Toy Model I: Deep Linear Networks

Mark Rhee

Chapter 2

This chapter is adapted from an interactive blog post on learningmechanics.pub *co-authored by Dhruva Karkada, Jamie Simon, and yours truly.*

In the previous chapter, we articulated why studying deep neural networks is so difficult. We discussed the three central axes of complexity: coupled dynamics, pointwise nonlinearities, and data complexity. One approach to tackling these complications, particularly beloved by physicists, is to build and study toy models. An example of this toy model approach is in the study of *deep linear networks*, i.e., deep neural networks with no nonlinear activation functions:

$$\hat{f}(x) = W_L \cdots W_2 W_1 x.$$

This toy model does away with one of the three axes complexity: pointwise nonlinearities.

We've already encountered a linear toy model in the previous chapter; namely, linear regression. We exactly solved for the training dynamics in the over-parameterized setting (Equation (**??**)) which revealed that 1) the loss decays exponentially, and 2) out of the many interpolating solutions, gradient descent converges to the solution of minimum L2 norm. While these findings are amusing, they don't tell us much about what we're actually interested in: deep neural networks. With linear regression, we've overly simplified the setup by not only getting rid of nonlinearities, but also coupled dynamics. With deep neural networks, we will attempt to retain coupled, nonlinear dynamics so as to understand some of the dynamical phenomena of neural network training. In particular, here are four pieces of folklore about neural network training that we will gain a quantitative understanding of through deep linear networks:

1. they preferentially learn certain functional directions before others[1];
2. they optimize to good (and sometimes global) minima despite having highly nonconvex loss landscapes;
3. at any given point in training, each weight matrix's typical updates are low-rank compared to the full matrix; and
4. they sometimes learn in a stepwise fashion with long plateaus punctuated by loss drops, especially when trained from very small initial weights.

We'll see that, amazingly, deep linear networks capture all of these behaviors in one solvable system.

Let's start by establishing the mathematical setting. Suppose we have a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ which consists of n sample-target pairs, where the samples $x \in \mathbb{R}^{d_{\text{in}}}$ and targets $y \in \mathbb{R}^{d_{\text{out}}}$ are both vectors. A deep linear network maps vector inputs through a chain of matrix multiplications:

[1] This can be thought of as an inductive bias, much like the minimum-norm bias of gradient descent in linear regression.

$$\hat{f}(x) = W_L \cdots W_2 W_1 x \tag{1}$$

and we will try to learn the weight matrices $\{W_\ell\}_{\ell=1}^L$ so that $\hat{f}(x_i) \approx y_i$. This model is *deep* because it consists of $L > 1$ trainable layers. It's *linear* because, despite the many parameters, it ultimately still represents a linear function of x: note that $\hat{f}(x) = W_{\text{tot}} x$, where $W_{\text{tot}} := W_L \cdots W_1$.

We'll train this model by minimizing the mean squared error using gradient descent:

$$\mathcal{L}(W_1, \ldots, W_L) = \frac{1}{n} \sum_{i=1}^{n} \|y_i - W_L \ldots W_1 x_i\|^2.$$

The summation over sample-target pairs can be thought of as an expectation over the dataset. With some matrix algebra, we can express the loss solely in terms of matrix quantities:

$$\begin{aligned} \mathcal{L}(W_1, \ldots, W_L) &= \mathbb{E}_{(x,y)\sim\mathcal{D}} \Big[\operatorname{Tr} (y - W_L \ldots W_1 x)(y - W_L \ldots W_1 x)^\top \Big] \\ &= \|\Sigma_{yx} \Sigma_{xx}^{-1/2} - W_L \ldots W_1 \Sigma_{xx}^{1/2}\|_{\text{F}}^2 + \text{const.} \end{aligned} \tag{2}$$

where $\Sigma_{xx} := \mathbb{E}_{\mathcal{D}}[xx^\top]$ is the input-input covariance, and $\Sigma_{yx} := \mathbb{E}_{\mathcal{D}}[yx^\top]$ is the output-input covariance. All of the statistical structure in the data is captured by these two covariance matrices. Consequently, information from the data enters the learning process solely through these matrices. The constant term[2] can be thought of as irreducible noise—ever-present no matter what the weights are.

1 Solving the learning dynamics

The beauty of the deep linear network is that you can exactly solve its learning dynamics, given some simplifying assumptions. Even though the function class is linear, the learning dynamics in parameter space are highly *non*linear, so it's surprising and convenient that it works out so nicely. We'll solve for the exact learning dynamics following a simplified version of the canonical derivation in Saxe et al. (2013). To keep the math as simple as possible, we'll solve the $L = 2$ case.

Here's a bird's-eye view of the strategy we'll take:

1. We'll write the gradient descent update equations, take the infinitesimal learning rate limit (gradient flow), and obtain differential equations describing the evolution of the weight matrices.
2. We'll look at the resulting equations and decide that they're too hard to solve. But, instead of giving up entirely, we'll add an assumption ($\Sigma_{xx} = I$) and argue that it's reasonable.
3. Then, the most complicated part: we'll argue that when the weights are initialized very small, the first part of optimization serves primarily to quickly align the weight matrices with each other and with the target.
4. We'll show that once the weights are aligned, the resulting matrix dynamics decouple into independent scalar ODEs. We integrate them to solve for the time course of learning, obtaining sigmoidal dynamics for each singular direction of the model.

[2] const. $= \Sigma_{yy} - \Sigma_{yx}(\Sigma_{xx})^{-1}(\Sigma_{yx})^\top$ is constant w.r.t. the weight matrices.

1.1 Write the gradient flow equations

For a 2-layer network, the mean squared error loss is:

$$\mathcal{L}(W_1, W_2) = \frac{1}{n}\sum_{i=1}^{n} \|y_i - W_2 W_1 x_i\|^2.$$

For maximal simplicity, we'll do the calculation in the case that all matrices are square: $x_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}^d$, and there are d hidden neurons (though more complicated, the analysis still works when the input, hidden, and output dimensions are different). Therefore W_1 and W_2 are both $d \times d$.

To train the model, we run gradient descent with learning rate η:

$$W_1^{(t+1)} = W_1^{(t)} - \eta \nabla_{W_1} \mathcal{L}|_{W_1^{(t)}}$$
$$W_2^{(t+1)} = W_2^{(t)} - \eta \nabla_{W_2} \mathcal{L}|_{W_2^{(t)}}$$

Discrete steps are hard to deal with. So we'll turn to a tool introduced in the previous chapter: gradient flow. With infinitesimally small learning rate, we get the following continuous differential equations instead of the discrete update rule above:

$$\frac{d}{dt} W_\ell = -\nabla_{W_\ell} \mathcal{L}.$$

Computing the gradients using the covariance matrices introduced in Equation (2), we arrive at the gradient flow equations:

$$\frac{d}{dt} W_1 = W_2^\top (\Sigma_{yx} - W_2 W_1 \Sigma_{xx}) \tag{3}$$
$$\frac{d}{dt} W_2 = (\Sigma_{yx} - W_2 W_1 \Sigma_{xx}) W_1^\top. \tag{4}$$

Before proceeding, let's build some intuition for what Equations (3) and (4) are telling us. We can infer what happens towards the beginning and end of training. In the beginning, learning is driven primarily by the statistical structure of the input-output relationship: when the initial weights are still small, $W_2 W_1 \Sigma_{xx} \ll \Sigma_{yx}$, and the parenthesized term in the gradient is dominated by Σ_{yx}. The input covariance Σ_{xx} plays a secondary role, acting as a *preconditioner* for the model. Training converges when the weights stop changing, $\frac{d}{dt} W_1 = \frac{d}{dt} W_2 = 0$. This is satisfied when the model converges to $W_2 W_1 = \Sigma_{yx} \Sigma_{xx}^{-1}$.

Equations (3) and (4) look quite similar. Concretely, one may observe that $(\frac{d}{dt} W_1) W_1^\top = W_2^\top (\frac{d}{dt} W_2)$. This observation leads us to a *conservation law*[3] in the dynamics:

$$\frac{d}{dt}(W_2^\top W_2 - W_1 W_1^\top) = 0. \tag{5}$$

It's called a conservation law because it states that the matrix quantity $H := W_2^\top W_2 - W_1 W_1^\top$ is dynamically conserved under gradient flow. If $H \approx 0$, we say that the two weights are "balanced." Therefore, a deep linear network that is initialized balanced will stay balanced, and vice versa. This conservation law implies that optimization trajectories have hyperbolic geometry.[4] A similar conservation law holds for arbitrarily deep linear networks; hyperbolic trajectories are a fundamental phenomenon of optimizing linear models using gradient descent.

[3] For the physicists in the audience, this conservation law is essentially a consequence of a continuous symmetry in the loss: $\mathcal{L}(W_1, W_2) = \mathcal{L}(AW_1, W_2 A^{-1})$ for $A \in \mathrm{GL}(n)$. Noether's theorem guarantees a corresponding conserved quantity.

[4] For intuition, consider that the scalar version $\frac{d}{dt}(x^2 - y^2) = 0$ parameterizes a hyperbola.

1.2 Whiten the input data

It's very difficult to integrate equations (1) and (2) when Σ_{yx} and Σ_{xx} don't commute.[5] Their left and right singular bases conflict with each other, and it's hard to precisely characterize this tension and its effect on the optimization dynamics. One way to get around this is by *whitening* the input data, i.e., linearly transforming the inputs such that $\Sigma_{xx} = I$.[6] Since the identity matrix commutes with everything, we can dodge this issue entirely, and Equations (3) and (4) simplify:

$$\frac{d}{dt}W_1 = W_2^\top(\Sigma_{yx} - W_2W_1) \tag{6}$$

$$\frac{d}{dt}W_2 = (\Sigma_{yx} - W_2W_1)W_1^\top. \tag{7}$$

Many applied math or physics problems are fundamentally about finding the right basis to work in. Solving Equations (6) and (7) is no different. With Σ_{xx} out of the picture, the statistical structure in these equations enters solely through Σ_{yx}. Thus, the natural left and right bases which diagonalize the problem are the left and right singular vectors of Σ_{yx}. Let's define the SVD of the target, $\Sigma_{yx} := U^\star\Lambda^\star V^{\star\top}$, and rotate our variables: $\Lambda^\star := U^{\star\top}\Sigma_{yx}V^\star$, $\tilde{W}_1 := W_1V^\star$, and $\tilde{W}_2 := U^{\star\top}W_2$. In terms of the new variables, the previous ODEs simplify to

$$\frac{d}{dt}\tilde{W}_1 = \tilde{W}_2^\top(\Lambda^\star - \tilde{W}_2\tilde{W}_1) \tag{8}$$

$$\frac{d}{dt}\tilde{W}_2 = (\Lambda^\star - \tilde{W}_2\tilde{W}_1)\tilde{W}_1^\top. \tag{9}$$

Since we initialize all the weights i.i.d. randomly, they do not introduce their own preferred basis, and the initial weight distribution remains unchanged under this transformation. Therefore, the rotated learning problem is exactly equivalent to the original. The main difference is that the target $\Lambda^\star$ is now diagonal. Its singular values $\lambda_\mu^\star$ are real-valued and non-negative since Σ_{yx} is positive-semidefinite.

We have simply rotated our entire learning problem into the SVD basis of Σ_{yx}. This helps make it clear that the data decomposes into independent and orthogonal "modes." Each mode μ pairs an *input direction* (i.e., the μ^{th} column of $V^\star$) with an *output direction* (i.e., the μ^{th} column of $U^\star$). The corresponding singular value $\lambda_\mu^\star$ indicates how prominent this distinction is in the data. In real data, larger singular values often correspond to coarser, more important structure; smaller ones often correspond to finer details.

An enlightening example about the SVD

The following example and figure are from Saxe et al. (2019).

Imagine a model is presented with an item i (e.g. a "Canary") represented as a one-hot input vector x_i. The model's objective is to predict a vector of features y_i, such as "Can Fly," "Has Wings," or "Is Yellow." Throughout training, the model experiences many such examples (x_i, y_i). The statistical structure of this environment is captured by the input-output correlation matrix $\Sigma_{xy} = \mathbb{E}_{\mathcal{D}}[y_i x_i^\top]$.

[5] Recall that diagonalizable (e.g. symmetric) matrices commute if and only if they share a common eigenbasis.

[6] You can always preprocess your data in this way, so it's not a terrible assumption. It's a different question whether this reflects practical ML pipelines.

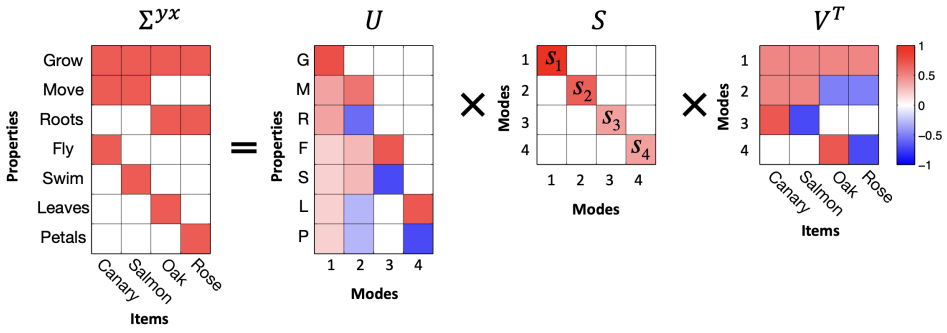

Figure 1: Modes link a set of coherently covarying properties with a set of coherently covarying items.

Using the SVD allows us to probe the data in terms of independent "modes." We refer to the columns of V (i.e. the rows of $V^\top$) as *input-analyzing* vectors, or *input modes*. The μth input mode determines the position of any given input item along an important 1-D semantic dimension. As a concrete example, consider the second row vector of $V^\top$: this vector acts as an "animal-plant" axis. Along this axis, animals (Canary, Salmon) have positive values while plants (Oak, Rose) have negative values.

We refer to the columns of U as *output-analyzing* vectors, or *output modes*. Similar to input modes, the μth output mode determines the position of any given output feature along an important 1-D semantic dimension. Every input mode has a corresponding output mode and vice versa. The "categorizing power" of any given axis is quantified by its singular value s_α. Larger singular values correspond to more important distinctions, while smaller singular values correspond to finer subordinate details.

1.3 Show that weight alignment happens quickly

Equations (8) and (9) are still quite complicated: the parameter dynamics remain coupled. Our strategy for attacking these equations will be to ask "under what conditions do the dynamics *decouple* into independent mode-wise scalar equations?"

It helps to continue thinking spectrally. Using the SVD, we may decompose any weight matrix into *directions* (singular vectors) and *magnitudes* (singular values). Decoupled dynamics arise when the directional structure has already been resolved, so that the only things left to evolve are the magnitudes.

To make this precise, we write the SVD of each weight matrix as $W_1 = U_1 S_1 V_1^\top$ and $W_2 = U_2 S_2 V_2^\top$. The rotated model $\tilde{W}_2 \tilde{W}_1$ reduces to a product of only the magnitude matrices $S_2 S_1$, with all the directional factors gone, when three conditions are satisfied:

1. $U_2 = U^\star$ — the left singular vectors of W_2 align with the output modes of the data.
2. $V_1 = V^\star$ — the right singular vectors of W_1 align with the input modes of the data.
3. $V_2 = U_1$ — the right singular vectors of W_2 match the left singular vectors of W_1.

We refer to these three conditions collectively as **weight alignment**. The key idea is that these conditions ensure that adjacent weight matrices interface coherently with each other and that the model is aligned with the target. If these conditions are satisfied, we only need the singular values S_2 and S_1 to evolve.

These weight alignment conditions are quite stringent. Nonetheless, one consistently observes this behavior empirically in the **small initialization regime**, where the parameters are initialized

near zero and must grow substantially during training. This regime naturally forces the network into alignment through two distinct mechanisms: one driving *inner alignment* (coherence between the layers) and one driving *outer alignment* (coherence with the data).

To contrast, for very wide networks in the large initialization regime, the model trains "lazily" in the sense that individual parameters barely move from their random starting values and thus the weight matrices do not develop any interesting spectral structure. Therefore phenomena such as sequential learning, plateaus, sudden transitions do not appear and the model's *internal representations* do not meaningfully evolve.

In the following subsections, we offer theoretical justification for why we should expect weight alignment to happen quickly in the small initialization regime. While not extremely complicated, they require some prior knowledge about perturbation theory and power iteration that we do not wish to explore deeply. We would thus not be offended if the reader took for granted that weight alignment happens quickly from small initialization. In fact, the original Saxe et al. (2013) paper assumed as such. We'll later see that the theory derived from this assumption matches empirical results, which, while not a substitute for rigorous theoretical justification, is a close second.

Inner alignment: the conservation law argument

The conservation law (Equation (5)) states that the difference between the weights' Gram matrices, $H := W_2^\top W_2 - W_1 W_1^\top$, is strictly conserved throughout training. In the small initialization regime, the initial Gram matrices (and consequently H) are tiny. Once the weights have grown substantially, H becomes small compared to the Gram matrices:

$$W_2(t)^\top W_2(t) = W_1(t) W_1(t)^\top + \text{perturbation}$$

The property $H \approx 0$ is called "balancedness." Given balancedness, we can employ facts about linear algebra to justify inner alignment. The Davis-Kahan $\sin(\theta)$ theorem states that given symmetric matrices A, B such that the difference $A - B = H$ is small, if the gaps between eigenvalues in A and B are much larger than the spectral norm of H, then the eigenspaces of A and B will be approximately equal. We can guarantee that the eigengaps in $W_1 W_1^\top$ and $W_2^\top W_2$ will quickly grow much larger than $\|H\|$ through small initialization. In the limit, this directly forces the right singular vectors of W_2 to align with the left singular vectors of W_1, giving us inner alignment ($V_2 = U_1$). Weyl's inequality also implies that the singular values of W_1 and W_2 must agree.

Outer alignment: the power iteration argument

While the conservation law explains why the layers align with each other, why do they align with the singular vectors of the *data*? Recall that the whitened gradient flow equation is

$$\frac{d}{dt} W_1 = W_2^\top (\Sigma_{yx} - W_2 W_1).$$

At initialization, the weights are tiny, and their product $W_2 W_1 \ll \Sigma_{yx}$. Because the model is effectively zero, the gradient equation simplifies dramatically:

$$\frac{d}{dt} W_1 \approx W_2^\top \Sigma_{yx}.$$

Notice the structure of the update: the weights are being repeatedly multiplied by the target matrix Σ_{yx}. This is the idea behind *power iteration*, the standard algorithm for pulling out a matrix's dominant singular vectors. During this early phase of training, the power iteration dynamic aggressively pulls the network's weights toward the top modes of the data. It rotates the top right singular vectors of W_1 toward the top right singular vectors of Σ_{yx} ($V_1 \to V$). By

the time the weights have grown large enough for the term W_2W_1 to matter, the power iteration phase has already locked the network's directional structure into the outer alignment conditions required to decouple the dynamics.

1.4 Decouple the dynamics and solve the scalar equation

Assuming weight alignment as defined above, Equations (3) and (4) become

$$\frac{d}{dt}S_1 = S_2(\Lambda^\star - S_2S_1) \tag{10}$$

$$\frac{d}{dt}S_2 = (\Lambda^\star - S_2S_1)S_1 \tag{11}$$

Observe that every matrix in these equations is now diagonal. This shows that weight alignment is a fixed point of the angular dynamics; once the weights are aligned, gradient flow will never misalign them. Furthermore, since the diagonal entries evolve independently, their dynamics decouple into mode-wise scalar equations.

In the small initialization regime, we can argue using the conservation law that $S_1 \approx S_2$ throughout training. Therefore, we can safely drop the subscript and approximate $S(t) := S_1(t) \approx S_2(t)$. The full model's end-to-end singular values are therefore given by S^2. The ODE for the μ^{th} end-to-end singular value, s_μ^2, is then given by the chain rule:

$$\begin{aligned}\frac{d}{dt}s_\mu^2 &= 2s_\mu\frac{d}{dt}s_\mu \\ &= 2s_\mu^2(\lambda_\mu^\star - s_\mu^2)\end{aligned}$$

This ODE is separable: we can move all the s_μ^2 terms to one side and the t terms to the other and integrate directly. The solution is

$$\boxed{s_\mu^2(t) = \frac{\lambda_\mu^\star}{1 + \left(\frac{\lambda_\mu^\star}{s_\mu^2(0)} - 1\right)e^{-2\lambda_\mu^\star t}}} \tag{12}$$

This is a logistic function in t. Each end-to-end singular value starts near its initial value $s_\mu^2(0)$, stays in that neighborhood for a while, rapidly transitions to its target value $\lambda_\mu^\star$, then saturates.

The final picture is that the full network map first undergoes a fast alignment period, followed by slow sigmoidal dynamics:

$$W_2(t)W_1(t) \approx U^\star S^2(t) V^{\star\top} = \sum_\mu s_\mu^2(t)\, u_\mu v_\mu^\top.$$

The network shares the same singular vectors as Σ_{yx} for most of training, and only the singular values change, each following its own sigmoidal trajectory.

We can check empirically whether Equation (12) accurately predicts the singular values of the model throughout training, and as it turns out, it does. The following figure is from Saxe et al. (2013).

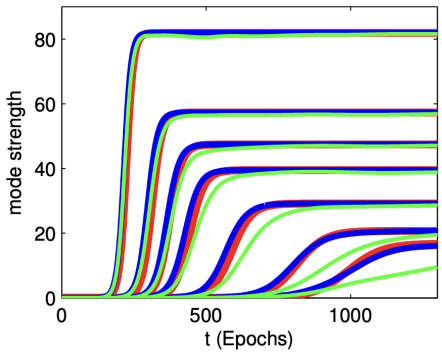

Figure 2: Dynamics of learning in a 2-layer neural network. Curves show the strength of the network's representation of seven modes of the input-output correlation matrix over the course of learning. Red traces show analytical curves. Blue traces show simulation of full dynamics of a 2-layer linear network from small random initial conditions. Green traces show simulation of a nonlinear 2-layer network with tanh activation functions.

2 Saddle-to-saddle training dynamics

How long does it take for mode μ to be learned? Starting from equation (10), we can solve for the halfway-time τ_μ at which $s_\mu^2(t=\tau_\mu) = \frac{1}{2}\lambda_\mu^\star$. Ignoring order-unity prefactors, we find that the time-to-learn scales as:

$$\tau_\mu \approx \frac{1}{\lambda_\mu^\star} \ln \frac{\lambda_\mu^\star}{s_\mu^2(0)}. \tag{13}$$

Now we see that there's a preferential learning order. Learning is fast when the mode strength is large: the coarse, dominant structure of the data is learned first. Finer details, i.e. modes with smaller singular values, take longer.

Notice that the separation between τ_μ and $\tau_{\mu+1}$ widens as the scale of initialization shrinks. We observe this empirically as well: as the initialization gets smaller, the sigmoidal dynamics separate in time, resulting in sequential learning in distinct stages. This explains the cascading staircase loss curves; each plateau corresponds to a period where no new mode is transitioning, and each sudden drop corresponds to a new mode "switching on."

What is the dynamical reason for this? The landscape contains a series of saddle points, including one at the origin, and the optimization trajectory passes nearby many of them. As the initialization scale shrinks, the trajectory passes ever-nearer to these saddle points, slowing down considerably in their vicinity (since gradients near saddle points are small by definition). These dynamics are often referred to as "saddle-to-saddle" dynamics.

Ultimately, the saddle-to-saddle dynamics act as an implicit regularizer: gradient descent from small initialization prefers simple, low-rank solutions, incrementing the rank sequentially as optimization proceeds. Since modes are learned in strict order of magnitude, a network stopped partway through training realizes only the top few modes of Σ_{yx}, leaving the rest still pinned near zero. In learning problems where the large modes carry signal and the small modes carry noise, this implicit bias directly improves generalization (Advani and Saxe, 2017).

3 So what've we learned from all this?

We started with four pieces of deep learning folklore that seemed like clues to deeper understanding:

1. deep learning learns components of its target function in a preferred order;
2. deep learning optimizes just fine, despite having a wickedly nonconvex loss landscape;
3. instantaneous training dynamics are often low-rank within each weight matrix; and
4. we tend to get stepwise, staircase loss functions, especially when training from small init.

Deep linear nets capture all of these. So, what now? Did all this reveal any deeper insight into the fundamental reasons for these folklore observations?

By and large, yes! We can revisit each of these four folklore beliefs with a new perspective.

1. The preferred learning order is a function of the dataset. In particular, it depends on both the statistics of the inputs in isolation (i.e. Σ_{xx}) and the choice of target function (determined by Σ_{yx}). All else equal, *larger* components of the target function that deal with *larger directions in the input space* are learned *faster*. Real nonlinear multi-layer perceptrons share this bias towards more quickly learning functions of larger input directions (Karkada et al., 2025b).
2. The nonconvex loss landscape of a deep linear network decomposes into many low-dimensional subproblems that are solved in parallel. While this isn't going to hold exactly for deep nonlinear nets, it seems reasonable to suspect that a similar intuition applies, especially given the (also folklore) belief that large models are full of sparse circuits that operate semi-independently.
3. In deep linear networks, the instantaneous low-rank behavior is due to the fact that only one singular value (or a handful, if we're not training from very small initialization) is rapidly growing at a time. When each singular value finally grows, it creates a new path from the input to the output, resembling a new circuit suddenly forming in the network. It's possible that deep nonlinear networks show low-rank behavior for a similar reason: that only a relatively small number of "circuits" are undergoing rapid development at a particular time. Of course, this is highly speculative and this explanation should be handled with care until tested.
4. In deep linear networks, this stepwise learning behavior is due to the fact that the parameters are sliding down a sequence of saddle points of decreasing order before converging to a global minimum. These saddle-to-saddle dynamics are the same essential reason this behavior is sometimes seen in deep nonlinear models. Deep linear networks are now understood as a good toy model for this behavior.

Every toy model has limits, and deep linear networks obviously don't capture everything about deep learning we want to understand. In particular, any phenomena that require learning a nonlinear function of data—which ends up being *most* of what we want to understand!—just can't be captured by this model. That said, as you can hopefully appreciate now, deep linear networks are a surprisingly good toy model of a handful of important *dynamical* aspects of network training.