Deep nets

10-701 Introduction to Machine Learning

Geoff Gordon and Pradeep Ravikumar

Reminders

- Quiz 2: Friday (usual place/time)
  - covers HW2 programming & written
  - designed for 15–20 min, you will have 30 min
  - should not require much studying: best advice is to look over your HW2 submission

Model = learnable program

Logistic regression

$z \leftarrow \sum_i w_i x_i$

return $\sigma(z)$

Perceptron

$z \leftarrow \sum_i w_i x_i$

if $z \geq 0$*:*
  return T
else:
  return F

Decision tree

if $x_1 \leq 3$*:*
  if $x_2 \leq 5$*:*
    return T
  else:
    return F
else:
  return T

- Can think of a model as a *decision function (program)* that has *learnable choices*
  - weight vector w in perceptron or logistic regression
  - split axes and thresholds in decision tree

Model = learnable program

Logistic regression

$z \leftarrow \sum_i w_i x_i$

return $\sigma(z)$

Perceptron

$z \leftarrow \sum_i w_i x_i$

if $z \geq 0$*:*

  return T

else:

  return F

Decision tree

if $x_1 \leq 3$*:*

  if $x_2 \leq 5$*:*

    return T

  else:

    return F

else:

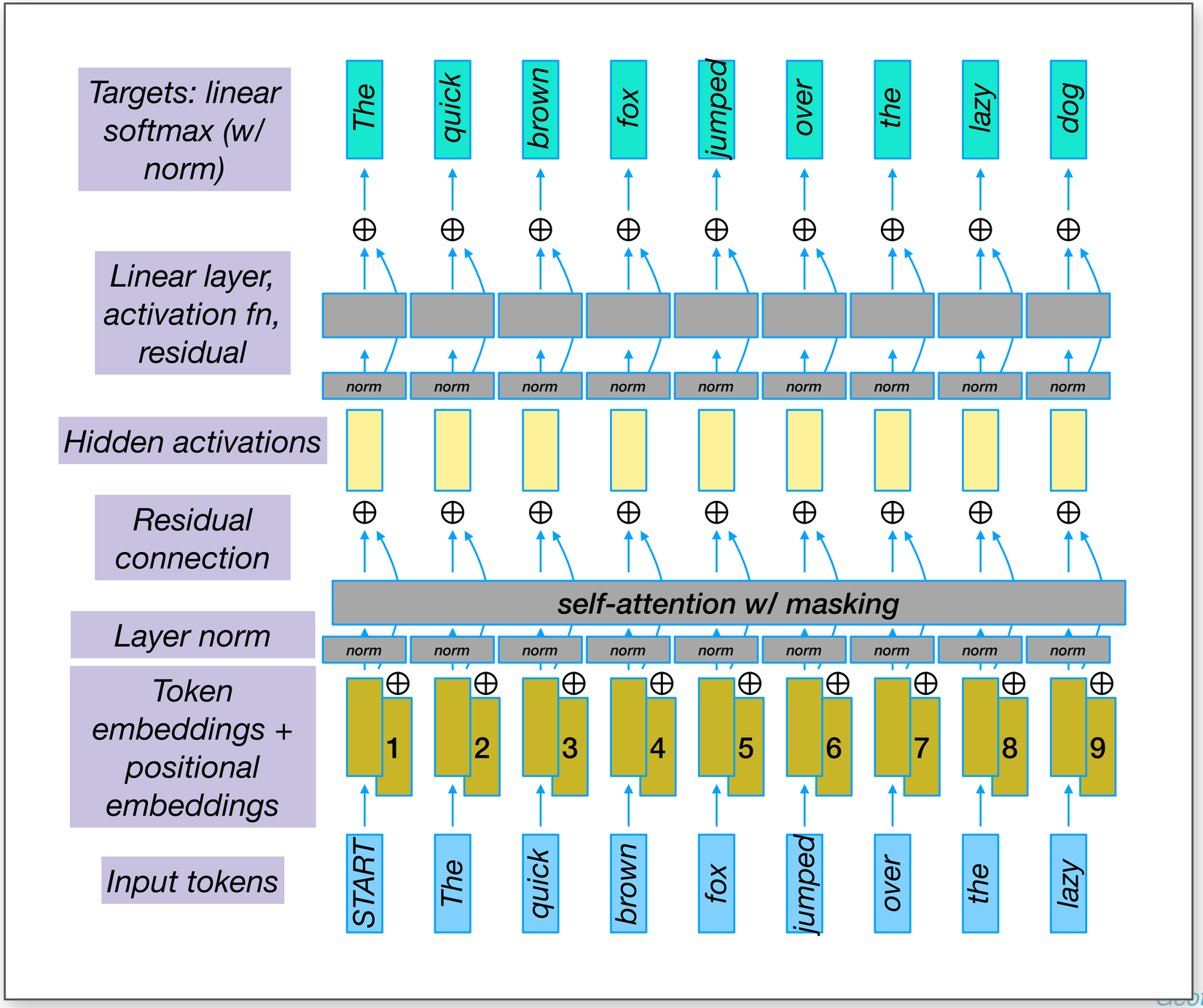
  return T

- Complete by adding an objective and optimizer
- E.g., mutual info and top-down induction (ID3)
- E.g., MLE + SGD (linear/logistic regression, deep nets)

Deep neural networks

- Decision functions composed from blocks of
  - trainable linear functions
  - fixed (not trainable) nonlinear functions, often chosen from a small list like $\sigma(z)$, $z/\|z\|$, $[z]_+$
  - with depth $\gg 1$
- Often:
  - entire function is continuously differentiable at least once wrt parameters (sometimes w/ sparse breakpoints)
  - trained by SGD
- Includes linear & logistic regression, perceptron as special cases *[Q: what about decision trees?]*

Deep neural networks



What is depth?

notation:

$$\max(0, z) = [z]_+$$

*is **ReLU** or **positive part**, one of the most common nonlinearities / activation functions for deep nets*

- Defn: ***depth*** = minimum number of sequential steps in a computation, given arbitrarily many parallel processors
- In contrast to ***width*** = max number of processors used, or ***work*** = total number of processor-steps
- E.g.,
  - $\max(0, 1 + 3 \cdot \max(0, 4x))$

What is depth?

- Width, depth, work are only defined up to constant factors, since we can fiddle with the capability of the processors
  - convention for deep nets: a processor (a ***neuron***) in one step computes linear function and fixed nonlinearity
  - a ***layer*** is a group of steps that can be performed in parallel

Example

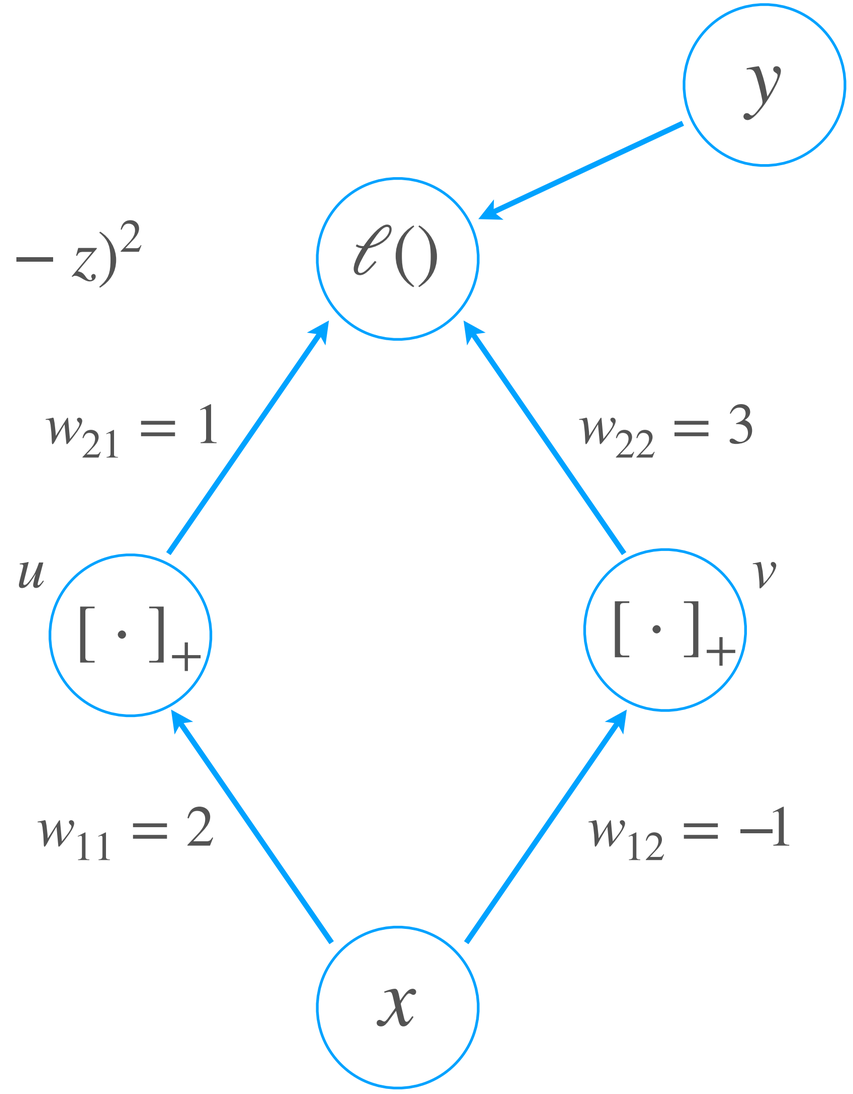
a simple neural network predicting label y from input x under squared error

$$\ell(y, z) = \frac{1}{2}(y - z)^2$$

2 layers

width 2

work 3



Label

Output

Hidden

Input

*each node computes a real number (an **activation**)*

*start at inputs, propagate toward outputs (**forward pass**)*

by convention, weights label edges, but are part of the computation of the incident node

try $x = 1, y = 1$

Example

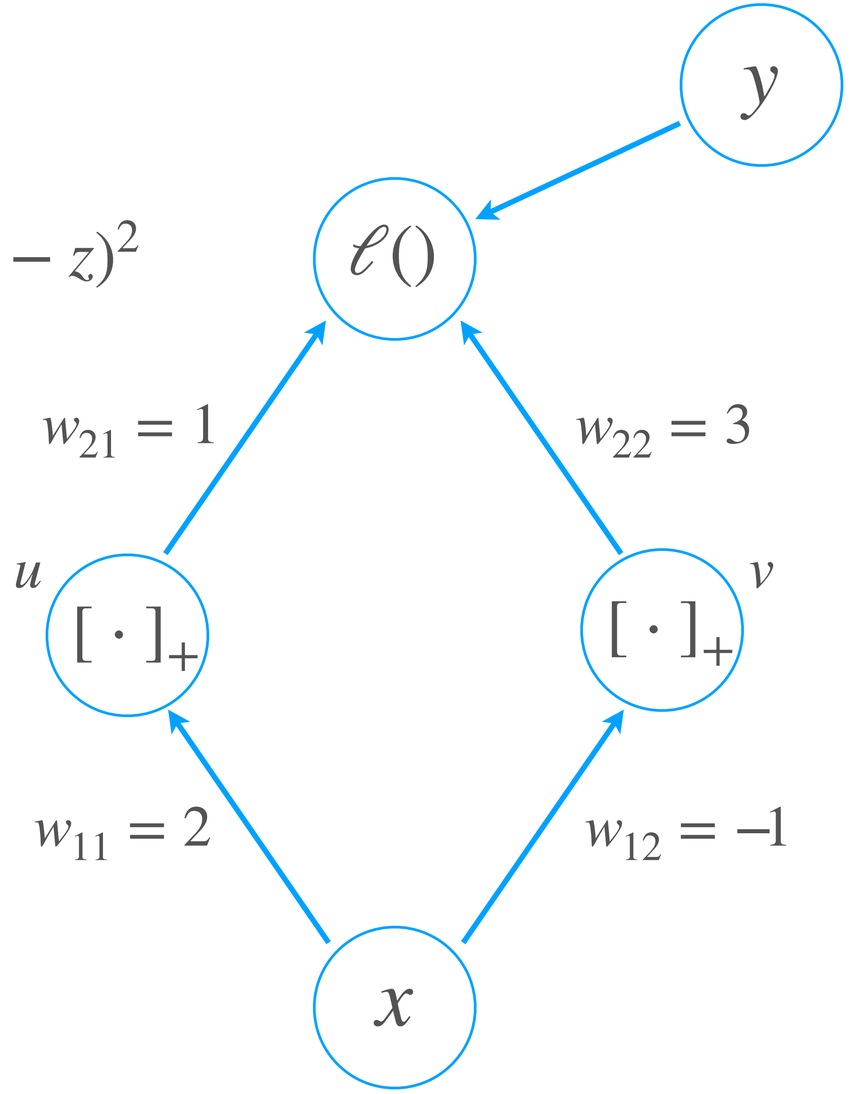
a simple neural network predicting label y *from input* x *under squared error*

$$\ell(y, z) = \frac{1}{2}(y - z)^2$$

2 layers

width 2

work 3



each node computes a real number (an ***activation****)*

*start at inputs, propagate toward outputs (****forward pass****)*

by convention, weights label edges, but are part of the computation of the incident node

How to train your ~~dragon~~ deep net

- SGD!
- Draw examples (x, y) from training set (or minibatches)
- Run network forward from x to get a prediction
- Compare to y to calculate loss function
- Calculate gradient g: derivative of loss wrt network weights
- Update network weights by subtracting a multiple of g
  - perhaps w/ momentum, Adam, etc.

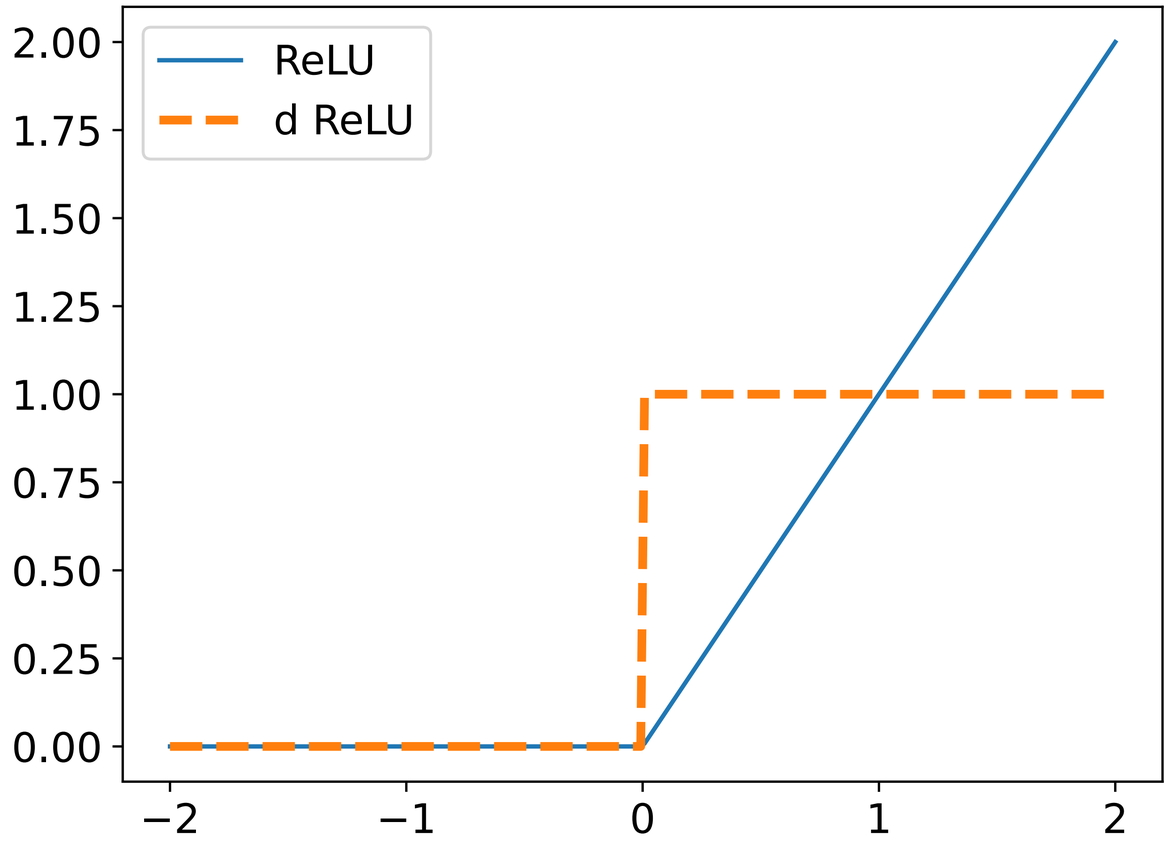
How to train your ~~dragon~~ deep net

how might we do this?

- SGD!
- Draw examples (x, y) from training set (or minibatches)
- Run network forward from x to get a prediction
- Compare to y to calculate loss function
- Calculate gradient g: derivative of loss wrt network weights
- Update network weights by subtracting a multiple of g
  - perhaps w/ momentum, Adam, etc.

Gradient of ReLU

$$[u]_+ = \begin{cases} u & \text{if } u \geq 0 \\ 0 & \text{if } u < 0 \end{cases}$$

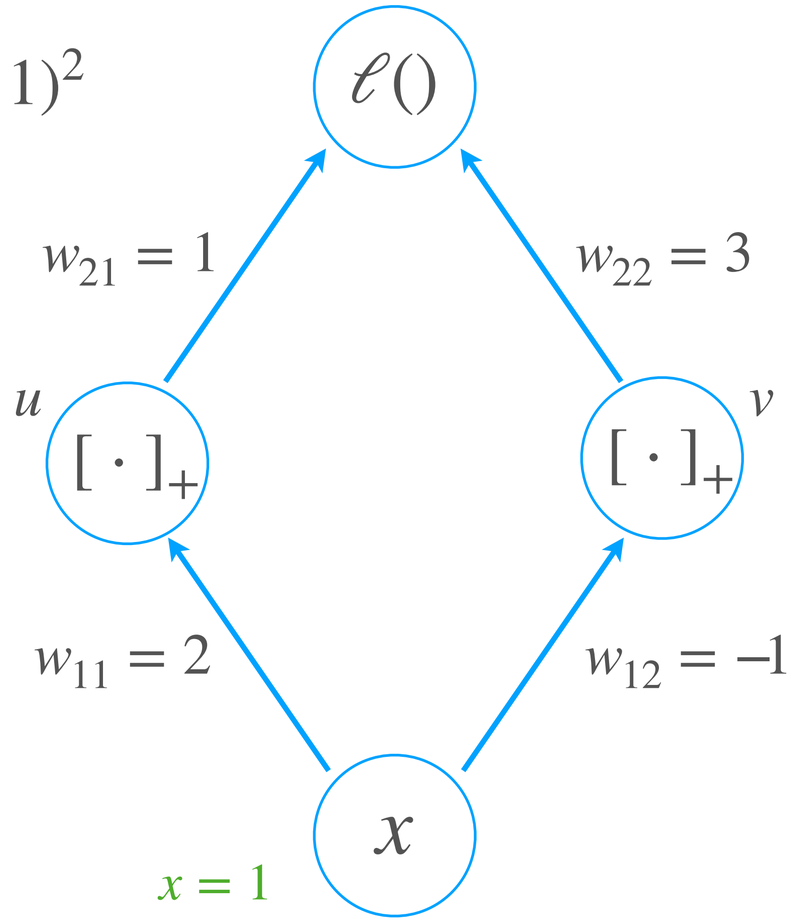


- Derivative of $[u]_+$: $S(u) = \begin{cases} & \text{if} \\ & \text{if} \end{cases}$

Calculate derivatives: first wrt x

fix $y = 1$ for simplicity

$$\ell(z) = \frac{1}{2}(z-1)^2$$



Output

Hidden

Input

Goal: find $d\ell$ as a function of dx

Intermediate: find $d\ell$ as a function of inputs of each node

Professor provides

$$x = 1$$

$w_{11} = 2$

$$u = [w_{11}x]_+$$

$w_{12} = -1$

$$v = [w_{12}x]_+$$

$$w_{21} = 1$$
$$w_{22} = 3$$

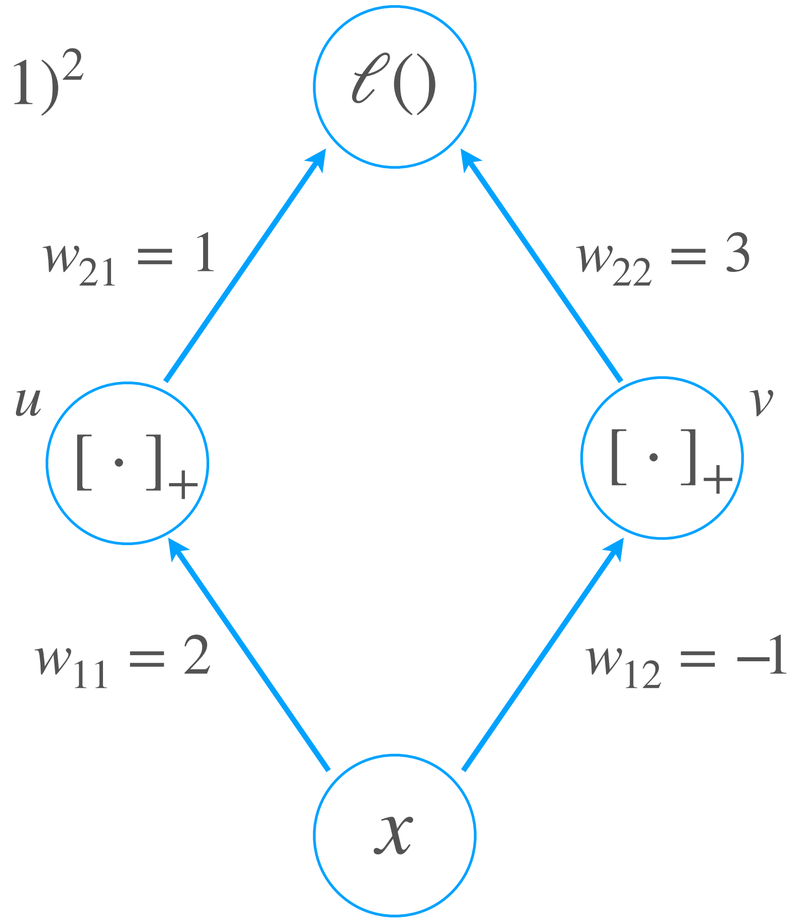
$$\ell = \frac{1}{2}(w_{21}u + w_{22}v - 1)^2$$

How did we get the derivative?

- Loop: start at top (output) node
  - calculate derivative of loss wrt inputs of top node
- Visit nodes in backward order, from output to input (a ***backward pass***)
  - so that we've looked at all successors of a node before we visit it, and know derivative of ***overall*** loss (at top node) wrt ***successor node's*** inputs
  - this is our loop invariant
- At each node, use chain rule to find derivative of loss wrt ***this node's*** inputs, using known derivatives wrt ***successor nodes'*** inputs

Calculate derivatives: now wrt w

$$\ell(z) = \frac{1}{2}(z-1)^2$$



Output

Hidden

Input

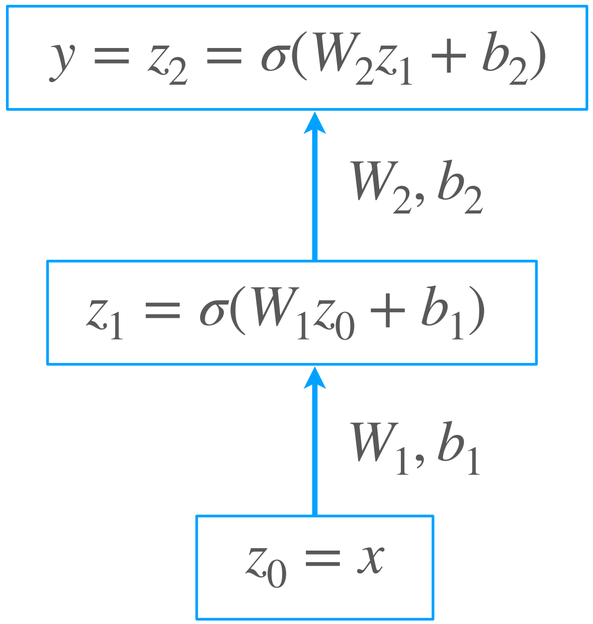
How did we get the derivative?

- At each node, we already computed all of the information we need (activations from forward pass and intermediate gradients from backward pass)
  - weight gradients can then be computed locally
- Typical setup:
  - store activations during forward pass, accumulate weight gradients during backward pass
  - but it's possible to mix cacheing and recomputation to change tradeoff of memory vs. runtime

Backprop

- Forward and backward passes are ***automatable***
  - a few simple operations we do over and over
  - linear function, activation function, activation derivative, chain rule
- Resulting algorithm called ***backward propagation*** or ***backprop***
- Responsible for neural networks' initial rise to popularity

Backprop setup



$z_t, b_t \in \mathbb{R}^{d_t}$

$W_t \in \mathbb{R}^{d_t \times d_{t-1}}$

- Suppose forward pass is: $z_i = \sigma(W_i z_{i-1} + b_i)$
  - activations z_i, activation function σ (componentwise), weights W_i, biases b_i (weight for constant feature)
  - that is, set $z_0 = x$, iterate $i = 1,2,\ldots,K$ computing $u_i = W_i z_{i-1} + b_i$ and $z_i = \sigma(u_i)$, read off output $y = z_K$

pre-activation

Backprop algorithm

- Goal: derivatives of loss L wrt weights W_i and biases b_i
  - as a byproduct, also wrt activation vectors z_i
- Loop invariant: we know $dL = \langle r_i, dz_i \rangle$
  - vector r_i is one of the activation derivatives
  - initialize at layer K w/ gradient of loss function (wrt its input, the final activation z_K)
- At each layer from K down to 1:
  - scale by derivative of activation f: define $v_i = f'(u_i) \circ r_i$
  - dL/db_i is just v_i
  - dL/dW_i is outer product $v_i z_{i-1}^\top$
- Update invariant: $r_{i-1} = W_i^\top v_i$

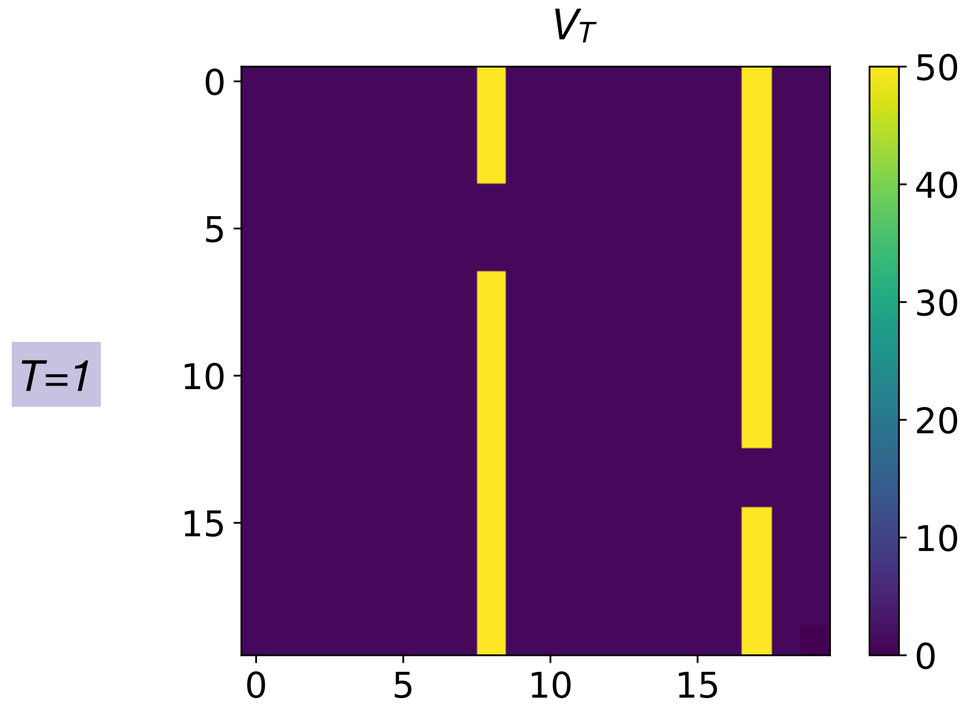
Fitting the model by SGD, redux

- Sample an (x, y)
  - forward pass to compute $f_w(x)$ using weights w, saving intermediate values (hidden activations)
  - find loss, e.g. $L = \frac{1}{2}(y - f_w(x))^2$
  - backward pass to compute dL as a linear function of dw
- Take an optimizer step on w: e.g., step downhill using derivative (coefficient of dw)
- Or, for each (x, y) in minibatch
  - do above, accumulate derivatives (coefficients of dw)
  - step using sum of derivatives

Why depth?

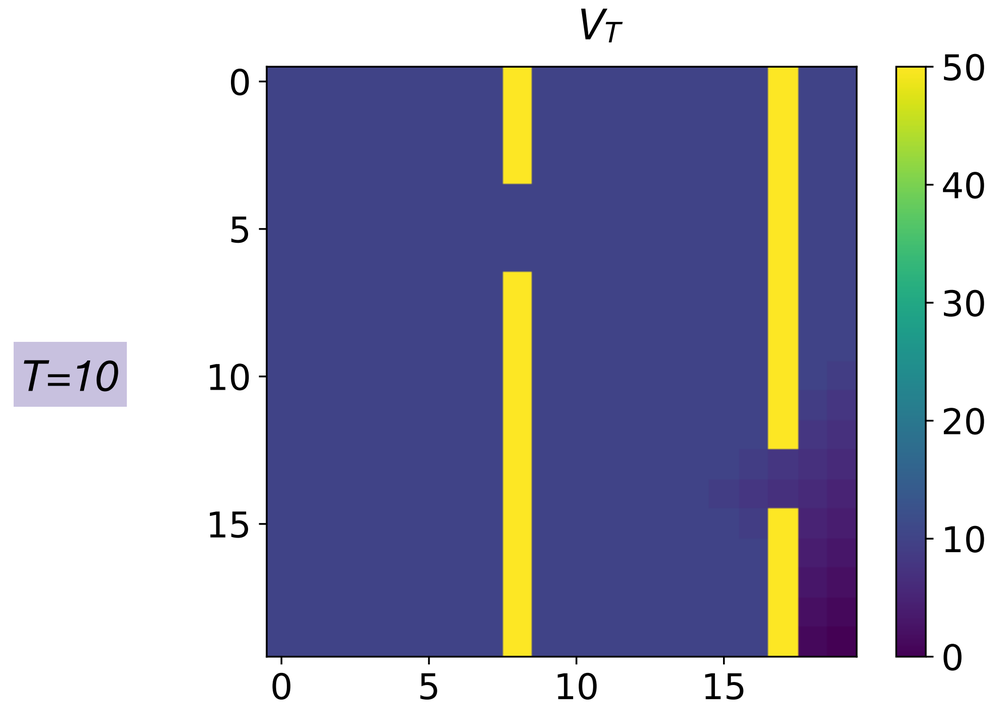
- ML needs two things to work:
  - expressivity: model has to be able to represent (something close to) the thing we want to learn
  - simplicity: need a regularizer or complexity measure that rules out bad-yet-consistent hypotheses — depth, width, work, norm of weights, …
- Tension:
  - expressivity: want to learn e.g. arbitrary Python code
  - simplicity: there are a ***lot*** of Python programs to search through, and we don't know how to rule out bad ones
- Depth seems to facilitate a tradeoff: simple families of deep programs contain useful functions at low complexity levels

Depth example: shortest paths



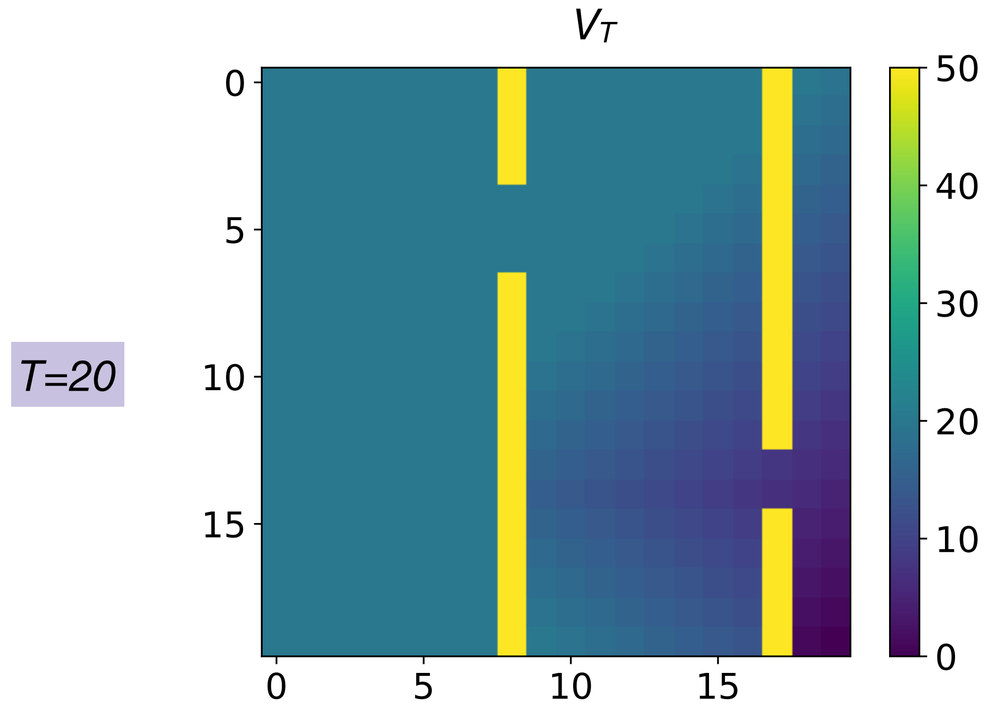
- Iterate a simple function:
  - matrix V, initialized to zero
  - constant matrix c, same size as V [cost of visiting state]
- For $t = 1,2,\ldots T$
  - $V \leftarrow c + \min(V[\text{left}], V[\text{right}], V[\text{up}], V[\text{down}])$
  - $V_{d,d} \leftarrow 0$ [goal state]

Depth example: shortest paths



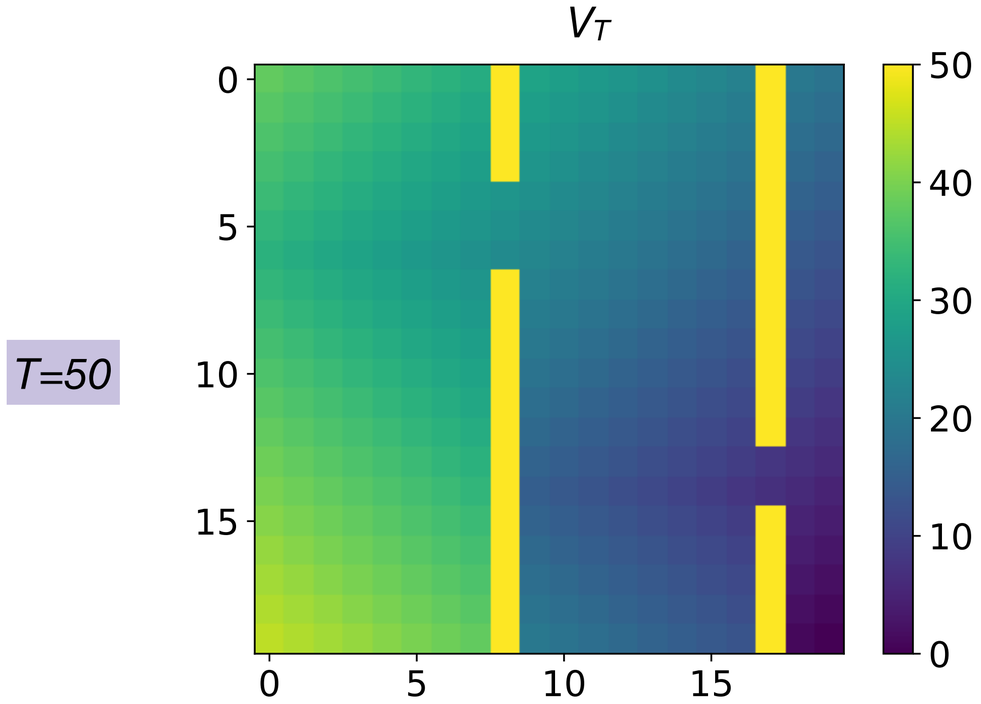
- Iterate a simple function:
  - matrix V, initialized to zero
  - constant matrix c, same size as V
- For $t = 1,2,\ldots T$
  - $V \leftarrow c + \min(V[\text{left}], V[\text{right}], V[\text{up}], V[\text{down}])$
  - $V_{d,d} \leftarrow 0$

Depth example: shortest paths



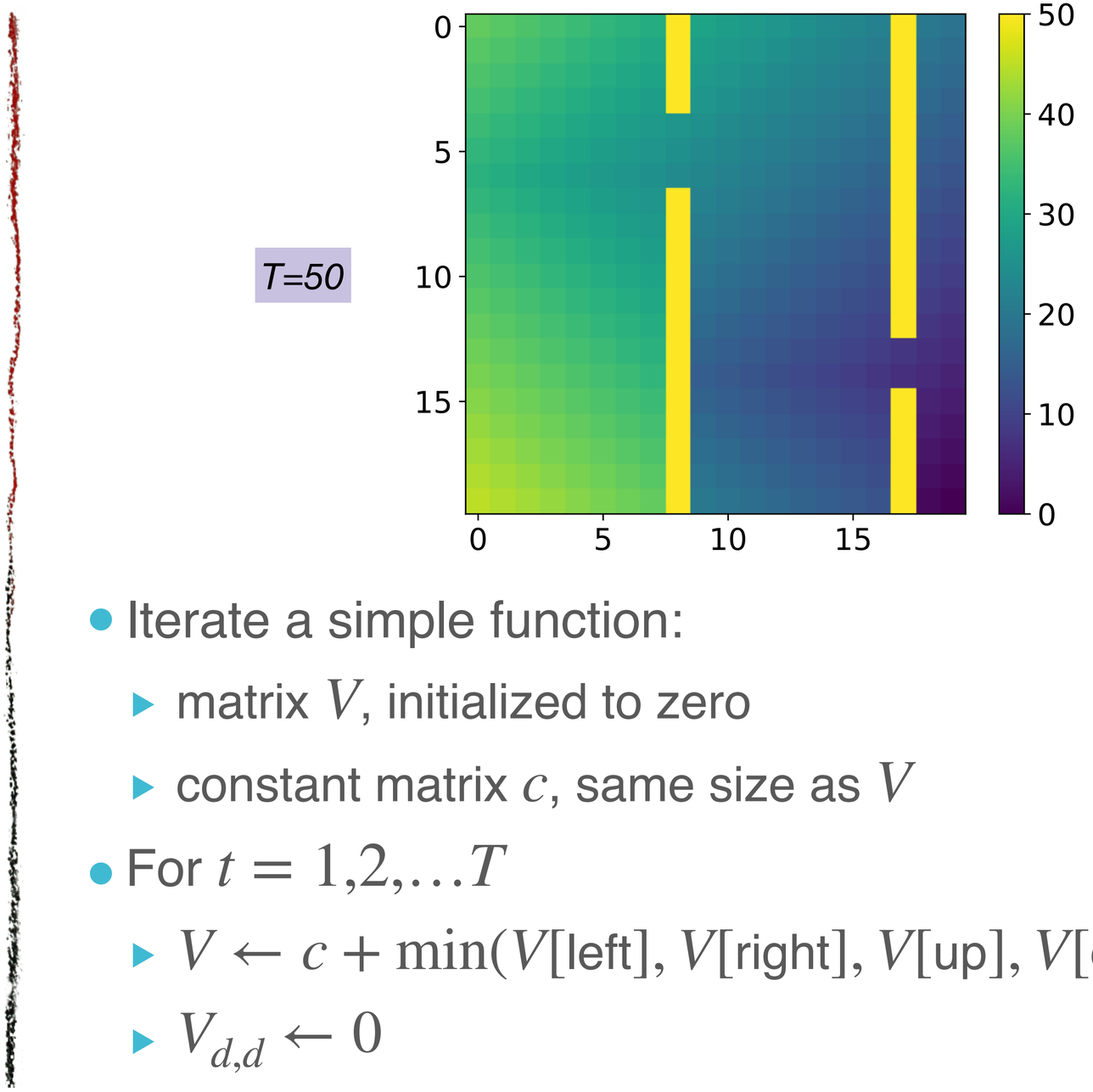
- Iterate a simple function:
  - matrix V, initialized to zero
  - constant matrix c, same size as V
- For $t = 1,2,\ldots T$
  - $V \leftarrow c + \min(V[\text{left}], V[\text{right}], V[\text{up}], V[\text{down}])$
  - $V_{d,d} \leftarrow 0$

Depth example: shortest paths



- Iterate a simple function:
  - matrix V, initialized to zero
  - constant matrix c, same size as V
- For $t = 1,2,\ldots T$
  - $V \leftarrow c + \min(V[\text{left}], V[\text{right}], V[\text{up}], V[\text{down}])$
  - $V_{d,d} \leftarrow 0$

Depth example: shortest paths



T=50

V_{ij} = *cost of shortest path*

$(i, j) \leftrightarrow$ *bottom right*

- Iterate a simple function:
  - matrix V, initialized to zero
  - constant matrix c, same size as V
- For $t = 1,2,\ldots T$
  - $V \leftarrow c + \min(V[\text{left}], V[\text{right}], V[\text{up}], V[\text{down}])$
  - $V_{d,d} \leftarrow 0$

Can prove that depth makes a big difference to expressivity

- For any k, compare programs that are
  - *shallow:* depth $\Theta(k)$, or
  - *deep:* depth $\Theta(k^3)$
- ***Thm*** *[Telgarsky, 2016]:* there are (many) functions that we can implement with *constant* width in deep programs but require *exponential* width $\Omega(2^k)$ even to approximate with shallow programs
  - processor step: can be quite general, e.g., piecewise polynomial fn of previous outputs

Why are classical models often shallow?

- Optimization
  - not obvious how to find good model in deep model class (need solutions to regularization, local minima, vanishing/ exploding gradients, sharpness, …)
- Generalization
  - deep model class typically has very high VC dimension / Rademacher complexity / anything else we can measure
  - need to co-design architecture and optimizer to get implicit regularization
- These used to be big problems
  - when I took 10-701:
    “don’t try to use SGD to train a model with depth > 2 or 3”
- Spoiler: not any more

Fitting the model, 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 most classes for models)

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

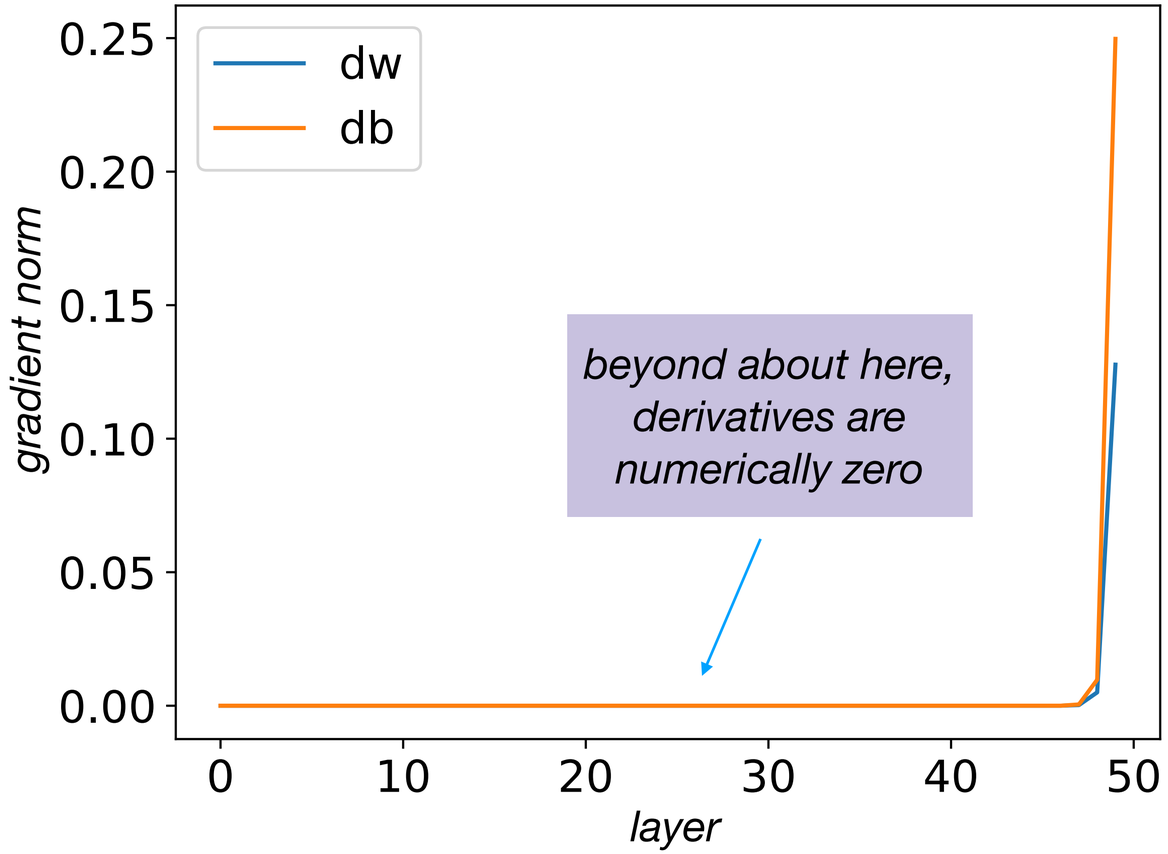
Make it deeper

- Now that we have autodiff, no problem differentiating a 50-layer network!
- Let's try it:
  - $x, z_i, w_i, b_i \in \mathbb{R}$
  - $z_0 = x$
  - $z_i = \sigma(w_i z_{i-1} + b_i)$, i=1..50
  - initialize $w_i, b_i \sim N(0, 0.1^2)$
- Forward pass: $z_{50} = 0.5167$
- Backward pass: →

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Make it deeper

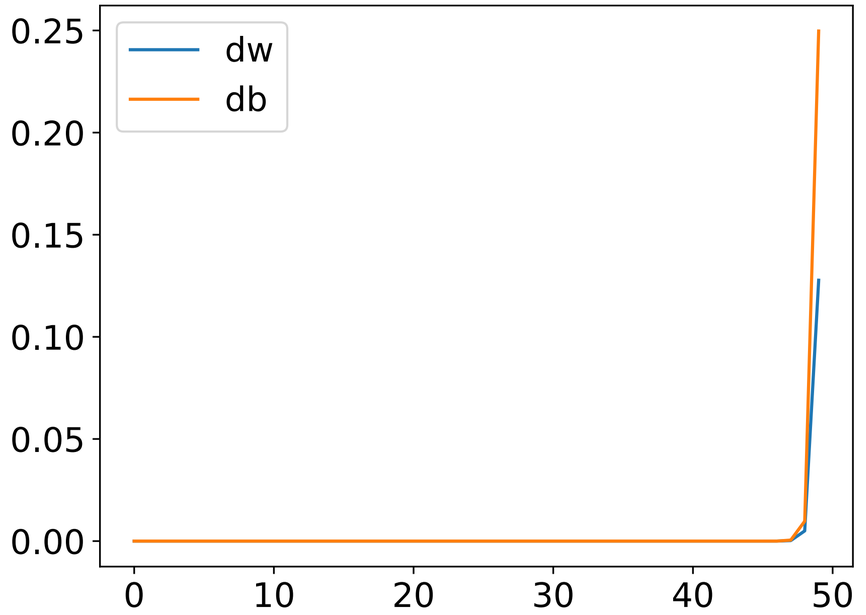
- Now that we have autodiff, no problem differentiating a 50-layer network!
- Let's try it:
  - $x, z_i, w_i, b_i \in \mathbb{R}$
  - $z_0 = x$
  - $z_i = \sigma(w_i z_{i-1} + b_i)$, i=1..50
  - initialize $w_i, b_i \sim N(0, 0.1^2)$
- Forward pass: $z_{50} = 0.5167$
- Backward pass: →



Is this a problem?

- In SGD: effectively no updates to w_i, b_i for $i \leq 45$, and truly zero updates for $i \leq 30$
  - so, we aren't really using a deep model: only last few layers are learnable
  - worse: in forward pass, $z_{30} \approx 0.5$ no matter what x is
    - $\mathrm{d}z_{30} = 0\,\mathrm{d}x$ numerically
    - i.e., we ignore the input

Why does this happen?



- Run backprop:

$$\begin{aligned} dz_{50} &= \sigma'(w_{50}z_{49} + b_{50})w_{50}dz_{49} \\ &= \sigma'(w_{50}z_{49} + b_{50})w_{50}\sigma'(w_{49}z_{48} + b_{49})w_{49}dz_{49} \\ &= \dots \end{aligned}$$

- all σ' terms $\approx \frac{1}{4}$, all w_i terms ≈ 0.1 in magnitude
  - exponential decay: called *vanishing gradients*

Exploding gradients

- Can get a related problem if we initialize to *large* weights
- Keep multiplying gradient by $w_i \rightarrow$ exponential growth
- But growth can be self-limiting:
  - e.g., with sigmoid nonlinearity: $\sigma'(z) \approx 0$ when z large, numerically 0 if $|z| \geq 17$ or so
- So, paradoxically, explosion can also lead to (near-)zero gradients

Just right

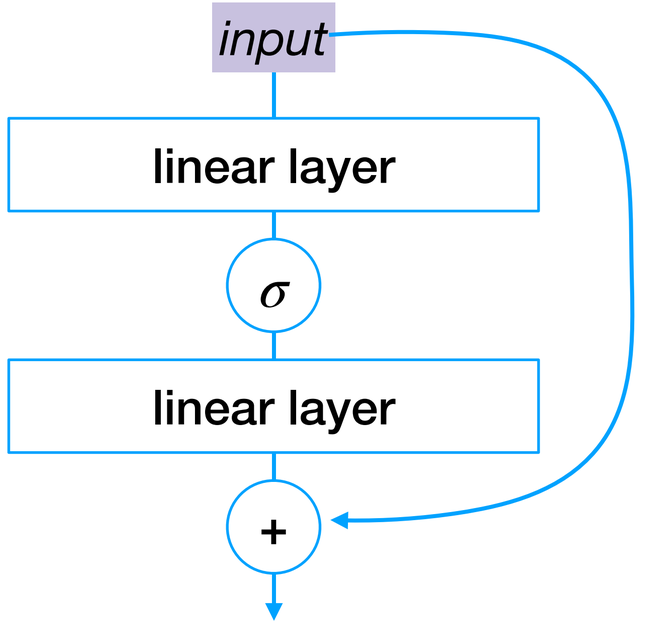
- What if we initialize to moderate-sized weights?
- Helps a lot
  - generally enough to train moderate-depth networks (say, 10 or so)
  - beyond that, it's hard to control for the contribution of σ'
- PyTorch does this by default
  - if we know approximate variance of vector z_t at layer t
  - and we choose variance V of elements of w_t
  - then we can calculate var$(z_t \cdot w_t)$
  - and multiply by an estimate of σ' to get variance of z_{t+1}
  - so adjusting V can keep gradients from growing or shrinking too much

Initialization matters

- Vanishing/exploding gradients
- Symmetry breaking
- ∃ results showing we can get a powerful model by appropriate random initialization, with training only adjusting the weights a tiny distance from starting values
- …

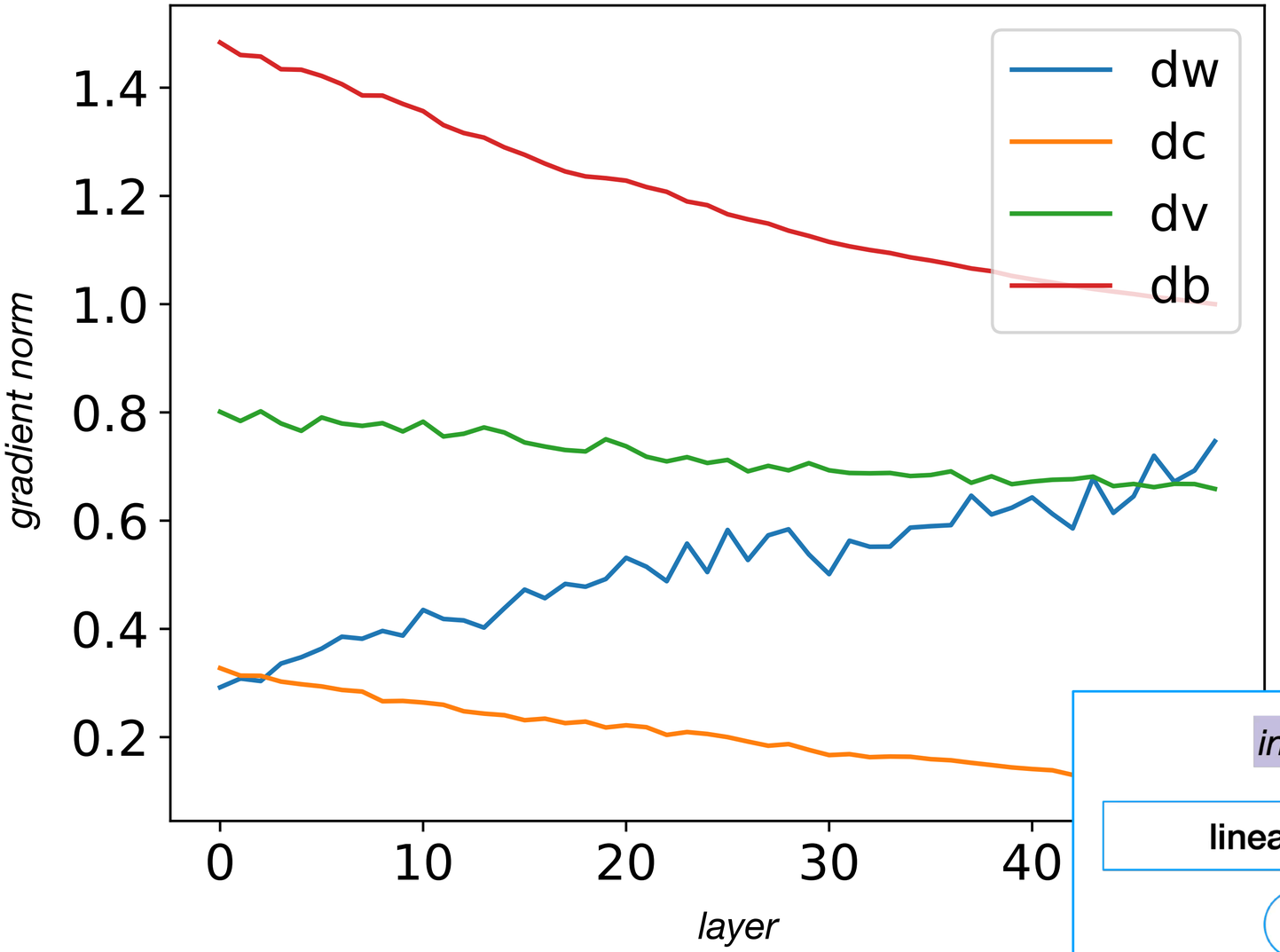
Better yet: residual connection

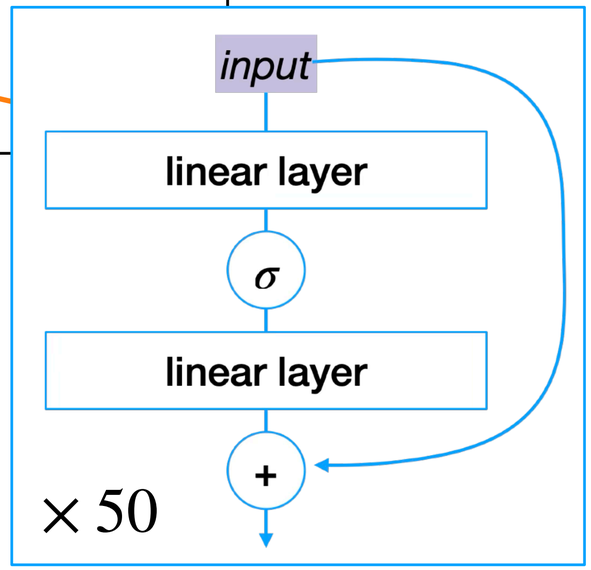
- Change the arrangement of layers:
  - $z_i = z_{i-1} + v_i \sigma(w_i z_{i-1} + c_i) + b_i$
  - can be any nonlinearity
  - weight matrices v_i, w_i
  - bias vectors b_i, c_i
- Sum node means we learn z_i as an *update* to z_{i-1}
- Second linear layer lets us learn sign of update, even if nonlinearity is always positive like $\sigma(z)$ or $[z]_+$
- Can stack this block like any layer
  - appears in famous architectures like ResNet, Transformer



This is an architecture diagram

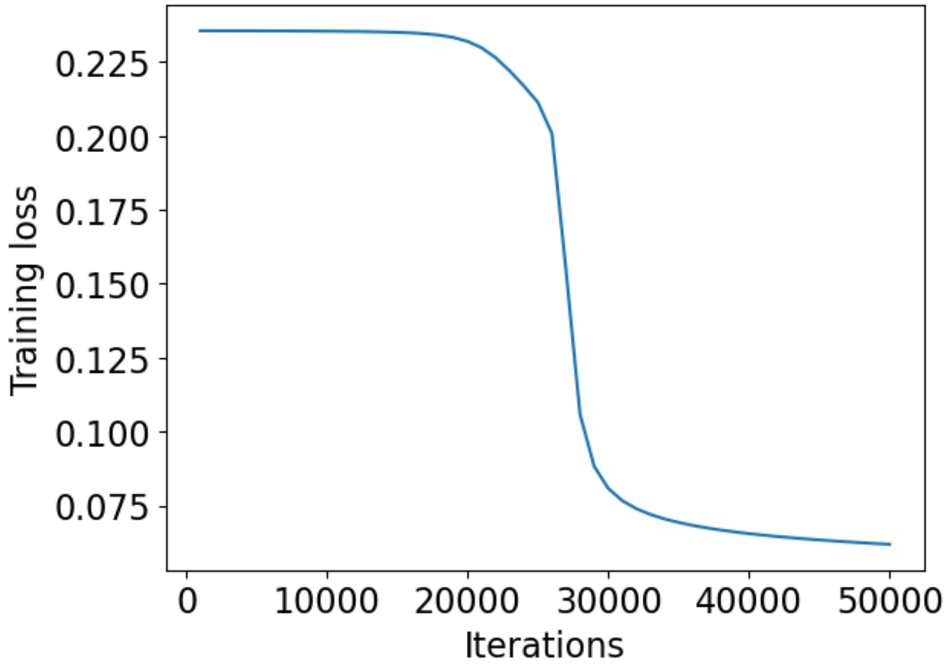
Gradient norm for residual blocks

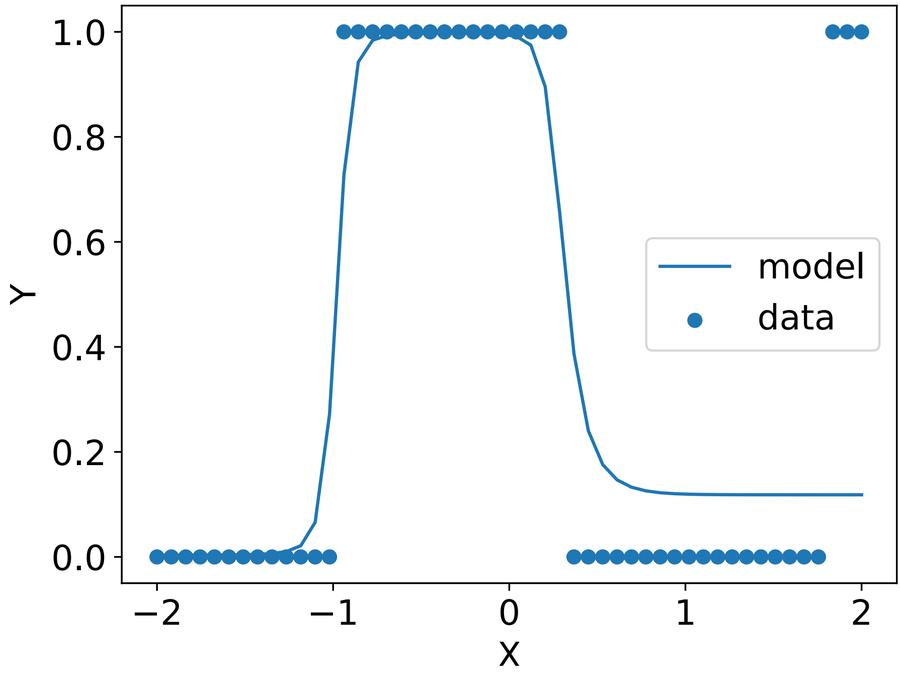




- 50 stacked residual blocks

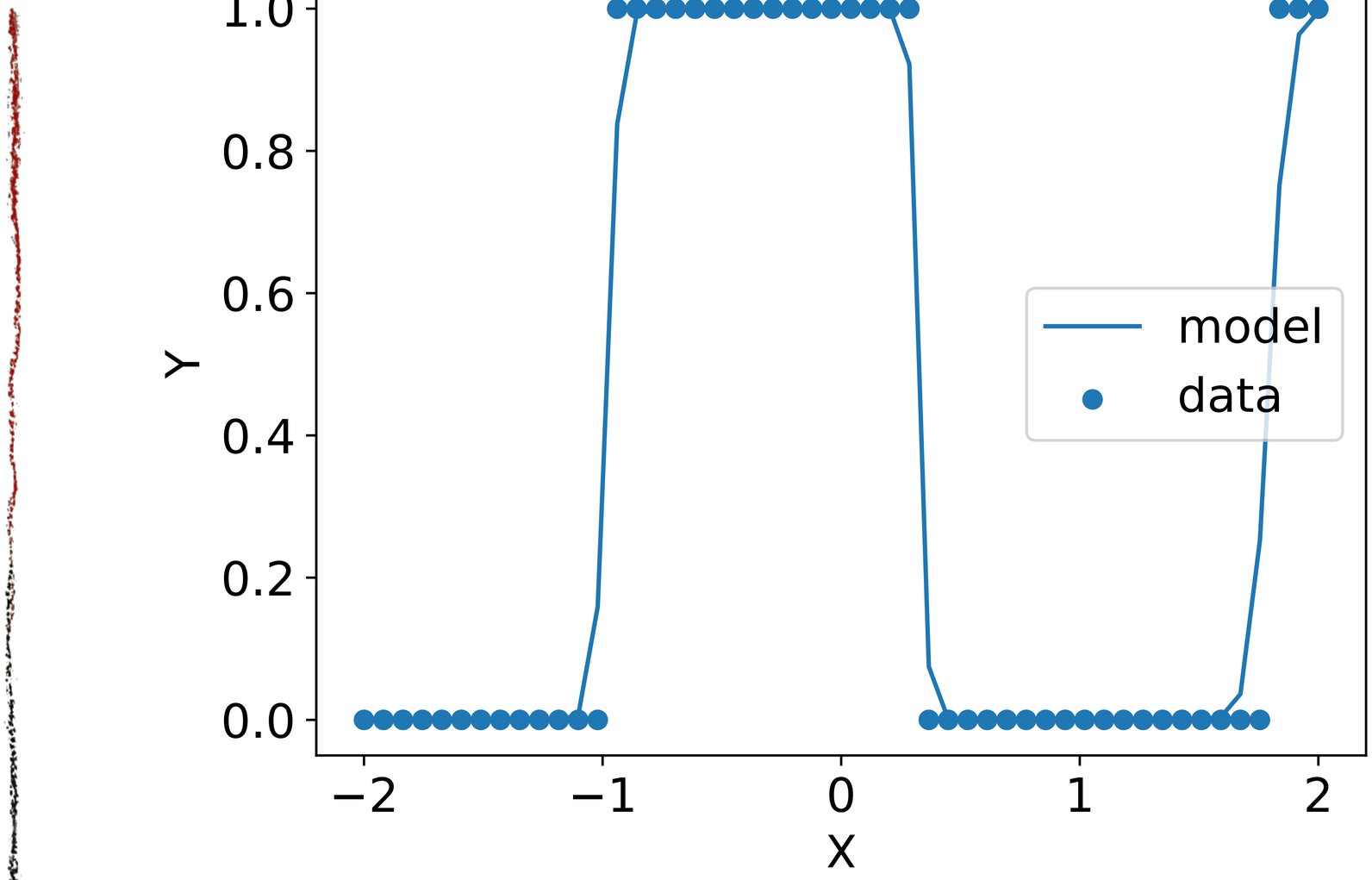
Bigger models seem to make GD/ SGD work better





- Simple dataset of 50 points, 3-layer network
  - enough width to fit training data exactly
- GD, learning rate = 0.01, min $\frac{1}{N}\sum_i (y_i - f_{W_{1:3}, b_{1:3}}(x_i))^2$
  - interestingly: GD doesn't find exact solution

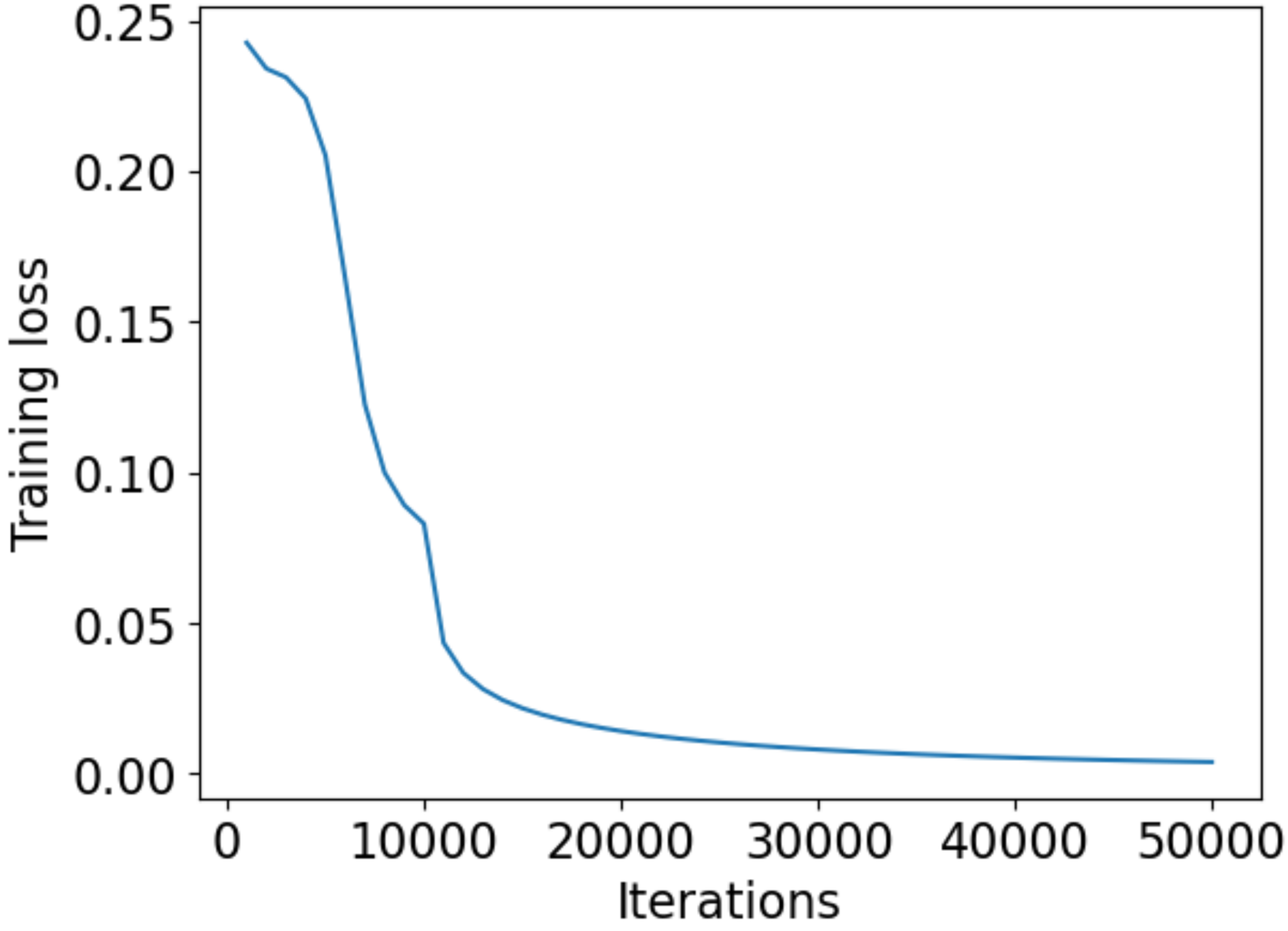
Wider network finds a better optimum



- Multiply width by 100 → perfect on training data

not obvious why

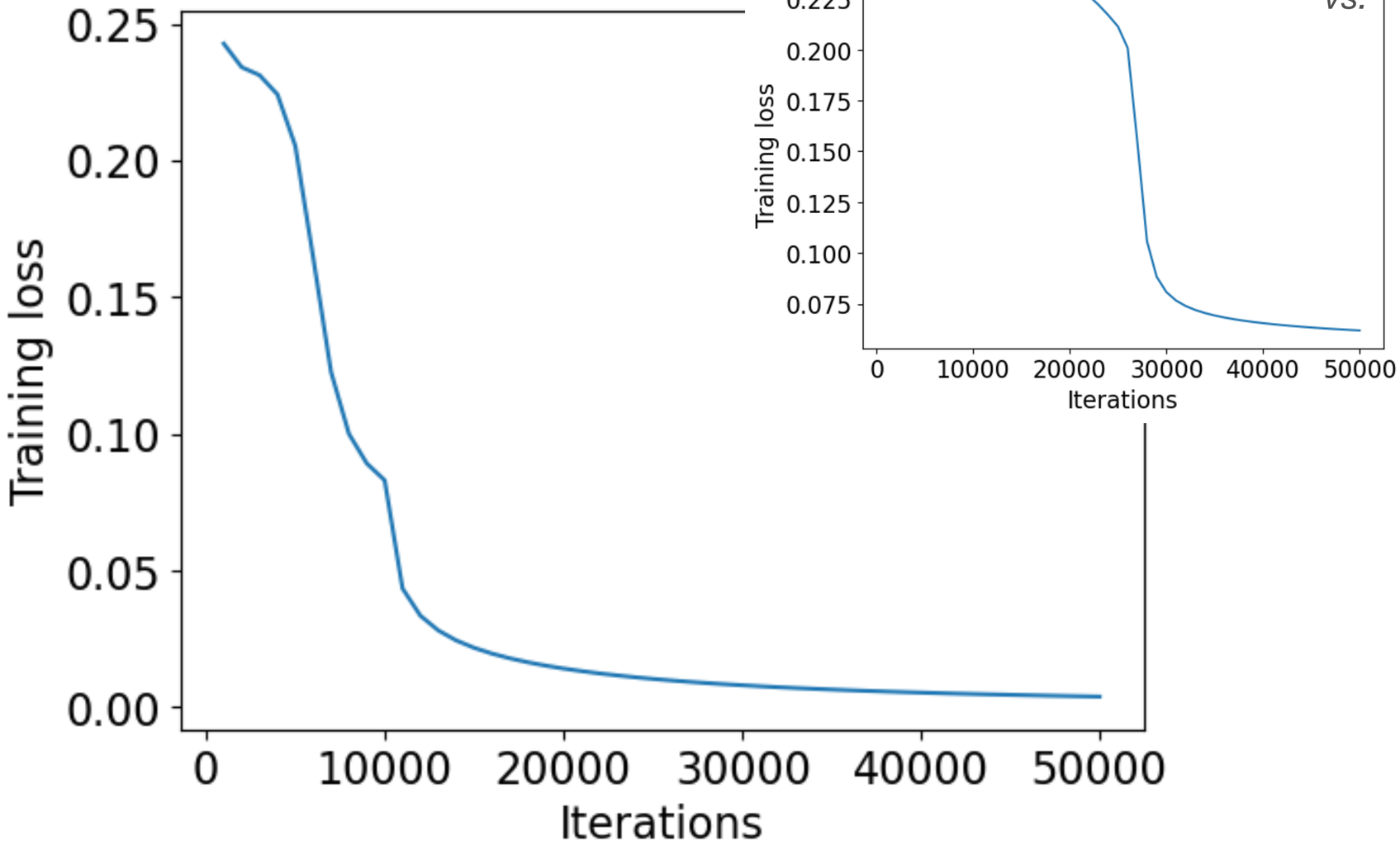
Wider optimizes faster



- Not just better classifier, but fewer epochs to train

not obvious why

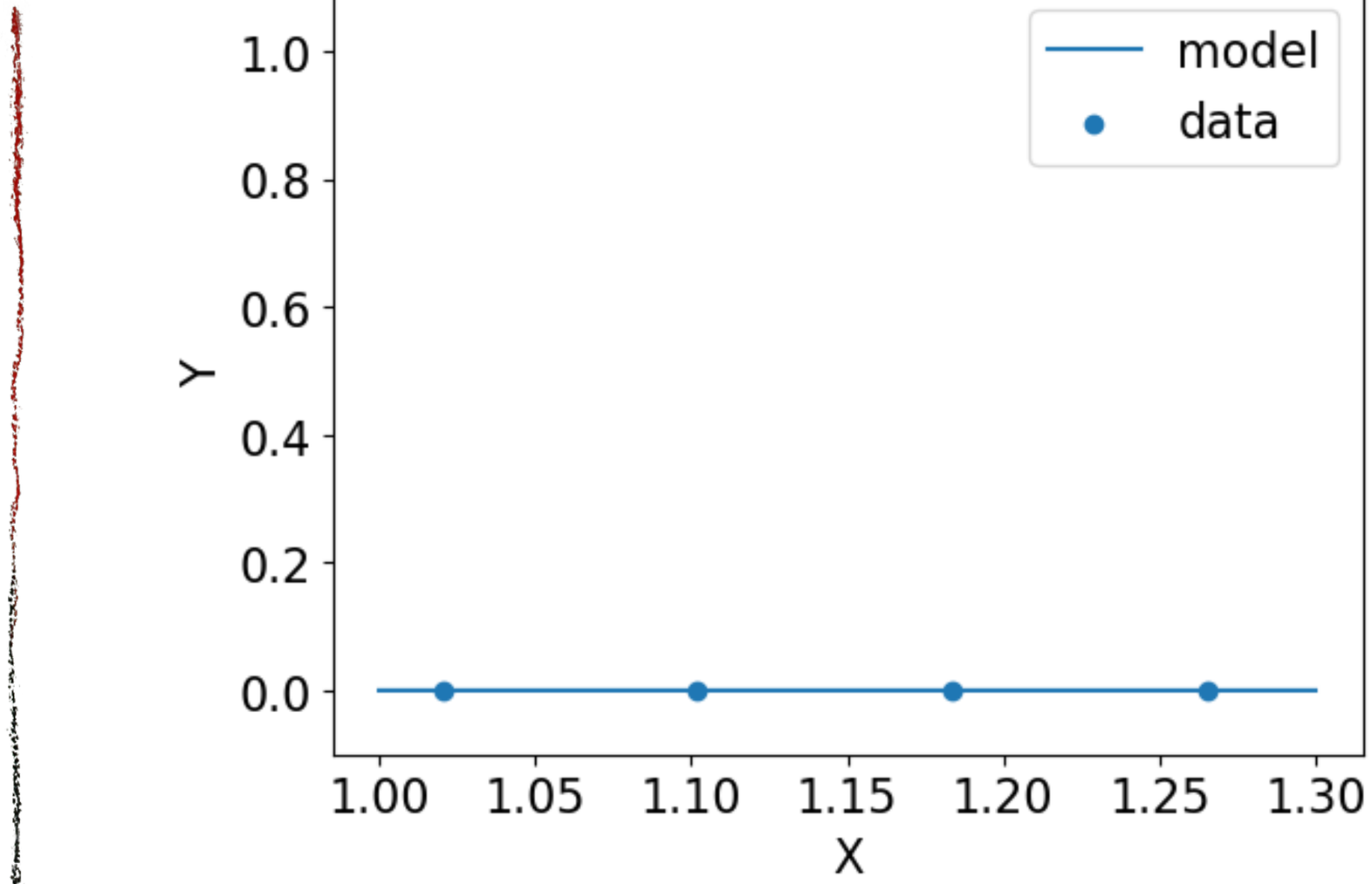
Wider optimizes faster



- Not just better classifier, but fewer epochs to train

not obvious why

Generalizes



- Not just better and faster, but generalizes

not obvious why

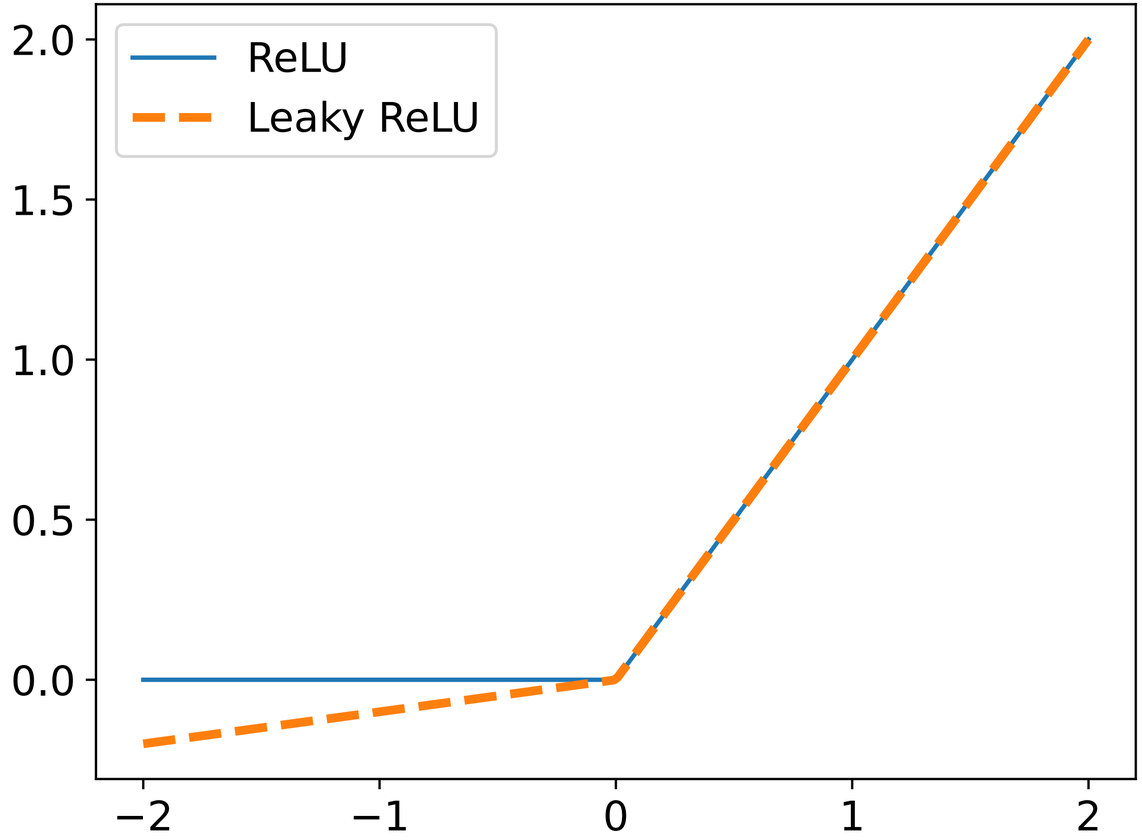
Nonlinearity

- So far we've seen $\sigma(\cdot)$, $[\cdot]_+$
  - variant of σ: softmax, $z \mapsto e^z / \sum e^z$, $\mathbb{R}^d \to \triangle$
  - should really be "soft argmax," but who am I to argue
- Many more choices:
  - leaky ReLU, SILU, GELU, gating, normalization, …

Dead zones

- Gradient of ReLU $[z]_+$ is 0 for $z < 0$
- OK if this is true for some examples
- If it's true for all examples, we're stuck: SGD has no gradient signal to help fix it
  - unit contributes nothing to learned function
  - might get lucky and SGD noise makes it wander back into life
  - else, a waste of GPU time and memory

Leaky ReLU

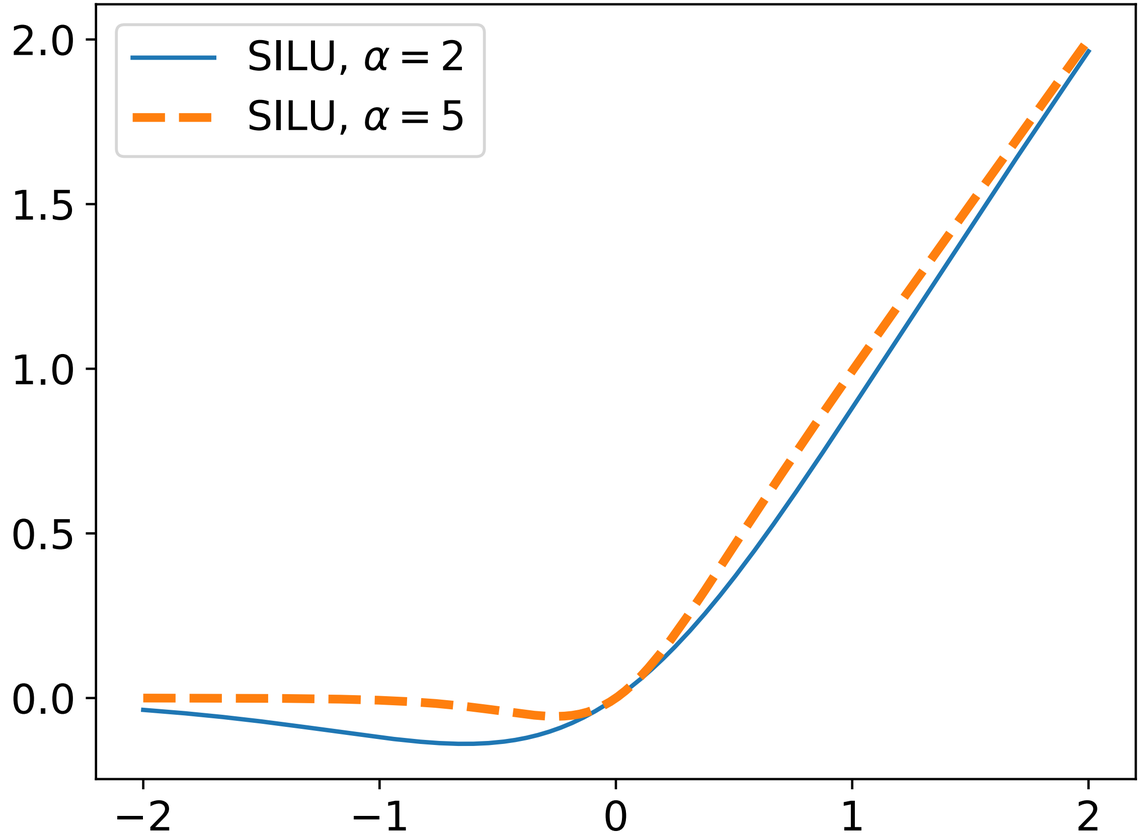


- ReLU can be written $\max(z,0)$
- Leaky ReLU: $\max(\alpha z, z)$ for parameter $\alpha > 0$
  - now we have nonzero gradient even when $z < 0$
- Can fix α or learn it by SGD

Large negative values

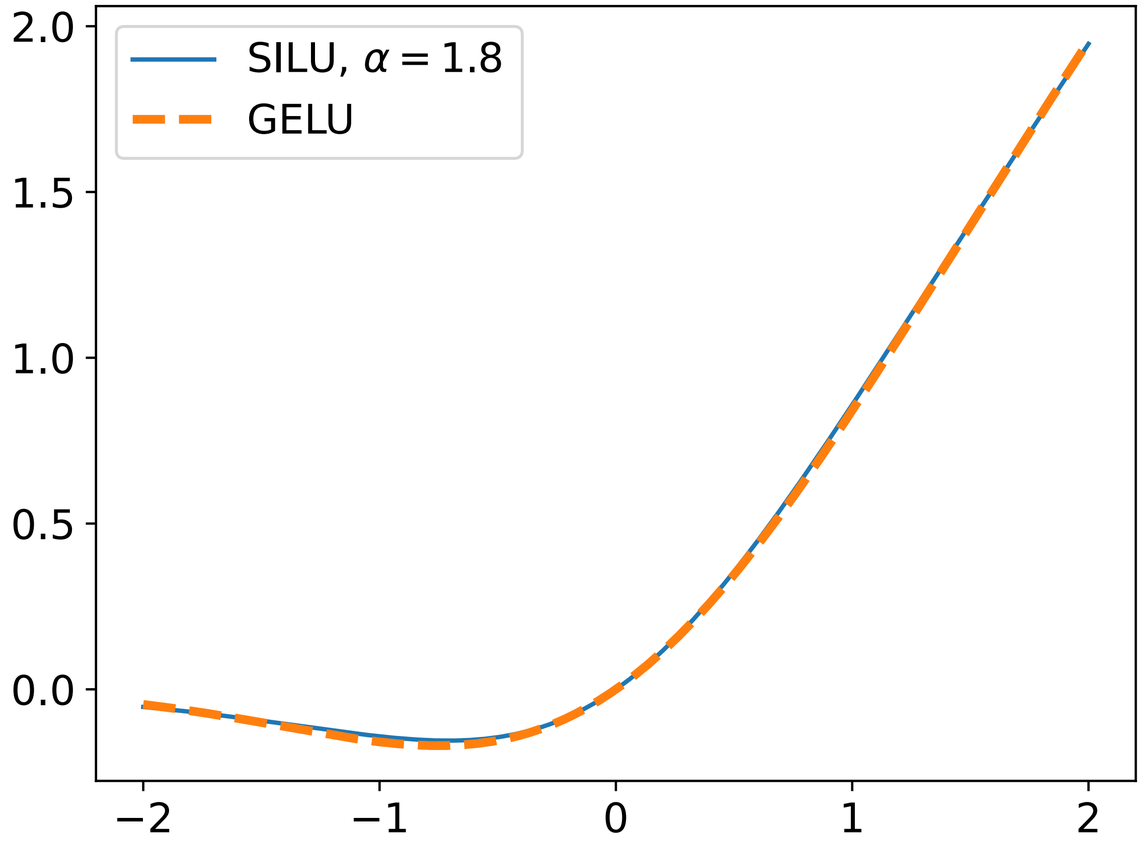
- Part of ReLU's benefit is that it zeros out large negative values of activation z
- Leaky ReLU doesn't do this: if $z \ll 0$, output can still have large absolute value
- Fixes: SILU, GELU
  - give up monotonicity to recover suppression of $z \ll 0$
  - note: can't have both

SILU



- Sigmoid linear unit = $z\,\sigma(\alpha z)$ for $\alpha > 0$
  - also called Swish
  - approaches ReLU for $\alpha \to \infty$

GELU

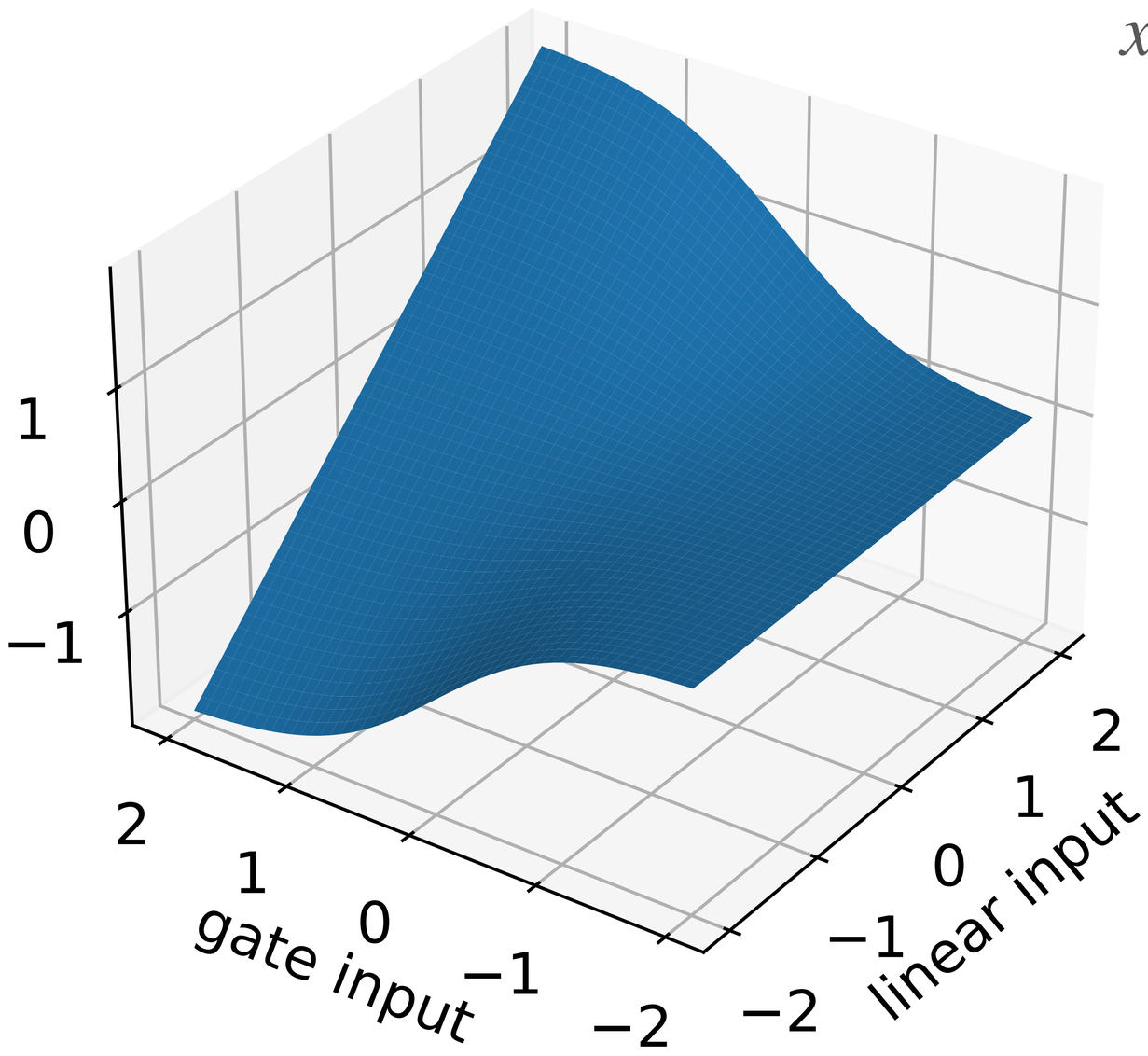


- Can do something like SILU with any CDF
- Currently popular: Gaussian CDF → GELU
  - Gaussian error linear unit, since the Gaussian CDF is related to the *error function* $\text{erf}(z)$

Gating

- Can think of ReLU, SILU, GELU as implementing a logical test (a *gate*) on activation $z = Wx + b$
  - if $z < 0$, suppress output (possibly differentiably)
- What if we want the gate to depend on something other than sign of z?
- Gated linear unit (GLU, aka multiplicative integration):
  - $(Ux + b) \circ \sigma(Wx + c)$
  - two separate linear functions of previous layer, second one gates the first
  - can replace σ with Gaussian CDF or anything else (ReLU, identity, …)
- Twice as many parameters (mostly matters for speed)

GLU



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

RMS norm

- Given $a = Wx \in \mathbb{R}^d$
  - define $\bar{a} = 0$
  - define $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(a_i - \bar{a})^2 + \epsilon = \frac{1}{d}\sum_{i=1}^{d} a_i^2 + \epsilon$
  - define $a' = \frac{a - \bar{a}}{\sigma} = \frac{a}{\sigma}$
- New version of layer, with RMS norm: $f(g \circ a' + b)$
  - only difference is not to subtract mean as first step
  - somewhat faster, similar performance

What to normalize?

- Recall: layer norm says the bag of hidden activations for any *single* example should have mean 0, variance 1
- Alternate goal: *each* hidden activation should have mean 0, variance 1 *across* examples
  - enforcing this property is called *batch norm*
  - can't measure it with just one example
  - makes batch norm more complicated to implement

Batch norm

- To the rescue: ***minibatch*** SGD
- Measure mean and variance in each minibatch
- Given $a_i = Wx_i \in \mathbb{R}^d$ for each example in a minibatch
  - define $\bar{a} = \frac{1}{|B|} \sum_{i \in B} a_i \quad \bar{a} \in \mathbb{R}^d$
  - define $\sigma^2 = \frac{1}{|B|} \sum_{i \in B} (a_i - \bar{a})^2 + \epsilon \quad \sigma \in \mathbb{R}^d$
  - define $a' = \frac{a - \bar{a}}{\sigma}$
- New version of layer, with batch norm: $f(g \circ a' + b)$

contrast w/ layer norm where mean, variance are **scalars**

Batch norm: test time

- We don't necessarily have a minibatch available at test time — so we can't compute $\bar{a}$ and σ!
- So, after we're done with SGD:
  - learn $\bar{a}$ and σ from entire training set
  - these parameters become part of our model

Exploding gradients

- All these normalization operations are differentiable
- When activations are small (as at initialization), normalization scales them up → they are sensitive to parameters W and previous layer activations z
  - causes exploding gradients!
- Must use a mitigation if we have normalization layers
  - most popular: residual connections