Why is Newton's method not used (anymore) in Machine Learning

Romain Lhotte

DL NLP

tl;dr

We could want to use Newton's method (that you should have encountered in earlier courses and in your Higher school preparatory classes days) to minimise the loss. Afterall, the loss is a function $f : \mathbb{R}^d \to \mathbb{R}$ so we could 'simply' look for the critical points[1] of our function[2] and Newton's method is perfect to find $x \in D_g$ such that $g(x) = 0$ for some[3] functions g.

You probably have not seen Newton's method applied to functions other than when $d = 1$. When $d > 1$, a Hessian matrix pops up. The rest of this document tries to convince you of why without having to admit the correctness of big powerful theorems. This Hessian matrix is the reason why we don't use Newton's method in modern Machine Learning: it requires computing and inverting that Hessian matrix, which is computationally expensive and memory-intensive (it's a matrix so it's quadratic in the number of parameters!!). Unimaginable for models with millions of parameters[4]. Instead, we use Stochastic Gradient Descent (SGD), which doesn't have any quadratic thing going on, making it efficient and suitable for large-scale optimization problems like training deep neural networks.

It's worth noting both SGD and Newton's method face difficulties in finding global minima in non-convex optimization settings, as the landscape may contain many local minima, saddle points, and flat regions. However, there is some folklore theory suggesting that the stochastic nature of SGD allows it to escape local minima more easily than purely deterministic methods like Newton's method. The noise introduced by sampling mini-batches can help the optimization process explore the loss surface and potentially avoid getting stuck in sharp local minima. We could extend Newton's method to use a 'subsampled Hessian approximation' but given the flaws above, I'm not aware of anyone who ever tried.

---

[1] In mathematics, a critical point is the argument of a function where the function derivative is zero.

[2] And this is a big leap, this is not always true, but let's pretend we have a nicely convex loss function so we can say minimum if and only if the derivative is 0

[3] See `https://en.wikipedia.org/wiki/Newton%27s_method#Failure_of_the_method_to_converge_to_the_root`, `https://en.wikipedia.org/wiki/Newton%27s_method#Oscillatory_behavior`, and `https://en.wikipedia.org/wiki/Newton%27s_method#Divergence_even_when_initialization_is_close_to_the_root`for failure modes. But those are not the main reason why we don't use Newton's method (anymore) in Machine Learning. So let's just pretend those issues don't exist.

[4] if you think a square can't hurt and won't blow up numbers, compare the gap between the monthly salary you'll get as an intern, and the same salary squared; and after that realization, realize that the gap is **larger** when we have millions/billions/trillions of parameters ($x \longmapsto x^2$ is convex).

Objective

We want to understand why the Hessian matrix naturally appears when using Newton's Method to find a minimum of a scalar-valued function of d variables. Let's use $d = 3$ to make the arguments simpler to follow:

$$x, y, z \in \mathbb{R}$$

and

$$f(x, y, z) \in \mathbb{R}$$

Step 1: The Gradient and Critical Points

To find a minimum, we look for critical points, which occur where the gradient is zero:

$$\nabla f(x, y, z) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} = \mathbf{0}$$

This can be rewritten as a system of linear equations:

$$\frac{\partial f}{\partial x}(x, y, z) = 0$$
$$\frac{\partial f}{\partial y}(x, y, z) = 0$$
$$\frac{\partial f}{\partial z}(x, y, z) = 0$$

Step 2: Let's revisit Newton's Method in One Variable

Recall Newton's method in one dimension: to solve $f'(x) = 0$, we use the iterative formula:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

This comes from approximating the derivative with its tangent line:

$$f'(x) \approx f'(x_n) + f''(x_n)(x - x_n)$$

and solving for the zero of the linear approximation:

$$f'(x_n) + f''(x_n)(x_{n+1} - x_n) = 0$$

.

Step 3: Generalizing to Multiple Dimensions

In multiple dimensions, we cannot simply apply the 1D rule to each variable separately. We must account for all these interactions simultaneously.
Let's find a step $\Delta\mathbf{x} = [\Delta x, \Delta y, \Delta z]^T$ that takes us from our current point $\mathbf{x}_n$ to a new point $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}$ where the gradient is (approximately/hopefully) zero[5].
Let's make a linear approximation for the first component of the gradient, $\frac{\partial f}{\partial x}$. Its value at the new point is approximately:

$$\frac{\partial f}{\partial x}(\mathbf{x}_{n+1}) \approx \frac{\partial f}{\partial x}(\mathbf{x}_n) + \frac{\partial^2 f}{\partial x^2}(\mathbf{x}_n)\Delta x + \frac{\partial^2 f}{\partial x \partial y}(\mathbf{x}_n)\Delta y + \frac{\partial^2 f}{\partial x \partial z}(\mathbf{x}_n)\Delta z$$

We want this to be zero. This gives us our first linear equation for the unknowns $\Delta x, \Delta y, \Delta z$:

$$\frac{\partial^2 f}{\partial x^2}(\mathbf{x}_n)\Delta x + \frac{\partial^2 f}{\partial x \partial y}(\mathbf{x}_n)\Delta y + \frac{\partial^2 f}{\partial x \partial z}(\mathbf{x}_n)\Delta z = -\frac{\partial f}{\partial x}(\mathbf{x}_n)$$

Step 4: Building the Full Linear System

We do the same for the other two components of the gradient ($\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial z}$), creating a linear approximation for each one and setting it to zero. This gives us a system of three linear equations for our three unknowns:

$$\begin{aligned}
\frac{\partial^2 f}{\partial x^2}\Delta x + \frac{\partial^2 f}{\partial x \partial y}\Delta y + \frac{\partial^2 f}{\partial x \partial z}\Delta z &= -\frac{\partial f}{\partial x} \\
\frac{\partial^2 f}{\partial y \partial x}\Delta x + \frac{\partial^2 f}{\partial y^2}\Delta y + \frac{\partial^2 f}{\partial y \partial z}\Delta z &= -\frac{\partial f}{\partial y} \\
\frac{\partial^2 f}{\partial z \partial x}\Delta x + \frac{\partial^2 f}{\partial z \partial y}\Delta y + \frac{\partial^2 f}{\partial z^2}\Delta z &= -\frac{\partial f}{\partial z}
\end{aligned}$$

(Note: All partial derivatives are evaluated at the current point $\mathbf{x}_n$).

Step 5: The Hessian Matrix Appears

This system of equations can be written beautifully in matrix form. What a coincidence, the matrix of coefficients of our unknowns is exactly the **Hessian matrix** of f.

$$\text{Hess}(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} \end{bmatrix}$$

Our system is therefore:

$$\text{Hess}(f)(\mathbf{x}_n) \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = - \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}$$

[5]that will work if approximating a function by its tangent line is good enough, which usually only kind of is, so we do multiple iterations, it really is the same logic as the 1D logic

Or more compactly:

$$\text{Hess}(f)(\mathbf{x}_n)\Delta\mathbf{x} = -\nabla f(\mathbf{x}_n)$$

Conclusion

How do we solve the matrix equation $A\mathbf{v} = \mathbf{b}$ for the vector $\mathbf{v}$? We use the matrix inverse (if the matrix is invertible, but let's just pretend it is): $\mathbf{v} = A^{-1}\mathbf{b}$. This is the multi-dimensional equivalent of dividing by a number.
To find our update step $\Delta\mathbf{x}$, we do exactly that:

$$\Delta\mathbf{x} = -\left[\text{Hess}(f)(\mathbf{x}_n)\right]^{-1}\nabla f(\mathbf{x}_n)$$

Since our new point is $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}$, we arrive at the final update rule for Newton's method in multiple dimensions:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \left[\text{Hess}(f)(\mathbf{x}_n)\right]^{-1}\nabla f(\mathbf{x}_n)$$

Which is bad, if $d = 10^9$ (1 billion parameter, a joke compared to current LLMs), there are $d^2 = 10^{18}$ things in that matrix. Remember that one gigabyte is one billion bytes, so to just have this matrix exist in your RAM, you would need ... a BILLION GIGABYTES OF RAM. And I'm not even accounting for the fact that a float is usually encoded on 4 bytes. So you would need four BILLION GIGABYTES OF RAM.