An opinionated introduction to deep learning

Eric P. Hanson

May 14, 2025

Outline

We will begin by formalizing deep learning as an optimization problem and briefly cover:

- terminology
- training
- gradient descent
- backpropagation

Then we will discuss how deep learning is not quite an optimization problem, at least in the way we formulated it. How do our goals vary from the simple formulation and what can we do about it?

Note: This is "Eric's take" rather than anything comprehensive. Everything discussed has many alternatives, variants, and generalizations — it's a big field.

Deep learning as optimization

Deep learning terminology

A deep learning model (a *network*) is a parametrized function $f(x; \theta)$, where:

- $\theta \in \mathbb{R}^p$ are the *parameters* of the network. In 2025, p is typically in the millions or billions, although smaller networks can still be practical and useful.
  - Llama 3 70B, an open-weights large language model (LLM) from Meta, has 70B parameters
- $x \in \mathbb{R}^n$ is the *input* to the network. While we often think of inputs as living in other domains, we will assume they have been injected into $\mathbb{R}^n$ prior to being passed to the network.
  - For a small 28x28 pixel grayscale image, $n = 28 * 28 = 784$.
  - For Llama 3 70B, $n = 8192$ token ids (its "context length"); the first layer embeds each into $\mathbb{R}^{8192}$ (its "embedding dimension"), $8192^2 \approx 67M$ numbers in all
- f is the *architecture*, typically a composition of simple functions called *layers*.
  - The architecture can be quite important, as a "good" architecture can make a problem tractable with much less data.

Deep learning terminology

- $y = f(x; \theta) \in \mathbb{R}^m$ is the output of the network.
  - The output dimension m varies widely. In classic tasks like regression or classification, m is usually much less than n; for example, $m = 1$ in binary classification.
  - However for generative models and LLMs, m can be quite large; for Llama 3 70B, for example, $m \approx 1B$.

Training

Training is optimizing a network f over its parameters $\theta \in \mathbb{R}^p$ to minimize a *loss* function $\ell : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$.

To train a network, one needs a "train set" consisting of pairs $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m$. Here, x are the inputs to the network, and y are the corresponding desired outputs.

Typically, training proceeds by attempting to solve the following optimization problem:

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{(x,y) \in \text{ train set}} \ell(f(x;\theta), y)$$

That is, we want to optimize over the parameters $\theta \in \mathbb{R}^p$ to minimize the loss ℓ of f over the train set.

This approach is sometimes called "empirical risk minimization": we seek to minimize the "risk" (loss) empirically over the data we have (the train set).

Gradient descent

Often, the optimization problem

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{(x,y) \in \text{ train set}} \ell(f(x;\theta), y)$$

is solved by variants of *gradient descent*. Gradient descent is a simple iterative algorithm. To minimize some function $F : \mathbb{R}^p \to \mathbb{R}$ over some values $\theta \in \mathbb{R}^p$, we start with some initial point θ_0, then iteratively compute

$$\theta_i := \theta_{i-1} - \gamma \nabla F(\theta_{i-1})$$

where $\gamma > 0$ is a *hyperparameter*[1] called the learning rate (or step size). Here ∇ is the gradient of F with respect to θ, also denoted ∇_θ for clarity sometimes.

In some scenarios, this procedure can be guaranteed to converge, meaning the sequence of θ_i tends toward some fixed limit. This is neither necessary nor sufficient for it to be useful.

[1] A hyperparameter is a value that is chosen before training and fixed, rather than optimized over

Applying gradient descent to deep learning models

Recall that the optimization problem we want to solve is:

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{(x,y) \in \text{ train set}} \ell(f(x;\theta), y)$$

To solve this with gradient descent (directly/naively), our function F must be

$$F(\theta) = \sum_{(x,y) \in \text{ train set}} \ell(f(x;\theta), y)$$

That is, to perform one update of gradient descent we pass through the entire train set. In practice, we often perform *stochastic gradient descent* instead:

1. Initialize with some point θ_0
2. At step i, draw a random pair (x_i, y_i) from the train set at random
3. Update the parameters as $\theta_i := \theta_{i-1} - \gamma \nabla_\theta \ell(f(x_i;\theta_{i-1}), y_i)$

Advantages of stochastic gradient descent (SGD)

Stochastic gradient descent has several important advantages over ordinary gradient descent:

1. if the train set is very large (or infinite, in the case of randomly generated data), we might never finish completing a single update. Instead, with SGD, a single update can be finished relatively quickly.
2. the random noise incurred by performing individual random updates can actually be helpful. The model "jumps around" more in parameter space over the course of training, which can help escape local minima[1].

[1]This is a thing people say, and it makes sense, but I haven't read any papers about it nor ever tried full-dataset gradient descent.

SGD variants

In practice, there are often many more adjustments made to SGD. For example:

- rather than performing updates with a single sample (x_i, y_i), a *batch* (or equivalently, *minibatch*) of such b samples is used, so the update step becomes

$$\theta_i := \theta_{i-1} - \gamma \nabla_\theta \sum_{j=1}^{b} \ell\Big(f\Big(x_{i_j}; \theta_{i-1}\Big), y_{i_j}\Big),$$

  where $\left\{\left(x_{i_j}, y_{i_j}\right) : j = 1, \ldots, b\right\}$ is the ith batch of training data.
- the learning rate γ may be set to decay over time ("learning rate scheduler") or depend on previous updates in some deterministic fashion (adaptive step size) or be divided by the batch size (convention)
- a "momentum" term may be added to the update rule, consisting of an exponentially decaying average of previous gradients
- many more...

Review: Jacobians

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a vector-valued function. Then we can write it as m scalar-valued functions $f_1, ..., f_m$ as:

$$f(\theta) = \begin{pmatrix} f_1(\theta) \\ f_2(\theta) \\ \vdots \\ f_m(\theta) \end{pmatrix}$$

Then the Jacobian matrix of f at θ is defined as:

$$J_f(\theta) = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} & \cdots & \frac{\partial f_1}{\partial \theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial \theta_1} & \frac{\partial f_m}{\partial \theta_2} & \cdots & \frac{\partial f_m}{\partial \theta_n} \end{pmatrix}$$

which is the $m \times n$ matrix of partial derivatives.

Computing the gradient

Recall that f is frequently composed of simpler functions called *layers*. The gradient of f can thus be computed via the chain rule from the gradient of each individual layer.

The chain rule says if $f = h \circ g$, then we can take the derivative with respect to θ as

$$J_f(\theta) = J_h(g(\theta)) \cdot J_g(\theta)$$

This is a matrix-multiplication between $J_h(g(\theta))$ and $J_g(\theta)$. If we had $f = h_1 \circ h_2 \circ ... \circ h_k$ for some value k, then

$$J_f(\theta) = J_{h_1}(h_2(...h_k(\theta)...)) \cdot J_{h_2}(h_3(...h_k(\theta)...)) \cdot ... \cdot J_{h_k}(\theta)$$

we have k matrix multiplications. The computational complexity of this operation depends on the sizes of all the individual matrices, and the *order* in which the product is conducted. Since matrix multiplication is *associative*, $ABC = A(BC) = (AB)C$, we can choose the order to minimize the computation time.

Backpropagation as optimized matrix multiplication associativity

Wikipedia gives the concrete case that if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then

- computing $(AB)C$ needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while
- computing $A(BC)$ needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

using that the straightforward multiplication of a matrix that is $X \times Y$ by a matrix that is $Y \times Z$ requires XYZ ordinary multiplications and $X(Y-1)Z$ ordinary additions.

Backpropagation is the insight that when $m \ll n$, that is, there are many fewer outputs than inputs, the best order in which to conduct this large matrix product is often from the output layer back toward the input.

Deep learning as human-in-the-loop meta-optimization

Deep learning is not a tractable mechanical optimization problem

The goals of deep learning frequently differ from solving the optimization problem

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{(x,y) \in \text{ train set}} \ell(f(x;\theta), y)$$

as follows:

- the loss function is often chosen for computational convenience (nice smooth derivatives). Perhaps the *actual* evaluation of the output on a sample requires an extensive simulation or is otherwise intractable.
- we don't actually care about the train set! Those are the examples we already labeled, we know the idealized output y for each x. We typically care about performance on new, potentially never-seen-before data.

Deep learning is not a tractable mechanical optimization problem

Typically, what we *want* to solve is more like:

$$\underset{\theta \in \mathbb{R}^p}{\text{minimize}} \sum_{x \in \text{ real world}} \text{weight}(x)\ \text{evaluate}(f(x;\theta))$$

where:

- "real world" represents the set of all possible inputs x
- $\text{weight}(x)$ captures that some inputs are more important than others and may have a higher weight. We may not know in advance which these are.
- $\text{evaluate}(f(x;\theta))$ represents some intractable evaluation like "is $f(x;\theta)$ the same output that a panel of human experts would provide after deliberative review of x" or "is $f(x;\theta)$ a 'good answer' to a plain-text question x encoded as a vector?"

and typically each of those items is infeasible to obtain or compute in general, and we have little hope that we can run some mechanical code (e.g. gradient descent) to solve this.

Deep learning as meta-optimization

So, how do we tackle deep learning problems? We manually approximate each quantity in the previous intractable problem and iterate, iterate, iterate ("grad-student descent").

Typically, this involves:

- collecting a train and test dataset
- iterating over update rules ("optimizers"), architectures, loss functions, initialization procedures, hyperparameters, and an increasing variety of other techniques (regularization, data augmentation, self-supervised pretraining, data resampling, ensembles of networks, etc.)
- until one finds a network that performs "well" on the test dataset, typically evaluated in many different domain-specific ways (not just an average loss)

One test dataset is not enough

The process of developing a network sometimes "burns" the test dataset: while the network wasn't trained on it, the optimization of generating the network has encoded dataset-specific properties into the network, such that it won't perform quite as well on truly unseen data. Alternatively, the test dataset may not be truly representative of real-world data.

So, a truly held-out test dataset must be collected and evaluated. If performance is not up-to-snuff, or the network shows performance issues down the line, the whole process is repeated.

Alternative philosophy: bootstrapping proxies

Another complementary way to see deep learning is as an opportunistic exploitation of proxies to the true problem we want to solve:

- Gradient-based optimization is an incredibly effective tool, so let's change our problem to one we can use it on
- Statistical sampling is powerful and the performance of statistical estimates typically depends on the number of samples you have, not the total population size.
  - e.g. you can get opinion polls of 300M Americans with ~3% margin of error by surveying just 1,000 people… as long as they are uniformly randomly sampled (they aren't)
  - truly representative test sets don't need to be huge (the problem is they aren't truly representative)

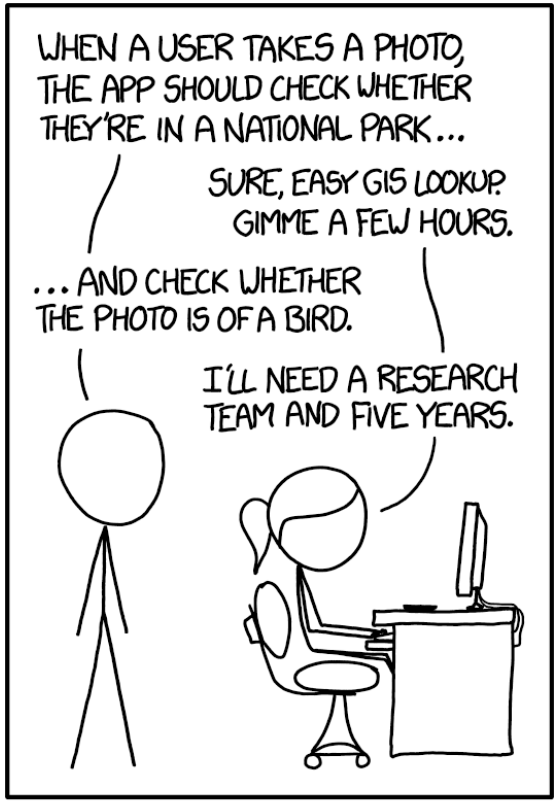
Instead of solving intractable problems, find a sequence of similar-but-tractable ones[1].

[1] with "tractable" being more important than "similar"

Why do all that?

Machine learning approaches (deep learning + classical ML) can take some problems that have been at < 50% solved for decades to 80% solved in days/weeks/months and to 95%+ solved in months/years, and frequently the best performance comes from deep learning models. This approach fundamentally improves the capabilities of computers.

That is not to say we can't do better than deep learning, nor that deep learning techniques can tackle every problem. We almost certainly can and will do better in time. But deep learning is likely an important step in the journey.



XKCD#1425 (September 2014)