10-301/601: Introduction to Machine Learning Lecture 13 – Backpropagation II

Geoff Gordon

with thanks to Henry Chai and Matt Gormley

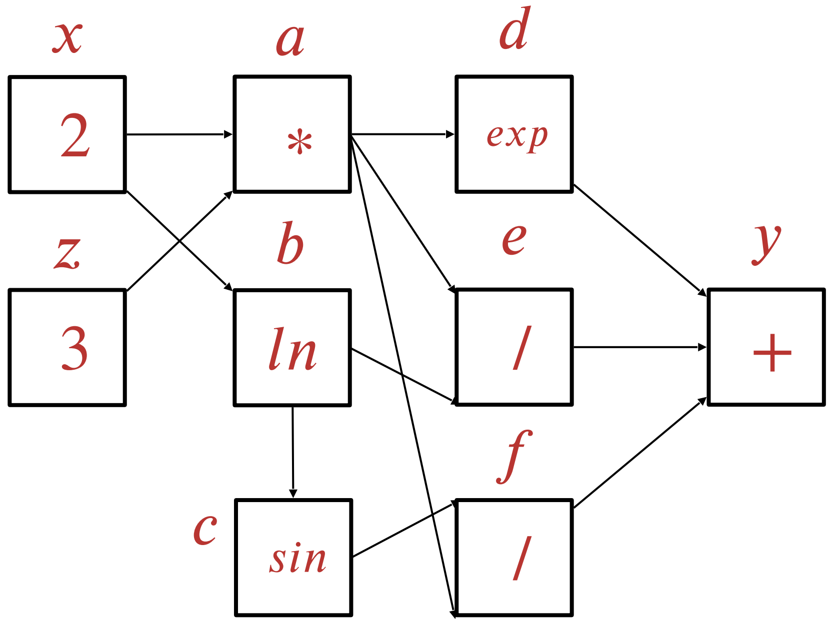
Recall: Automatic Differentiation (reverse mode)

- Given $y = f(x, z) = e^{xz} + \frac{xz}{\ln(x)} + \frac{\sin(\ln(x))}{xz}$

what are $\frac{\partial y}{\partial x}$ and $\frac{\partial y}{\partial z}$ at $x = 2, z = 3$?

- First define some intermediate quantities, draw the computation graph and run the "forward" computation

$$a = xz$$
$$b = \ln(x)$$
$$c = \sin(b)$$
$$d = e^a$$
$$e = \frac{a}{b}$$
$$f = \frac{c}{a}$$
$$y = d + e + f$$



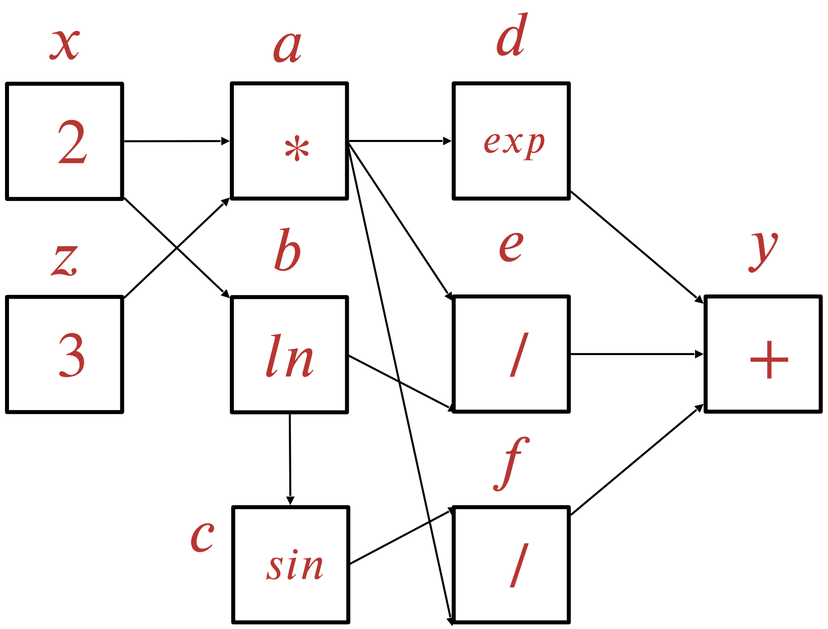
Example courtesy of Matt Gormley

Recall: Automatic Differentiation (reverse mode)

- Given $y = f(x,z) = e^{xz} + \dfrac{xz}{\ln(x)} + \dfrac{\sin\left(\ln(x)\right)}{xz}$

what are $\dfrac{\partial y}{\partial x}$ and $\dfrac{\partial y}{\partial z}$ at $x = 2, z = 3$?

- Then compute partial derivatives, starting from y and working back



$$g_y = \frac{\partial y}{\partial y} = 1$$

$$g_d = g_e = g_f = 1$$

$$g_c = \frac{\partial y}{\partial c} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial c} = g_f\left(\frac{1}{a}\right)$$

- $$g_b = \frac{\partial y}{\partial b} = \frac{\partial y}{\partial e}\frac{\partial e}{\partial b} + \frac{\partial y}{\partial c}\frac{\partial c}{\partial b} = g_e\left(-\frac{a}{b^2}\right) + g_c\left(\cos(b)\right)$$

- $$g_a = \frac{\partial y}{\partial a} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial a} + \frac{\partial y}{\partial e}\frac{\partial e}{\partial a} + \frac{\partial y}{\partial d}\frac{\partial d}{\partial a} = g_f\left(\frac{-c}{a^2}\right) + g_e\left(\frac{1}{b}\right) + g_d\left(e^a\right)$$

$$g_x = \frac{\partial y}{\partial x} = \frac{\partial y}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial y}{\partial a}\frac{\partial a}{\partial x} = g_b\left(\frac{1}{x}\right) + g_a(z)$$

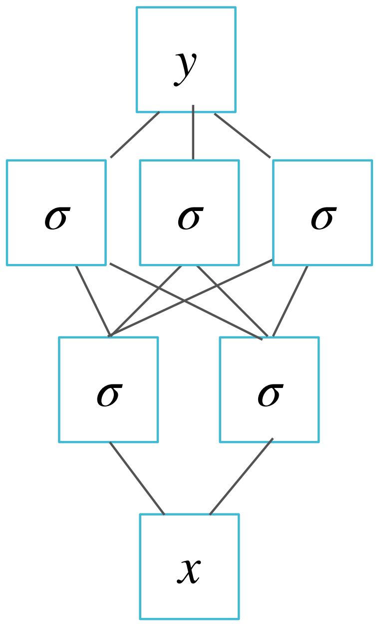
$$g_z = \frac{\partial y}{\partial z} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial z} = g_a(x)$$

Computation graph conventions

- The diagram represents *an algorithm*
- Nodes are rectangles with one node per input, output, or intermediate variable in the algorithm
- Each node is labeled with the function that it computes (inside the box) and the variable name (outside the box)
  - if argument order matters, it's left-to-right for where edges enter the box
- Edges are directed and do not have labels
- We can make a computation graph for a neural network
  - Each weight, feature value, label and *bias term* appears as a node
  - We *can* include the loss function

Neural Network Diagram Conventions

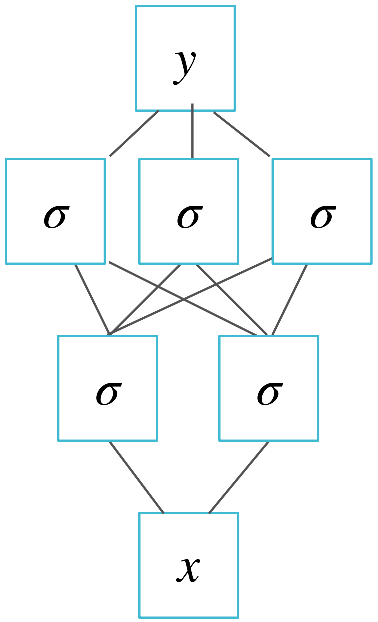
- The diagram represents a *neural network*
- Nodes are circles or squares with one node per input, hidden, or output unit
- Each node may be labeled with the variable corresponding to the hidden unit and/or with its activation function
- Edges are directed and each edge can be labeled with its weight
- The diagram typically does *not* include any nodes related to the loss computation



often in papers, network diagrams are simplified: grouping nodes, omitting normalization, …

Conversion

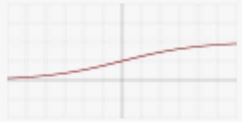
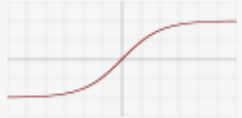
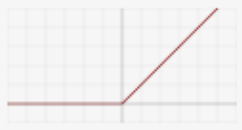
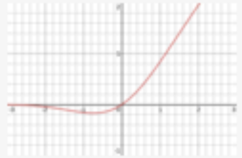
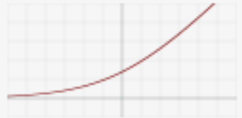
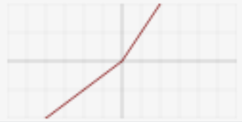
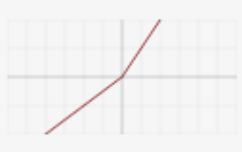
- We can *convert* a neural network diagram into a computation graph



Neural net

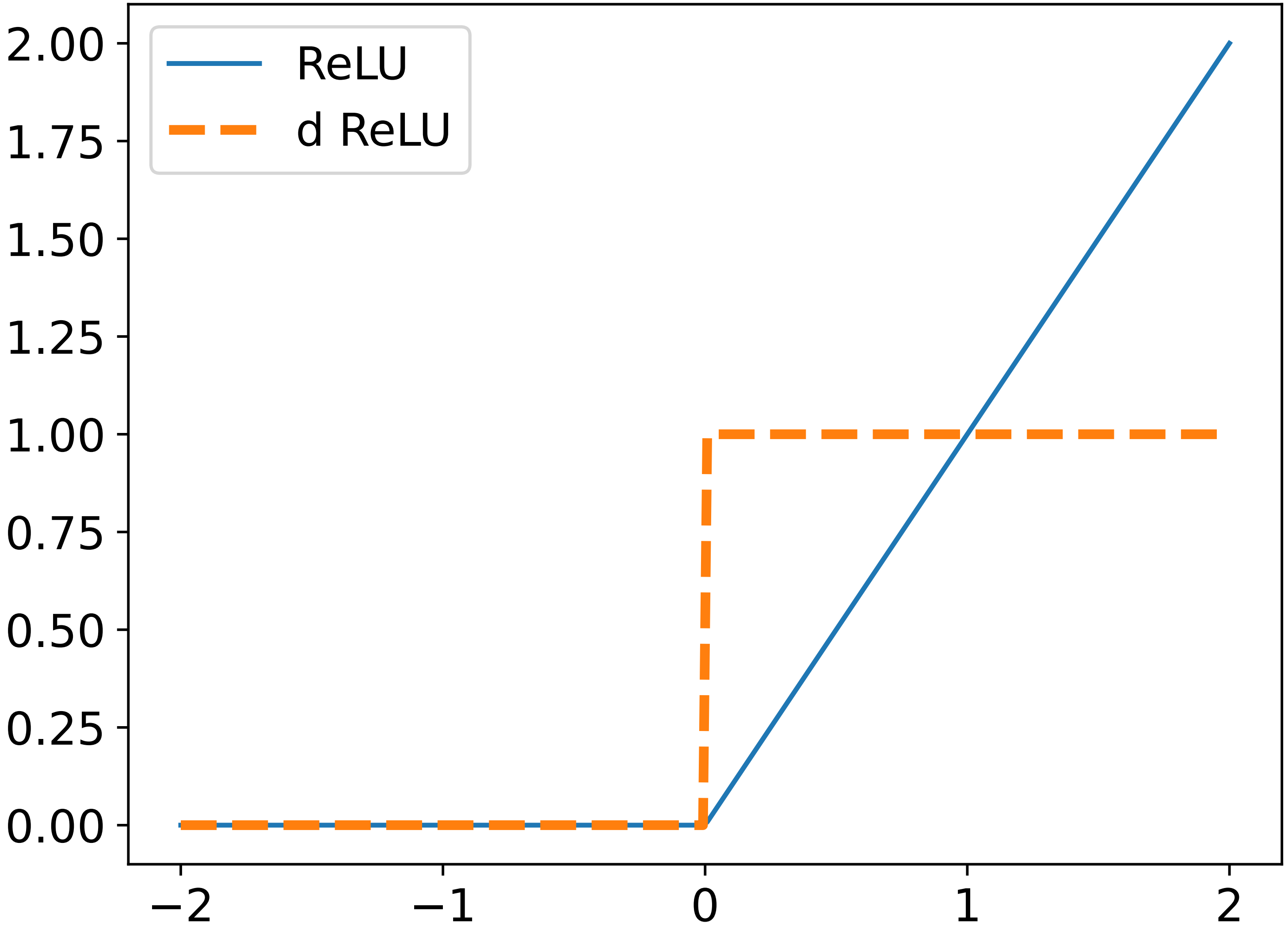
Computation graph

Lots of activation functions!

Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1+e^{-x}}$
Hyperbolic tangent (tanh)		$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified linear unit (ReLU)[7]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max\{0, x\} = x\mathbf{1}_{x>0}$
Gaussian Error Linear Unit (GELU)[4]		$\frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $= x\Phi(x)$
Softplus[8]		$\ln(1 + e^x)$
Exponential linear unit (ELU)[9]		$\begin{cases} \alpha\left(e^x - 1\right) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ with parameter α
Leaky rectified linear unit (Leaky ReLU)[11]		$\begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
Parametric rectified linear unit (PReLU)[12]		$\begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameter α

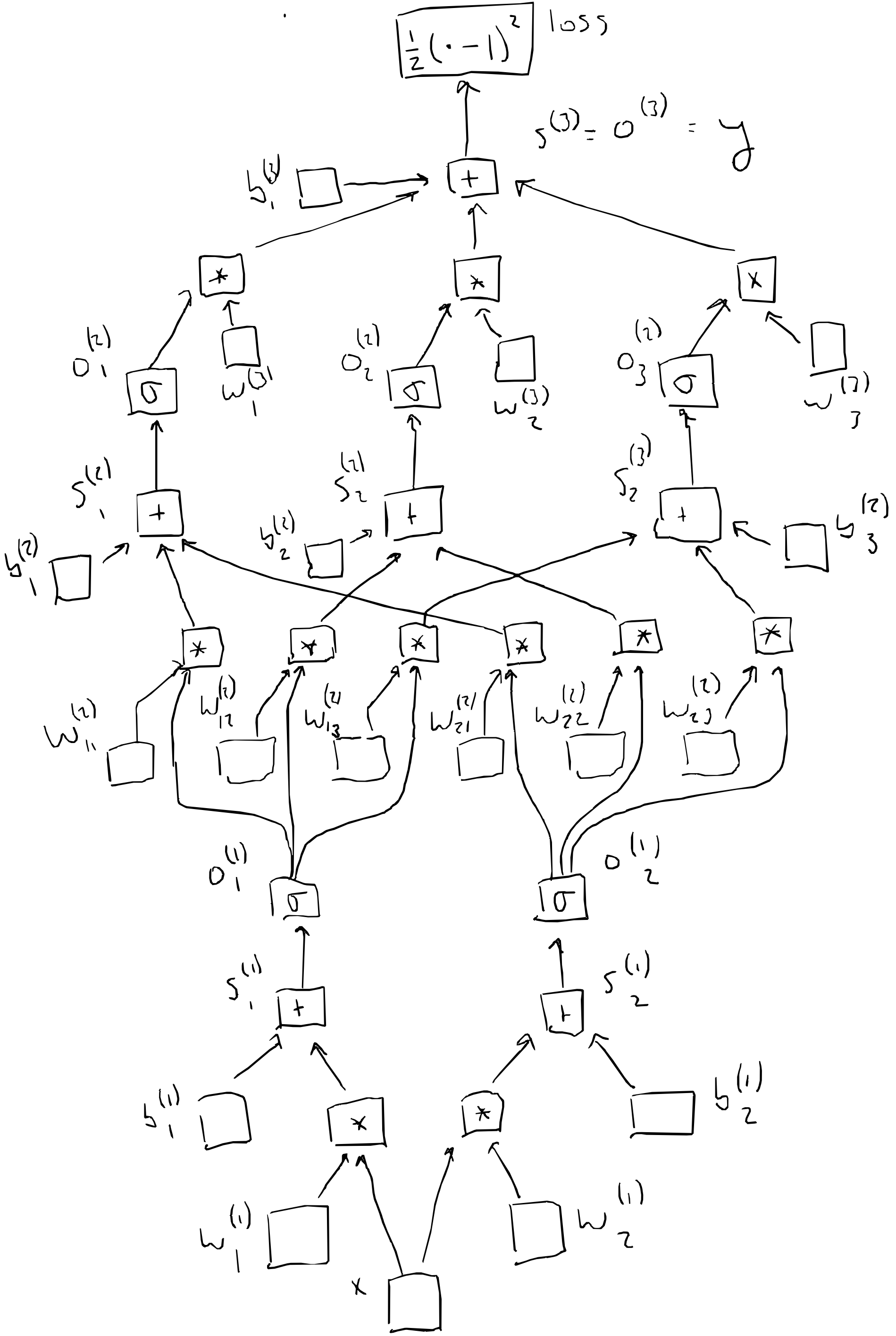
Source: https://en.wikipedia.org/wiki/Activation_function

Derivative of ReLU



- We'll use ReLU for our example, since it has a simple derivative

Forward pass



σ = ReLU

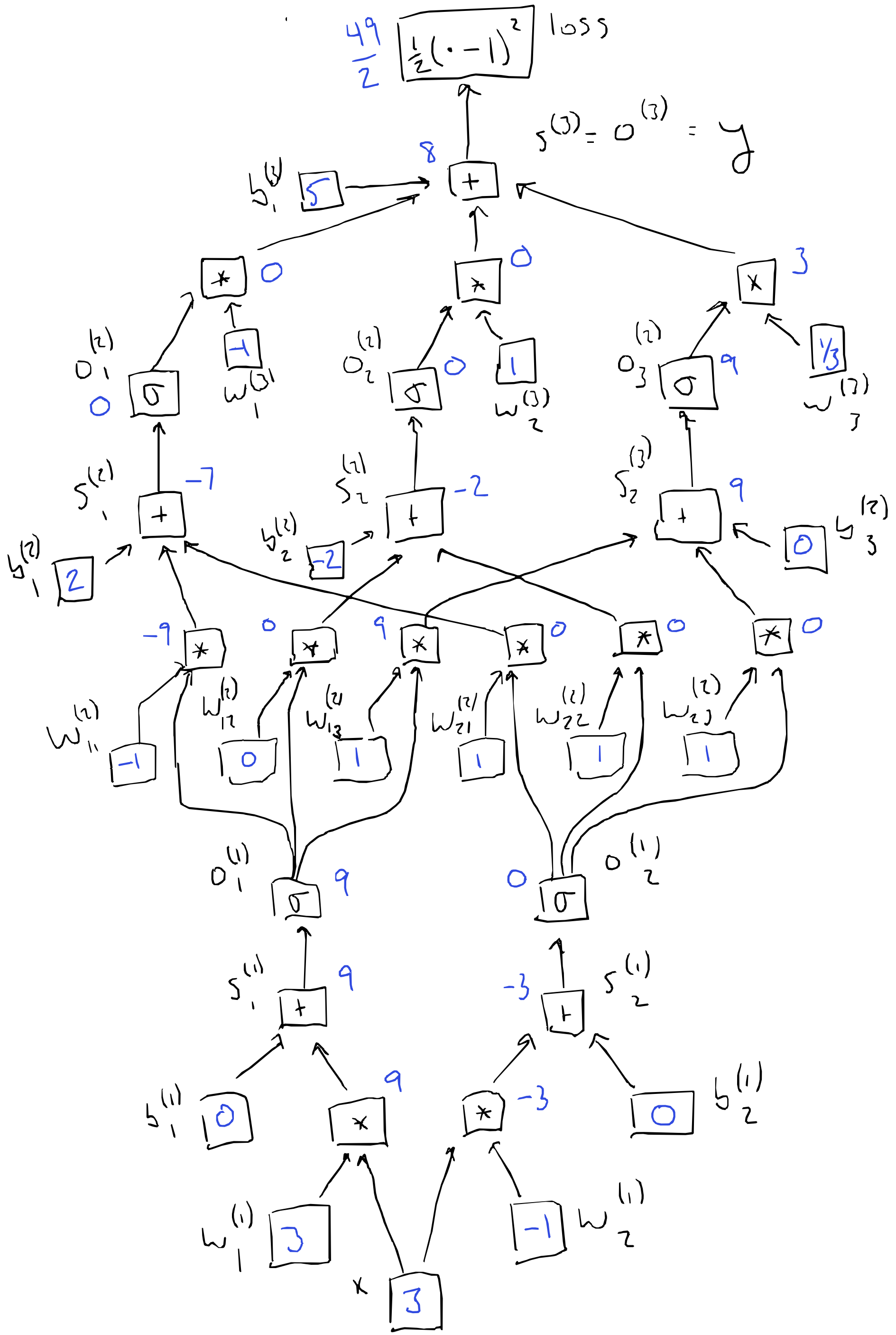
$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

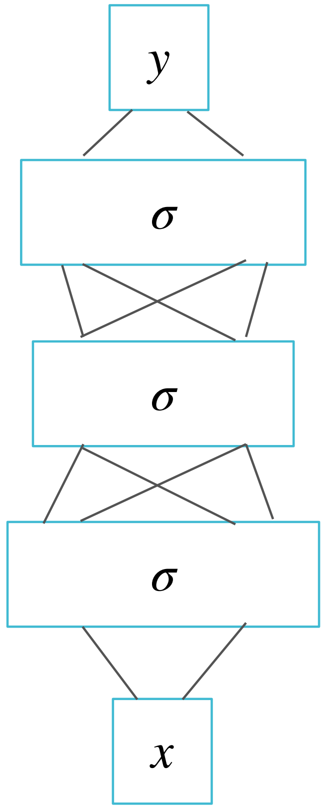
$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

$$\text{loss} = \frac{1}{2}(y - 1)^2$$

Compute derivatives (backward pass)

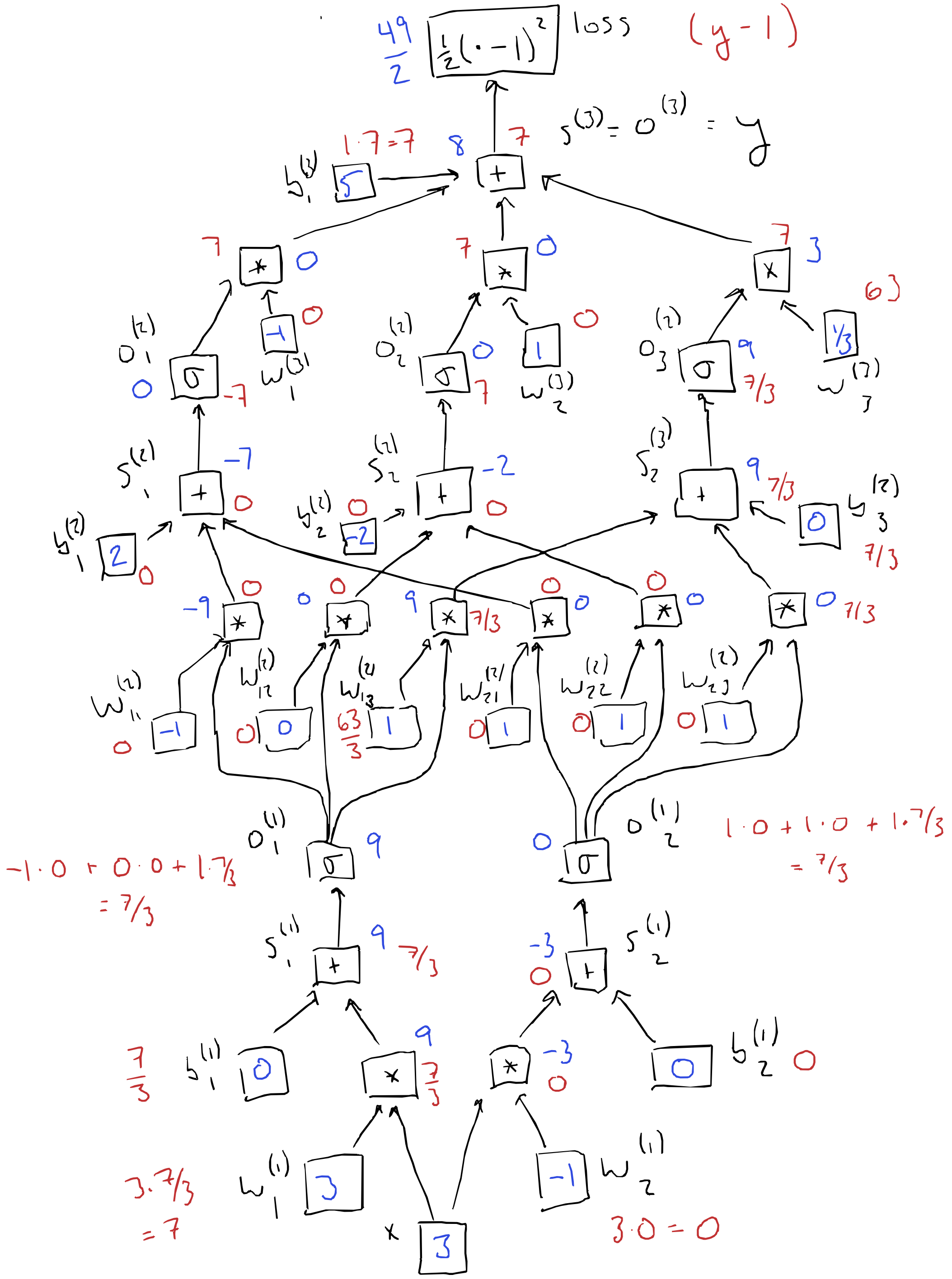


Matrix form



- Assume network composed of layers
- Before and after activations: $s^{(l)}, o^{(l)}$
- Weights, biases $W^{(l)}, b^{(l)}$
- Dimensions

What did we do?

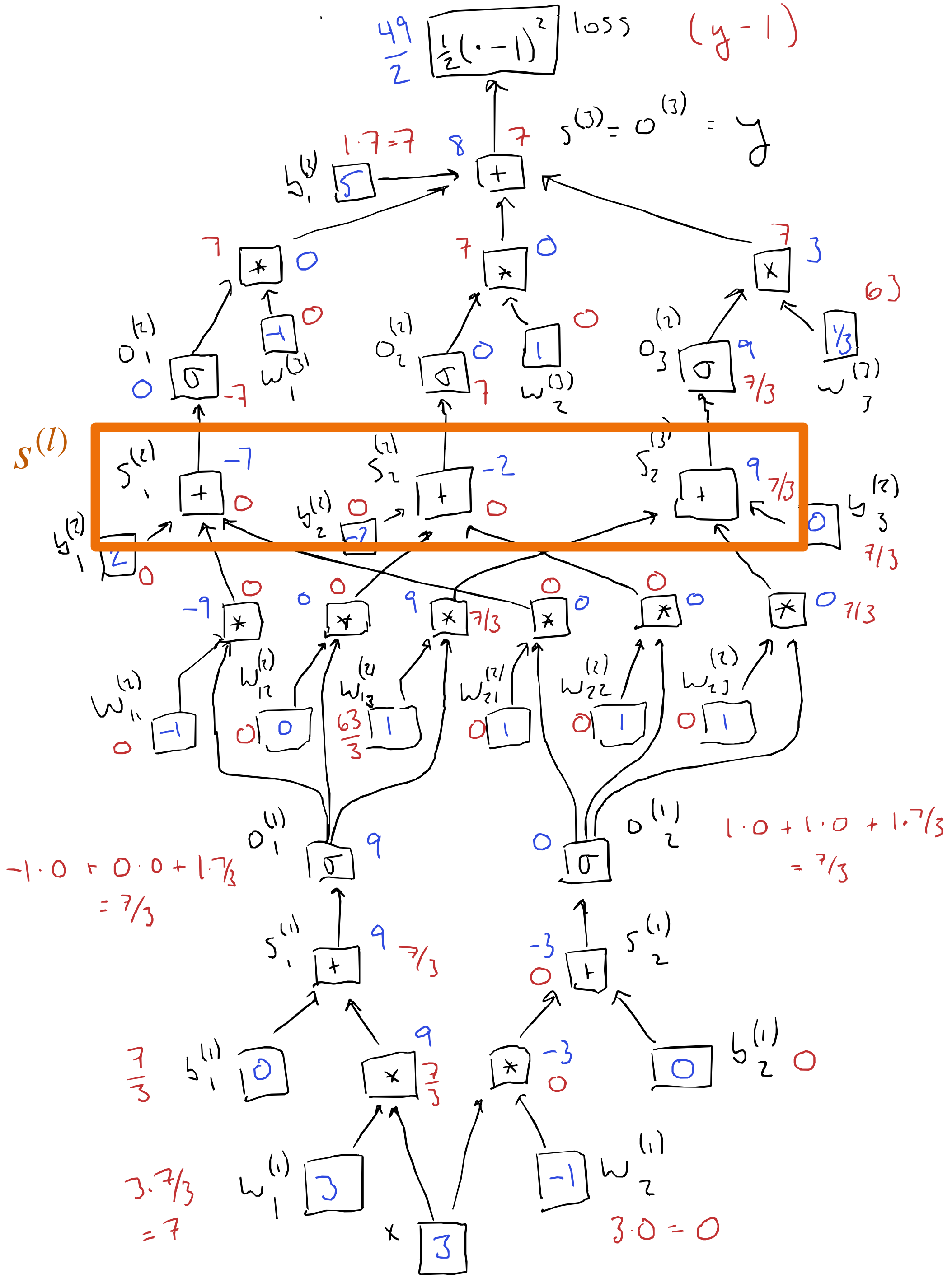


$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

What did we do?

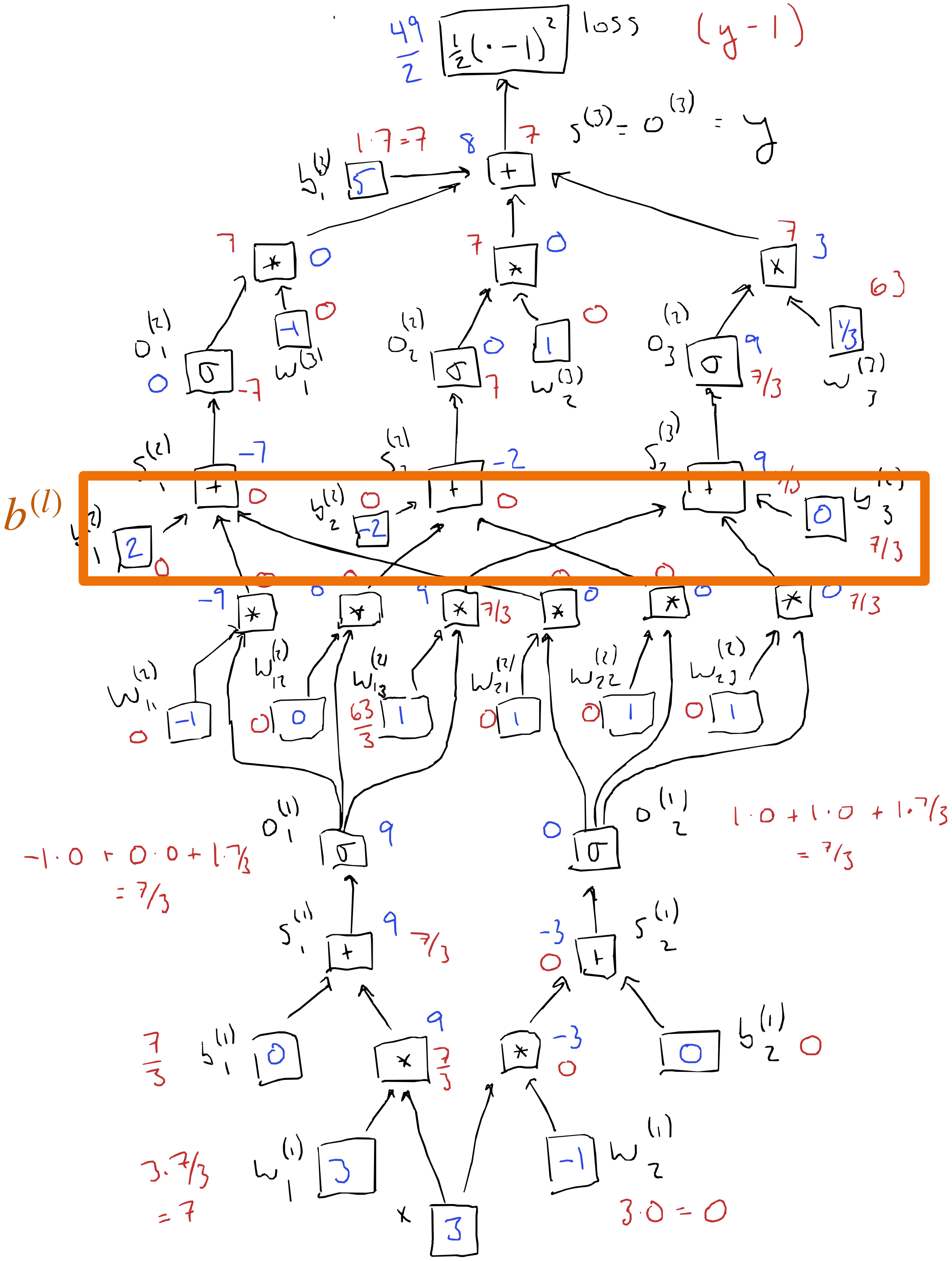


$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

What did we do?

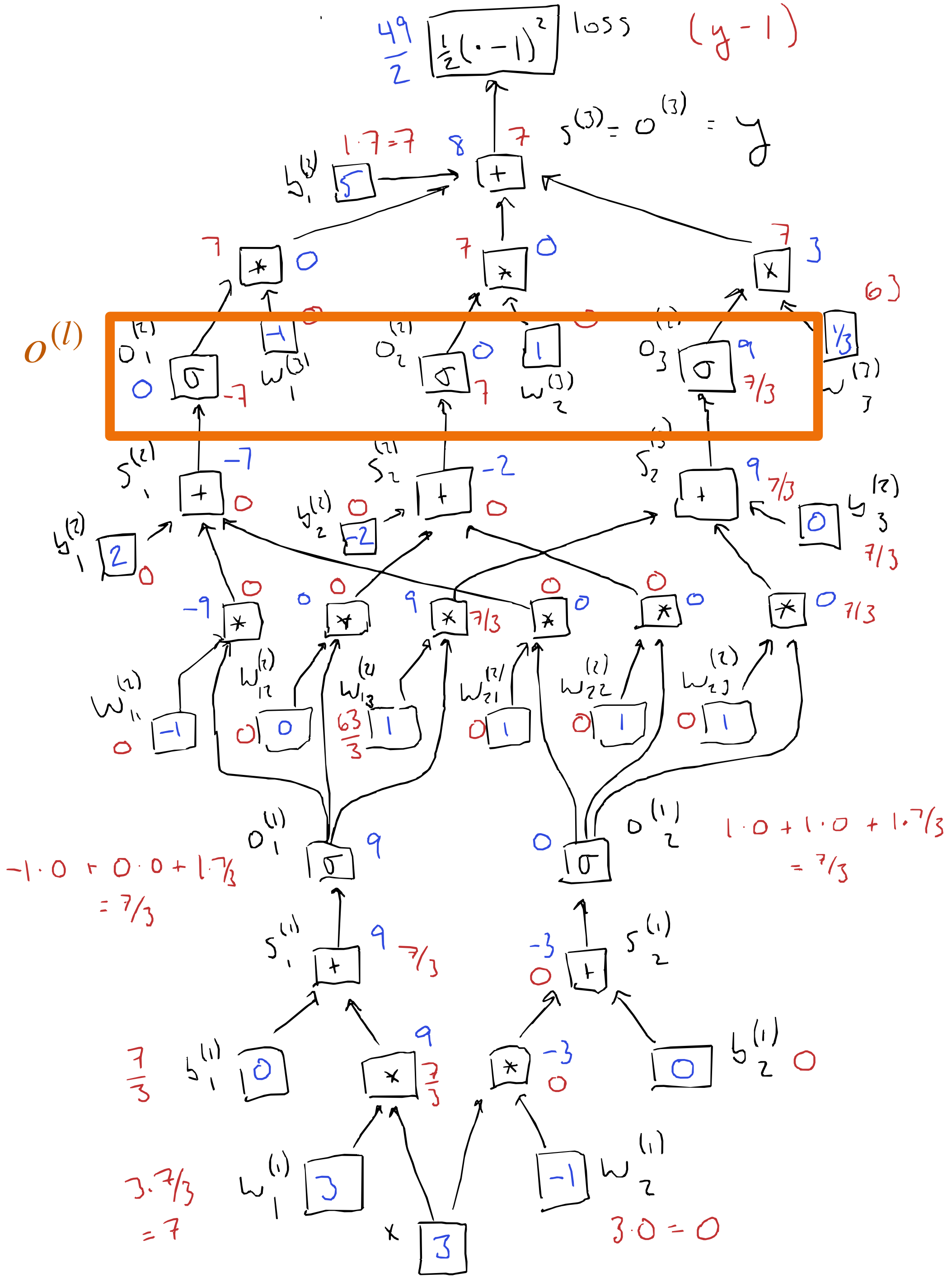


$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

What did we do?

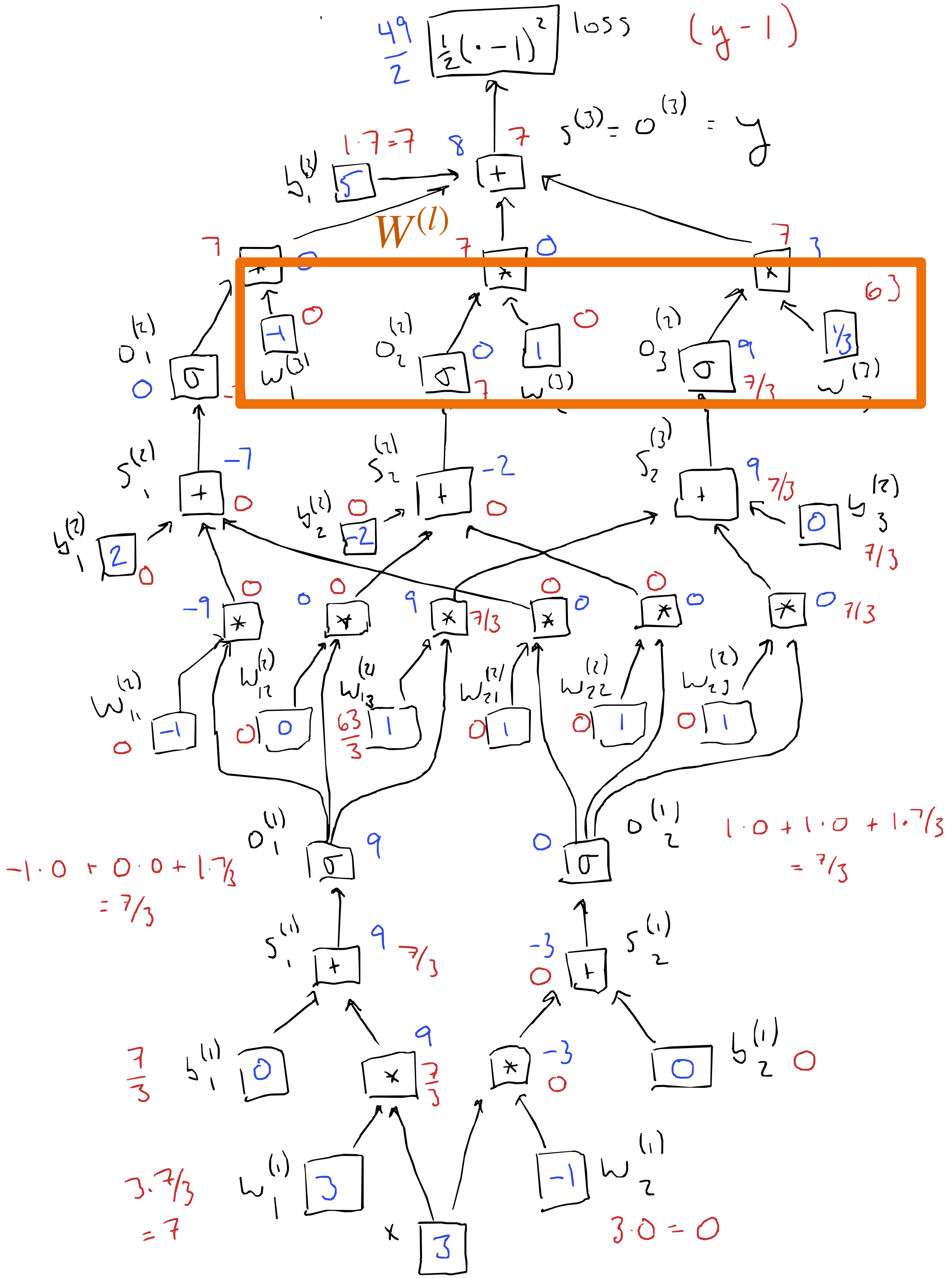


$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

What did we do?

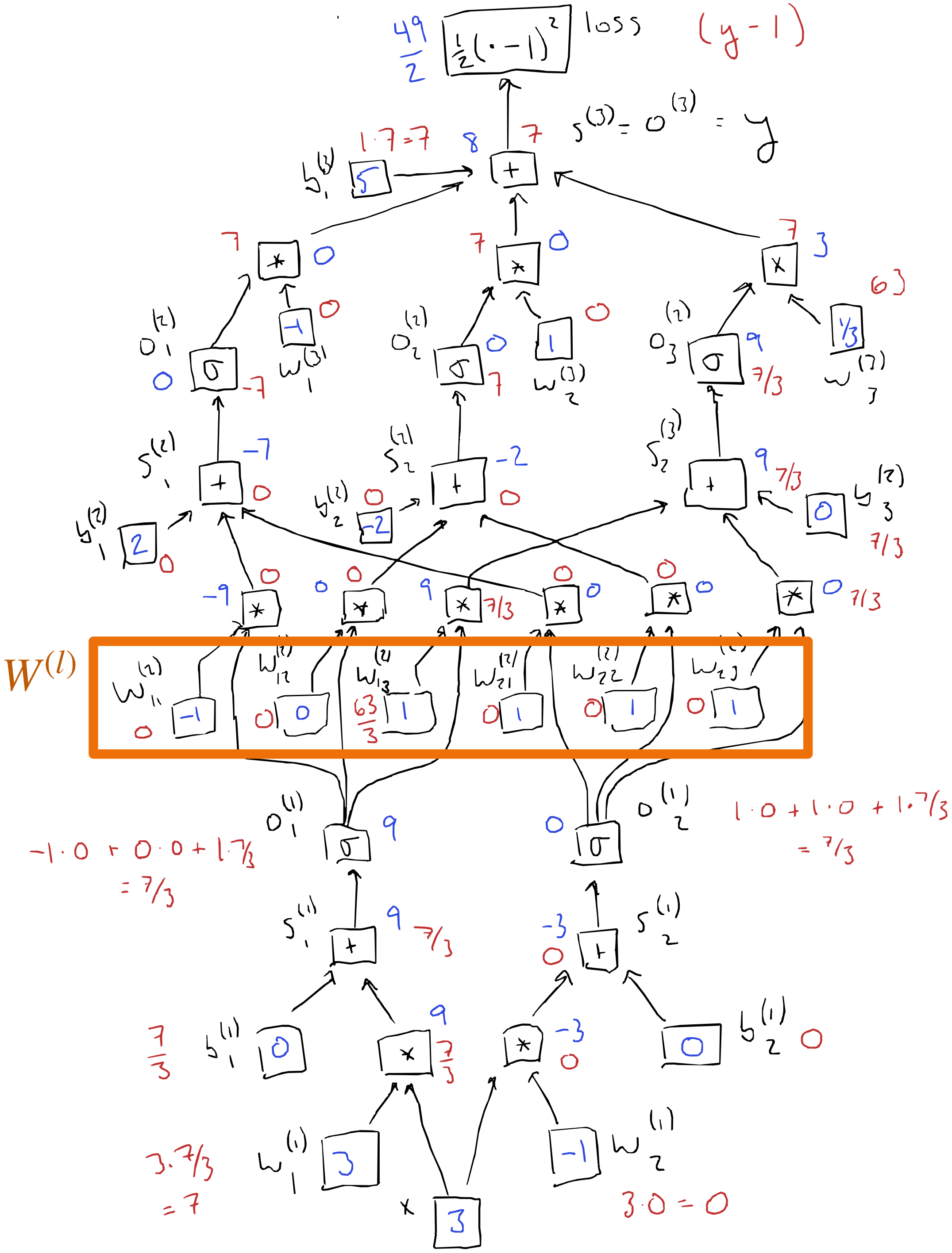


$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

What did we do?

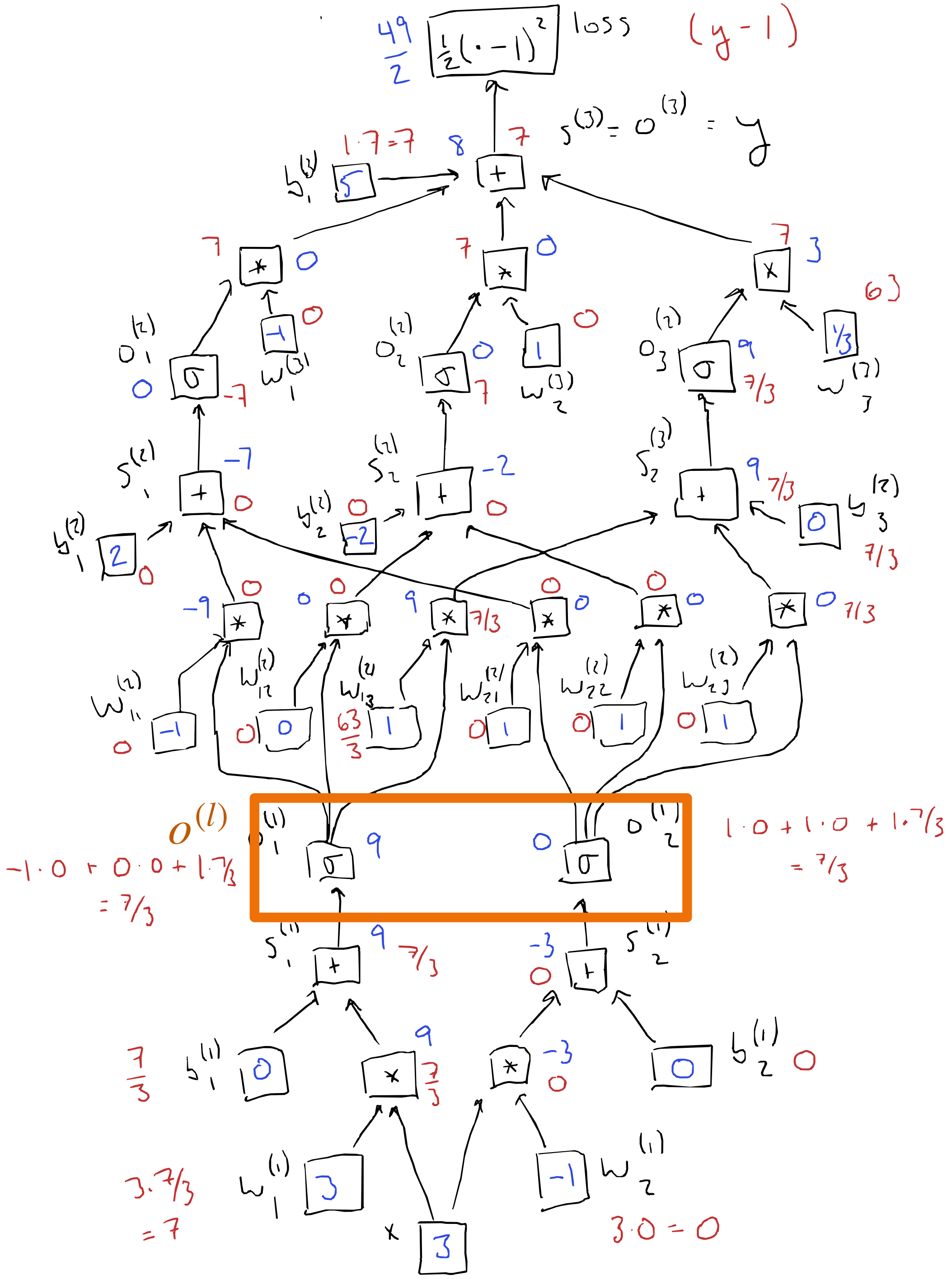


$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

What did we do?



$$W^{(1)} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

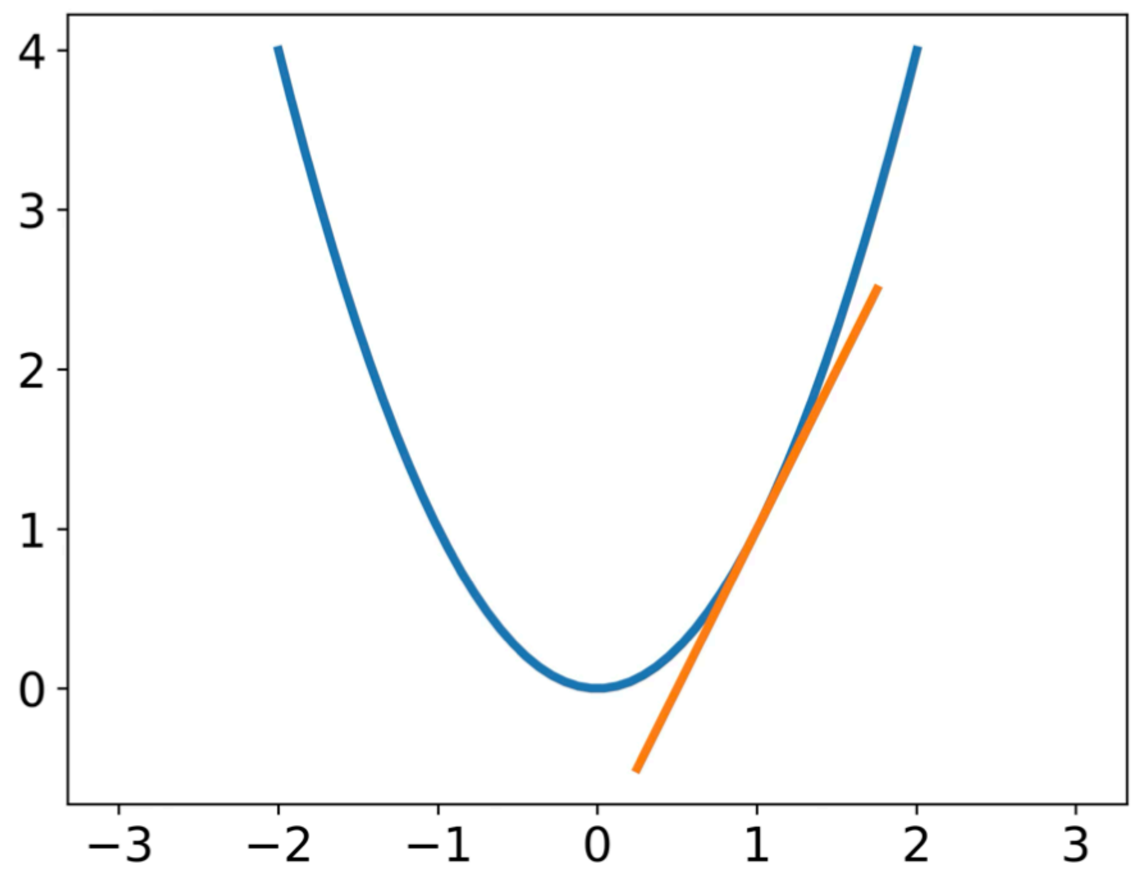
$$W^{(2)} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} -1 & 1 & \frac{1}{3} \end{bmatrix} \quad b^{(3)} = 5$$

Backprop

- Assume we've done a forward pass to compute loss
- Initialize $r \leftarrow \frac{\partial \text{loss}}{\partial o^{(L)}}$ (evaluated at $o^{(L)}$)
- For $l = L, L-1, \ldots, 1$
  - $r \leftarrow \frac{\partial \text{loss}}{\partial s^{(l)}} = r \circ \sigma'(o^{(l)})$ [componentwise]
  - output: $\frac{\partial b^{(l)}}{\partial \text{loss}} = r$ $\frac{\partial W^{(l)}}{\partial \text{loss}} = r(o^{(l-1)})^\top$
  - $r \leftarrow \frac{\partial o^{(l-1)}}{\partial \text{loss}} = (W^{(l)})^\top r$

Derivatives: intuition

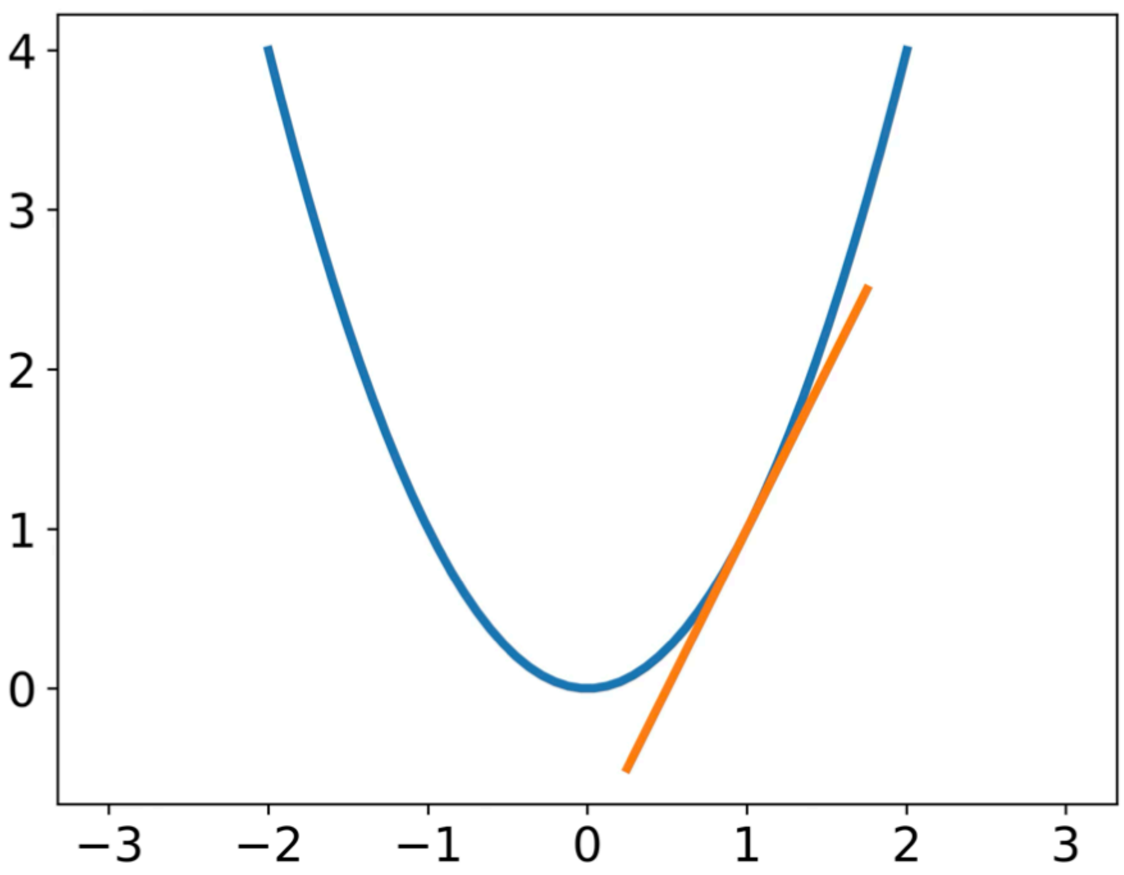


$m = n = 1$

$dy = y - f(\hat{x})$
$dx = x - \hat{x}$

- Function $y = f(x)$
  - $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $f \in \mathbb{R}^n \to \mathbb{R}^m$
- Derivative is a *linear fn* that *locally* approximates f near $\hat{x}$
  - $dy = a\,dx$, where $a \in \mathbb{R}$ is the derivative

Derivatives: intuition



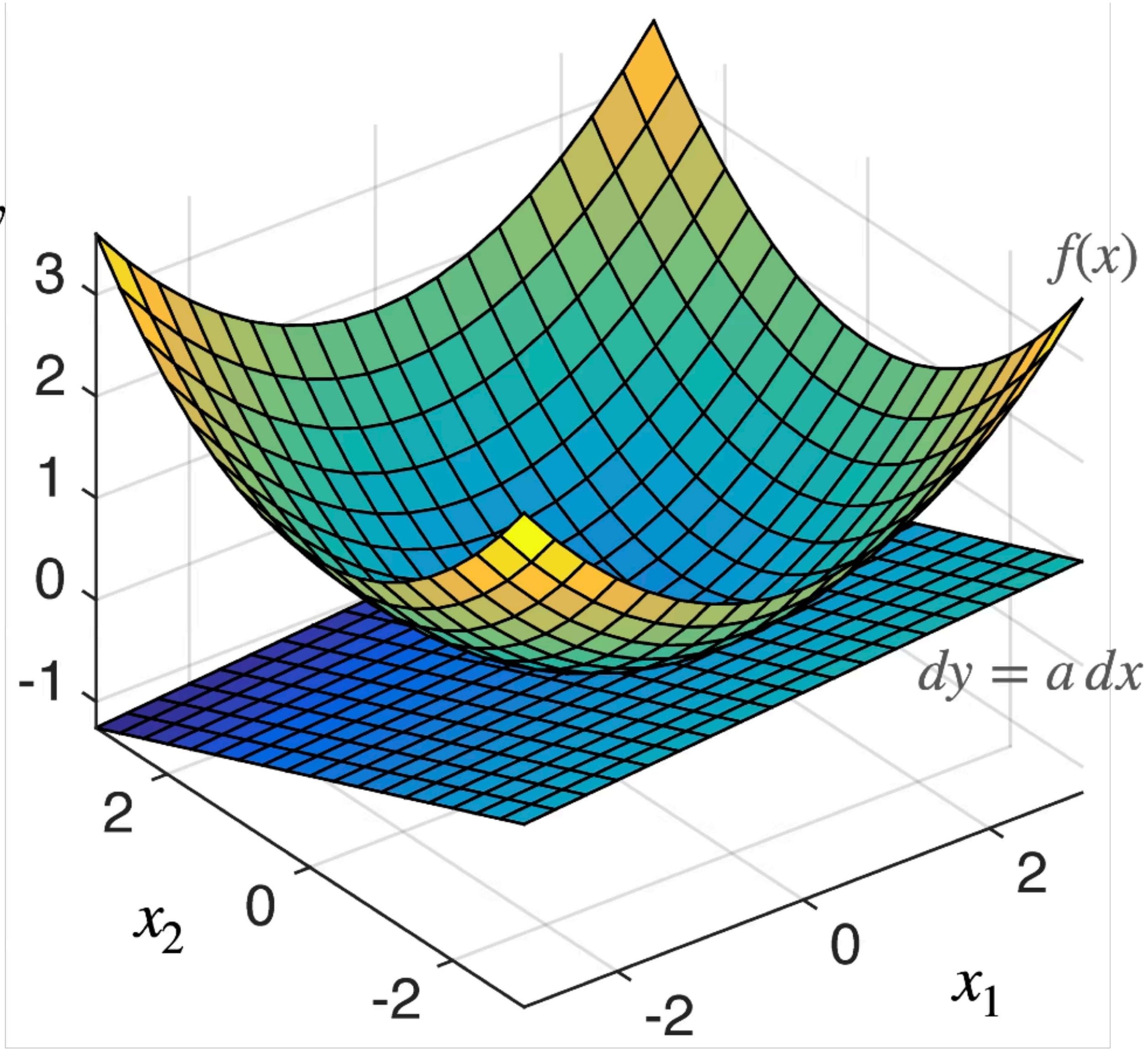
$m = n = 1$

$dy = y - f(\hat{x})$
$dx = x - \hat{x}$

- Function $y = f(x)$
  - $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $f \in \mathbb{R}^n \to \mathbb{R}^m$
- Derivative is a *linear fn* that *locally* approximates f near $\hat{x}$
  - $dy = a\,dx$, where $a \in \mathbb{R}$ is the derivative

think of this as a synonym for $\frac{dy}{dx} = a$

Higher dimensions

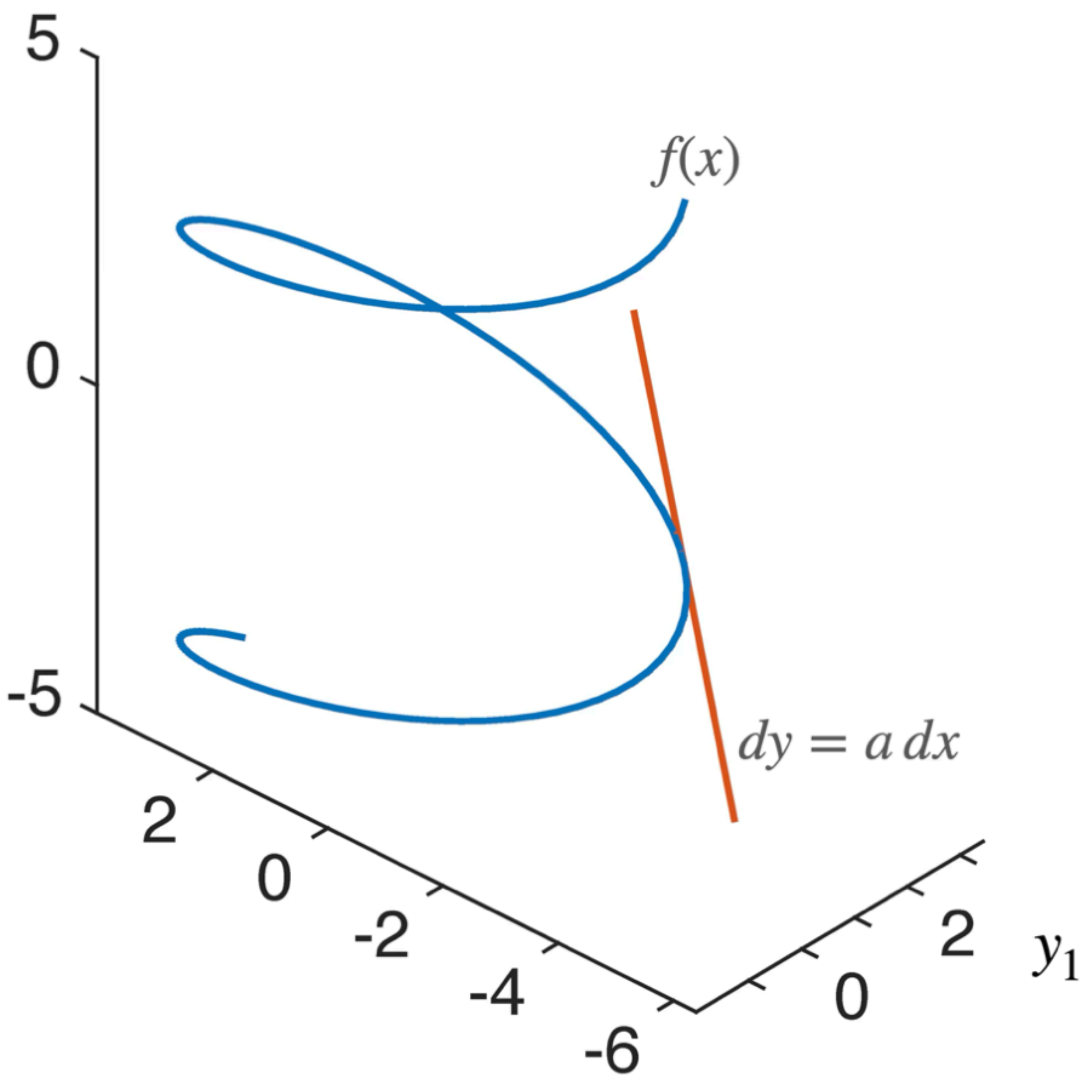


$\mathbb{R}^n \to \mathbb{R}^m$

$m = 1, n = 2$

Function $y = f(x)$ $\quad dy = a\,dx \quad a \in \mathbb{R}^{1\times 2}$

Higher dimensions



Function $y = f(x)$ $\quad dy = a\,dx \quad a \in \mathbb{R}^{3\times 1}$

$\mathbb{R}^n \to \mathbb{R}^m$

$m = 3, n = 1$

Notation for linear functions

input shape (with respect to)

output shape (derivative of)

	Scalar	Vector	Matrix
Scalar	$ds = a\,dt$	$ds = u \cdot dv$	$ds = \langle M, dN \rangle$
Vector	$du = v\,ds$	$du = M^\top dv$	$\times$
Matrix	$dM = N\,ds$	$\times$	$\times$

- a, s, t: scalars u, v: column vectors M, N: matrices
- note: $ds = a\,dt$ is a synonym for $a = \frac{ds}{dt}$, etc.
- $\times$ means no common notation; see torch.einsum
- derivative = coefficient of ds, dt, dv, dN on RHS

Geoff Gordon

Notation for linear functions

output shape (derivative of)

input shape (with respect to)

	Scalar	Vector	Matrix
Scalar	$ds = a\,dt$	gradient	$ds = \langle M, dN \rangle$
Vector	velocity	Jacobian	$\times$
Matrix	$dM = N\,ds$	$\times$	$\times$

- a, s, t: scalars u, v: column vectors M, N: matrices
- note: $ds = a\,dt$ is a synonym for $a = \frac{ds}{dt}$, etc.
- $\times$ means no common notation; see torch.einsum
- derivative = coefficient of ds, dt, dv, dN on RHS

Geoff Gordon

Conventions

- When derivative is a vector or matrix, our convention:
  - ***denominator layout:*** first coordinate of the derivative corresponds to the denominator
  - e.g., $u \in \mathbb{R}^m \quad v \in \mathbb{R}^n \quad \Rightarrow \frac{d\vec{u}}{d\vec{v}} \in$
- Taking derivative of a scalar, our convention:
  - derivative is ***same shape*** as argument
  - e.g., $u \in \mathbb{R} \quad v \in \mathbb{R}^{m \times n} \quad \Rightarrow \frac{d\vec{u}}{d\vec{v}} \in$

Denominator layout examples

Types of Derivatives	scalar	vector
scalar	$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x}\right]$	$\frac{\partial \mathbf{y}}{\partial x} = \left[\frac{\partial y_1}{\partial x} \quad \frac{\partial y_2}{\partial x} \quad \cdots \quad \frac{\partial y_N}{\partial x}\right]$
vector	$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_P} \end{bmatrix}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_2} \\ \vdots & & & \\ \frac{\partial y_1}{\partial x_P} & \frac{\partial y_2}{\partial x_P} & \cdots & \frac{\partial y_N}{\partial x_P} \end{bmatrix}$

Table courtesy of Matt Gormley

Scalar derivative examples

Types of Derivatives	**scalar**
scalar	$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x}\right]$ $\leftarrow$ matches previous slide!
vector	$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_P} \end{bmatrix}$ $\leftarrow$ matches previous slide!
matrix	$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} & \frac{\partial y}{\partial X_{12}} & \cdots & \frac{\partial y}{\partial X_{1Q}} \\ \frac{\partial y}{\partial X_{21}} & \frac{\partial y}{\partial X_{22}} & \cdots & \frac{\partial y}{\partial X_{2Q}} \\ \vdots & & & \vdots \\ \frac{\partial y}{\partial X_{P1}} & \frac{\partial y}{\partial X_{P2}} & \cdots & \frac{\partial y}{\partial X_{PQ}} \end{bmatrix}$

Table courtesy of Matt Gormley

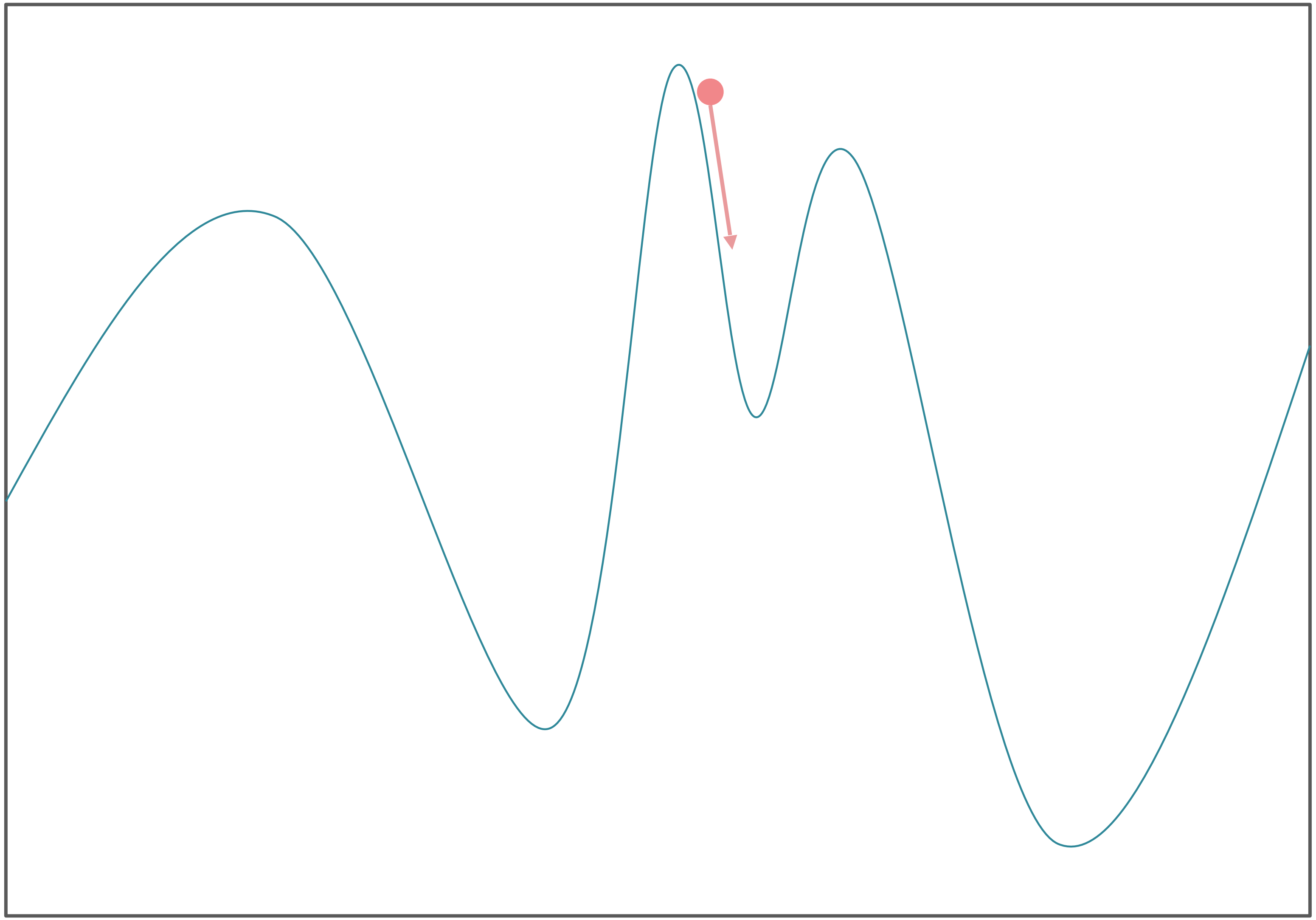
Autodiff version

- Use `Tensor` class for all arrays
  - keeps the “paper trail” of who depends on whom
- For each example in minibatch
  - run code that starts w/ (x, y) and ends w/ `loss += ...`
- Call `loss.backward()` to run autodiff
- Take an optimizer step (e.g., SGD) using accumulated derivative
- Clear derivative storage (e.g., SGD optimizer has `zero_grad()` method, as do many others)

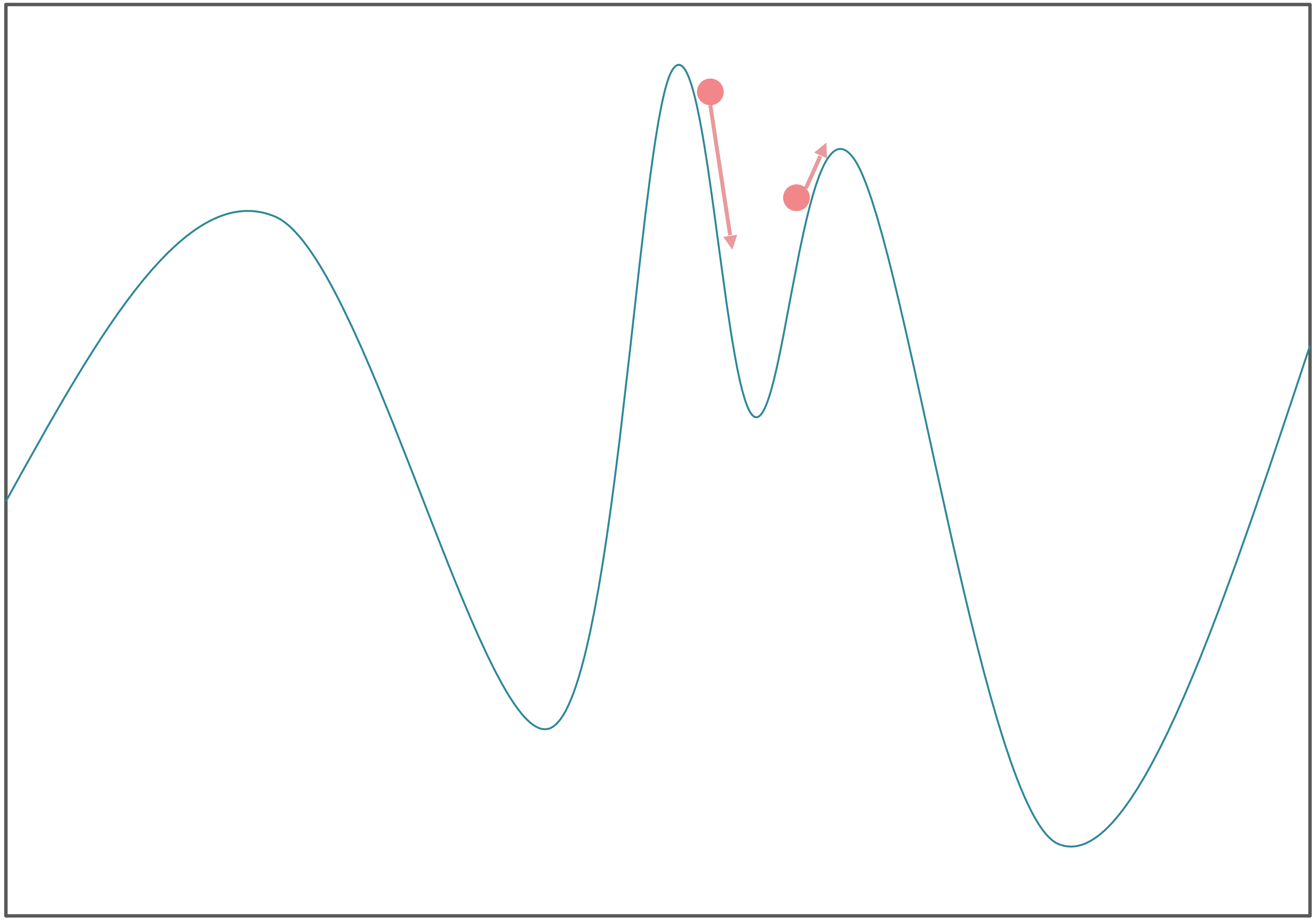
Now that we have the gradients

- Typical optimizer: minibatch SGD with momentum
  - and something to adjust learning rates (RMSprop, Adam)
  - and some kind of regularizer (weight decay, dropout, early stopping)
- And possibly some modifications to the network to make optimization easier
  - some kind of normalization to guide $s^{(l)}$ near "interesting part" of activation function (layer norm, batch norm)
  - tools to keep gradient sizes manageable (ReLU, residual connections)

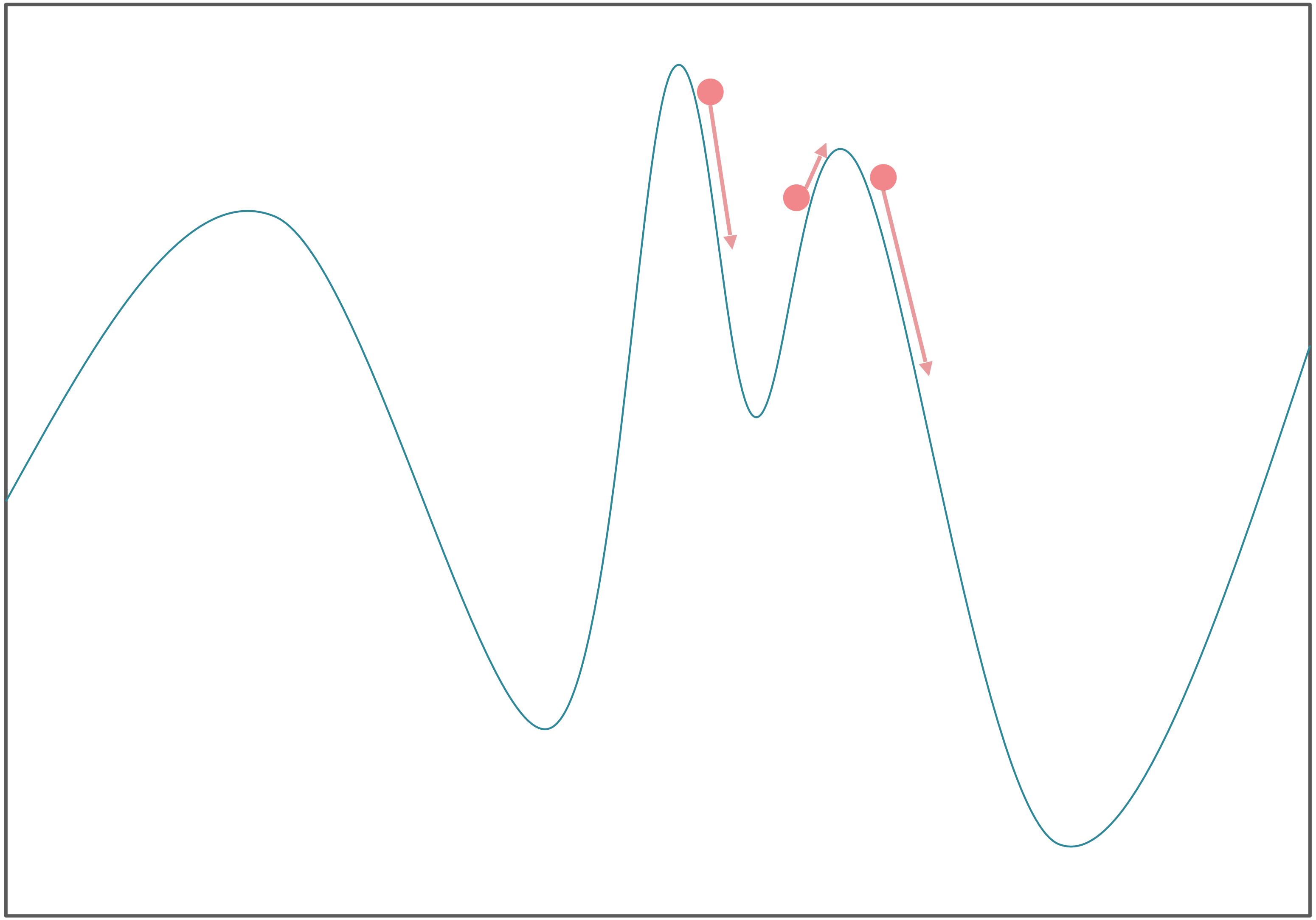
Mini-batch Stochastic Gradient Descent with Momentum for Neural Networks



Mini-batch Stochastic Gradient Descent with Momentum for Neural Networks

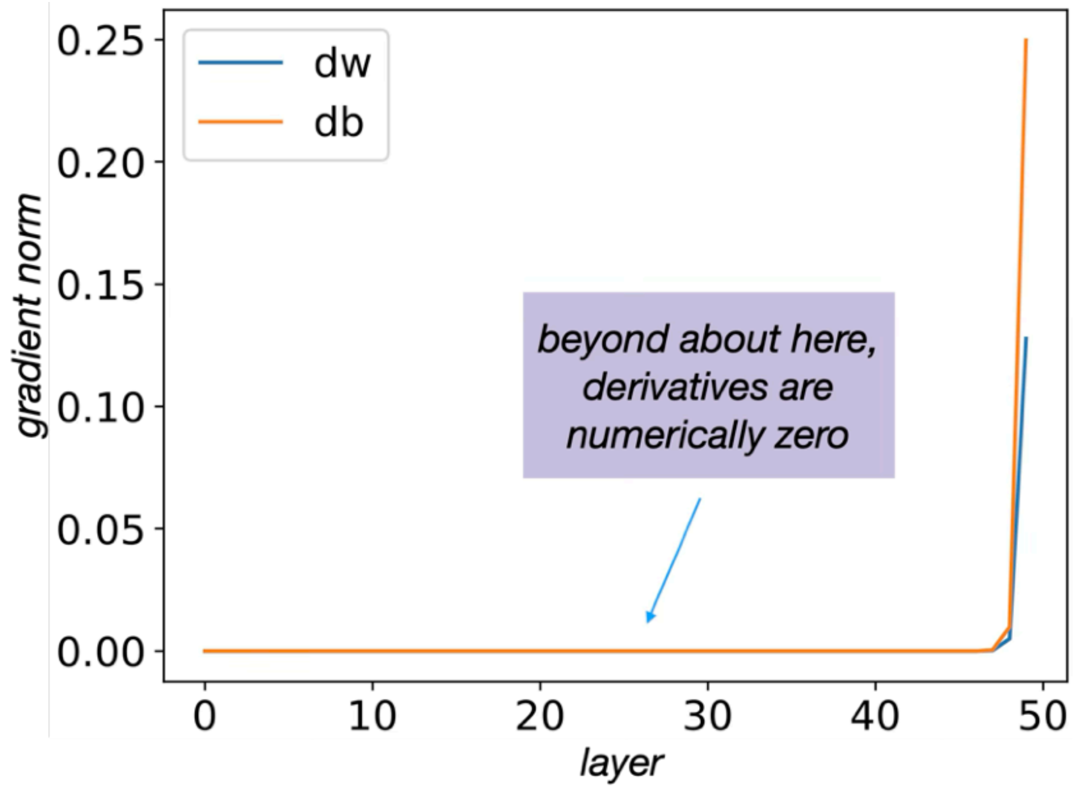


Mini-batch Stochastic Gradient Descent with Momentum for Neural Networks



Vanishing gradients

- Now that we have autodiff, no problem differentiating a 50-layer network!
- Sigmoid activations, initialize $w_i, b_i \sim N(0, 0.1^2)$
- Forward pass: $z_{50} = 0.5167$
- Backward pass: $\rightarrow$



Problem!

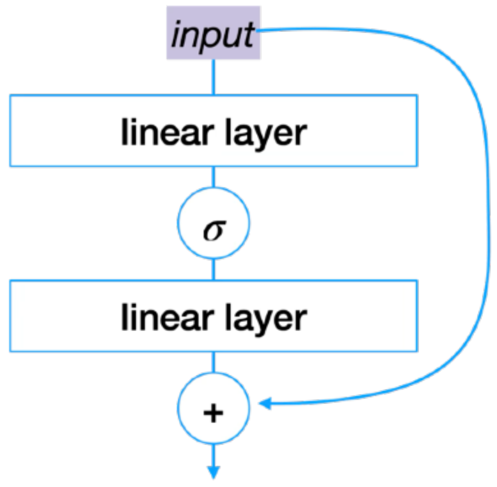
- Effectively zero updates to parameters for layers before 45, and truly zero updates for layers before 30
- so, we aren't really using a deep model: only last few layers are learnable
- worse: in forward pass, output of layer 30 is the same no matter what input is

Why?

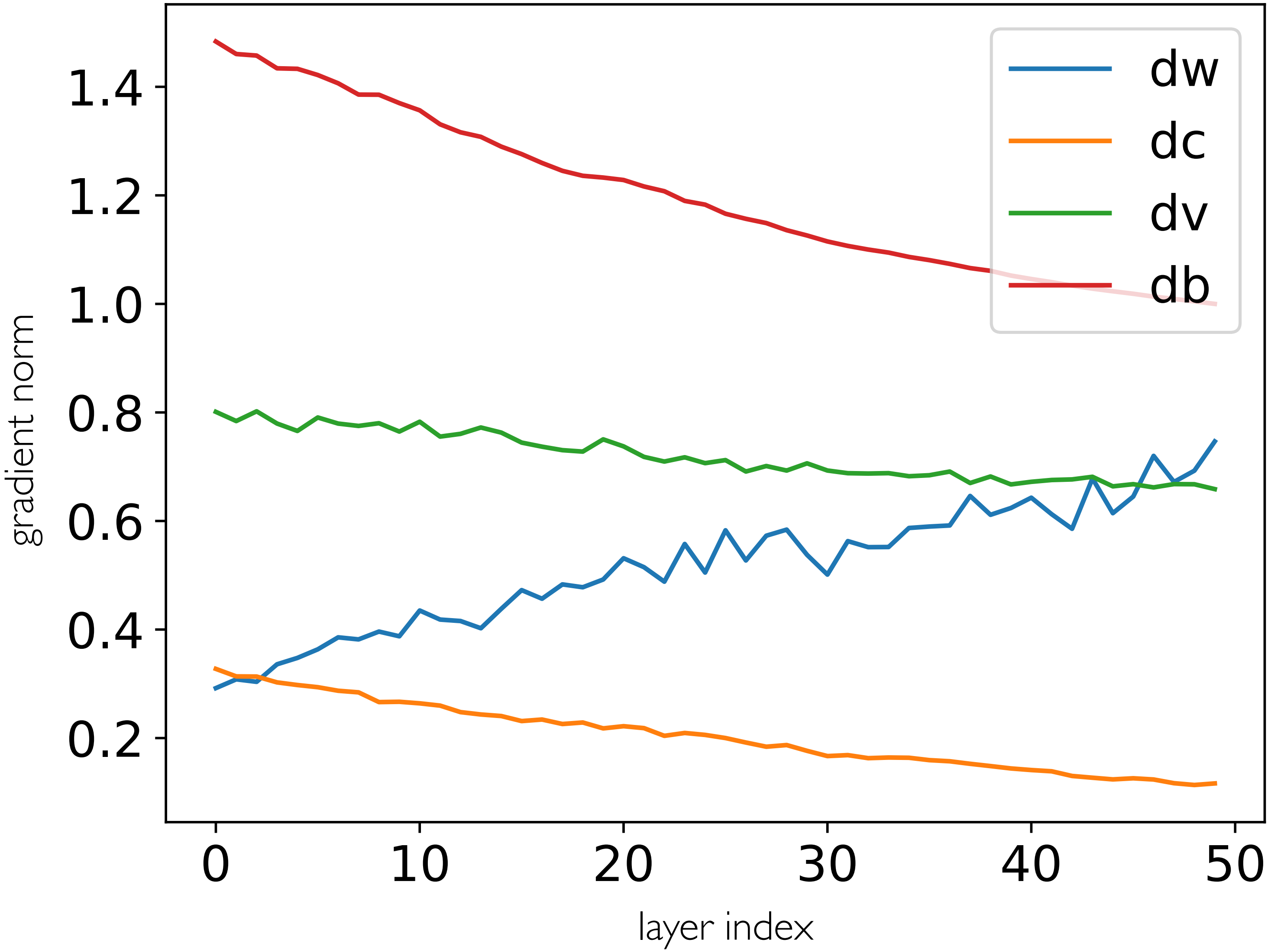
- Recursion for backprop:
  - $r \leftarrow \frac{\partial \text{loss}}{\partial s^{(l)}} = r \circ \sigma'(o^{(l)})$ [componentwise]
  - $r \leftarrow \frac{\partial o^{(l-1)}}{\partial \text{loss}} = (W^{(l)})^\top r$
- Factors are $\ll 1$ in magnitude: exponential decay!

One fix: residual connection

- Change the arrangement of layers:
  - $o^{(l)} = o^{(l-1)} + V^{(l)}\sigma(W^{(l)}o^{(l-1)} + c^{(l)}) + b^{(l)}$
  - any nonlinearity (sigmoid, ReLU, …)
  - two weight matrices and two bias vectors
- Learn layer l as an *update* to layer $l-1$ instead of a transformation
- Second linear layer allows $o^{(l)}$ to be +ve or –ve
- Stack this block like any other layer
  - appears in famous architectures like ResNet, transformer

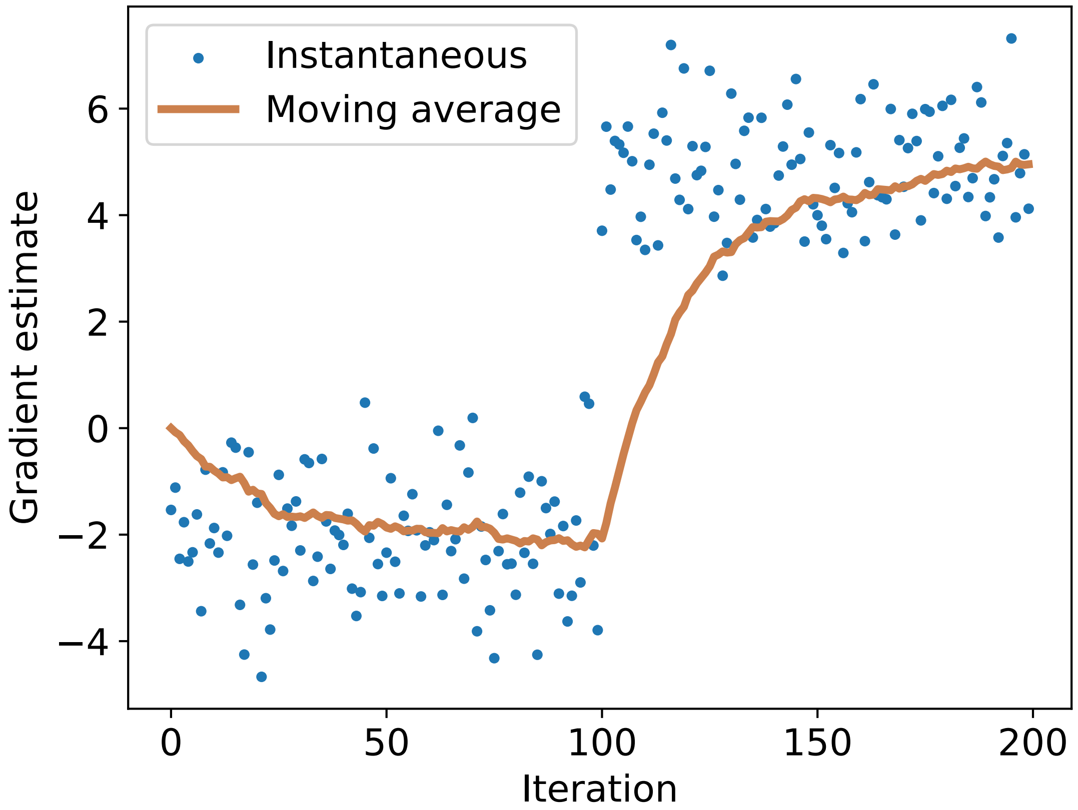


Fixes the problem (why?)



50 stacked residual blocks, same init

Tracking gradient statistics



$\beta = 0.95$

trades off *noise* vs. *transient*

- Huge class of methods: track statistics of gradient estimates (like mean and variance), use them to compensate for curvature in different directions
- Most common tracking: exponential moving average

$$m_{t+1} = \beta m_t + (1-\beta) g_t \quad \beta \in (0,1)$$

Geoff Gordon

Track the gradient mean: momentum

g_t is gradient estimate at step t

- Recall momentum:

$$m_{t+1} = \beta m_t + (1-\beta) g_t$$
$$\theta_{t+1} = \theta_t - \gamma m_{t+1}$$

- Two parameters: moving average rate $\beta \in (0,1)$, learning rate $\gamma > 0$
- Memory for one array same size as θ_t
- Instead of using gradient directly, update θ in direction of *average gradient*

Track the gradient variance: Adam

- Adam ("ADAptive Moment estimation"):

$$m_{t+1} = \beta_m m_t + (1 - \beta_m) g_t$$

gradient estimate g_t

$$v_{t+1} = \beta_v v_t + (1 - \beta_v) g_t^2$$

$(\cdot)^2$ is componentwise

$$\hat{m}_{t+1} = \frac{1}{1 - \beta_m^t} m_{t+1}, \quad \hat{v}_{t+1} = \frac{1}{1 - \beta_v^t} v_{t+1}$$

transient correction

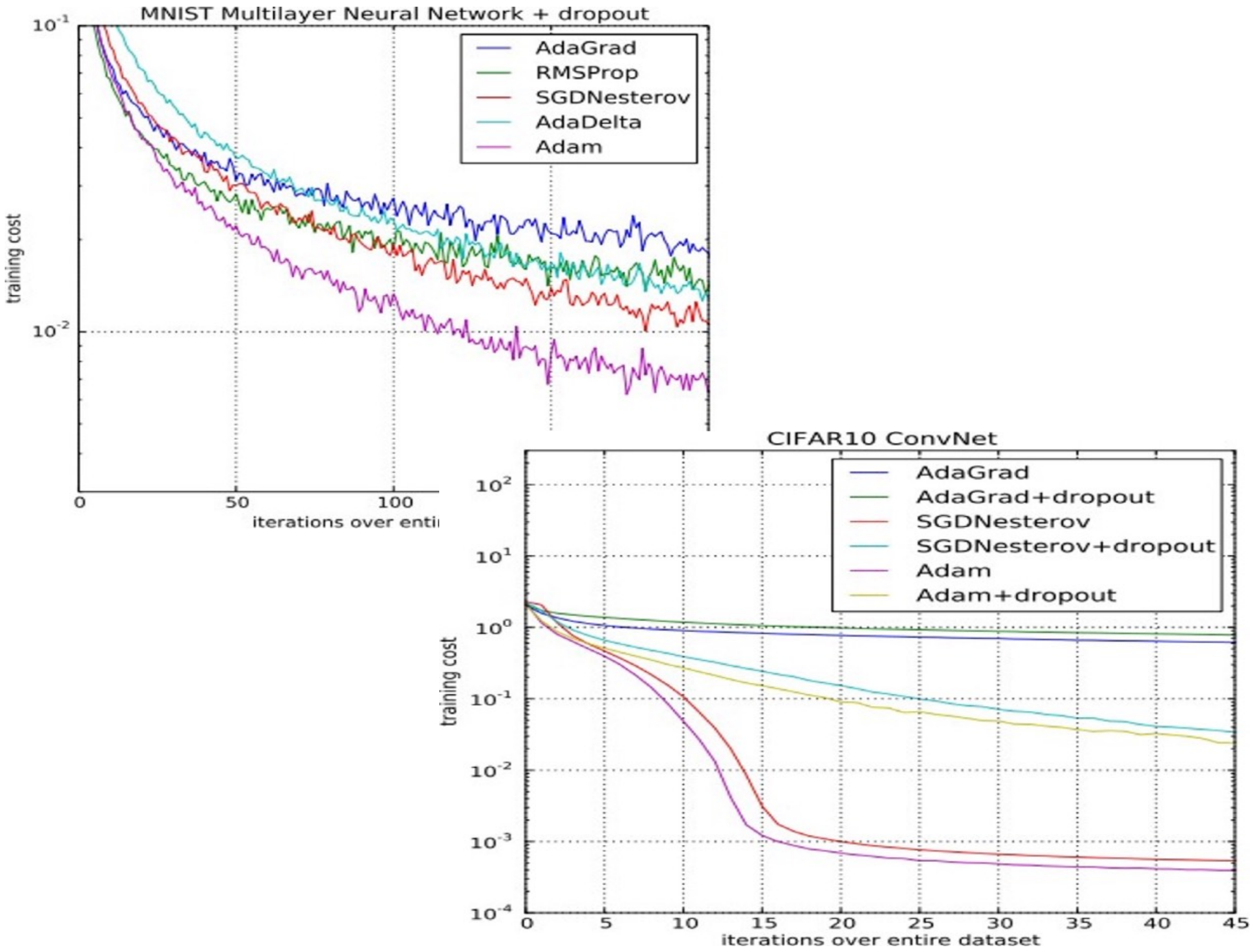
$$\theta_{t+1} = \theta_t - \gamma \frac{1}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1}$$

$\sqrt{\cdot}$ is componentwise

- Four hyperparameters: moving average $\beta_m, \beta_v \in (0,1)$, learning rate $\gamma > 0$, curvature regularizer $\epsilon > 0$
- Memory for two arrays same size as θ_t
- Warning! Not proven to converge (and in fact does not converge in some cases)
    - there are other methods that try to use gradient variance while guaranteeing better theoretical properties, but none uniformly better than Adam in practice

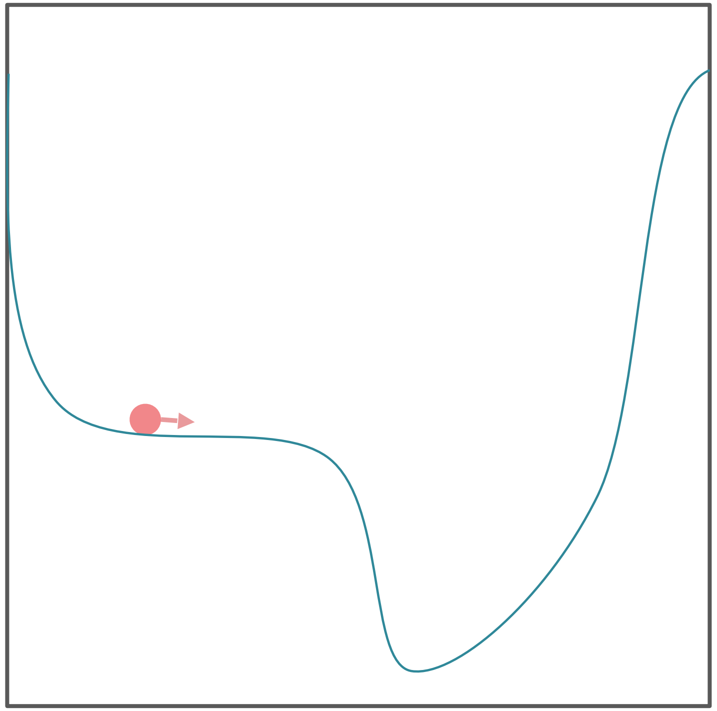
[figure credit: Kingma & Ba, 2015]

Adam performance



Terminating Gradient Descent

- For non-convex surfaces, the gradient’s magnitude is often not a good metric for proximity to a minimum



Terminating Gradient Descent "Early"

- For non-convex surfaces, the gradient's magnitude is often not a good metric for proximity to a minimum
- Combine multiple termination criteria: e.g., only stop if enough iterations have passed and the improvement in error is small
- Alternatively, terminate early by using a validation data set: if the validation error starts to increase, just stop!
  - Or go a bit past minimum and backtrack
  - Early stopping asks like regularization by **<u>limiting how much of the hypothesis set</u>** is explored

Neural Networks and Regularization

- Minimize $\ell_{\mathcal{D}}^{AUG}\left(W^{(1)}, \dots, W^{(L)}, \lambda_C\right)$

$$= \ell_{\mathcal{D}}\left(W^{(1)}, \dots, W^{(L)}\right) + \lambda_C \Omega\left(W^{(1)}, \dots, W^{(L)}\right)$$

e.g. L2 regularization

$$\Omega\left(W^{(1)}, \dots, W^{(L)}\right) = \sum_{l=1}^{L} \sum_{i=0}^{d^{(l-1)}} \sum_{j=1}^{d^{(l)}} \left(w_{j,i}^{(l)}\right)^2$$

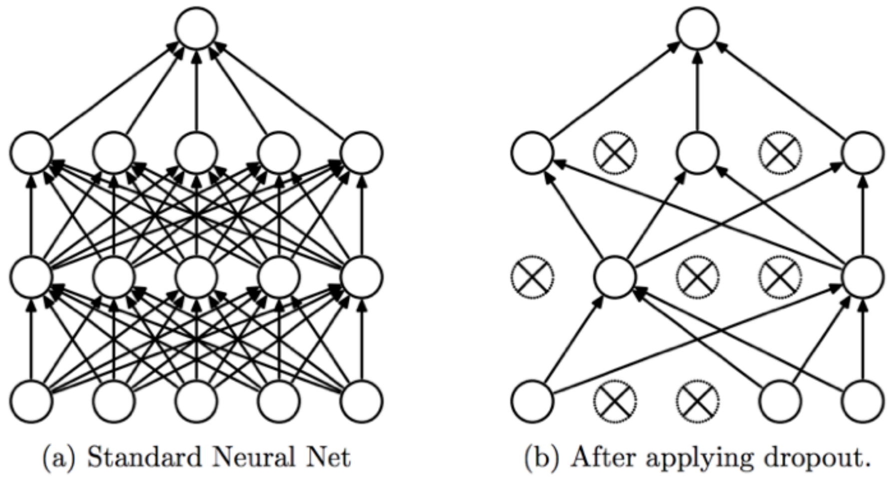
Neural Networks and “Strange” Regularization (Srivastava et al., 2014)

- Dropout
  - In each iteration of gradient descent, randomly remove some of the nodes in the network

Source: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Neural Networks and “Strange” Regularization (Srivastava et al., 2014)

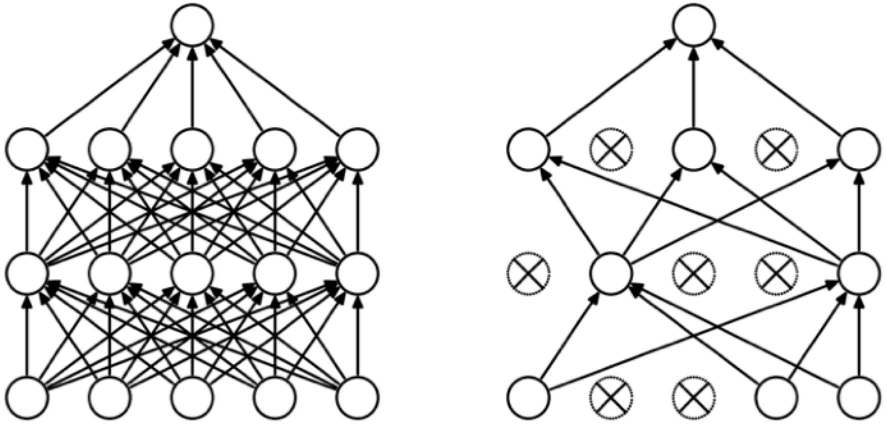
- Dropout
  - In each iteration of gradient descent, randomly remove some of the nodes in the network



Source: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Neural Networks and “Strange” Regularization (Srivastava et al., 2014)

- Dropout
  - In each iteration of gradient descent, randomly remove some of the nodes in the network
  - Compute the gradient using only the remaining nodes
  - The weights on edges going into and out of “dropped out” nodes are not updated



(a) Standard Neural Net (b) After applying dropout.

Source: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Normalize

- Typical layer: $f(Wx + b)$: matrix W, vector x, b, componentwise nonlinear activation f
- Many activations are most interesting near zero
  - sigmoid's non-constant region
  - ReLU's derivative discontinuity
  - …
- Idea: auto-shift and auto-scale inputs to be in this neighborhood
  - but let the network learn to move away if needed
- Hope: normalization helps generalization
  - in practice: effects poorly understood, but can be positive

Layer norm

- Given $a = Wx \in \mathbb{R}^d$
  - define $\bar{a} = \frac{1}{d}\sum_{i=1}^{d} a_i$
  - define $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d} (a_i - \bar{a})^2 + \epsilon$
  - define $a' = \frac{a - \bar{a}}{\sigma}$

tiny number, to prevent divide-by-zero

- New version of layer, with layer norm: $f(g \circ a' + b)$
  - new componentwise *gain* parameter g lets us pick scale
  - existing bias parameter b lets us shift away from zero
  - but by default (if $g = (1, 1, \ldots)^\top$ and $b = 0$), inputs to f have mean zero and variance 1 across layer for each example

Backpropagation Learning Objectives

You should be able to…

- Differentiate between a neural network diagram and a computation graph
- Construct a computation graph for a function as specified by an algorithm
- Carry out backpropagation on an arbitrary computation graph
- Construct a computation graph for a neural network, identifying all the given and intermediate quantities that are relevant
- Instantiate the backpropagation algorithm for a neural network
- Instantiate an optimization method (e.g. SGD) and a regularizer (e.g. L2) when the parameters of a model are comprised of several matrices corresponding to different layers of a neural network
- Apply the empirical risk minimization framework to learn a neural network
- Use the finite difference method to evaluate the gradient of a function
- Employ basic matrix calculus to compute vector/matrix/tensor derivatives.