

SciML - Gradients and Optimization

Mark Asch - IMU/VLP/CSU

2023

Program

1. Optimization:
 - (a) Gradients and Optimization.
 - (b) Automatic Differentiation: Computational Graphs, Backpropagation and Adjoint.
 - (c) Optimization and AD with PyTorch.
2. Physics constrained learning (PCL).
3. Physics Induced Neural Networks (PINN).
4. Operator-based learning.

Recall: Optimization

- Optimization is at the very heart of:
 - ⇒ Machine Learning
 - ⇒ Inverse Problems (including Data Assimilation)
 - ⇒ 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

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

or find \mathbf{x}_* that satisfies

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

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

Recall: Gradients

- Necessary and sufficient conditions for finding a (local) minimum, 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 $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is continuously differentiable and if $\mathbf{p} \in \mathbb{R}^n$, then*

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

where the **gradient vector**

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

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

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

and

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

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*

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

- 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.
- ⇒ 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}_* .
- ⇒ But we can (and do) use this necessary condition to solve the system of n algebraic equations in n unknowns defined by (3).
- ⇒ Then we have to sort out and decide which of the candidate points are indeed **minimizers**.
- ⇒ 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.

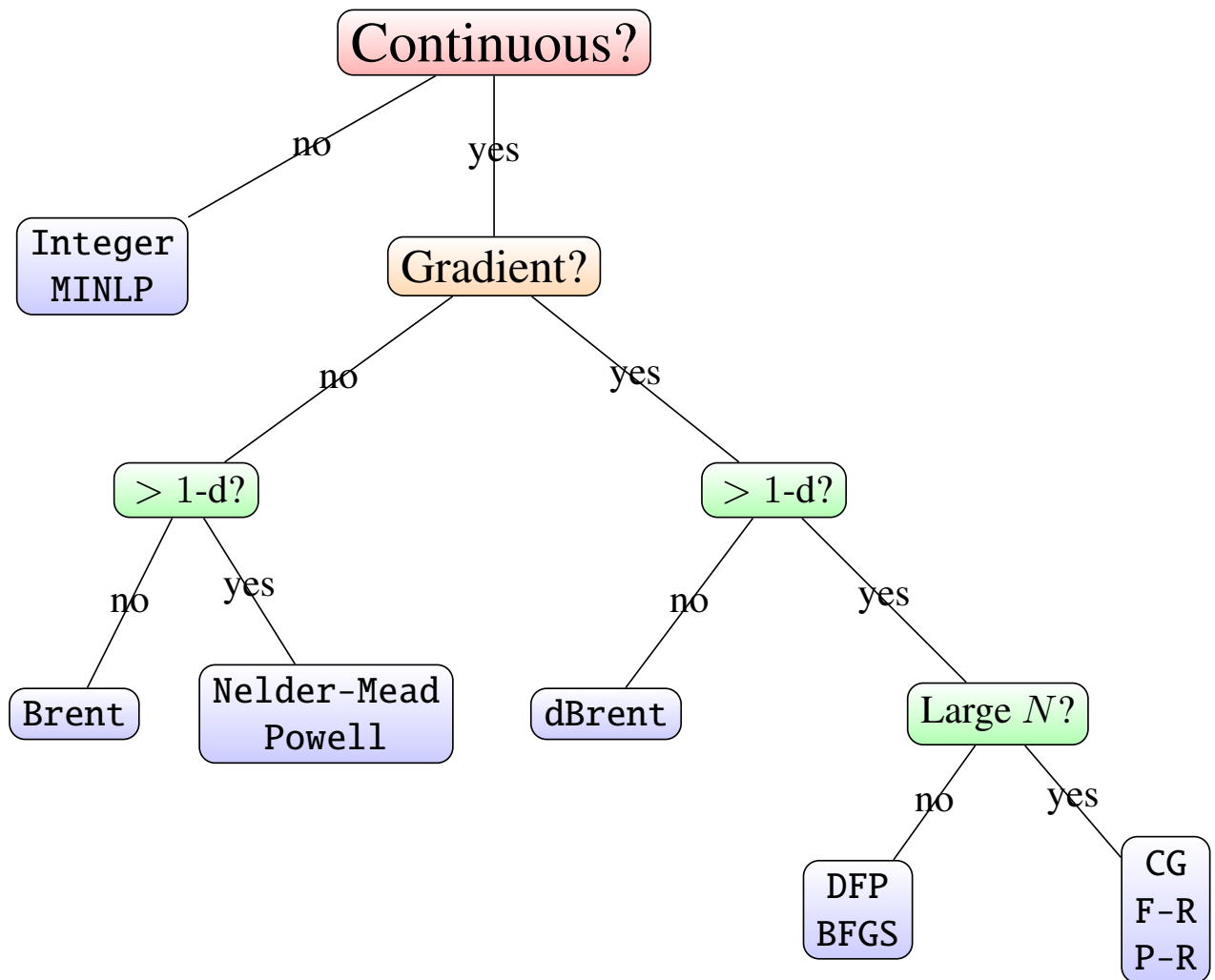
Theorem 3 (Second-Order Sufficient Condition). *Suppose for a point \mathbf{x}_* the function f has first- and second-order derivatives. Suppose also that the gradient of f is zero and that the Hessian is positive definite at \mathbf{x}_* . Then \mathbf{x}_* is a (strict) local minimum of f .*

- 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
 - ⇒ the ML part usually implies a large number of weights
 - ⇒ 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:
 - ⇒ SGD (stochastic gradient descent) and its variants, notably ADAM
 - ⇒ LBFGS (quasi-Newton) when higher order is needed and feasible

Recall: Gradient Descent

- Most minimization algorithms have the following basic structure:

```
while  $\mathbf{x}^{(k)}$  is not a minimum
    calculate the step direction  $\mathbf{p}^{(k)}$ 
        with  $\|\mathbf{p}^{(k)}\| = 1$ 
    calculate the step size  $\alpha^{(k)}$ 
    update  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$ 
     $k = k + 1$ 
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 $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is a differentiable function and $\mathbf{x}_0 \in \mathbb{R}^n$ is an initial guess, then the following SDG algorithm computes $\mathbf{x}_* = \operatorname{argmin}_{\mathbf{x}} f(\mathbf{x})$.

- 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.

⇒ However, this method is rarely used in practice---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.

⇒ Suppose that

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

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

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

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 $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$

⇒ 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**

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

⇒ 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**:

```
x = x0  
for  $k = 0, 1, 2, \dots$   
    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.

⇒ 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

⇒ $\alpha_k > 0$ is the usual linesearch parameter

⇒ 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

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

is the descent direction.

- Note that:
 - \Rightarrow When $B_k = I$, we have the **gradient descent** method, and when $\alpha_k = 1$, steepest decent.
 - \Rightarrow When $B_k = \nabla^2 f(\mathbf{x}_k)$, $\alpha_k = 1$, we have **Newton's** method.
 - \Rightarrow 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,
 - \Rightarrow the Broyden-Fletcher-Goldfarb-Shanno (**BFGS**) and
 - \Rightarrow the Davidon-Fletcher-Powell (DFP) methods.
- Both are what are called **low-rank updates**,

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

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

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

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

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

where we have defined

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

- The **DFP update** is given by

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

where

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

- 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.

⇒ The **Sherman-Morrison-Woodbury** formula gives

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

and

$$H_{k+1}^{\text{DFP}} = H_k - \frac{H_k \mathbf{y}_k \mathbf{y}_k^T H_k}{\mathbf{y}_k^T H_k \mathbf{y}_k} + \frac{\mathbf{s}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}, \quad (8)$$

for the BFGS and DFP methods respectively.

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

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

- 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 Hessian, 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.

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 a special implementation of **gradient descent**, already seen above.
 - ⇒ 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

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

⇒ Seek a point x where the cost function—usually a mismatch function—attains its minimal value.

⇒ **Gradient descent** uses all samples at once,

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

⇒ **Stochastic gradient** (descent) uses just **one sample** at a time,

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

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

⇒ 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

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

where we have used the definition of mathematical expectation

- Since we sample from a uniform distribution,

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

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

⇒ Thus, we obtain an **unbiased estimator** of the true gradient from just a single point.

- ⇒ 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.
- ⇒ 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.

⇒ 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

⇒ Genetic Algorithms

⇒ 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 $k = 0, 1, \dots, K - 1$

 sample uniformly $j \in \{1, \dots, n\}$

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

Stochastic Gradient: practical 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,

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

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:

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

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 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

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

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) , $i = 1, \dots, n$, 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

⇒ $\mathcal{L}(f(x), y)$ 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

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

- 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,

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

- 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):
 - ⇒ function approximation
 - ⇒ linear regression
 - ⇒ comparison between LBFGS and ADAM

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.