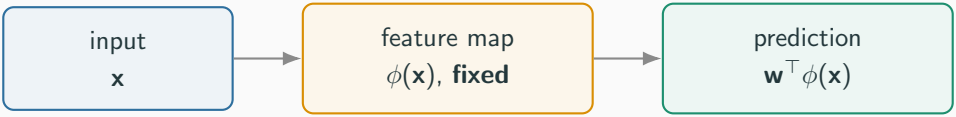
Neural Network Basics; Optimization for Deep Learning

What does a network learn, and why is fitting one so hard?

CSCI 567 — Machine Learning • Lecture 07 • Oct 23, 2026

Every model so far fixed the feature map; a neural network learns it



Linear and logistic regression, the Perceptron, kernels, SVM, PCA: every model before the exam fits this template, with ϕ chosen in advance, by hand or through a kernel.

A neural network is a linear model with a learned feature map.

One change: ϕ is learned by gradient descent, together with the final linear layer. That change makes the gradient flow through ϕ and makes the objective non-convex, which is what the rest of this lecture is about.

Outline

1. **Neural network basics**: from one neuron to an L-layer network, and backpropagation, the exact gradient at the cost of one forward pass.

2. **Activation functions**: why a nonlinearity is needed, and why ReLU replaced the sigmoid in hidden layers.

3. **Hard optimization and the optimizers built for it**: vanishing and exploding gradients, saddle points, and curvature, each paired with its fix: SGD, momentum, RMSProp, Adam, AdamW.

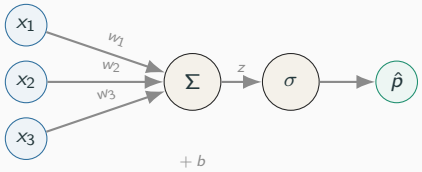
4. **Regularization in neural networks**: weight decay, early stopping, data augmentation, dropout.

5. **Inductive bias** (optional, not examined): does the feature map actually get learned (three animations), and what an architecture believes before training (Deep Image Prior).

1. Neural Network Basics

From one neuron to backpropagation

Logistic regression is one unit; K units side by side are a hidden layer



$\mathbb{P}(\mathbf{y} = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$: a weighted sum of the inputs, squashed by the sigmoid. This is one **unit** (neuron); everything below is built from copies of it.

Place K units side by side, each with its own weights and bias, all reading the same input $\mathbf{x} \in \mathbb{R}^d$:

$$z_j = \mathbf{w}_j^\top \mathbf{x} + b_j, \quad x_j^{(1)} = g(z_j), \quad j = 1, \ldots, K.$$

g is a nonlinearity (Section 2 says which); for now think of σ. With $W^{(1)} \in \mathbb{R}^{d \times K}$ holding the $\mathbf{w}_j$ as columns:

$$\mathbf{z}^{(1)} = (W^{(1)})^\top \mathbf{x} + b^{(1)} \in \mathbb{R}^K$$
$$\mathbf{x}^{(1)} = g(\mathbf{z}^{(1)}) \in \mathbb{R}^K$$

$\mathbf{x}^{(1)}$ is a **hidden layer**: K new numbers computed from $\mathbf{x}$, each one a pattern unit j detects.

The general L-layer network, and the bridge made literal

A linear model on top of the hidden layer, $\hat{\mathbf{y}} = \mathbf{w}^\top \mathbf{x}^{(1)} + b^{(2)}$, is a linear model on a feature map that was fit, not designed. Stacking L hidden layers, each reading the previous one:

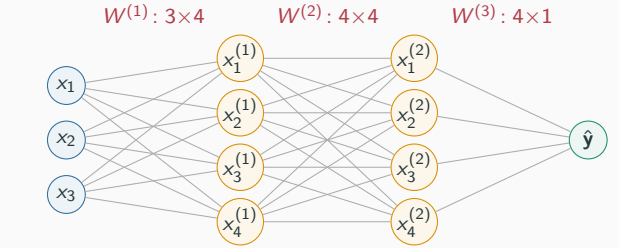
$$\mathbf{x}^{(0)} := \mathbf{x}, \qquad \mathbf{z}^{(l)} = \big(W^{(l)}\big)^\top \mathbf{x}^{(l-1)} + b^{(l)}, \qquad \mathbf{x}^{(l)} = g\big(\mathbf{z}^{(l)}\big), \qquad l = 1, \ldots, L,$$

then a linear output layer $\hat{\mathbf{y}} = f_\theta(\mathbf{x}) := \big(W^{(L+1)}\big)^\top \mathbf{x}^{(L)} + b^{(L+1)}$, with $\theta := \{W^{(l)}, b^{(l)}\}_{l=1}^{L+1}$. This is a **feedforward network** (multilayer perceptron).

$$f_\theta(\mathbf{x}) \;=\; \big(W^{(L+1)}\big)^\top \mathbf{x}^{(L)} \;+\; b^{(L+1)} \text{ is exactly } \mathbf{w}^\top \phi(\mathbf{x}) \text{ with}$$
$$\mathbf{w} \;:=\; W^{(L+1)} \text{ and learned feature map } \phi(\mathbf{x}) \;:=\; \mathbf{x}^{(L)}$$

For classification, replace the output layer by a softmax (Lecture 05) and the squared loss below by cross-entropy; nothing else changes.

The network as a picture



Every edge is a weight; every non-input node adds a bias and applies g (the output node is linear). This $d = 3$, $K = 4$, $L = 2$ network has 32 weights and 9 biases. The hidden layers are the learned feature map $\phi(\mathbf{x}) = \mathbf{x}^{(2)}$; the last four edges are $\mathbf{w}$. Width K and depth L are the two knobs: about K^2 weights per hidden layer, L layers.

The forward pass is cheap; a naive gradient is not

With hidden width K there are $O(K^2L)$ parameters. The **forward pass** runs the recursion for $l = 1, \ldots, L+1$ and **caches** every $\mathbf{z}^{(l)}, \mathbf{x}^{(l)}$: one $O(K^2)$ matrix-vector product per layer, $O(K^2L)$ in all.

Training needs $\nabla_\theta \ell_n$ for each example, with $\ell_n := \frac{1}{2}(\mathbf{y}_n - \hat{\mathbf{y}}_n)^2$. The naive way: nudge one parameter, rerun the forward pass, read off the change, repeat for every parameter.

$O(K^2L)$ parameters $\times$ $O(K^2L)$ per forward pass $=$ $O(K^4L^2)$ per gradient

Backpropagation gives the *exact* gradient for *every* parameter at the cost of about one more forward pass, paid for in memory: every cached activation must be kept

Backprop, step 1: the error signal, and its value at the output

For each layer define the **error signal**, the loss's sensitivity to that layer's pre-activations:

$$\delta^{(l)} := \frac{\partial \ell_n}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{d^{(l)}}, \qquad l = 1, \ldots, L+1,$$

with $d^{(l)}$ the width of layer l ($d^{(L+1)} = 1$ for a scalar output). The plan: compute the $\delta^{(l)}$ backward from the output, one layer at a time; every weight gradient then follows in one multiplication.

Output layer. It is linear, $\hat{\mathbf{y}}_n = z^{(L+1)}$, so

$$\delta^{(L+1)} = \frac{\partial}{\partial z^{(L+1)}} \left[\frac{1}{2} (\mathbf{y}_n - z^{(L+1)})^2 \right] = \hat{\mathbf{y}}_n - \mathbf{y}_n,$$

the residual, the same signal that drove linear regression in Lecture 02. With softmax and cross-entropy it is $\hat{\mathbf{p}} - \mathbf{e}_{\mathbf{y}_n}$, again a residual.

Backprop, step 2: one chain-rule step per layer

Suppose $\delta^{(l+1)}$ is known. $z_i^{(l)}$ affects the loss only through $x_i^{(l)} = g(z_i^{(l)})$ and then through every $z_j^{(l+1)} = \sum_i W_{ij}^{(l+1)} x_i^{(l)} + b_j^{(l+1)}$ it feeds, so the chain rule sums over those paths:

$$\frac{\partial \ell_n}{\partial z_i^{(l)}} = \sum_{j=1}^{d^{(l+1)}} \underbrace{\frac{\partial \ell_n}{\partial z_j^{(l+1)}}}_{=\delta_j^{(l+1)}} \cdot \underbrace{\frac{\partial z_j^{(l+1)}}{\partial x_i^{(l)}}}_{=W_{ij}^{(l+1)}} \cdot \underbrace{\frac{\partial x_i^{(l)}}{\partial z_i^{(l)}}}_{=g'(z_i^{(l)})} .$$

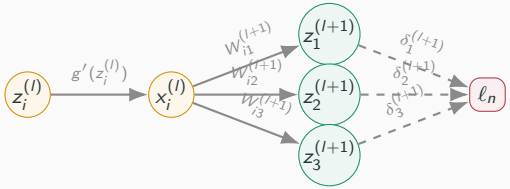
The sum is the i-th entry of $W^{(l+1)}\delta^{(l+1)}$, and the last factor is common to all paths:

Backward recursion.

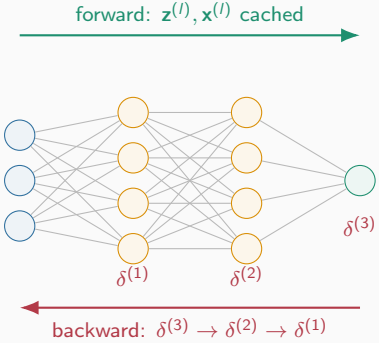
$$\delta^{(l)} = \left(W^{(l+1)}\delta^{(l+1)}\right) \odot g'\left(\mathbf{z}^{(l)}\right), \qquad \odot = \text{elementwise product}$$

Start from $\delta^{(L+1)} = \hat{\mathbf{y}}_n - \mathbf{y}_n$ and apply this for $l = L, \ldots, 1$, with g' evaluated at the cached $\mathbf{z}^{(l)}$.

The same two steps on the picture



One unit's fan-out: $z_i^{(l)}$ reaches the loss through $x_i^{(l)}$ and then through each $z_j^{(l+1)}$. One product of three factors per path, summed. The dashed edges are everything downstream, already summarized by $\delta_j^{(l+1)}$.



The whole algorithm: one left-to-right sweep that stores every layer's $\mathbf{z}^{(l)}, \mathbf{x}^{(l)}$, then one right-to-left sweep producing $\delta^{(3)}, \delta^{(2)}, \delta^{(1)}$, each from the one to its right.

Backprop, step 3: parameter gradients, and the total cost

Since $z_j^{(l)} = \sum_i W_{ij}^{(l)} x_i^{(l-1)} + b_j^{(l)}$, each weight's gradient is the δ at its head times the cached activation at its tail:

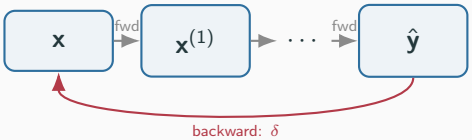
$$\frac{\partial \ell_n}{\partial W_{ij}^{(l)}} = \delta_j^{(l)}\, x_i^{(l-1)}, \qquad \frac{\partial \ell_n}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

In matrix form, $\partial \ell_n / \partial W^{(l)} = \mathbf{x}^{(l-1)} (\delta^{(l)})^\top$, an outer product of two vectors already in hand.



Cost. Each $\delta^{(l)}$ is one matrix-vector product, $O(K^2)$, so the backward pass costs $O(K^2 L)$, the same as the forward pass. The full gradient costs about two forward passes, not one per parameter.

Backprop is reverse-mode automatic differentiation; the price is memory



every $\mathbf{x}^{(l)}, \mathbf{z}^{(l)}$ from the forward pass must be kept until the backward pass uses it: backprop trades memory for compute

Checked numerically (Colab, Part 1): a from-scratch implementation agrees with central differences to a relative error of 1.8×10^{-10} on all 37 parameters of a small network.

Nothing here is specific to neural networks. Any composition of differentiable maps with many inputs and one scalar output can be differentiated this way: one backward sweep gives the derivative with respect to every input, because each intermediate sensitivity is computed once and reused by everything upstream. This is **reverse-mode automatic differentiation**, and it is what PyTorch and JAX run.

Forward mode pushes a derivative forward through the same graph, one input direction at a time. It is cheap only with few inputs, the opposite of training, where inputs are the millions of parameters and the output is one loss.

2. Activation Functions

Why nonlinearity, and which one

Without a nonlinearity, depth buys nothing

Nothing so far required g to be nonlinear. Set $g(\mathbf{z}) = \mathbf{z}$ at every layer and see what depth buys.

Fact. A composition of affine maps is affine. With g = identity, a network of any depth computes an affine function of $\mathbf{x}$, which linear regression already does.

Proof. For affine maps $h_1(\mathbf{x}) = A_1\mathbf{x} + c_1$ and $h_2(\mathbf{x}) = A_2\mathbf{x} + c_2$,

$$h_2(h_1(\mathbf{x})) = A_2(A_1\mathbf{x} + c_1) + c_2 = (A_2A_1)\mathbf{x} + (A_2c_1 + c_2),$$

again affine. By induction the network's $L + 1$ affine layers collapse to a single $\tilde{W}^\top\mathbf{x} + \tilde{b}$. ■

Every gain from depth comes from g bending the input at each layer. The next two slides ask which bend to use, and the answer is decided by g', not g.

Common choices: sigmoid, tanh, ReLU, Leaky ReLU

Sigmoid. $\sigma(z) = 1/(1+e^{-z})$, range $(0,1)$. $\sigma'(z) = \sigma(z)(1-\sigma(z)) \leq \frac{1}{4}$, with equality only at $z = 0$.

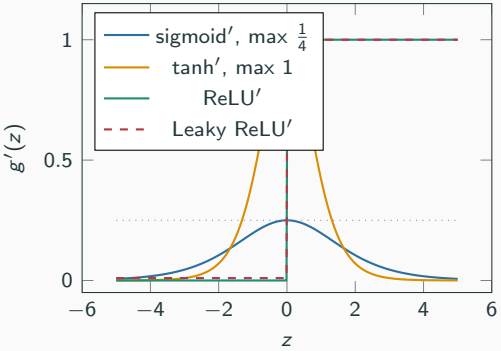
Tanh. $\tanh(z) = 2\sigma(2z) - 1$, range $(-1,1)$. $\tanh'(z) = 1 - \tanh^2(z) \leq 1$, equality only at $z = 0$.

ReLU. $g(z) = \max(0, z)$. $g'(z) = \mathbb{1}[z > 0]$: exactly 1 for $z > 0$, exactly 0 for $z < 0$.

Leaky ReLU. $g(z) = \max(0.01z, z)$: a small slope instead of 0 for $z < 0$, so no unit is ever fully dead.

All four act elementwise. Sigmoid remains the right choice at an *output* that must be a probability; the question here is hidden layers, where only g' matters, because g' is what the backward recursion multiplies by.

The derivatives, and saturation



Definition. g **saturates** in a direction if $g'(z) \to 0$ as z moves that way.

- Sigmoid and tanh saturate on *both* sides: a unit with large $|z|$ passes almost no gradient, so the weights feeding it stop moving. Sigmoid also shrinks every gradient by at least 4, saturated or not.
- ReLU saturates only for $z < 0$ (a "dead" unit) and passes gradients unchanged for $z > 0$. A dead unit never recovers; Leaky ReLU's 0.01 floor lets it.

The next section multiplies L of these factors together.

3. Hard Optimization, and the Optimizers Built for It

Three problems, each with its fix

An exact gradient is necessary, but not sufficient: three difficulties

Every objective so far (MSE, cross-entropy, hinge, each with a convex regularizer) was *convex*: any critical point is a global minimum, and gradient descent with a sensible step gets there.

A network's loss is **non-convex** in θ. Permuting the hidden units leaves f_θ unchanged, so a minimizer has $K!$ copies with equal loss; convexity would force their average to be at least as good, and it generically is not.

Three difficulties follow, each paired below with its fix:

1. **Vanishing and exploding gradients** $\rightarrow$ activations, initialization, residuals.
2. **Plateaus around saddle points** $\rightarrow$ mini-batch noise and momentum (bad local minima are a modeling problem: architecture, pre-training, curriculum).
3. **Curvature** caps the step size and differs across layers $\rightarrow$ momentum, then per-parameter steps (RMSProp, Adam, AdamW).

Vanishing gradients: a product of L factors

$$\delta^{(l)} = \left(W^{(l+1)}\delta^{(l+1)}\right) \odot g'\left(\mathbf{z}^{(l)}\right)$$



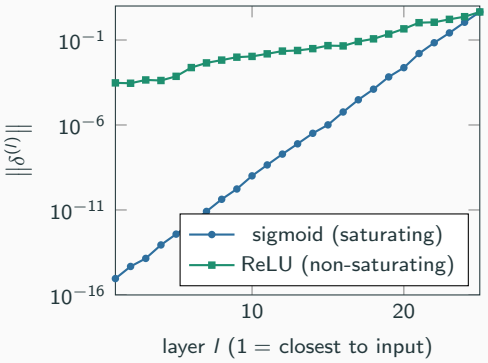
By the time the recursion reaches an early layer it has multiplied by g' about L times. If every g' factor is at most $c < 1$, the signal shrinks like c^L: geometrically in depth.

Sigmoid units. $c = \frac{1}{4}$ at best, at every layer. Even if the weight matrices contribute exactly 1, a 20-layer sigmoid network's first-layer gradient is shrunk by at least

$$\left(\tfrac{1}{4}\right)^{20} = \tfrac{1}{2^{40}} \approx 9 \times 10^{-13},$$

a factor of a trillion, from the activations alone. In practice the weights make it worse, not better.

Measured: error-signal norm by layer, sigmoid vs. ReLU



A straight line on the log axis is geometric decay.

Setup (Colab, Part 2). A 25-layer, width-20 network with the *same* weights for both activations, drawn at scale $\sqrt{1/K}$ so the weights alone neither shrink nor grow the signal. The backward pass starts from the same $\delta^{(25)}$ and runs the recursion layer by layer.

	sigmoid	**ReLU**
$\|\delta^{(25)}\|$	4.5	4.5
$\|\delta^{(1)}\|$	9.2×10^{-16}	3.0×10^{-4}

Sigmoid loses 15 orders of magnitude over 25 layers; ReLU loses 4, because about half its units are inactive at any layer. At the first layer ReLU's signal is 3×10^{11} times larger.

Exploding gradients, the symptom, and the three remedies

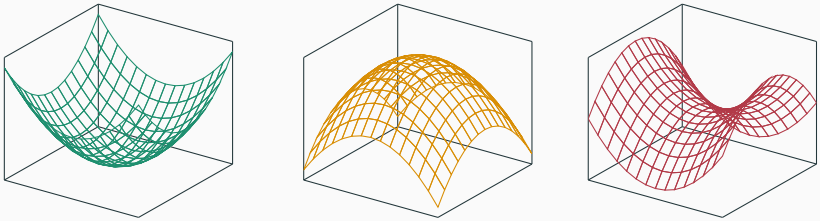
If the weight matrices consistently have singular values above 1, the same product **grows** geometrically and updates become large enough to blow training up. Either way the cause is the chain rule multiplying many factors. The symptom: with vanishing gradients the early layers train far too slowly or not at all; with exploding gradients the loss diverges.

Three remedies, one per place a factor can be fixed:

- **Activation choice**: a non-saturating g removes the systematic $c < 1$ factor (this lecture).
- **Weight initialization**: scale each $W^{(l)}$ so that neither the weight nor the activation factors start out systematically above or below 1 (Discussion 07, Xavier and He).
- **Residual connections**: give the gradient an additive path that bypasses the product (Discussion 07, and Lecture 09's Transformers).

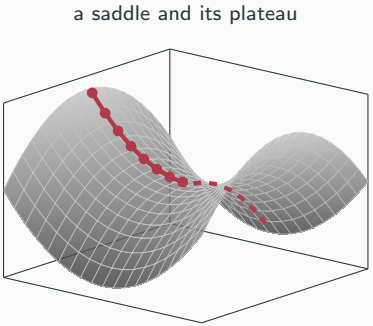
Critical points: minimum, maximum, or saddle, decided by the curvatures

$x^2 + y^2$: $(+,+)$, minimum $\quad -(x^2 + y^2)$: $(-,-)$, maximum $\quad x^2 - y^2$: $(+,-)$, saddle

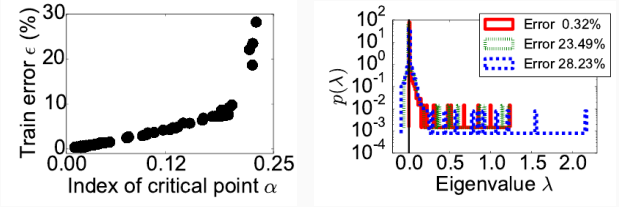


At a critical point ($\nabla \ell = \mathbf{0}$) look at the curvature along each direction. All up: a local minimum. All down: a maximum. Mixed: a **saddle**, which still descends somewhere, so it slows gradient descent but cannot trap it. In d dimensions there are 2^d sign patterns and only two are not saddles, so for d in the millions the saddle is the generic critical point; the next slide measures this.

Dauphin et al. (2014): the generic obstacle is the plateau around a saddle



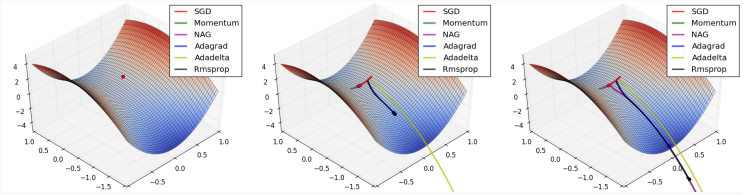
The red path rolls down the wall, then crawls across the flat plateau around the saddle (steps shrink with the gradient) until the descending direction (dashed) takes over. Nothing traps it; it is only slow.



Dauphin et al. (2014), Fig. 1(a),(b), MNIST: critical points of a trained net, error vs. *index* α (fraction of directions curving down); curvature histograms at three of them.

Argument. Treat the d curvature signs as coin flips, more often negative at higher loss: all d non-negative (a true minimum) is exponentially unlikely except near the bottom.
Measurement. Higher error, higher index, on one rising curve; only the near-zero-error points are minima ($\alpha \approx 0$ at 0.32%), everything above is a saddle, and every histogram piles up at $\lambda \approx 0$: the plateau's flat directions.

The fix for saddles: noise and momentum carry the iterate off the plateau

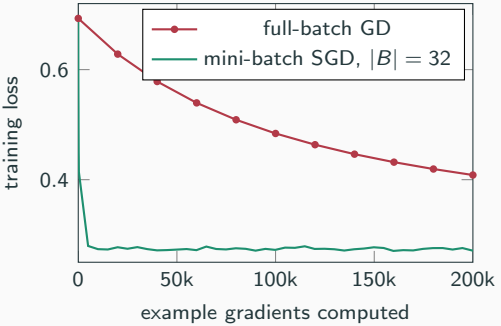


Alec Radford's animation, via Ruder (2016): start, one third, end. A saddle, flat along one axis, curving down along the other; all optimizers from the same point. **Click the picture to open the GIF.**

Plain gradient descent sits on the flat axis: a tiny gradient means a tiny step. Three things get it moving, and the rest of this section introduces each: **mini-batch noise** (a noisy gradient is never exactly zero along the descending direction), **momentum** (velocity from the way down carries it across), and **per-parameter steps** (a tiny gradient divided by its own tiny size is a full step).

Bad local minima do exist, but they are fought with the model, not the optimizer: architecture, pre-training, curriculum, a larger learning rate. The optimizer's job is to cross plateaus fast.

SGD: trade one exact gradient for hundreds of cheap, noisy ones



Logistic regression, $n = 20{,}000$, $d = 20$, same $\eta = 0.5$. After one pass over the data GD is at 0.63 and SGD at 0.27, a value GD reaches only after 40 passes.

The full gradient $\nabla\hat{\mathcal{R}}(\mathbf{w}) = \frac{1}{n}\sum_i \nabla\ell_i(\mathbf{w})$ costs one backprop per example; with n in the millions, one step is a pass over the data.

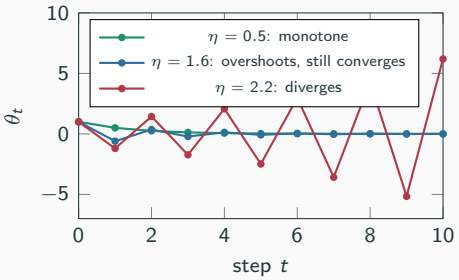
Mini-batch SGD uses a random batch B of 32 to a few thousand examples instead:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta\, g_t, \quad g_t = \frac{1}{|B|}\sum_{i\in B} \nabla\ell_i(\mathbf{w}_{t-1}).$$

one step costs a batch, not n; a batch is one matrix product per layer, which GPUs do well; and the noise carries the iterate off plateaus a full gradient would sit on

The noise also means SGD never settles exactly, so η is **decayed** over training. Every method below keeps g_t and changes only what is done with it.

Curvature: the steepest direction caps the step size, so the flattest one crawls



Gradient descent on a bowl of curvature $\beta = 1$ from $\theta_0 = 1$: each step multiplies θ by $1 - \eta\beta$, so the step overshoots for $\eta > 1/\beta$ and the overshoots grow for $\eta > 2/\beta$.

A gradient step assumes the slope holds for the whole step; where the loss curves up sharply the step overshoots the bottom.

the step size must stay below about $2/\beta_{\max}$, $\beta_{\max}$ the curvature of the **steepest** direction

The same η is then used along the flattest direction, curvature $\beta_{\min}$, where the error shrinks by only $1 - \beta_{\min}/\beta_{\max}$ per step: crossing it takes on the order of $\kappa := \beta_{\max}/\beta_{\min}$ steps, the **condition number**. A long narrow valley is the picture: steep walls force tiny steps, tiny steps crawl along the floor (Rosenbrock, below: $\kappa \approx 2500$).

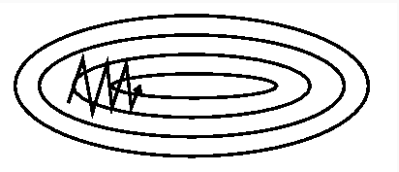
SGD with momentum (Polyak, 1964)

Momentum keeps a running velocity instead of using each gradient once:

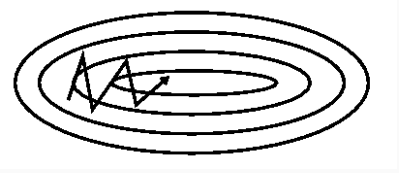
$$v_t = \rho\, v_{t-1} + g_t, \qquad \mathbf{w}_t = \mathbf{w}_{t-1} - \eta\, v_t,$$

$v_0 = \mathbf{0}$, $\rho \in [0, 1)$, commonly 0.9.

Unrolled, $v_t = \sum_{s<t} \rho^s g_{t-s}$: a sum of recent gradients with memory about $1/(1-\rho) = 10$ steps. Gradients that flip sign across the valley **cancel** inside v_t; gradients that keep their sign along it **add up**, to $10\times$ a single gradient.



SGD: zig-zags across the valley

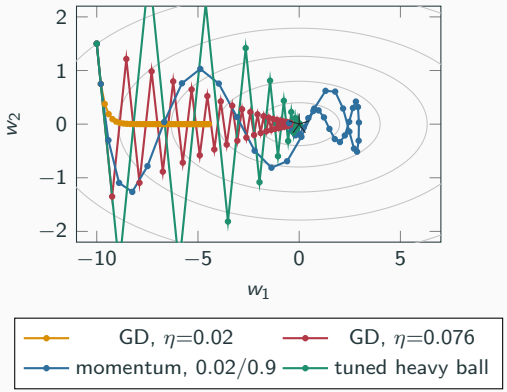


SGD with momentum: the ball rolls along it

Figures from Ruder (2016).

With ρ and η tuned to the valley it needs about $\sqrt{\kappa}$ steps where SGD needs κ (Polyak 1964): at $\kappa = 2500$, 50 times fewer. The next slide shows it.

Seen on an ill-conditioned bowl: crawl, zig-zag, or momentum

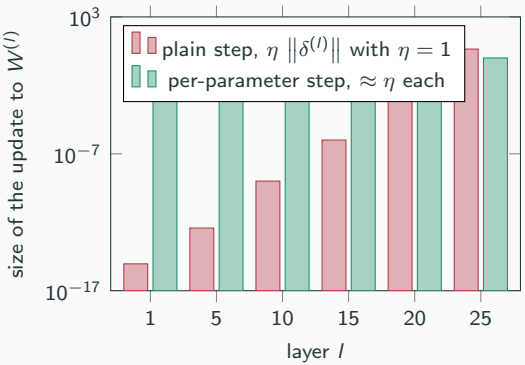


$f(\mathbf{w}) = \frac{1}{2}(w_1^2 + 25\,w_2^2)$, so $\kappa = 25$; every run starts at $(-10, 1.5)$ and takes 40 steps (full gradients, so plain GD).

	$f(\mathbf{w}_{40})$	to 10^{-2}
GD, $\eta = 0.02$	9.9	> 100
GD, $\eta = 0.076$	0.096	55
momentum, 0.02/0.9	0.13	61
tuned heavy ball	3×10^{-10}	19

At the safe step GD crawls; momentum at the same step is $80\times$ lower after 40 steps. GD only keeps up at its ceiling, where it zig-zags (red). Polyak's tuning ($\rho = 0.44$, $\eta = 0.11$) reaches 10^{-2} in 19 steps against GD's best 55.

Problem: one step size, but gradient scales that differ by 10^{15} across layers



The sigmoid network's measured $\left\|\delta^{(l)}\right\|$ from earlier, read as the size of each layer's update under one global step size.

Momentum, like SGD, multiplies every coordinate's gradient by the *same* η. But a gradient's scale reflects where the parameter sits, not how far it should move: in the deep sigmoid network the first layer's gradient is 10^{15} times smaller than the last's.

any η small enough for the last layer leaves the first layer frozen; any η large enough for the first layer blows the last one up

The same mismatch appears, less extremely, between weights and biases, between wide and narrow layers, and over training. The cure is to give each parameter its own step.

RMSProp (Hinton, 2012): a step size per parameter

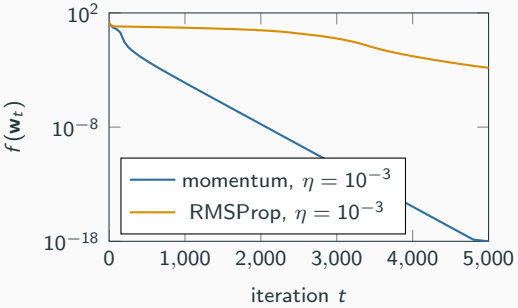
RMSProp divides each coordinate's gradient by a running estimate of its own size:

$$v_t = \beta\, v_{t-1} + (1-\beta)\, g_t^2, \qquad \mathbf{w}_t = \mathbf{w}_{t-1} - \eta\, \frac{g_t}{\sqrt{v_t} + \epsilon} \qquad \text{(elementwise)},$$

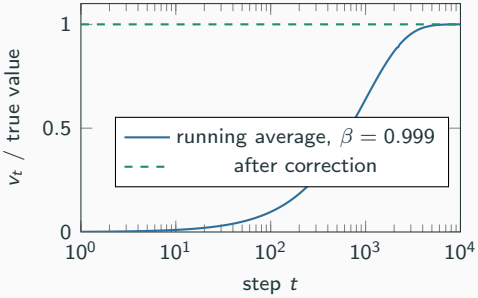
Hinton's defaults $\beta = 0.9$, $\eta = 10^{-3}$; $\epsilon \approx 10^{-8}$ guards the division.

- $\sqrt{v_t}$ is the recent typical size of each coordinate's gradient, so $g_t/\sqrt{v_t}$ is about ± 1: **every parameter moves by about η per step**, whether its gradients are 10 or 10^{-3}. A layer with vanishing gradients is no longer left behind.
- Without momentum it moves a constant $\approx \eta$ per coordinate, so it cannot accelerate along a valley. (It fixes Adagrad, whose sum of *all* past squared gradients drove its steps to zero.)

Problem: RMSProp cannot accelerate, and its first steps are miscalibrated



Rosenbrock valley: RMSProp advances a fixed $\approx 10^{-3}$ per coordinate per step and is still at $f = 1.6 \times 10^{-3}$ after 5000 steps; momentum is at 10^{-19}.



A running average started at 0 underestimates: at $t = 1$ it holds 0.1% of the true g^2 and needs about 3000 steps to reach 95%. Dividing by too small a $\sqrt{v_t}$ makes the first steps far too large.

Two gaps: RMSProp has the per-parameter scale but not momentum's accumulation along a valley, and any running average started at 0 is biased for its first few thousand steps. Adam closes both.

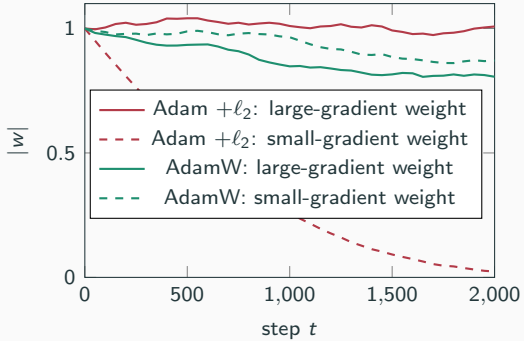
Adam (Kingma and Ba, 2015): momentum plus RMSProp

Adam keeps both running averages, the gradient (m_t, momentum) and its square (v_t, RMSProp), and takes the momentum step scaled per parameter:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \qquad \mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$
$$m_0 = v_0 = \mathbf{0}; \text{ defaults } \beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 10^{-8},\ \eta = 10^{-3}.$$

- **Bias correction.** Both averages start at $\mathbf{0}$, so for the first steps they are too small by the factor $1-\beta^t$ (at $t=1$, v_1 is only 0.1% of g_1^2). Dividing by $1-\beta^t$ undoes this exactly; after a few thousand steps it does nothing.
- Each coordinate moves by about η per step whatever its gradient's scale, so η is set on the scale of the weights; 10^{-3} is a good default. Cost: two extra vectors the size of θ.

Problem: ℓ_2 regularization inside Adam is not weight decay



Two weights starting at 1, zero-mean gradient noise of scale 10 and 0.01, $\lambda = 0.1$, $\eta = 10^{-3}$, defaults otherwise. Pure decay gives $(1 - \eta\lambda)^{2000} = 0.82$.

Lecture 03's penalty $\frac{\lambda}{2} \|\mathbf{w}\|^2$ adds $\lambda\mathbf{w}$ to the gradient. For SGD that is **weight decay**: every weight shrinks by the factor $1 - \eta\lambda$ each step.

inside Adam the $\lambda\mathbf{w}$ term is divided by $\sqrt{\hat{v}_t}$ like everything else: a weight with large gradients gets almost no decay, a weight with small gradients gets crushed

On the left, the large-gradient weight ends at 1.01 (no decay at all) and the small-gradient one at 0.02, where true decay would leave both near 0.82. The regularizer no longer means what it says, and η and λ cannot be tuned separately.

AdamW (Loshchilov and Hutter, 2019): weight decay done right

The fix is to take the decay out of the gradient entirely.

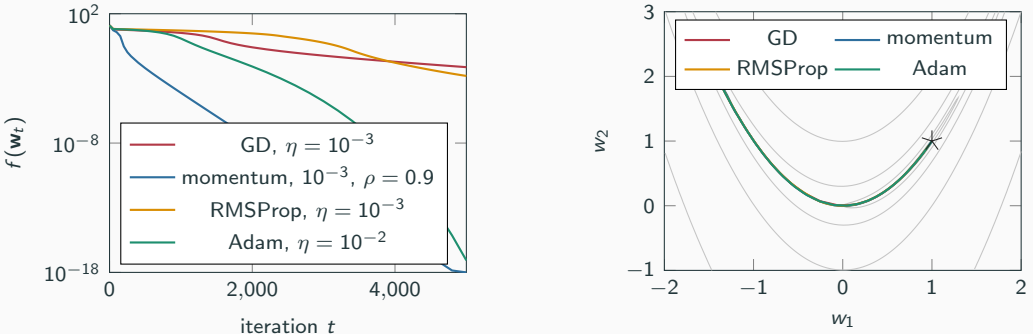
AdamW: keep λ out of the gradient and apply the decay to the weights directly,

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta\Big(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda\, \mathbf{w}_{t-1}\Big),$$

with m_t, v_t computed from the data gradient alone.

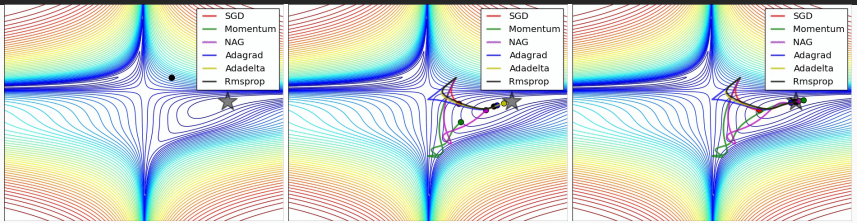
Every weight now shrinks by the same factor $1 - \eta\lambda$ per step regardless of its gradient history (both weights on the previous slide end near 0.8), and η and λ can be tuned separately. AdamW is the default optimizer for training transformers and most other large models today; typical λ is 0.01 to 0.1.

Measured: all four on the Rosenbrock valley, 5000 steps from the same start



$f(x, y) = (1 - x)^2 + 100\,(y - x^2)^2$ (Colab, Part 3), from $\mathbf{w}_0 = (-1.5, 2.5)$; grey: level sets of $\log_{10} f$; star: the minimum $(1, 1)$. Final loss: GD 7×10^{-3}, still creeping along the floor; RMSProp 2×10^{-3}, steady $\approx \eta$ per coordinate; momentum 3×10^{-19} after one overshoot of the floor (the loop near the start); Adam 8×10^{-18}, equal-length steps that turn the corner cleanly. GD is only slow, not wrong: 50,000 steps also reach 10^{-18}, and $\eta = 5 \times 10^{-3}$ diverges within 10 steps.

The same optimizers on the Beale function, and which one to use



Alec Radford's animation, via Ruder (2016); start, one third, end: the Beale function's contours, all optimizers from the same point (star: the minimum). **Click the picture to open the GIF.**

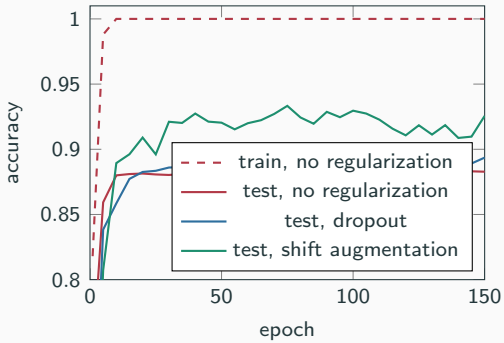
Momentum overshoots like a rolling ball and comes back; the per-parameter methods head straight for the minimum; plain SGD is still on its way at the end.

Which one. AdamW for transformers and most large models; Adam without weight decay; SGD with momentum and a schedule in computer vision (cheaper in memory). RMSProp survives mainly inside Adam.

4. Regularization in Neural Networks

Weight decay, early stopping, data augmentation, dropout

A network can memorize its training set; the toolbox that stops it



A $784 \to 512 \to 512 \to 10$ ReLU network trained with Adam on 500 MNIST digits, tested on 10,000. Training accuracy hits 100% by epoch 10; test accuracy stalls at 88%. Dropout (0.2 input, 0.5 hidden) gives 89%, random ± 2-pixel shifts 93%.

An overparameterized network fits its training set exactly (Lecture 04); what it does elsewhere is decided by what else constrains it. Four tools, two of them new:

- **Weight decay**: Lecture 03's ℓ_2, applied as AdamW does it.
- **Early stopping**: keep the checkpoint with the best validation score.
- **Data augmentation**: more training data from label-preserving views.
- **Dropout**: randomly silence units during training.

Normalization (Discussion 07) regularizes only as a side effect (Ioffe and Szegedy 2015).

Data augmentation: tell the network which changes do not change the label



One training digit and five label-preserving views: shifted, rotated $\pm 15°$, with pixel noise, with 30% of pixels erased. Each epoch sees a fresh random view.

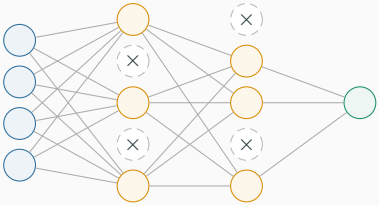
Each transformation is prior knowledge: "a shifted 0 is still a 0," which 500 examples cannot teach on their own, supplied the way a convolution supplies translation invariance. Images: crops, flips, rotations, color jitter, erasing; audio: time shifts, noise; text: word dropout, back-translation. AlexNet (Krizhevsky et al. 2012) trained on random crops and flips and overfit substantially without them.



mixup: $0.6 \cdot$ "0" $+ 0.4 \cdot$ "9"

Modifying the targets too. *mixup* (Zhang et al. 2018) trains on convex combinations of pairs of examples *and* their labels: the mixed digit gets the soft target $(0.6, 0.4)$. *Label smoothing* puts $1 - \varepsilon$ on the true class and spreads ε over the rest. Both stop the logits from running off to infinity.

Dropout (Srivastava et al., 2014): train a random sub-network each step



One training step: the crossed units are switched off, together with their edges; the next step draws a different mask.

During training, multiply each hidden activation by an independent coin flip:

$$\tilde{\mathbf{x}}^{(l)} = \mathbf{x}^{(l)} \odot \frac{\mathbf{m}}{1-p}, \qquad m_j \sim \text{Bernoulli}(1-p),$$

p = 0.5 for hidden layers, about 0.2 for inputs; at test time use the full network with no mask.

Why it works. A unit cannot rely on a specific partner, so units stop *co-adapting*. Every mask is a sub-network; training samples exponentially many with shared weights, and the test-time network averages them cheaply (the $1/(1-p)$ keeps the expected activation fixed): Lecture 08's bagging inside one model.

5. Inductive Bias (optional)

What the architecture already believes

Not examined; read it for the ideas behind Lectures 09 and 10.

From implicit regularization (the optimizer) to inductive bias (the architecture) (optional)

Recall Lecture 04. When many $\mathbf{w}$ fit the training data exactly, gradient descent from $\mathbf{w}_0 = \mathbf{0}$ does not pick an arbitrary one:

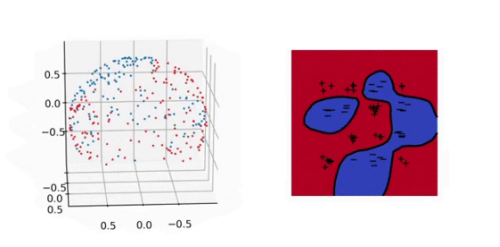
> it converges to the **minimum-norm** interpolating solution, a property of the optimizer's path, not of the objective

That was **implicit regularization**: the algorithm, not the loss, picks the solution. A network adds a second source of preference that exists before any training happens:

> **Inductive bias.** The functions a model class represents cheaply and reaches early under gradient descent, as fixed by the *architecture*, before seeing any data.

Two questions, answered by watching training: is the feature map learned at all (three

Does the feature map get learned? Experiment 1: train only the output layer (optional)



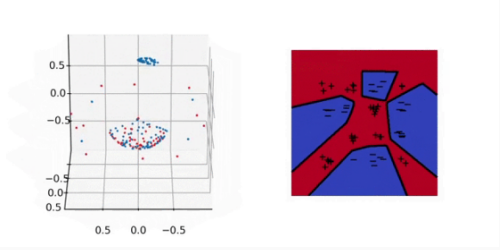
Animation by L. Chizat (Chizat and Bach 2020, francisbach.com); plays in Acrobat, else the final frame shows. **Click the picture to open the GIF.** Strip: start, middle, end.

Setup. A wide two-layer ReLU network on a 2-D classification task, trained by gradient descent on the exponential loss with the hidden weights frozen at their random values and only the output weights trained. Left: one particle per hidden neuron at $|a_j|\, b_j$, red or blue by the sign of its output weight, the unit sphere drawn as "infinity". Right: the training points, the predictor's sign in color, the boundary in black.

Observation. The particles' directions never change; only their distance from the origin grows, while the boundary settles into a smooth curve around the clusters.

Takeaway. With the hidden layer frozen this is Lecture 04's random-feature kernel machine, heading for the kernel's max-margin classifier. No

Experiment 2: train both layers from a small initialization (optional)



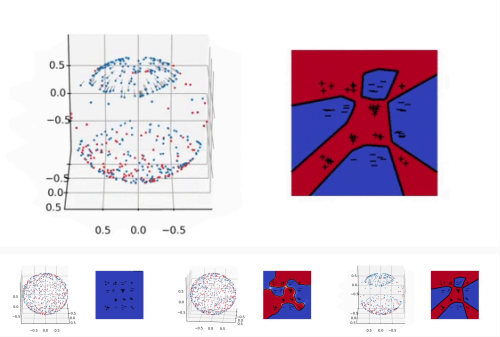
Animation by L. Chizat (Chizat and Bach 2020, francisbach.com); plays in Acrobat, else the final frame shows. **Click the picture to open the GIF.** Strip: start, middle, end.

Setup. Same data, network, and loss; now both layers are trained, starting from hidden weights spread over a small sphere.

Observation. The particles move: they leave the sphere and gather into a handful of clusters, one per learned direction, while the boundary becomes polygonal with one edge per cluster. The particles at the bottom started with a bias too negative to ever activate, and never move.

Takeaway. The hidden layer has learned features, and the predictor is of a different kind: Chizat and Bach (2020) show it converges to a max-margin classifier for a norm that charges the total weight of the ReLUs used, an ℓ_1-type preference for few ReLUs. The clustering is that preference made visible; no fixed kernel gives such sharp, sparse boundaries cheaply.

Experiment 3: the same run from a larger initialization: lazy first, features later (optional)



Animation by L. Chizat (Chizat and Bach 2020, francisbach.com); plays in Acrobat, else the final frame shows. **Click the picture to open the GIF.** Strip: start, middle, end.

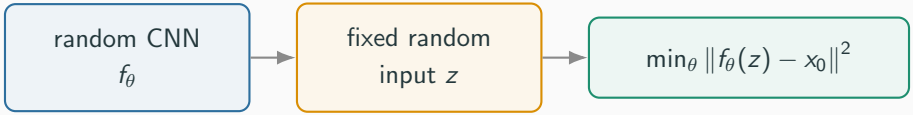
Setup. Identical to Experiment 2, except that the hidden weights start on a larger sphere and the step size is reduced to match.

Observation. For a long early phase the particles do not visibly move while the boundary already separates the data smoothly; later they move, cluster, and the boundary turns polygonal, the same end state as Experiment 2.

Takeaway. The initialization scale selects the regime. A large start gives the *lazy regime*: the network acts as a linear model in its parameters, a kernel machine for the neural tangent kernel (Jacot et al. 2018; Chizat, Oyallon, Bach 2019). A small start gives feature learning. Here the lazy phase is transient, so one run shows both. Caveats: two-layer ReLU nets, infinite width, qualitative.

Deep Image Prior (Ulyanov et al., 2018): fit a random network to one image (optional)

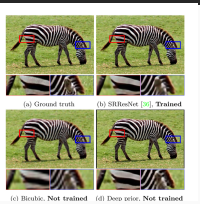
Take one corrupted image x_0: noisy, or with a hole, or downsampled. Take a convolutional network f_θ with random weights and a fixed random input z, and fit θ by gradient descent on the usual squared error:



one image, no labels, no training set, **ever**: the only data the procedure sees is the corrupted image it is reconstructing

Given enough steps the overparameterized network reproduces the corruption too. The finding is about the *path*: stopped at the right time, $f_\theta(z)$ is a clean version of x_0, because natural-image structure is reached early and the corruption only late.

The result: an untrained network matches a trained one (optional)

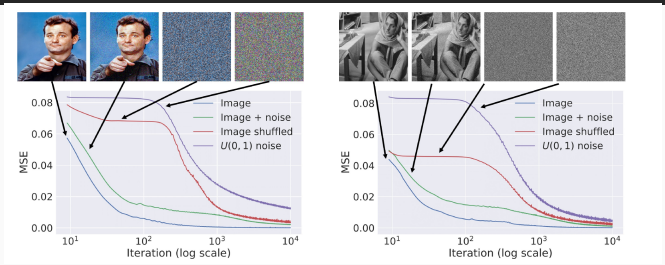


Ulyanov, Vedaldi, Lempitsky (2018), Figure 1: 4× super-resolution. (a) ground truth, (b) SRResNet, **trained** on a large dataset, (c) bicubic upsampling, (d) deep image prior, not trained.

(d) has seen zero training images, yet stopped at the right time it looks close to (b) and far sharper than (c).

The convolutional architecture alone encodes a strong prior over natural images. Local filters, shared across positions and composed across scales, make image-like outputs far easier for gradient descent to reach than noise. The optimizer decides how fast a network finds *some* good weights; the architecture decides what "good" means.

Why stopping early works: the paper's learning curves (optional)



Ulyanov, Vedaldi, Lempitsky (2018), Figure 4: reconstruction error vs. iteration (log scale) when the same random network is fit to a natural image, the image plus noise, the image with its pixels shuffled, and pure noise; two images, two panels.

The clean image is fit fastest; noise slows the fit; shuffled pixels (same histogram, no spatial structure) and pure noise are slowest, with a long plateau before the network memorizes them. So on a corrupted image the path passes close to the clean version early and reaches the corruption only much later, and stopping in between returns the restoration. Nothing in the objective tells signal from noise; the architecture's prior does.

6. Wrap-up

One causal arc, start to finish

The whole arc

1. **NN basics**: a network is $\mathbf{w}^\top \phi(\mathbf{x})$ with ϕ learned. Backprop computes the exact gradient for the cost of about two forward passes, by running $\delta^{(l)} = (W^{(l+1)}\delta^{(l+1)}) \odot g'(\mathbf{z}^{(l)})$ backward.

2. **Activations**: without a nonlinearity depth collapses to one affine map. Sigmoid saturates and shrinks every gradient by 4; ReLU passes gradients unchanged.

3. **Hard optimization, and its fixes**: the backward product shrinks like c^L or explodes (activations, initialization, residuals); critical points in $\mathbb{R}^d$ are saddles whose plateaus slow training (mini-batch noise, momentum); the steepest curvature caps the step size and curvature differs across layers (momentum's $\sqrt{\kappa}$; RMSProp, Adam, and AdamW give each parameter its own step).

4. **Regularization**: an overparameterized network memorizes; weight decay, early stopping, data augmentation (label-preserving views, a prior in data form), and dropout (random sub-networks, an ensemble in one model) keep the test error down.

5. **Inductive bias** (optional): the initialization scale decides whether the hidden layer learns

Coming up

Discussion 07: Architecture Components Practicum. Convolutions, layer normalization, residual connections, and the Xavier/He initialization formulas that address this lecture's vanishing and exploding gradients directly.

HW3 is announced today, due Mon 11/2.

Lecture 08: Ensemble Methods. Bagging, boosting, gradient boosting, and the neural-network versions (parameter averaging, mixture of experts) this lecture makes possible.

This lecture's Colab notebook (`nn_backprop_optimization_demo.ipynb`) is the source of every "measured" slide: NumPy only, fixed seeds, so every number can be rerun.

References

- Polyak. *Some Methods of Speeding Up the Convergence of Iteration Methods.* USSR Computational Mathematics and Mathematical Physics, 4(5), 1964.
- Kingma, Ba. *Adam: A Method for Stochastic Optimization.* ICLR 2015; arXiv:1412.6980.
- Dauphin, Pascanu, Gulcehre, Cho, Ganguli, Bengio. *Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization.* NeurIPS 2014, pp. 2933–2941.
- Ulyanov, Vedaldi, Lempitsky. *Deep Image Prior.* CVPR 2018.
- He, Zhang, Ren, Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* ICCV 2015.
- Loshchilov, Hutter. *Decoupled Weight Decay Regularization.* ICLR 2019; arXiv:1711.05101.
- Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* JMLR 15, 2014. Krizhevsky, Sutskever, Hinton. *ImageNet Classification with Deep Convolutional Neural Networks.* NeurIPS 2012. Zhang, Cisse, Dauphin, Lopez-Paz. *mixup.* ICLR 2018. Ioffe, Szegedy. *Batch Normalization.* ICML 2015.
- Ruder. *An overview of gradient descent optimization algorithms.* ruder.io, 2016; arXiv:1609.04747 (RMSProp: Hinton, Coursera Lecture 6e, 2012, unpublished).
- Chizat, Bach. *Implicit Bias of Gradient Descent for Wide Two-layer Neural Networks Trained with the Logistic Loss.* COLT 2020; arXiv:2002.04486. Animations: Bach and Chizat's 2020 posts *Gradient descent for wide two-layer neural networks, I and II.* francisbach.com