zero, which indicates a stationary point.
2. The second condition ensures that the constraints are satisfied by the optimal solution.
3. The third condition states that the Lagrange multipliers associated with the inequality constraints must be non-negative.
4. The fourth condition expresses the complementary relationship between the Lagrange multipliers and the constraint functions. If a constraint is not active (i.e., its corresponding $g_i(x_*) < 0$), then its Lagrange multiplier must be zero. If a constraint is active (i.e., its corresponding $g_i(x_*) = 0$), then its Lagrange multiplier can be positive or zero.

Remark 1. It's important to note that the KKT conditions

are necessary conditions for optimality, but they are not always sufficient.

Satisfying the KKT conditions is a strong indication that a point is optimal, but further analysis might be needed in some cases.

Keep in mind that while the KKT conditions might appear complex, optimization libraries in Octave/Matlab/NumPy/SciPy handle them automatically when solving constrained optimization problems—see Examples.

Equality Constraints Only

- In (P)DE constrained optimization that we will encounter in SciML, there is often onoy an equality constraint.

- When there are only equality constraints in a constrained optimization problem, the Karush-Kuhn-Tucker (KKT) conditions simplify a bit.

- Let's consider the case when there are only equality constraints. The general problem is:

$$(\text{COE}) \begin{cases} \text{maximize or minimize} & f(\mathbf{x}) \\ \text{subject to} & h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots \end{cases}$$

- In this case, the KKT conditions are as follows:

1. **Stationarity Condition**:

$$\nabla f(x_*) + \sum (\mu_j \nabla h_j(x_*)) = 0$$

2. **Primal Feasibility**:

$$h_j(x_*) = 0,$$

for $j = 1, 2, ..., p$

3. **Dual Feasibility**: μ_j (Lagrange multipliers associated with equality constraints) are unrestricted in sign.

4. **Complementary Slackness**:

$$\mu_j h_j(x) = 0,$$

for $j = 1, 2, ..., p$. This implies that either $\mu_j = 0$ or $h_j(x) = 0$.

- In this simplified scenario with only equality constraints, the complementary slackness condition means that either the Lagrange multiplier associated with an equality constraint is zero, or the constraint itself is zero.

- It is important to note that in this case, the dual feasibility condition is less restrictive because the Lagrange

multipliers associated with equality constraints are unrestricted in sign. This is in contrast to the case with inequality constraints, where the Lagrange multipliers associated with inequalities must be non-negative.

- These conditions still ensure that the optimal solution x* satisfies the equality constraints while optimizing the objective function.

- As mentioned before, optimization libraries like SciPy handle the KKT conditions automatically, so you don't need to manually implement them when solving optimization problems.

Example

Let's consider an example where we want to minimize a simple quadratic function subject to an equality constraint.

- We will formulate the problem and then also solve it using Python, while also discussing the KKT conditions for this scenario.

- **Problem**:

  $\Rightarrow$ Minimize: $f(x) = x^2 + y^2$
  $\Rightarrow$ Subject to:$x + y = 1$

- Step 1: **Formulate the Lagrangian**:

  $\Rightarrow$ The Lagrangian is defined as:

$$L(x, \mu) = f(x) + \mu h(x) = x^2 + y^2 + \mu(x + y - 1)$$

- Step 2: **Calculate the Stationarity Condition**:

⇒ Taking the gradient of the Lagrangian with respect to x and setting it equal to zero:

$$\nabla L(x, \mu) = [2x + \mu, 2y + \mu] = [0, 0]$$

⇒ From the first component of the gradient, we get: $2x + \mu = 0$ and thus $x = -\mu/2$
⇒ From the second component of the gradient, we get: $2y + \mu = 0$ and thus $y = -\mu/2$

- Step 3: **Primal Feasibility**:

⇒ Given the equality constraint: $x + y = 1$
⇒ Substitute the expressions for x and y from the stationarity condition: $(-\mu/2) + (-\mu/2) = 1$, so $-\mu = 1$, and thus $\mu = -1$

- Step 4: **Dual Feasibility**:

⇒ $\mu = -1$ (which is unrestricted in sign, satisfying the dual feasibility condition).

- Step 5: **Complementary Slackness**:

$$\mu * (x + y - 1) = -1 * (x + y - 1) = 0$$

This implies that either $\mu = 0$ or $x + y - 1 = 0$.

- Step 6: Calculate the **Optimal Solution** and Value:

  $\Rightarrow$ Substitute $\mu = -1$ into the expressions for x and y :

$$x = -(-1)/2 = 0.5, \quad y = -(-1)/2 = 0.5$$

- Calculate the optimal value of the objective function:

$$f(x, y) = x^2 + y^2 = 0.5^2 + 0.5^2 = 0.5$$

- **Conclusion**: The optimal solution is

$$x_* = 0.5, \quad y_* = 0.5,$$

  and the optimal value of the objective function is

$$f(x, y) = 0.5.$$

- In this example, we explicitly calculated the Lagrangian, the stationarity condition, primal and dual feasibility, and complementary slackness.

- The solution process demonstrates how the Lagrange multiplier (-1) influences the optimal solution and confirms the KKT conditions for this specific problem.

OPTIMIZATION - PDE CONSTRAINED

PDE-Constrained Optimization

- In inverse problems, DA in particular, we are confronted with a special type of constrained optimization, PDE-constrained optimization

  $\Rightarrow$ this is closely associated with an adjoint approach—see Basic Course 11_DA.adj.pdf

  $\Rightarrow$ others (not dealt with here): linear programming (simplex method), quadratic programming, MINLP, etc.

OPTIMIZATION - MULTI-OBJECTIVE

Multiobjective Optimization

- In SciML, we are often confronted with multi-term objective functions that minimize

  $\Rightarrow$ the training error of the ML component
  $\Rightarrow$ the approximation error of the underlyig physical constraints

- These are notoriously difficult optimization problems, and there is no one-size-fits-all approach to determining optimal weights because it often depends on the specific problem, domain expertise, and the goals of optimization.

  $\Rightarrow$ The choice of method should align with your understanding of the problem and the preferences of stakeholders.
  $\Rightarrow$ Additionally, it may be beneficial to consider multiple weight combinations and their impact on the Pareto front to make well-informed decisions.

- Determining the optimal weights for the weighted sum approach in multiobjective optimization is a crucial step, and there are systematic methods to help you find suitable weights. The choice of weights directly impacts the trade-off between the objectives. Here are some systematic approaches for determining optimal weights:

  1. **A Priori Preference Information:** If you have prior knowledge about the relative importance of the objectives, you can use that information to assign weights. For example, if L1 represents cost and L2 represents performance, you might assign a higher weight to cost if cost reduction is more critical. This approach relies on expert judgment.
  2. **Analytical Hierarchy Process (AHP):** AHP is a decision-making method that quantifies preferences using pairwise comparisons. You can ask decision-makers to compare the importance of objectives relative to each other. AHP then computes weights that reflect these preferences. There are Python libraries like 'pyanp' that can help you implement AHP.
  3. **Data-Driven Approaches:** If you have access to historical data, you can use machine learning or data-driven methods to learn the optimal weights.

For example, you can frame the determination of weights as a supervised learning problem, where the inputs are features of the data, and the output is a weight vector that minimizes prediction error.

4. **Multi-Criteria Decision Analysis (MCDA):** MCDA methods like the PROMETHEE (Preference Ranking Organization Method for Enrichment Evaluations) or ELECTRE (Elimination Et Choix Traduisant la Realite) can help you determine weights based on decision-maker preferences. These methods consider both the value and the shape of the preference function.
5. **Pareto-Based Methods:** If you don't have explicit preferences but want to explore the Pareto front, you can use multi-objective optimization algorithms like NSGA-II or MOEA/D. These algorithms provide a set of solutions on the Pareto front, allowing you to assess trade-offs and make informed decisions.
6. **Sensitivity Analysis:** Perform a sensitivity analysis by varying the weights systematically and observing how the Pareto front changes. This can help you understand the impact of different weight combinations on the solutions.

7. **Interactive Methods:** Interactive methods involve decision-makers in the weight determination process. You can use interactive visualizations to allow decision-makers to explore the trade-offs and adjust the weights interactively.
8. **Randomized Search:** If you are uncertain about the relative importance of objectives, you can perform a randomized search over a range of weights and analyze the resulting Pareto fronts to identify promising trade-offs.

Bibliography

References

[1] J. Nocedal, S. Wright. *Numerical Optimization.* Springer, 2006.

[2] M. Asch. *Digital Twins: from Model-Based to Data-Driven.* SIAM, 2022.

[3] L. Bottou, F.Curtis, J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60 (2018), pp.223–311.