Scientific Machine Learning 02
Classifying Handwritten Digits & Gradient Descent

Donghyun Ko

January 4, 2026

Handwritten digit recognition is a perfect first "real" classification task for practice: pixels in, digit out. We will encode targets in the right way (one-hot), train a two-layer ANN, and optimize with *gradient descent* by including its practical workhorse, *stochastic gradient descent (SGD)*.

Contents

1 Why this post? What you'll learn **1**

2 Classifying handwritten digits **2**
2.1 Problem definition 2
2.2 Encoding the outputs 3
2.3 The MNIST dataset 3

3 Learning with Gradient Descent **4**
3.1 Mathematical definition of the classification task 4
3.2 Gradient Descent (GD) 5
3.3 Stochastic Gradient Descent (SGD) 7

1 Why this post? What you'll learn

We turn last time's neuron/ANN basics into a working classifier:

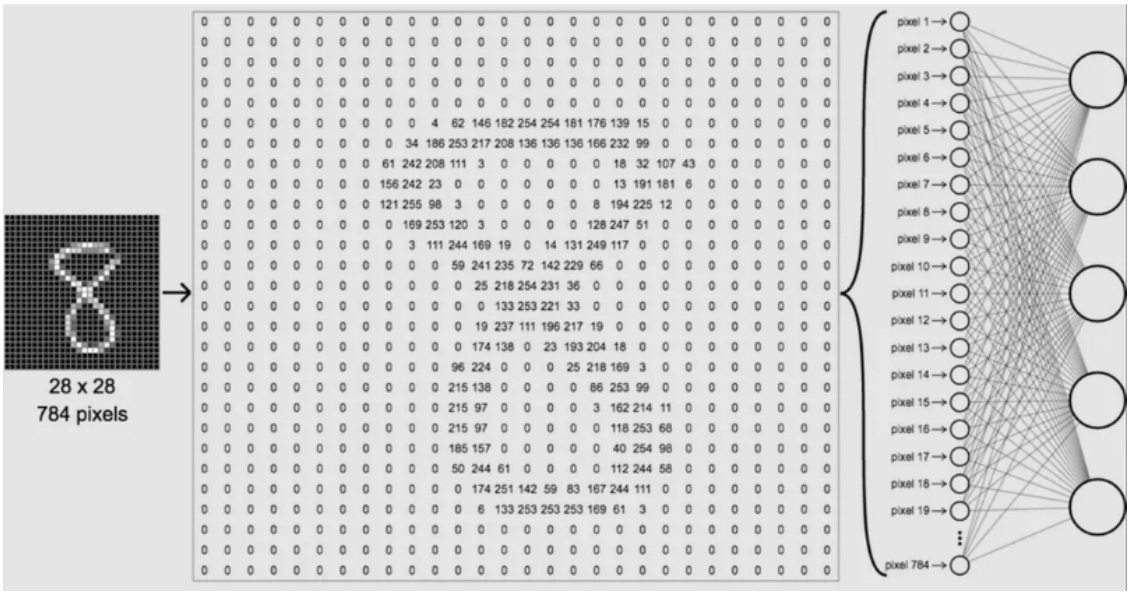
- How to pose handwritten digit recognition as a clean ML problem.
- Why one-hot encoding beats ordinal labels for multi-class digits.
- How quadratic cost and (stochastic) gradient descent actually drive learning.
- What mini-batch, batch size, iterations, and epochs mean in practice.

Learning goals — Move from concept to training loop: define the model and labels, choose a sensible cost, take stable gradient steps, and scale with mini-batches.

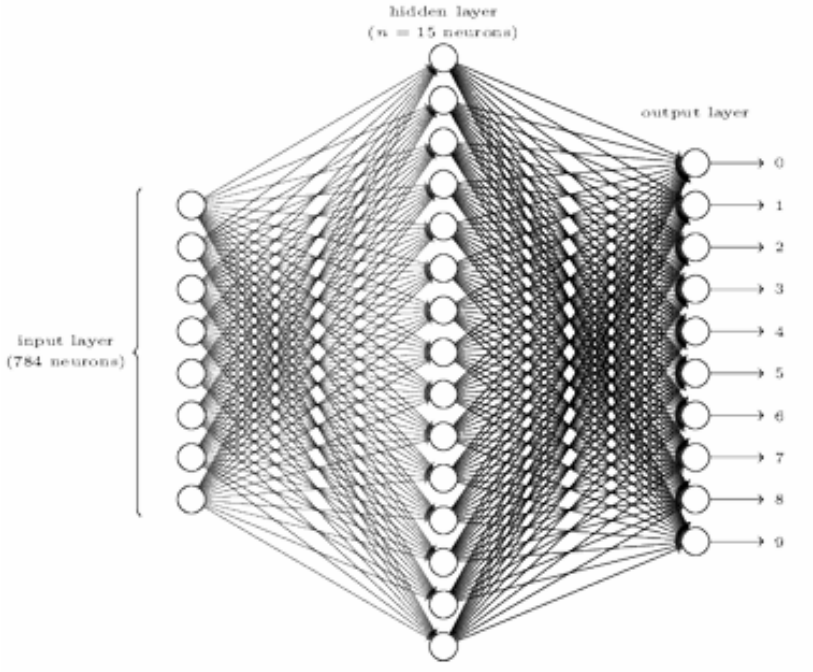
2 Classifying handwritten digits

2.1 Problem definition

We want a network that maps a gray 28×28 image of a digit to its class between 0 and 9 (10 discrete outcomes in total). Flatten each image to a vector $x \in \mathbb{R}^{784}$ (values in $[0, 1]$), feed it to a *two-layer* ANN (one hidden layer), and read the output neuron with the largest activation value as the predicted digit.



Flattened image by 28×28 pixels: 0 is black, 255 is white, and different degrees of grays in between



A simple $784 \to 15 \to 10$ network for MNIST

Architecture (minimal but effective)

- *Input*: 784 nodes (one per pixel).
- *Hidden*: 15 sigmoid units (small, on purpose).
- *Output*: 10 sigmoid (or softmax) units, one per digit 0–9.

How the prediction is picked. Compute all 10 output activations and choose the index with the largest value. Training will push the true class's unit to be highest.

2.2 Encoding the outputs

You could design the output layer in three ways: (i) Use *10 output neurons* (one per class 0-9), (ii) Use *4 binary neurons* (since $2^4 = 16$ patterns), or (iii) Use *1 neuron* to output the digit value (ordinal encoding). **Empirically, the 10-neuron one-hot scheme works best**: each class has its own output unit and incoming weights, giving a clean signal where exactly one unit should fire. In general, categorical labels must be converted to numbers, but the encoding must match the variable type. *Ordinal encoding* (assigning integers like 1,2,3) is appropriate only when a *natural rank order* truly exists; using it on non-ordered classes (like digit identities) creates spurious numeric distances and can degrade performance. *One-hot encoding* avoids this by representing each class as a vector of 0/1s with a single "hot" index, preserving class independence and typically yielding better learning dynamics. Below is an example for 3.

$$\text{One-hot}(3) = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^\top$$

10-output encoding (decimal)	4-output encoding (binary)	Ordinal Encoding	output number
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]	[0, 0, 0, 0]	0	0
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]	[0, 0, 0, 1]	1	1
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]	[0, 0, 1, 0]	2	2
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]	[0, 0, 1, 1]	3	3
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]	[0, 1, 0, 0]	4	4
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]	[0, 1, 0, 1]	5	5
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]	[0, 1, 1, 0]	6	6
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]	[0, 1, 1, 1]	7	7
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]	[1, 0, 0, 0]	8	8
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]	[1, 0, 0, 1]	9	9

Why one-hot works well. Each class has its own output and its own incoming weights, allowing the network to specialize features per digit; only one unit should "fire" at a time, which is a clean training signal.

2.3 The MNIST dataset

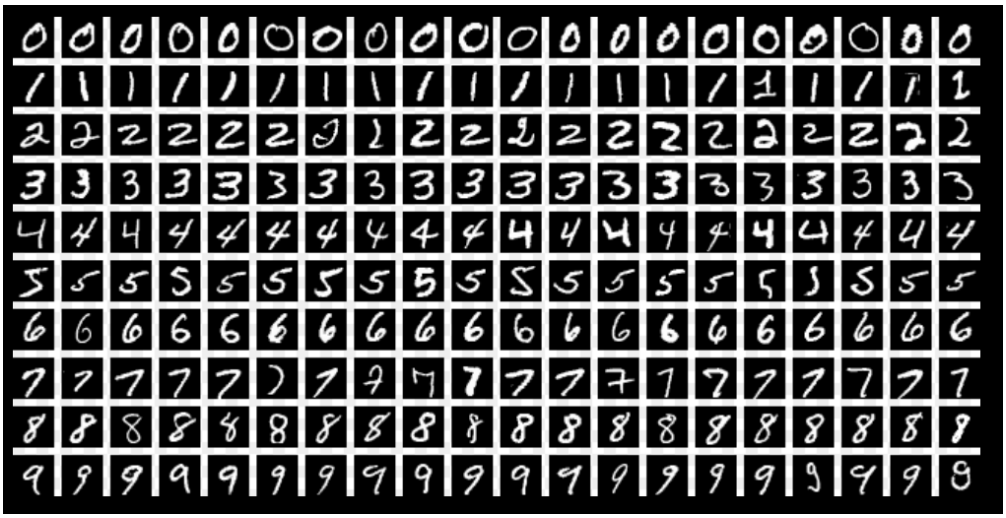
The *MNIST* dataset contains tens of thousands of scanned images of handwritten digits together with their *correct class labels*. The name "MNIST" comes from the fact that it is

a *Modified* subset of two digit datasets collected by *NIST* (the U.S. National Institute of Standards and Technology). MNIST comes in two parts:

- **Training part:** *60,000* grayscale images of size 28×28. These are handwriting samples from *250 writers*: half were U.S. Census Bureau employees and half were high school students.
- **Test part:** *10,000* grayscale images of size 28×28 used strictly for evaluation.

> **Good test design.** The *test* images were collected from a *different set of 250 people* than those who produced the *training* images (with the same Census/High-School split). This separation helps us judge whether a trained ANN can recognize digits written by people it *never saw* during training.

In practice, we train the network on the 60k training images and report performance on the 10k test images. Each image is typically scaled to $[0, 1]$ and flattened to a vector $x \in \mathbb{R}^{784}$ before being fed to the network.



3 Learning with Gradient Descent

3.1 Mathematical definition of the classification task

Let $x \in \mathbb{R}^{784}$ be one training image data (28×28 grayscale digit, flattened). Let $y = y(x) \in \mathbb{R}^{10}$ be its one–hot target: the k-th entry is 1 if the image shows the digit k, and 0 otherwise. Our goal is to find weights W and biases b so that the network output $a = a(x; W, b) \in \mathbb{R}^{10}$ *approximates* $y(x)$ for *all* training inputs. This can be done by optimizing the model parameters with proper cost functions(also called objective function) with a variety of optimization algorithms such as Gradient Descent(GD), Stochastic GD, mini-batch GD, momentum, RMSProp, Adam, and more.

Quadratic (MSE) cost. To quantify mismatch, define the (nonnegative) quadratic cost

$$C(W,b) = \frac{1}{2n}\sum_x \| y(x) - a(x;W,b) \|_2^2 = \frac{1}{2n}\sum_x \| y(x) - a \|^2, \tag{1}$$

where n is the total number of training samples and a is the vector of outputs from the network when x is input (i.e., the activation value). This cost is *smooth* in (W,b) and becomes small when a is close to $y(x)$ on many x. The quadratic cost works perfectly well for understanding the basics of learning in ANNs. In the future, we will explore other cost functions.

Why not "% of correct" as the cost function? (i) The number of images correctly classified is *not* a smooth function of (W,b); (ii) Making small changes to the parameter(W and b) may not cause any change at all in the number of training images classified correctly, making it difficult to figure out how to change W and b to get improved performance, so we get no gradient signal; (iii) The quadratic cost is relatively smooth, letting us compute small, helpful updates that reduce error.

3.2 Gradient Descent (GD)

To minimize a function, we can use calculus to find the minimum analytically if the explicit form of the function is known. Mathematically, we could compute derivatives and then use them to find places where the function reaches an optimum value. However, ANNs have cost functions that depend on so many variables(i.e. weights and biases) in an extremely complicated way. Using calculus to minimize it just won't work! Instead, gradient descent(GD) is one of the most popular algorithms to perform optimization and by far the most common way to optimize ANNs. It is an iterative optimization algorithm used to find the minimum value of a function. Consider a differentiable cost function, $C(\mathbf{v})$ with $\mathbf{v} = [v_1, \ldots, v_p]^\top$ collecting *all* network parameters (weights and biases). If we change $\mathbf{v}$ by a small amount $\Delta\mathbf{v} = (\Delta v_1, \ldots, \Delta v_p)^\top$, first-order Taylor expansion gives us:

$$\Delta C \;\approx\; \frac{\partial C}{\partial v_1}\Delta v_1 + \cdots + \frac{\partial C}{\partial v_p}\Delta v_p \;=\; \nabla C \cdot \Delta\mathbf{v}, \tag{2}$$

where ∇C is a column vector called a gradient such that $\nabla C = \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_p}\right)$. Equation (2) shows that the gradient linearly relates changes in parameters to the change in cost.

Descent direction and update rule To *minimize* the loss C, we need to take a small step Δv in the direction *opposite* to the gradient, since the gradient points along the steepest increase of C by definition. So, we choose:

$$\Delta\mathbf{v} \;=\; -\eta\,\nabla C, \qquad \eta > 0 \tag{3}$$

so that

$$\Delta C \;\approx\; \nabla C \cdot (-\eta \nabla C) \;=\; -\eta\,\|\nabla C\|^2 \;\leq\; 0 \tag{4}$$

The small positive η is *learning rate*. Repeating the move (3) yields the iterative GD update and this guarantees that if we move v in the following way, the function C will always decrease:

$$\mathbf{v} \;\to\; \mathbf{v}' \;=\; \mathbf{v} - \eta\,\nabla C, \tag{5}$$

and we keep doing this over and over until C cannot be decreased anymore ($\Delta C \approx 0$) and it reaches a minimum.

Why is $-\nabla C$ the best *direction* for a fixed step size? Our goal is to pick Δv that reduces C as much as possible. When the step is small, the change in C is well-approximated by the linear term $\Delta C \approx \nabla C \cdot \Delta v$, so minimizing C amounts to making this inner product as negative as possible. Because $\nabla C \cdot \Delta v = \|\nabla C\|\,\|\Delta v\| \cos\theta$, making this inner product very negative means choosing Δv opposite the gradient ($\theta = 180°$), which lowers C the fastest. To show this, we restrict the step to a fixed size $\|\Delta \mathbf{v}\| = \varepsilon$ and want to reduce C as much as possible. Minimizing the linearized change $\Delta C \approx \nabla C \cdot \Delta \mathbf{v}$ under this constraint is a classic Cauchy–Schwarz application:

$$||\nabla C \cdot \Delta \mathbf{v}|| \;\le\; \|\nabla C\|\,\|\Delta \mathbf{v}\| \;=\; \|\nabla C\|\,\varepsilon, \tag{6}$$

where the equality holds if and only if ∇C and Δv are parallel. Thus, the choice that *most decreases* C (most negative inner product) is the step in the direction $-\nabla C$. Equivalently, the step (3) is optimal among all small moves of the same size, with $\eta = \varepsilon/\|\nabla C\|$. So, Gradient Descent can be viewed as a way of taking small steps in the direction which does the most to decrease C.

Geometric intuition. The linearized change is a scaled cosine: $\Delta C \approx \|\nabla C\|\,\varepsilon\,\cos\theta$. For a fixed step length ε, the only way to reduce C as fast as possible is to make the angle θ as large as possible (i.e., 180°), hence *steepest descent* along $-\nabla C$. This is the same conclusion as Cauchy–Schwarz: the bound $||\nabla C \cdot \Delta \mathbf{v}|| \le \|\nabla C\|\,\varepsilon$ is tight exactly when the two vectors are parallel (here, anti-parallel for maximal decrease). Another intuition is that because the step length is fixed, we minimize the linearized change $\Delta C \approx \nabla C \cdot \Delta v = \|\nabla C\|\,\varepsilon \cos\theta$. This is smallest when $\cos\theta = -1$ ($\theta = \pi$), i.e., when Δv points *exactly opposite* to the gradient. Equivalently, the Lagrange multiplier solution of $\min_{\|\Delta v\|=\varepsilon} \nabla C \cdot \Delta v$ yields $\Delta v^* = -\varepsilon\,\nabla C/\|\nabla C\|$, not $+\varepsilon\,\nabla C/\|\nabla C\|$. Choosing $+\nabla C$ would *maximize* the increase in C.

Applying GD to the ANN cost The idea is to use Gradient Descent to find the weights w_k and biases b_l which minimize the cost C (W,b). Applying (5) for individual parameters gives us an update rule such that:

$$w_k \leftarrow w_k - \eta\,\frac{\partial C}{\partial w_k}, \qquad b_\ell \leftarrow b_\ell - \eta\,\frac{\partial C}{\partial b_\ell} \tag{7}$$

For classification with one-hot targets $y(x)$ and network outputs $a(x; W, b) = a$, we use the quadratic (MSE) cost written as an average:

$$C(W,b) \;=\; \frac{1}{2n}\sum_x \Big\|y(x) - a\Big\|^2 \;=\; \frac{1}{n}\sum_x C_x, \tag{8}$$

where the *individual* (per-sample) cost is

$$C_x \;=\; \frac{\left\|y(x)-a\right\|^2}{2} \tag{9}$$

In order to compute the full gradient ∇C, we first calculate ∇C_x for each training sample and average them:

$$\nabla C \;=\; \frac{1}{n}\sum_x \nabla C_x \tag{10}$$

This "full-batch" gradient is exact but can be slow to evaluate when n is large, motivating the stochastic (mini-batch) approximation introduced in the next subsection.

3.3 Stochastic Gradient Descent (SGD)

SGD speeds up learning by estimating the full gradient from a small, randomly chosen set of m training samples—called a *mini-batch*—drawn without replacement within an epoch. By averaging the per-sample gradients over this set, $\frac{1}{m}\sum_{i=1}^{m}\nabla C_{x_i} \approx \nabla C$, we obtain a good, inexpensive approximation to the true full gradient, which accelerates gradient descent and thus learning. Provided the sample size 'm' is large enough, we expect that the average value of the ∇C_{x_i} be roughly equal to the average over all ∇C_x (i.e. the full-batch gradient) and be far cheaper to compute per update, that is:

$$\frac{1}{m}\sum_{i=1}^{m}\nabla C_{x_i} \;\approx\; \frac{1}{n}\sum_x \nabla C_x = \nabla C \tag{11}$$

where the first sum is over the 'm' samples in the mini-batch, and the second sum is over the entire set of training data.

Mini-batch update rules. Using (11), the GD update rules become

$$\begin{aligned} w_k \;\to\; w_k' \;&=\; w_k - \eta\,\frac{\partial C}{\partial w_k} \;=\; w_k - \eta\,\frac{1}{m}\sum_{i=1}^{m}\frac{\partial C_{x_i}}{\partial w_k}, \\ b_\ell \;\to\; b_\ell' \;&=\; b_\ell - \eta\,\frac{\partial C}{\partial b_\ell} \;=\; b_\ell - \eta\,\frac{1}{m}\sum_{i=1}^{m}\frac{\partial C_{x_i}}{\partial b_\ell}, \end{aligned} \tag{12}$$

where the sums are over all the training examples $x_1, x_2, ..., x_m$ in the current mini-batch. After the update, sample a new mini-batch and repeat. When every training example has been used once, we have completed one *epoch*; then we start a new epoch.

Important Terminologies

- **Epoch**: one full pass through the training set (all mini-batches used once). Because a full epoch is usually too large to feed to the ANN at once, we split the data into several smaller *mini-batches* and update after each mini-batch.
- **Batch size** m: the number of samples processed together before a single parameter update (size of one mini-batch).
- **Iteration**: one mini-batch update (one application of (12)); thus, an epoch contains $\frac{n}{m}$ iterations when n is the total number of training samples.
- **Example (MNIST).** With $n = 60{,}000$ and $m = 100$, there are 600 iterations per epoch; For 10 epochs as an example $\Rightarrow$ 6,000 updates total. (If we choose to do 10 epochs, we will go through all the n = 60,000 training images 10 times.)
- **Update frequency.** Weights and biases are updated in *every* batch.

Online / incremental learning. Setting $m = 1$ gives online SGD (update after each single example). Expect large oscillations in both the cost and the parameter values.

Why do we need multiple epochs? One epoch rarely guarantees convergence for a fixed learning rate. With finite training data, SGD needs time to *extract* the signal from noisy mini-batch estimates, so revisiting the same samples over multiple epochs improves convergence. It is also good practice to *adjust the learning rate per epoch* (e.g., decay based on learning curves) to stabilize and accelerate training. Finally, because datasets are limited, multiple epochs let the model make better use of the available information—giving the algorithm time to converge without requiring more (often impractical) data.

Local vs. global minima. Many ML costs are multi-modal, which means it can have multiple local minimums, only one of which is the global minimum that we want. GD isn't guaranteed to find the global minimum, yet in practice it often works extremely well. (For a convex treatment, see Boyd & Vandenberghe, *Convex Optimization*.)

This is a posting that I summarized with study-purpose and is adapted from lecture notes of NE-795(Scientific Machine Learning), given by professor Xu Wu, NC State University.