The Learning Mechanics Introduction to Neural Networks

Mark Rhee

Note 1

1 Introduction

In 2024, one half of the Nobel Prize in Chemistry was awarded to Demis Hassabis and John Jumper of Google DeepMind for developing AlphaFold—an AI model that predicts protein structures form amino acid sequences. Despite this monumental achievement, the *protein folding problem* remains unsolved; that is, we still lack a first-principles understanding of how protein structures emerge from a linear sequences of amino acid residues[1]. An AI system capable of generating highly accurate protein structure predictions will likely have internalized meaningful regularities about protein folding that we ourselves cannot yet articulate. The problem is that we don't have the ability to recover such knowledge learned by the model. Part of the reason is that we lack a first-principles understanding of how AlphaFold does what it does. It turns out, training a model to predict does not necessarily mean we will comprehend how it predicts.

The same problem persists with Large Langauge Models (LLMs). As of 2026, frontier labs have spent hundreds of billions of dollars scaling LLMs, guided more or less by an empirical observation: that more compute, more data, and more parameters seem to produce better models. By some estimates, data center investment now accounts for roughly half of U.S. GDP growth, surpassing total consumer spending as a share of the economy[2]. And yet ask any honest AI researcher and they will tell you that we have barely scratched the surface in terms of understanding *why* or *how* any of this works.

In both cases, capabilities have far out-paced comprehension. We can build these systems, deploy them, stake a Nobel Prize or a national economy on them, yet we cannot explain why they work or what they have learned. This is the practical motivation for *Learning Mechanics*. Beyond the practical stakes, the astounding empirical success of deep learning is also a genuine scientific mystery that should spark our curiosity. The ingredients for deep learning are remarkably simple, yet the results are not. Understanding of how deep learning works would reshape our understanding of intelligence itself: how structure emerges from data, why certain neural representations arise, how representations are used for computation, and what it even means to learn.

A note on scope. Deep learning has a long and intricate history, and the literature surrounding even the simple setup we are about to introduce is vast—spanning decades of work on architectures, optimization, and theory. This note deliberately sets nearly all of it aside. The goal here is not a survey but an *on-ramp*: the most direct route to the questions and tools of Learning Mechanics. We will introduce only what we need, when we need it.

[1] Source: https://www.pnas.org/doi/10.1073/pnas.2214423119
[2] Source: https://americanaffairsjournal.org/2026/02/understanding-the-llm-bubble/

2 The First (Giant) Step: Deep Neural Networks

At the heart of modern machine learning are *deep neural networks*. Nearly every headline-grabbing AI system of the past decade (AlphaGo, GPT, AlphaFold, Stable Diffusion, etc.) is, at its core, a deep neural network trained by gradient descent on a large dataset—or some other similar architecture that heavily relies on deep neural networks. If we want a scientific theory of modern AI, we need a scientific theory of neural networks.

2.1 The Deceptively Simple Setup

Consider a standard supervised learning task. We are presented with P pairs of data points $\{x_i, y_i\}_{i=1}^P$, each consisting of an input vector x_i and an output vector y_i. The goal is to learn a function $\hat{f}$, i.e., a model, such that $\hat{f}(x_i) \approx y_i$ for all $i = 1, \ldots, P$. The canonical example of a supervised learning task is that of handwritten digit recognition. Here, each input $x \in \mathbb{R}^d$ is a flattened image of a handwritten digit, where d is the number of pixels and each element records the intensity of the corresponding pixel. The output vector $y \in \mathbb{R}^{10}$ corresponding to x is a one-hot vector indicating which digit (0, 1, ..., 9) the image depicts. The goal is to learn a function $\hat{f}$ that maps raw pixel intensities to the correct digit.

A neural network is a particularly powerful choice of model. In its simplest form, it applies a linear transformation to the input and then passes the result through a *point-wise nonlinearity* σ, e.g., a Rectified Linear Unit (ReLU):

$$\hat{f}(x) = \sigma(Wx). \tag{1}$$

Since the composition of linear transformation and nonlinearity occurs only once, we say this network has 1 layer. Stacking L such operations yields a *deep* neural network:

$$\hat{f}(x) = W_L(\sigma(W_{L-1}\sigma(\ldots\sigma(W_1 x)))). \tag{2}$$

So now we have the *architecture*, but how shall we set the values of the weight matrices $W_1, \ldots, W_L$? The answer is that we will *learn* them through a process called *training*.

Training consists of minimizing a *loss function*, e.g., Mean Squared Error (MSE):

$$\mathcal{L}_{\text{MSE}}(\theta) = \sum_{i=1}^{P} ||y_i - \hat{f}(x_i)||^2. \tag{3}$$

We minimize the loss function by *randomly initializing* the weight matrices and updating them through *Gradient Descent (GD)*:

$$W_l \leftarrow W_l - \eta \nabla_{W_l} \mathcal{L} \tag{4}$$

where θ represents the learnable parameters in the model (the elements of the weight matrices in this case) and η is some small positive number called the *learning rate*.

That's it. That's pretty much the whole setup: an **architecture**, some **data**, a **loss function**, and a **learning rule**. Granted, practitioners have learned to add various bells and whistles to this simple setup to get everything to work in the real-world. On the architectural side, modern networks incorporate residual connections, convolutional blocks, and attention mechanisms. On the learning rule side, vanilla gradient descent is typically replaced by minibatch variants such

as Stochastic Gradient Descent (SGD) or more sophisticated optimizers like Adam or Muon and regularization techniques such as weight decay, dropout, or batch normalization may be employed. These modifications matter enormously in practice, but at the heart of it all is the vanilla setup given by Equations (2), (3), and (4). Understanding deep neural networks is a prerequisite for saying anything scientifically meaningful about the more complex architectures built on top of them, and ultimately for building new ones on principled theoretical grounds rather than empirical trial and error.

2.2 The Guiding Questions

A number of natural questions arise when one first encounters a deep neural network. We can organize some of them neatly around the life cycle of a model: what we set up *before* training, what happens *during* training, and what we are left with *after*.

Before training. The process of training is almost by definition an automatic one. The model learns which parameters are best suited to minimize the loss without any human intervention. However, before training, there are many *hyperparameters* that must be set *a priori*. Some such hyperparameters include the dimensions of the weight matrices, the depth of the network, the learning rate, the initialization scheme and scale, the length of training, and the choice of nonlinearity. The main question here is:

> *Given a task, can we principally determine an optimal set of hyperparameters prior to training?*

During training. Consider the loss as a function of the weights. Each point in the space of all possible weights, or the *weight-space*, has an associated loss. At initialization, the model is randomly dropped somewhere in weight-space where the loss is likely high. Throughout training, the model traverses the loss landscape to find a set of weights where the loss small. In analogy to physics, we may picture the model as a ball rolling through the hills and valleys of the *loss landscape*. This prompts the following question:

> *What laws, analogous to the laws of physics, govern how a network moves through its loss landscape during training?*

After training. Here lies perhaps the deepest and most pressing questions. In practice, deep neural networks are heavily *over-parameterized*: the dimensions of the weight matrices are typically so large that there are far more tunable parameters than training examples, more than enough to simply *memorize* the data and fail on anything new. Yet in practice they generalize to new examples unseen during training. This begs the question:

> *Why don't over-parameterized deep neural networks just memorize? How do they find solutions that generalize?*

A widely held answer—what has become folklore in many pockets of the research community—is that, instead of memorizing, deep neural networks build structured *internal representations* of the data. When we pass an input x through the network, each layer outputs a vector of activations: an internal re-encoding of the input. The first layer outputs $z_1 = \sigma(W_1 x)$, the second layer outputs $z_2 = \sigma(W_2 z_1)$, and so on. We call these intermediate vectors $z_1, z_2, \ldots$ internal representations, and the claim is that the network reshapes them layer by layer to expose exactly the structure a task needs, so that inputs calling for similar outputs come to look alike. This is a thread we will follow through much of the course:

> *What internal representations do deep neural networks build, how do they form during training, and why do they support generalization?*

These are some of the guiding questions of Learning Mechanics. They are all interconnected and

often, the attempt to answer one sheds light on the answer to another. Throughout this course, we will tackle many of them. However, before diving in, it's worth asking why these questions are difficult to answer in the first place when the setup seems so straightforward; where is the complexity arising from when the ingredients are so simple?

3 Why is it Hard to Study Deep Neural Networks?

Usually, the first and primary hurdle to studying complex systems is *opacity*. Consider a biological cell: to understand what makes it tick, we must painstakingly infer its inner workings from a limited, noisy set of observations, each giving us a partial, indirect view. The "equations of motion" are not handed to us; we must guess at them. The same is true of the brain, the economy, or the climate: the systems are partially hidden, and much of the science consists of figuring out what the variables even look like.

Neural networks are different. Every weight, every activation, every gradient, every loss value is exactly knowable at every step of training. The "laws of motion," i.e., gradient descent, are written down explicitly in a few lines of math. We can freeze training, perturb any weight, and watch exactly what happens. In the history of science, we have rarely been handed a complex system this transparent.

So why are deep neural networks hard to study? The difficulty is not opacity, but *complexity*, and it comes in a few distinct flavors:

- **Coupled dynamics.** Gradient descent on a deep network is a coupled nonlinear dynamical system in millions to trillions of variables. The update to any given weight depends, by the chain rule, on the current values of many other weights across the network. Layers talk to each other. Change one weight in layer 3, and the gradients flowing through layer 17 change too.
- **Pointwise nonlinearities.** The nonlinearity σ is what allows neural networks to learn nonlinear functions, but it is also what makes them analytically painful to work with. A ReLU or sigmoid introduces non-smooth or saturating behaviors that make exact analytical solutions virtually impossible in the general case.
- **Data complexity.** Even if we did fully understood the network, we would still have to reckon with the data. Real datasets—natural images, natural language, protein sequences—have rich, high-dimensional, poorly characterized structure. This matters because everything the model learns comes from the data. Thus, unavoidably, to understand deep neural networks, we must understand data.

The depth, nonlinearity, and data of a deep neural network form axes of complexity that all interact with each other. The art of Learning Mechanics is figuring out which axes we can turn off, simplify, or take to a limit, so that the remaining problem becomes tractable without throwing away the phenomenon we want to understand.

3.1 The Learning Mechanics Way

So, studying neural networks is hard. Some say, it's so hard that questions such as those proposed in Section 2.2 are unanswerable. The Learning Mechanic argues that this is not the case. The Learning Mechanic believes that with the right tools, we can develop a *fundamental, mathematical, predictive, comprehensive, intuitive,* and *useful* theory to understand them. What are the right tools, then? Well, as it turns out, physicists have been developing theories that fit the above desiderata for centuries. A non-exhaustive list of tools beloved by physicists is:

simplify, take limits, find the right variables, and look for universality. Let's look at what simplifying and taking limits can look like in the context of Learning Mechanics.

4 Linear Regression: The Compulsory Starting Point

One way to *simplify* the study of deep neural networks is to get rid of the nonlinearity σ. Without nonlinearities, the deep neural network from Equation (2) becomes a *deep linear network*:

$$\hat{f}(x) = W_L W_{L-1} \cdots W_1 x. \tag{5}$$

A deep linear network computes a linear function of x (it's just a product of matrices), but gradient descent on the individual W_l matrices is a genuinely nonlinear dynamical system. One can observe real deep learning phenomena such as *saddle-to-saddle dynamics* in a setting simple enough to solve exactly. Tackling deep linear networks is quite an arduous task—mainly due to the coupled dynamics—and it's the subject of a future lecture. For now let's look at the simplest possible case where the depth $L = 1$. We won't see much interesting deep learning phenomena, but this analysis will serve as a great introduction to the tools of Learning Mechanics.

4.1 Simplify: No Nonlinearities & Depth $= 1$

When depth $= 1$, the model from Equation (5) reduces to

$$\hat{f}(x) = Wx. \tag{6}$$

This is simply linear regression, or least squares. Linear regression is often considered uninteresting as a learning algorithm because we know exactly what it learns, how it learns it and the strict limitations it has due to its inexpressiveness. However, it's a good setting to become acquainted with gradient descent due to the lack of coupled dynamics and nonlinearities. And, in fact, it's also a good setting to learn about an interesting phenomenon in gradient descent called *implicit regularization*. Linear regression is the first example of many we will encounter where the setting is simple but the phenomenon (implicit regularization in this case) is subtle and deep.

For maximal simplicity, take the output to be one-dimensional (the multi-dimensional case is simply many parallel one-dimensional cases stacked on each other). Write $w \in \mathbb{R}^d$ for the single weight vector. The goal is to find w such that for every input vector $x_i \in \mathbb{R}^d$ and output scalar $y_i \in \mathbb{R}$ in the training data set, $y_i \approx w^\top x_i$. The MSE loss is given by $\sum_{i=1}^P (y_i - w^\top x_i)^2$. We may equivalently write this by stacking the inputs into a *design matrix* $X \in \mathbb{R}^{P \times d}$ whose i-th row is $x_i^\top$, and stacking the targets into $y \in \mathbb{R}^P$ such that the MSE loss is:

$$\mathcal{L}(w) = \|y - Xw\|^2 = \left\| \begin{bmatrix} y_1 \\ \vdots \\ y_P \end{bmatrix} - \begin{bmatrix} — & x_1^\top & — \\ & \vdots & \\ — & x_P^\top & — \end{bmatrix} \begin{bmatrix} | \\ w \\ | \end{bmatrix} \right\|^2.$$

This loss is a convex quadratic with respect to the weight vector w, i.e., a single bowl in weight-space, with no hills or saddles to get stuck on. Its gradient with respect to w is $\nabla_w \mathcal{L} = -2X^\top(y - Xw)$, so a gradient descent update looks like:

$$w \leftarrow w + 2\eta\, X^\top (y - Xw).$$

The update is *linear* in w: no products of weights, no coupling, just a fixed matrix acting on the residual. This is exactly the simplicity we bought by dropping depth.

The classical setting is the well-determined regime where $P \geq d$ (more samples than number of parameters) with X having linearly independent, high-dimensional column vectors. In this regime, the matrix product $X^\top X$ is invertible. We say gradient descent has converged when the gradient equals zero; thus, setting the gradient equal to zero, we can compute the weight vector w^* that gradient descent converges to:

$$\begin{aligned} \nabla_w \mathcal{L} = -2X^\top (y - Xw^*) &= 0 \\ X^\top X w^* &= X^\top y \\ w^* &= (X^\top X)^{-1} X^\top y. \end{aligned}$$

So, when the problem is well-determined, there exists a unique solution. However what we're actually interested in is not the well-determined regime, but the over-parameterized regime where $P < d$ such that there are more parameters than number of data points (recall that this is the regime that deep neural networks work in in practice). We will assume that the rows are linearly independent so that $XX^\top$ is invertible.

In the over-parameterized regime, there is an entire subspace $\mathcal{U}$ of weights that achieve the same minimal loss on the training set. Mathematically, this is because a wide, underdetermined matrix X is guaranteed to have a nontrivial null space $\text{null}(X) \neq \emptyset$. This means that for any $w \in \mathcal{U}$, if $\Delta \in \text{null}(X)$, then $X(w + \Delta) = Xw$ such that $w + \Delta \in \mathcal{U}$. So, there are infinitely many elements in $\mathcal{U}$—which one will gradient descent converge to? To answer this question, we shall introduce one of the most widely utilized tools in Learning Mechanics: the *gradient flow* regime.

4.2 Take a Limit: $\eta \to 0$

The infinitesimal learning rate limit ($\eta \to 0$) is so common that it has a name: *gradient flow*. Taking this continuous-time limit allows us to apply powerful tools from the theory of ordinary differential equations (ODEs) to study the training trajectories. Instead of discrete steps, we can track the parameters moving smoothly over continuous time t.

In the gradient flow regime, we consider the weight vector w as a function of time such that its derivative with respect to time $\dot{w}$ equals the negative gradient of the loss $-\nabla_w \mathcal{L}$:

$$\dot{w} = -\nabla_w \mathcal{L} = 2X^\top (y - Xw).$$

We can solve this differential equation using a substitution. Define the residual vector as $r := y - Xw$ such that $\dot{w} = 2X^\top r$. Then,

$$\begin{aligned} X\dot{w} &= 2XX^\top r \\ \implies \dot{r} &= -2XX^\top r, \end{aligned}$$

which is a standard first order homogeneous linear differential equation with solution:

$$r(t) = e^{-2XX^\top t} r(0).$$

Substituting this into the gradient flow equation,

$$\dot{w} = 2X^\top e^{-2XX^\top t}(y - Xw_0),$$

where w_0 is the weight vector at initialization. Integrating both sides with respect to time,

$$\begin{aligned} w(T) - w_0 &= \int_0^T \left[2X^\top e^{-2XX^\top t}(y - Xw_0)\right] dt \\ &= 2X^\top \left(\int_0^T e^{-2XX^\top t} dt\right)(y - Xw_0). \end{aligned}$$

The matrix integral $\int_0^T e^{-2XX^\top t} dt$ can be evaluated since $XX^\top$ is positive-semidefinite in the over-parameterized regime (assuming the rows are independent). In particular,

$$\begin{aligned} \int_0^T e^{-2XX^\top T} dt &= -\frac{1}{2}(XX^\top)^{-1} e^{-2XX^\top t}\Big|_0^T \\ &= \frac{1}{2}(XX^\top)^{-1}(I_P - e^{-2XX^\top T}), \end{aligned}$$

where I_P is the identity matrix of dimension $P \times P$. Substituting this expression for the integral, we obtain our final solution:

$$w(T) - w_0 = X^\top (XX^\top)^{-1}(I_P - e^{-2XX^\top T})(y - Xw_0). \tag{7}$$

Congratulations, you just analytically solved for the training dynamics of over-parameterized linear regression! We can evaluate the above expression at any point during training to see what the weights will look like.

This expression is certainly intimidating at first glance. However, there are some nice ways to interpret it. Observe that $e^{-2XX^\top T}$ decays exponentially, so it goes to 0 as the training time $T \to \infty$. Once this term is small, $w(T)$ stops changing, so gradient descent has converged. Thus, for over-parameterized linear regression, the training loss decays exponentially. And plugging in the expression for $w(T)$ into the loss, we can see that once the $e^{-2XX^\top T}$ has decayed to 0, the loss equals 0. So, gradient descent converges to a perfectly *interpolating* solution. But recall that there are infinitely many solutions in the over-parameterized regime. We can learn more about the specific solution that gradient descent converges to by looking at what happens if $w_0 = 0$, i.e., the parameters are initialized at 0. Then in the limit $T \to \infty$, we get the following simple solution:

$$\lim_{T \to \infty} w(T) = X^\top (XX^\top)^{-1} y =: w^*. \tag{8}$$

Is there anything special about this particular solution?

One observation is that w^* is contained entirely within the row space of X, i.e., it's contained within the span of all the input vectors. Since we assumed the input vectors are linearly independent, w^* is the unique solution contained in this span. Recall the linear algebra fact that the row space of a matrix is the orthogonal complement of its null space. Thus, any solution $w \in \mathcal{U}$ can be orthogonally decomposed such that one component $w^{\|}$ is in the row space of X ($w^{\|} = w^*$ by uniqueness) and the other $w^{\perp}$ is in the null space. Since the solution that gradient descent converges to is entirely in the row space of X, it must be the solution of minimum L2 norm; i.e., for any $w \in \mathcal{U}$,

$$\|w\|^2 = \|w^*\|^2 + \|w^\perp\|^2 \geq ||w^*||^2.$$

This is interesting because we never explicitly constrained gradient descent to find the minimum-norm solution. Instead, it *implicitly regularized* itself with a bias toward small norm. This can be thought of as a rudimentary *simplicity bias*.

4.3 A Note on Toy Models

Linear regression is so simple that we know exactly what happens to its weights during training (Equation (7)) and what solution it learns (Equation (8)). And we were able to derive both fairly easily. However, because it's so simple, it doesn't tell us much of anything about deep neural networks, i.e., it's not a very good toy model for deep neural networks. Recall the simplifications we made and the limits we took to get from Equation (2) (a deep neural network) to Equation (8) (the training dynamics of linear regression): 1) we got rid of all nonlinearities, 2) we set depth=1, and 3) we took the limit $\eta \to \infty$. We effectively set aside all the difficulties of studying neural networks presented in Section 3: coupled dynamics, nonlinearities, and data complexity. As a general rule of thumb, we should keep at least one axis of complexity if we want to learn something useful about actual deep neural networks. Ultimately, the right way to check whether our toy model is representative of the real thing is through empirics—run similar experiments on both that test for the phenomenon of interest, and see if the results look similar. We will wrestle with this problem of finding good toy models that are *simple but not too simple* for the rest of the course.