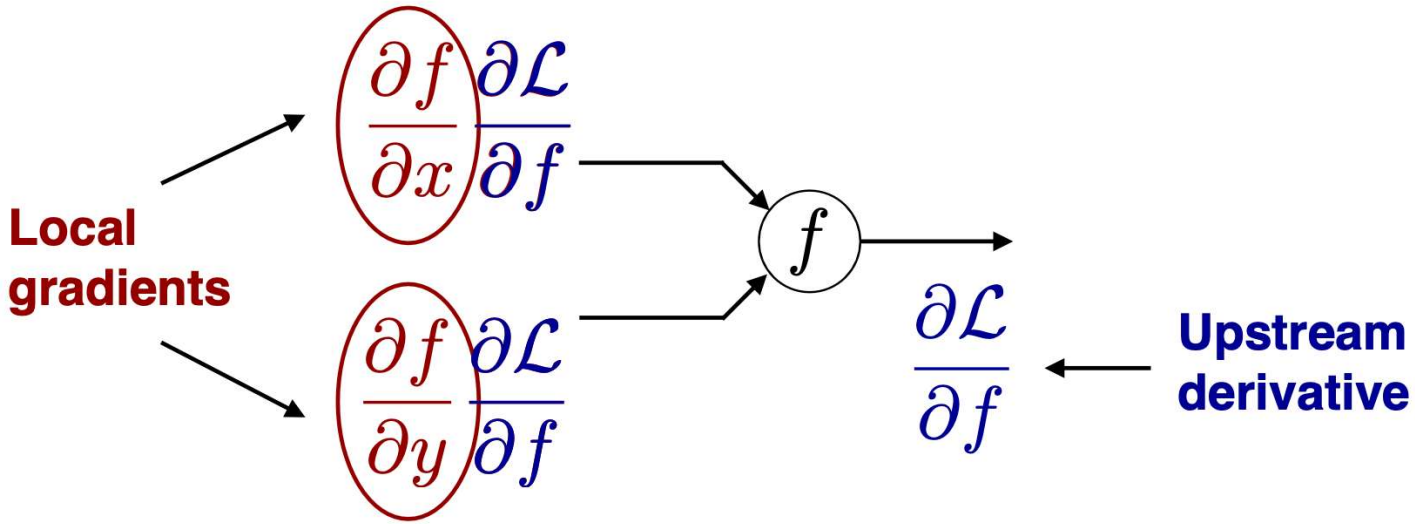
Lecture 7: Backpropagation & Initialization

Announcements:

- HW #3 is due, **Friday, Feb 7**, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

Idea: computational graphs apply to gradients



The basic intuition of backpropagation is that we break up the calculation of the gradient into **small and simple steps**. Each of these nodes in the graph is a straightforward gradient calculation, where we multiply an **input** (the "upstream derivative") with a **local gradient** (an application of the chain rule).

Composing all of these gradients together returns the overall gradient.

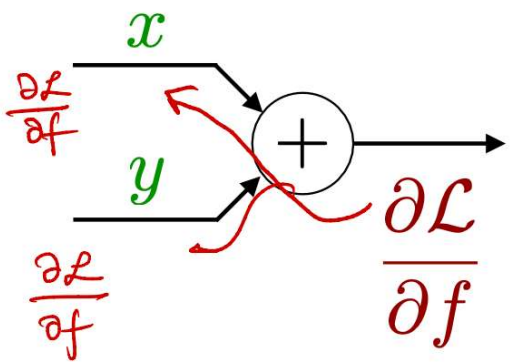
A gate view of gradients

Interpreting backpropagation as gradient “gates”:

$$f = x + y$$

$$\frac{\partial f}{\partial x} = 1$$

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



A gate view of gradients

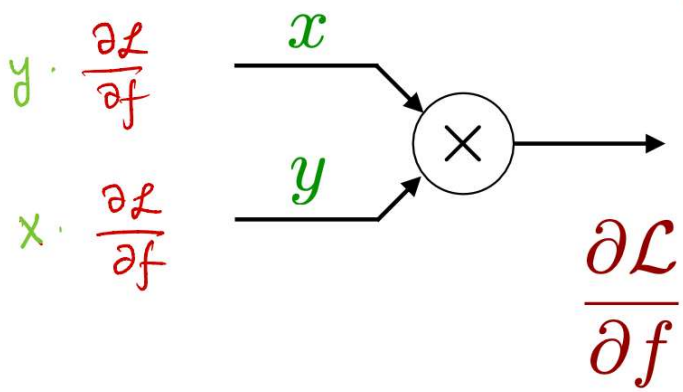
Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient

$f = x \cdot y$

$\frac{\partial f}{\partial x} = y$

$\frac{\partial f}{\partial y} = x$



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

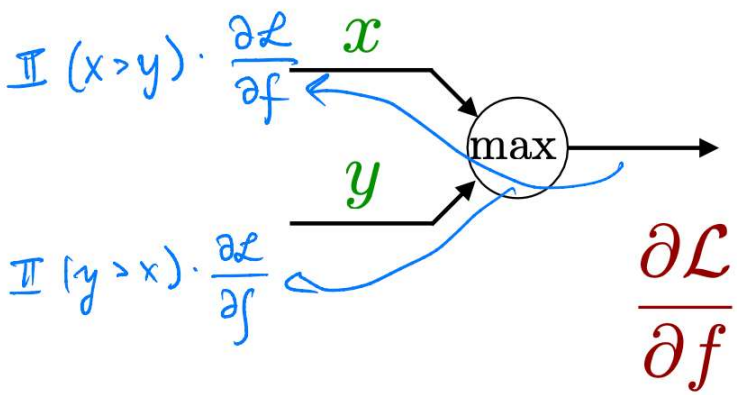
Add gate: distributes the gradient
Mult gate: switches the gradient

$\text{relu}(x) = \max(x, 0)$

$f = \max(x, y)$

$$\frac{\partial f}{\partial x} = \begin{cases} 1, & \text{if } x > y \\ 0, & \text{else} \end{cases}$$

$$= \mathbb{I}\{x > y\}$$



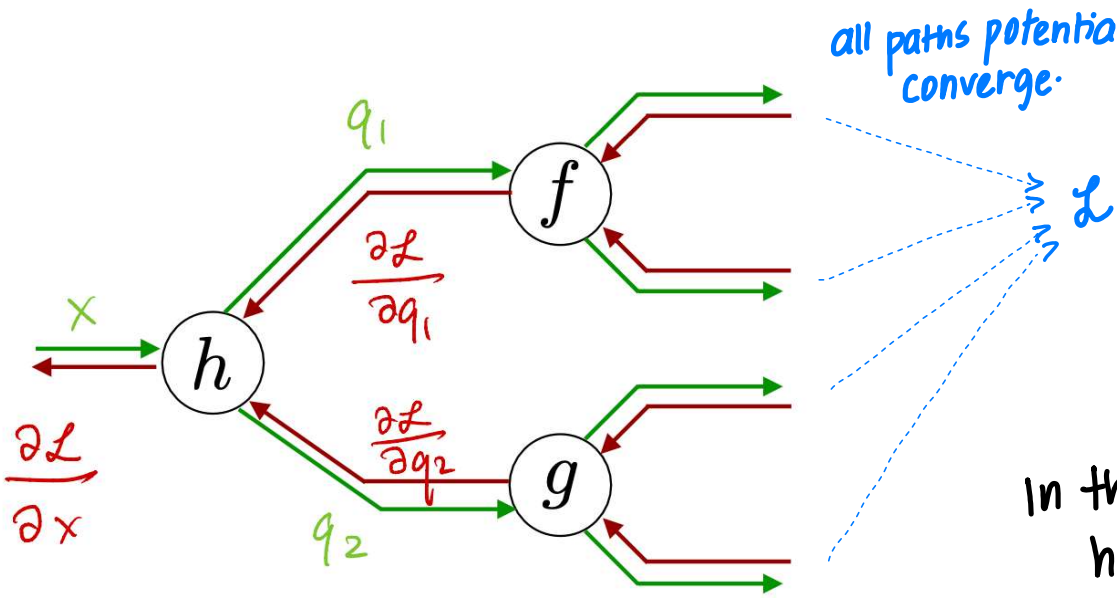
What happens when two gradient paths converge?

$q_1 = h(x)$

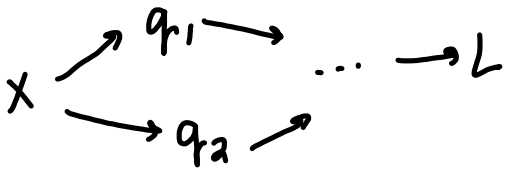
$q_2 = h(x)$

Law of total derivatives:

$$\frac{\partial \mathcal{L}}{\partial x} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial q_i} \cdot \frac{\partial q_i}{\partial x}$$



In this course, $h(x) = x$



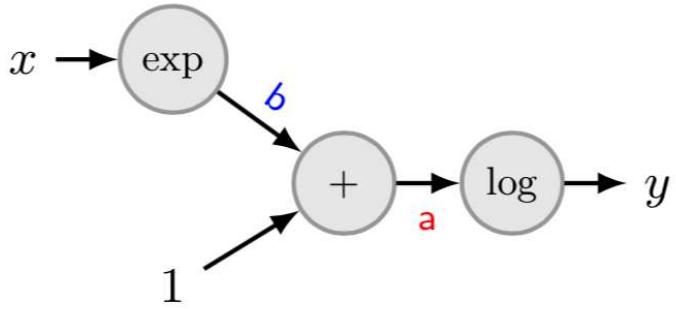
$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial q_1} + \frac{\partial \mathcal{L}}{\partial q_2}$$

Backpropagation example: softplus

- $y = \text{softplus}(x) = \log(1 + \exp(x))$. We can analytically calculate the derivative.

$$\frac{dy}{dx} = \frac{\exp(x)}{1 + \exp(x)}$$

Let's now calculate it via backpropagation.



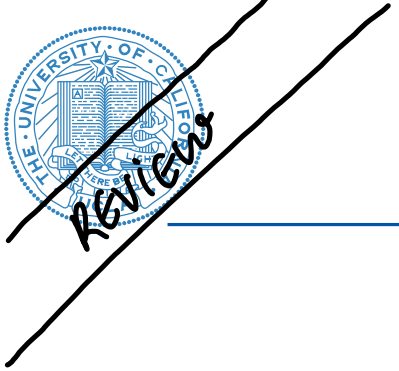
Here, we have that $b = \exp(x)$ and that $a = 1 + \exp(x)$. Applying the chain rule, we have that:

$$\begin{aligned}
\frac{dy}{da} &= \frac{d}{da} \log a = \frac{1}{a} \\
\frac{dy}{db} &= \frac{da}{db}\frac{dy}{da} = \frac{dy}{da} \\
\frac{dy}{dx} &= \frac{db}{dx}\frac{dy}{db} = \exp(x)\frac{1}{a}
\end{aligned}$$

Multivariate backpropagation

To do multivariate backpropagation, we need a multivariate chain rule.

$h_2 = \text{relu}(wh_1 + b)$



Derivative of a scalar w.r.t. a vector

$$\nabla_{\mathbf{x}} y = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$$

In other words, the gradient is:

- A *vector* that is the same size as $\mathbf{x}$, i.e., if $\mathbf{x} \in \mathbb{R}^n$ then $\nabla_{\mathbf{x}} y \in \mathbb{R}^n$.
- Each dimension of $\nabla_{\mathbf{x}} y$ tells us how small changes in $\mathbf{x}$ in that dimension affect y. i.e., changing the ith dimension of $\mathbf{x}$ by a small amount, Δx_i, will change y by

$$\frac{\partial y}{\partial x_i} \Delta x_i$$

We may also denote this as:

$$(\nabla_{\mathbf{x}} y)_i \, \Delta x_i$$

Derivative of a scalar w.r.t. a matrix

The derivative of a scalar, y, with respect to a matrix, $\mathbf{A} \in \mathbb{R}^{m \times n}$, is given by:

$$\nabla_{\mathbf{A}} y = \begin{bmatrix} \frac{\partial y}{\partial a_{11}} & \frac{\partial y}{\partial a_{12}} & \cdots & \frac{\partial y}{\partial a_{1n}} \\ \frac{\partial y}{\partial a_{21}} & \frac{\partial y}{\partial a_{22}} & \cdots & \frac{\partial y}{\partial a_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial a_{m1}} & \frac{\partial y}{\partial a_{m2}} & \cdots & \frac{\partial y}{\partial a_{mn}} \end{bmatrix}$$

Like the gradient, the i, jth element of $\nabla_{\mathbf{A}} y$ tells us how small changes in a_{ij} affect y.

Note:

- If you search for the derivative of a scalar with respect to a matrix, you may find people give a transposed definition to the one above.
- Both are valid, but you must be consistent with your notation and use the correct rules. Our notation is called "denominator layout" notation; the other layout is called "numerator layout" notation.
- In the denominator layout, the dimensions of $\nabla_{\mathbf{A}} y$ and $\mathbf{A}$ are the same. The same holds for the gradient, i.e., the dimensions of $\nabla_{\mathbf{x}} y$ and $\mathbf{x}$ are the same. In the numerator layout notation, the dimensions are transposed.
- More on this later, but in "denominator layout," the chain rule goes right to left as opposed to left to right.

Derivative of a vector w.r.t. a vector

In neural network layers, we will have affine transformations

$y = Wx$

$(n)\quad (n \times m)(m)$

Derivative of a vector w.r.t. a vector

$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$ scalars.

Let $\mathbf{y} \in \mathbb{R}^n$ be a function of $\mathbf{x} \in \mathbb{R}^m$. What dimensionality should the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$ be?

- e.g., to see how $\Delta\mathbf{x}$ modifies y_i, we would calculate:

$$\Delta y_i = \nabla_{\mathbf{x}} y_i \cdot \Delta\mathbf{x}$$

→ want to know how much y_i would change if I wiggle x.

- This suggests that the derivative ought to be an $n \times m$ matrix, denoted $\mathbf{J}$, of the form:

Each row is a gradient that tells me how wiggling all elements of x wiggles an element of y

$$\Delta y = \begin{bmatrix} \Delta y_1 \\ \Delta y_2 \\ \vdots \\ \Delta y_n \end{bmatrix} \quad \mathbf{J} = \begin{bmatrix} (\nabla_{\mathbf{x}} y_1)^T \\ (\nabla_{\mathbf{x}} y_2)^T \\ \vdots \\ (\nabla_{\mathbf{x}} y_n)^T \end{bmatrix} \begin{bmatrix} \Delta x \end{bmatrix}$$

"Jacobian"

$$= \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

$x \in \mathbb{R}^m$

$y \in \mathbb{R}^n$

$J \in \mathbb{R}^{n \times m}$

The matrix would tell us how a small change in $\Delta\mathbf{x}$ results in a small change in $\Delta\mathbf{y}$ according to the formula:

$$\Delta\mathbf{y} \approx \mathbf{J}\Delta\mathbf{x}$$

Derivative of a vector w.r.t. a vector (cont.)

$$\frac{\partial y_1}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} \\ \vdots \\ \frac{\partial y_1}{\partial x_n} \end{bmatrix}$$

The matrix $\mathbf{J}$ is called the Jacobian matrix.

A word on notation:

- In the denominator layout definition, the denominator vector changes in a column.

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} & \frac{\partial y_2}{\partial x} & \cdots & \frac{\partial y_n}{\partial x} \end{bmatrix}$$

$$\nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

"denominator layout" ⟸ dimension of $\mathbf{x}$ goes first when you take grad.

$$\triangleq \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

- Hence the notation we use for the Jacobian would be:

$$\mathbf{J} = (\nabla_{\mathbf{x}}\mathbf{y})^T = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T$$

$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{h1} & w_{h2} & \cdots & w_{hn} \end{bmatrix}$

Derivative of a vector w.r.t. a vector

$y = Wx$
$y \in \mathbb{R}^h$
$x \in \mathbb{R}^n$
$W \in \mathbb{R}^{h \times n}$

Example: derivative of a vector with respect to a vector

The following derivative will appear later on in the class. Let $\mathbf{W} \in \mathbb{R}^{h \times n}$ and $\mathbf{x} \in \mathbb{R}^n$. We would like to calculate the derivative of $f(\mathbf{x}) = \mathbf{W}\mathbf{x}$ with respect to $\mathbf{x}$.

$$\nabla_{\mathbf{x}} \mathbf{W}\mathbf{x} = \nabla_{\mathbf{x}} \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n \\ \vdots \\ w_{h1}x_1 + w_{h2}x_2 + \cdots + w_{hn}x_n \end{bmatrix} \begin{matrix} \rightarrow y_1 \\ \rightarrow y_2 \\ \\ \rightarrow y_h \end{matrix}$$

$$= \begin{bmatrix} w_{11} & w_{21} & \cdots & w_{h1} \\ w_{12} & w_{22} & \cdots & w_{h2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \cdots & w_{hn} \end{bmatrix}$$

$$= \mathbf{W}^T$$

$\frac{\partial y_1}{\partial x_1}$ $\frac{\partial y_1}{\partial x_2}$ $\frac{\partial y_1}{\partial x_n}$ $\frac{\partial y_2}{\partial x_1}$ $\frac{\partial y_2}{\partial x_2}$ $\frac{\partial y_2}{\partial x_n}$

$y = Wx$
$\nabla_x y = W^T$

$x \in \mathbb{R}^n$
$y \in \mathbb{R}^h$
$W \in \mathbb{R}^{n \times h}$

Hessian: the generalization of the second derivative

$f(x) \in \mathbb{R}$
$x \in \mathbb{R}^m$

Hessian

The Hessian matrix of a function $f(\mathbf{x})$ is a square matrix of second-order partial derivatives of $f(\mathbf{x})$. It is composed of elements:

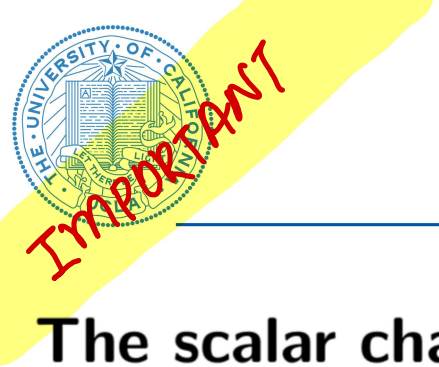
diagonal elements are second derivatives

$$\mathbf{H} = \begin{bmatrix} \frac{\partial f}{\partial x_1^2} & \frac{\partial f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_m} \\ \frac{\partial f}{\partial x_2 \partial x_1} & \frac{\partial f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_2 \partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_m \partial x_1} & \frac{\partial f}{\partial x_m \partial x_2} & \cdots & \frac{\partial f}{\partial x_m^2} \end{bmatrix}$$

$\nabla_x f(x) \in \mathbb{R}^m$

$\nabla_x \left(\nabla_x f(x)\right) \in \mathbb{R}^{m \times m}$

We can denote this matrix as $\nabla_{\mathbf{x}} \left(\nabla_{\mathbf{x}} f(\mathbf{x})\right)$. We often denote this simply as $\nabla_{\mathbf{x}}^2 f(\mathbf{x})$.



The scalar chain rule

The scalar chain rule states that if $y = f(x)$ and $z = g(y)$, then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Intuitively: the chain rule tells us that a small change in x will cause a small change in y that will in turn cause a small change in z, i.e., for apropriately small Δx,

$$\begin{aligned} \Delta y &\approx \frac{dy}{dx}\Delta x \\ \Delta z &\approx \frac{dz}{dy}\Delta y \\ &= \frac{dz}{dy}\frac{dy}{dx}\Delta x \end{aligned}$$

Vector chain rule

Chain rule for vector valued functions

In the "denominator" layout, the chain rule runs from right to left. We won't derive this, but we will check the dimensionality and intuition.

Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, and $\mathbf{z} \in \mathbb{R}^p$. Further, let $\mathbf{y} = f(\mathbf{x})$ for $f : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{z} = g(\mathbf{y})$ for $g : \mathbb{R}^n \to \mathbb{R}^p$. Then,

$$\nabla_{\mathbf{x}} \mathbf{z} \quad = \quad \nabla_{\mathbf{x}} \mathbf{y} \, \nabla_{\mathbf{y}} \mathbf{z}$$

Equivalently:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \quad = \quad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}}$$

- Note that $\nabla_{\mathbf{x}} \mathbf{z}$ should have dimensionality $\mathbb{R}^{m \times p}$.
- As $\nabla_{\mathbf{x}} \mathbf{y} \in \mathbb{R}^{m \times n}$ and $\nabla_{\mathbf{y}} \mathbf{z} \in \mathbb{R}^{n \times p}$, the operations are dimension consistent.

Vector chain rule

setup: $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $z \in \mathbb{R}^p$; $y = f(x)$, $z = g(y)$

1. $\Delta x \longrightarrow \Delta z$ (How does wiggling x change z?)
2. $\Delta x \longrightarrow \Delta y \longrightarrow \Delta z$

1. $\Delta x \longrightarrow \Delta z$: $\Delta z \approx \left(\frac{\partial z}{\partial x}\right)^T \Delta x$ → Jacobian.

$(p) \quad (p \times m)(m)$

2. $\Delta x \longrightarrow \Delta y$: $\Delta y \approx \left(\frac{\partial y}{\partial x}\right)^T \Delta x$

$\Delta y \longrightarrow \Delta z$: $\Delta z \approx \left(\frac{\partial z}{\partial y}\right)^T \Delta y \approx \left(\frac{\partial z}{\partial y}\right)^T \left(\frac{\partial y}{\partial x}\right)^T \Delta x$

$$\left(\frac{\partial z}{\partial x}\right)^T = \left(\frac{\partial z}{\partial y}\right)^T \left(\frac{\partial y}{\partial x}\right)^T$$

$$\Longrightarrow \boxed{\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \cdot \frac{\partial z}{\partial y}}$$

→ Runs from right to left.

$(m \times p) \quad (m \times n) \quad (n \times p)$

Vector chain rule

Chain rule for vector valued functions (cont.)

- Intuitively, a small change $\Delta\mathbf{x}$ affects $\Delta\mathbf{z}$ through the Jacobian $(\nabla_{\mathbf{x}}\mathbf{z})^T$

$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{x}}\mathbf{z})^T \Delta\mathbf{x} \quad (1)$$

- The chain rule is intuitive, since:

$$\Delta\mathbf{y} \approx (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x}$$
$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{y}}\mathbf{z})^T \Delta\mathbf{y}$$

Composing these, we have that:

$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{y}}\mathbf{z})^T (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x} \quad (2)$$

- Combining equations (1) and (2), we arrive at:

$$(\nabla_{\mathbf{x}}\mathbf{z})^T = (\nabla_{\mathbf{y}}\mathbf{z})^T (\nabla_{\mathbf{x}}\mathbf{y})^T$$

which, after transposing both sides, reduces to the (right to left) chain rule:

$$\nabla_{\mathbf{x}}\mathbf{z} = \nabla_{\mathbf{x}}\mathbf{y}\, \nabla_{\mathbf{y}}\mathbf{z}$$

$y \in \mathbb{R}^m$

$x \in \mathbb{R}^n$

$W \in \mathbb{R}^{m \times n}$

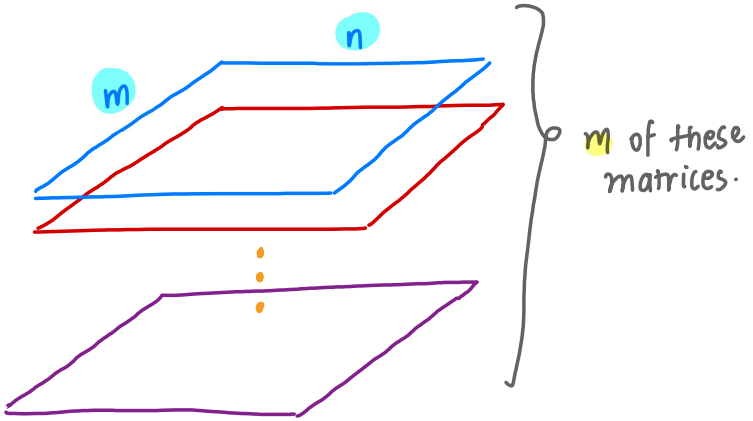
Tensor derivatives

$y = Wx$

$\frac{\partial y}{\partial W}$ will give a 3D tensor $\Longrightarrow$ $\frac{\partial y}{\partial W} \in \mathbb{R}^{m \times n \times m}$

$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$ $\frac{\partial y_1}{\partial W} \in \mathbb{R}^{m \times n}$ [derivative of a scalar w.r.t. matrix W is a matrix]

$\frac{\partial y_2}{\partial W} \in \mathbb{R}^{m \times n}$



Derivatives of tensors

Occasionally in this class, we may need to take a derivative that is more than 2-dimensional. For example, we may want to take the derivative of a vector with respect to a matrix. This would be a 3-dimensional tensor. The definition for this would be as you expect. In particular, if $\mathbf{z} \in \mathbb{R}^p$ and $\mathbf{W} \in \mathbb{R}^{m \times n}$, then $\nabla_{\mathbf{W}} \mathbf{z}$ is a three-dimensional tensor with shape $\mathbb{R}^{m \times n \times p}$. Each $m \times n$ slice (of which there are p) is the matrix derivative $\nabla_{\mathbf{W}} z_i$.

Note, these are sometimes a headache to work with. We typically can find a shortcut to perform operations without having to compute and store these high-dimensional tensor derivatives. The next slides show an example.

Tensor derivatives

Multivariate chain rule and tensor derivative example

Consider the squared loss function:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{W}\mathbf{x}^{(i)} \right\|^2$$

Here, $\mathbf{y}^{(i)} \in \mathbb{R}^m$ and $\mathbf{x}^{(i)} \in \mathbb{R}^n$ so that $\mathbf{W} \in \mathbb{R}^{m \times n}$. We wish to find $\mathbf{W}$ that minimizes the mean-square error in linearly predicting $\mathbf{y}^{(i)}$ from $\mathbf{x}^{(i)}$.

$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} (y^{(i)} - Wx^{(i)})^T (y^{(i)} - Wx^{(i)})$

$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} z^{(i)T} z^{(i)}$

define $z^{(i)} = y^{(i)} - Wx^{(i)}$

define $\varepsilon^{(i)} = z^{(i)T} z^{(i)}$ $\Longrightarrow$ $\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} \varepsilon^{(i)}$

omit superscript (i) — For simplicity

$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial z}{\partial W} \cdot \frac{\partial \mathcal{L}}{\partial z}$

$(m \times n)$ $(m \times n \times m)$ $(m \times 1)$

$(m \times n \times 1)$

dimensionality check

Multivariate chain rule example

We consider one example, $\varepsilon^{(i)} = \frac{1}{2}\left\|\mathbf{y}^{(i)} - \mathbf{W}\mathbf{x}^{(i)}\right\|^2$.

We wish to calculate $\nabla_{\mathbf{W}}\varepsilon^{(i)}$.

To do so, we define $\mathbf{z}^{(i)} = \mathbf{y}^{(i)} - \mathbf{W}\mathbf{x}^{(i)}$. Then, $\varepsilon^{(i)} = \frac{1}{2}(\mathbf{z}^{(i)})^T\mathbf{z}^{(i)}$. For the rest of the example, we're going to drop the superscripts (i) and assume we're working with the ith example (to help notation). Then, we need to calculate is:

$$\nabla_{\mathbf{W}}\varepsilon = \nabla_{\mathbf{W}}\mathbf{z}\nabla_{\mathbf{z}}\varepsilon$$

Multivariate chain rule example

Now, we need to calculate $\nabla_{\mathbf{W}}\mathbf{z}$. This is a three dimensional tensor with dimensionality $m \times n \times m$.

This makes sense dimensionally, because when we multiply a $(m \times n \times m)$ tensor by a $(m \times 1)$ vector, we get out a $(m \times n \times 1)$ tensor, which is equivalently an $(m \times n)$ matrix. Because $\nabla_{\mathbf{W}}\varepsilon$ is an $(m \times n)$ matrix, this all works out.

Multivariate chain rule example

Setup: $z \in \mathbb{R}^m$, $W \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$

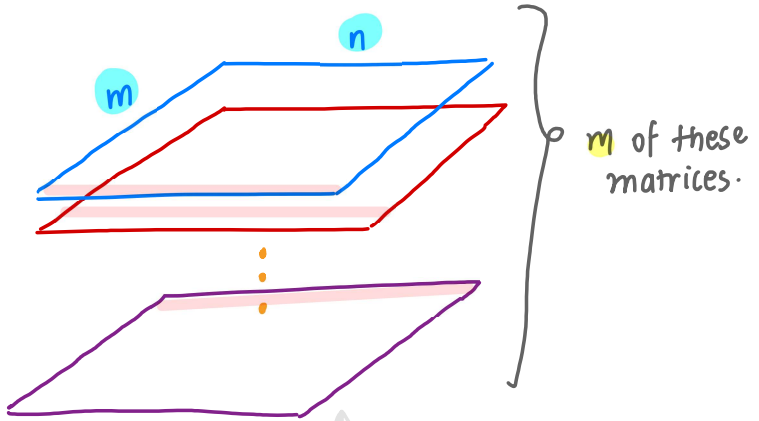
$z = y - Wx$

$z = y_k - \sum_{j=1}^{n} w_{kj} x_j$

$\frac{\partial z}{\partial W} \in \mathbb{R}^{m \times n \times m}$

$\frac{\partial z_k}{\partial W} \in \mathbb{R}^{m \times n}$ (I will have m of these matrices $k = 1, \ldots, m$)

differentiate z_k w.r.t. every element of W.



$$\frac{\partial z_k}{\partial W} = \frac{\partial}{\partial W}\left[y_k - \sum_j W_{kj} x_j\right]$$

$$\frac{\partial z_k}{\partial w_{ip}} = 0 - \frac{\partial}{\partial w_{ip}}\left[\sum_j w_{kj} x_j\right] = -\frac{\partial}{\partial w_{ip}}\left[w_{k1}x_1 + w_{k2}x_2 + \cdots + w_{kn}x_n\right]$$

each element of W?

y_k has no matrix W

(1) When $i \neq k \Rightarrow \frac{\partial z_k}{\partial w_{ip}} = 0$

$i = 2, i = 3 \Rightarrow \frac{\partial}{\partial w_{2p}}\left[w_{31}x_1 + \cdots + w_{3n}x_n\right]$

(2) When $i = k \Rightarrow \frac{\partial z_k}{\partial w_{ip}} = -x_p$

$\frac{\partial z_k}{\partial W}$ is an $m \times n$ matrix

$$\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ -x_1 & -x_2 & -x_3 \cdots -x_n \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

k^{th} row

Multivariate chain rule and tensor derivative example (cont.)

$\nabla_{\mathbf{W}}\mathbf{z}$ is an $(m \times n \times m)$ tensor.

- Letting z_k denote the kth element of $\mathbf{z}$, we see that each $\frac{\partial z_k}{\partial \mathbf{W}}$ is an $m \times n$ matrix, and that there are m of these for each element z_k from $k = 1, \ldots, m$.
- The *matrix*, $\frac{\partial z_k}{\partial \mathbf{W}}$, can be calculated as follows:

$$\frac{\partial z_k}{\partial \mathbf{W}} = \frac{\partial}{\partial \mathbf{W}} \sum_{j=1}^{n} -w_{kj} x_j$$

and thus, the (k, j)th element of this matrix is:

$$\left(\frac{\partial z_k}{\partial \mathbf{W}} \right)_{k,j} = \frac{\partial z_k}{\partial w_{kj}} = -x_j$$

It is worth noting that

$$\left(\frac{\partial z_k}{\partial \mathbf{W}} \right)_{i,j} = \frac{\partial z_k}{\partial w_{ij}} = 0 \qquad \text{for } k \neq i$$

Multivariate chain rule example

Multivariate chain rule and tensor derivative example (cont.)

Hence, $\frac{\partial z_k}{\partial \mathbf{W}}$ is a matrix where the kth row is $\mathbf{x}^T$ and all other rows are the zeros (we denote the zero vector by $\mathbf{0}$). i.e.,

$$\frac{\partial z_1}{\partial \mathbf{W}} = \begin{bmatrix} -\mathbf{x}^T \\ \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} \qquad \frac{\partial z_2}{\partial \mathbf{W}} = \begin{bmatrix} \mathbf{0}^T \\ -\mathbf{x}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} \qquad \text{etc...}$$

Now applying the chain rule,

$$\frac{\partial \varepsilon}{\partial \mathbf{W}} = \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \frac{\partial \varepsilon}{\partial \mathbf{z}}$$

is a tensor product between an $(m \times n \times m)$ tensor and an $(m \times 1)$ vector, whose resulting dimensionality is $(m \times n \times 1)$ or equivalently, an $(m \times n)$ matrix.

$\mathcal{E} = z^T z$

$z = y - Wx$

Multivariate chain rule example

$$\frac{\partial \mathcal{E}}{\partial W} = \frac{\partial z}{\partial W} \cdot \frac{\partial \mathcal{E}}{\partial z} = \sum_{k=1}^{m} \frac{\partial z_k}{\partial W} \cdot \frac{\partial \mathcal{E}}{\partial z_k}$$

$$\frac{\partial \mathcal{E}}{\partial z} = \begin{bmatrix} \frac{\partial \mathcal{E}}{\partial z_1} \\ \frac{\partial \mathcal{E}}{\partial z_2} \\ \vdots \\ \frac{\partial \mathcal{E}}{\partial z_m} \end{bmatrix}$$

$(m \times n) \quad (m \times n \times m) \quad (m \times 1)$

$$= \frac{\partial \mathcal{E}}{\partial z_1} \cdot \begin{bmatrix} -x^T - \\ 0 \\ 0 \\ \vdots \end{bmatrix} + \frac{\partial \mathcal{E}}{\partial z_2} \begin{bmatrix} 0 \\ -x^T - \\ 0 \\ \vdots \end{bmatrix} + \cdots$$

$$= -\begin{bmatrix} \frac{\partial \mathcal{E}}{\partial z_1} x^T \\ \frac{\partial \mathcal{E}}{\partial z_2} x^T \\ \vdots \\ \frac{\partial \mathcal{E}}{\partial z_m} x^T \end{bmatrix} = -\frac{\partial \mathcal{E}}{\partial z} x^T$$

won't actually see (-) in neural network layers.

$(m \times 1)(1 \times n)$

$\Downarrow$

$(m \times n)$

→ Chain rule from above simplifies to inner product.

$\Downarrow$

So we never have to store sparse matrices.

In denominator layout, chain rule goes from right to left

Multivariate chain rule and tensor derivative example (cont.)

We carry out this tensor-vector multiply in the standard way.

$$
\begin{aligned}
\frac{\partial \varepsilon}{\partial \mathbf{W}} &= \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \frac{\partial \varepsilon}{\partial \mathbf{z}} \\
&= \sum_{i=1}^{m} \frac{\partial z_i}{\partial \mathbf{W}} \left(\frac{\partial \varepsilon}{\partial \mathbf{z}} \right)_i \\
&= \frac{\partial \varepsilon}{\partial z_1} \begin{bmatrix} -\mathbf{x}^T \\ \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} + \frac{\partial \varepsilon}{\partial z_2} \begin{bmatrix} \mathbf{0}^T \\ -\mathbf{x}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} + \cdots \\
&\quad + \frac{\partial \varepsilon}{\partial z_m} \begin{bmatrix} \mathbf{0}^T \\ \mathbf{0}^T \\ \vdots \\ -\mathbf{x}^T \end{bmatrix}
\end{aligned}
$$

Multivariate chain rule example

Multivariate chain rule and tensor derivative example (cont.)

Continuing the work from the previous page...

$$\frac{\partial \varepsilon}{\partial \mathbf{W}} = -\begin{bmatrix} \frac{\partial \varepsilon}{\partial z_1}\mathbf{x}^T \\ \frac{\partial \varepsilon}{\partial z_2}\mathbf{x}^T \\ \vdots \\ \frac{\partial \varepsilon}{\partial z_m}\mathbf{x}^T \end{bmatrix}$$

$$= -\frac{\partial \varepsilon}{\partial \mathbf{z}}\mathbf{x}^T$$

Multivariate chain rule example

Hence, with a final application of the chain rule, we get that

$$\begin{aligned} \nabla_{\mathbf{W}} \varepsilon &= -\mathbf{z}\mathbf{x}^T \\ &= -(\mathbf{y} - \mathbf{W}\mathbf{x})\mathbf{x}^T \end{aligned}$$

Setting this equal to zero, we find that for one example,

$$\mathbf{W} = \mathbf{y}\mathbf{x}^T(\mathbf{x}\mathbf{x}^T)^{-1}$$

Summing across all examples, this produces least-squares.

ℓ = loss

$\frac{\partial \ell}{\partial z} \in \mathbb{R}^m$, $\frac{\partial \ell}{\partial w} \in \mathbb{R}^{m \times n}$

Multivariate chain rule example

$\nabla_x W_x = W^T$

$\nabla_W W_x$ "should look like x^T"

A few notes on tensor derivatives

- In general, the simpler rule can be inferred via pattern intuition / looking at the dimensionality of the matrices, and these tensor derivatives need not be explicitly derived.
- Indeed, actually calculating these tensor derivatives, storing them, and then doing e.g., a tensor-vector multiply, is usually not a good idea for both memory and computation. In this example, storing all these zeros and performing the multiplications is unnecessary.
- If we know the end result is simply an outer product of two vectors, we need not even calculate an additional derivative in this step of backpropagation, or store an extra value (assuming the inputs were previously cached).

$$\frac{\partial \ell}{\partial W} = \frac{\partial z}{\partial W} \cdot \frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial z} x^T$$

(mxn) (mx1) (mx1)(1xn)

→ SHORTCUT THAT WORKS.

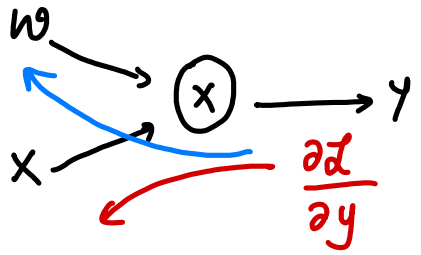
NOT ACTUALLY 1xn or x^T.

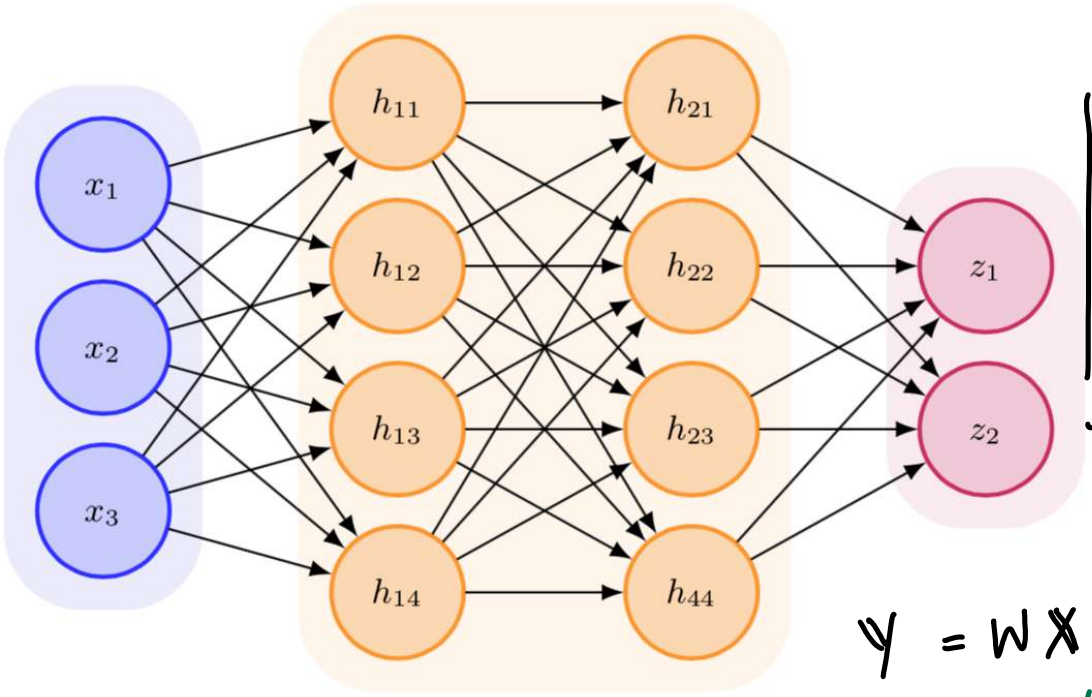
It's a 3D tensor (mxnxm) and when you multiply it, it simplifies to multiplying on the right by x^T.

Back to backpropagation

$y = wx$

$h_2 = f(wh_1)$





$$\frac{\partial L}{\partial x} = W^T \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} x^T$$

$Y = WX$

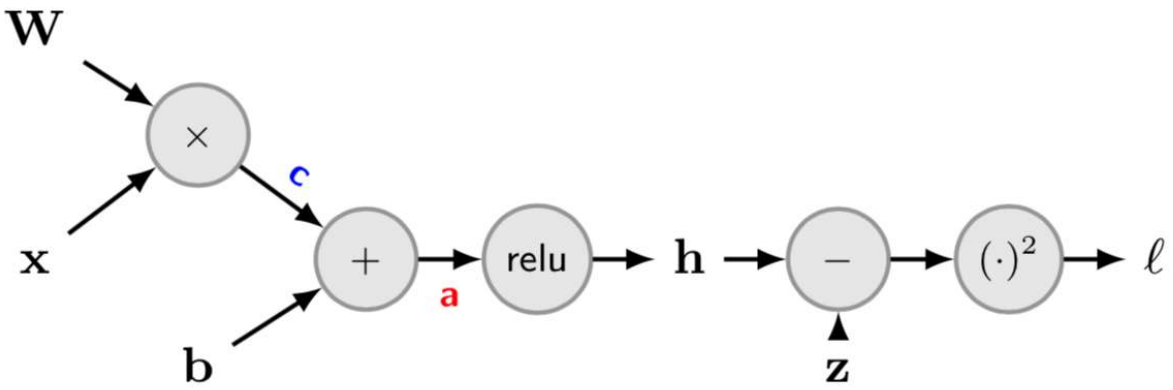
$(n \times p) \quad (n \times m)(m \times p)$

$$\frac{\partial L}{\partial X} = W^T \frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} X^T$$

Backpropagation: neural network layer

Here, $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output $\ell = (\mathbf{h} - \mathbf{z})^2$ where $\mathbf{z}$ is some target value.

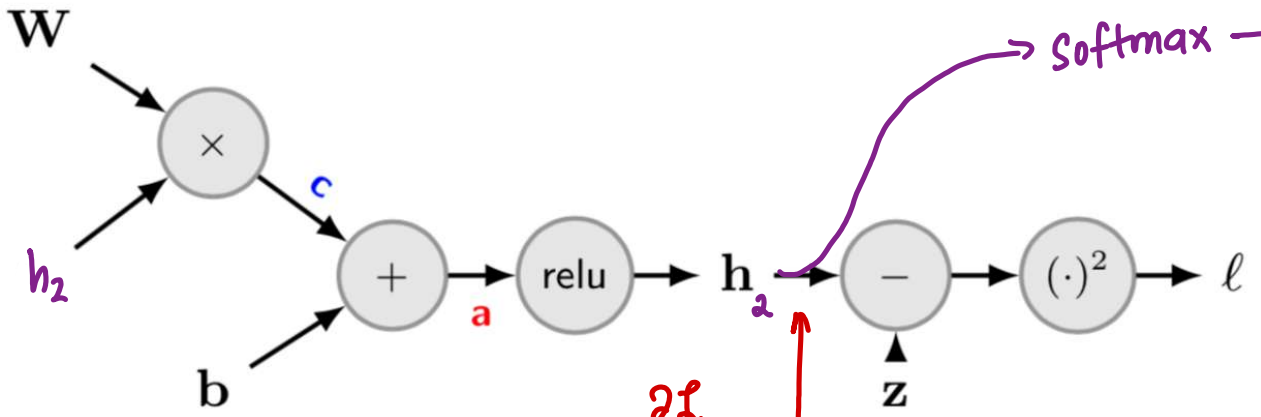


A few things to note:

- Note that $\nabla_{\mathbf{h}} \ell = 2(\mathbf{h} - \mathbf{z})$. We will start backpropagation at $\mathbf{h}$ rather than at ℓ.
- In the following backpropagation, we'll have need for elementwise multiplication. This is formally called a Hadamard product, and we will denote it via $\odot$. Concretely, the ith entry of $\mathbf{x} \odot \mathbf{y}$ is given by $x_i y_i$.

Backpropagation for a neural network layer

$h_2 = \text{relu}(wh_1 + b)$



$$\text{relu}(a) = \begin{bmatrix} \text{relu}(a_1) \\ \text{relu}(a_2) \\ \vdots \end{bmatrix}$$

vector

$$\frac{\partial \mathcal{L}}{\partial a} = \mathbb{1}\{a > 0\} \odot \frac{\partial \mathcal{L}}{\partial h_2}.$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial c} = \frac{\partial \mathcal{L}}{\partial a}$$

$$\frac{\partial \mathcal{L}}{\partial h_1} = w^T \frac{\partial \mathcal{L}}{\partial c}$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial c} h_1^T$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$x \odot y = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \end{bmatrix}.$$

Backpropagation: neural network layer (cont.)

Applying the chain rule, we have:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{a}} &= \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}} \\
\frac{\partial \ell}{\partial \mathbf{c}} &= \frac{\partial \ell}{\partial \mathbf{a}} \\
\frac{\partial \ell}{\partial \mathbf{x}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}}
\end{aligned}$$

A few notes:

- For $\frac{\partial \mathbf{c}}{\partial \mathbf{x}}$, see example in the Tools notes.
- Why was the chain rule written right to left instead of left to right? This turns out to be a result of our convention of how we defined derivatives (using the "denominator layout notation") in the linear algebra notes.
- Though maybe not the most satisfying answer, you can always check the order of operations is correct by considering non-square matrices, where the dimensionality must be correct.

Backpropagation: neural network layer (cont.)

What about $\frac{\partial \ell}{\partial \mathbf{W}}$? In doing backpropagation, we have to calculate $\frac{\partial \mathbf{c}}{\partial \mathbf{W}}$ which is a 3-dimensional tensor.

However, we also intuit the following (informally):

- $\frac{\partial \ell}{\partial \mathbf{W}}$ is a matrix that is $h \times m$.
- The derivative of $\mathbf{W}\mathbf{x}$ with respect to $\mathbf{W}$ "ought to look like" $\mathbf{x}$.
- Since $\frac{\partial \ell}{\partial \mathbf{c}} \in \mathbb{R}^h$, and $\mathbf{x} \in \mathbb{R}^m$ then, intuitively,

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T$$

This intuition turns out to be correct, and it is common to use this intuition. However, the first time around we should do this rigorously and then see the general pattern of why this intuition works.

Now we have the gradients, we can do gradient descent

With the gradients of the parameters, we can go ahead and now apply our learning algorithm, gradient descent.

However, we'll soon find that if we do this naively, performance won't be great **for neural networks** (you'll see this on HW #3).

There are many other important considerations we now need to consider before we can train these networks adequately.

Regularizations and training neural networks

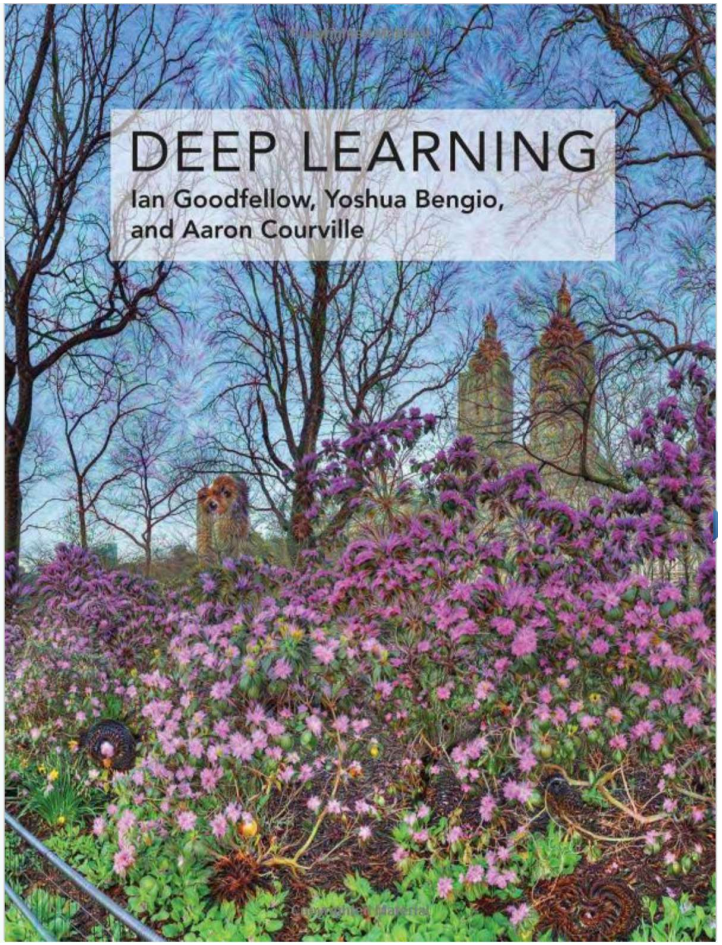
In this lecture, we'll talk about specific techniques that aid in training neural networks. This lecture will focus on a few topics that are relevant for training neural networks well. The next lecture will then focus on optimization.

- Weight initialization in neural networks
- Batch normalization
- Regularizations
  - L1 + L2 normalization
  - Dataset augmentation
  - Model ensembles / Bagging
  - Dropout

Regularizations and training neural networks

Reading:

Deep Learning, 7 (intro), 7.1, 7.2, 7.4, 7.5, 7.7, 7.9, 7.11, 7.12 (skim), 8.7.1



Lecture 8: Regularization and optimization

Announcements:

- HW #3 is due **Friday, Feb 7**, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

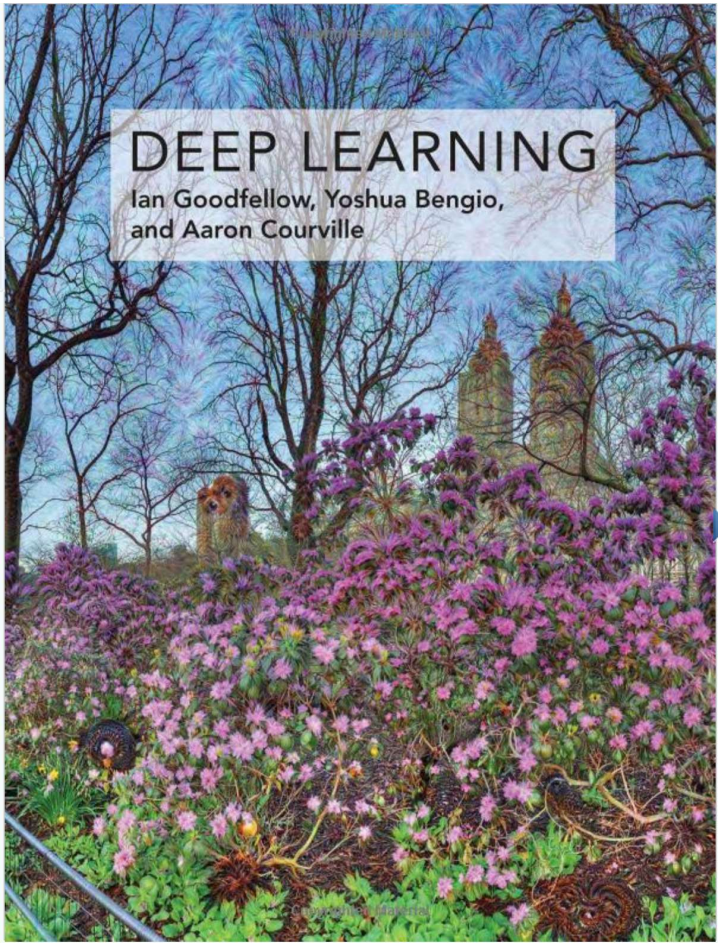
- The midterm is in just over 2 weeks on February 19, 2025.
  - You are allowed 4 cheat sheets (each an 8.5 x 11 inch paper). You can fill out both sides (8 sides total). You can put whatever you want on these cheat sheets.
  - It will cover material up to and including lecture on February 12, 2025.
  - Past exams are uploaded to Bruin Learn (under “Modules” —> “past exams”).
  - You may bring a calculator to the exam.
  - You **must do the exam in pen.**

  - MSOL students will take the exam remotely on Saturday, February 22, from 2-3:50pm. We will send out details to the MSOL students closer to the date.

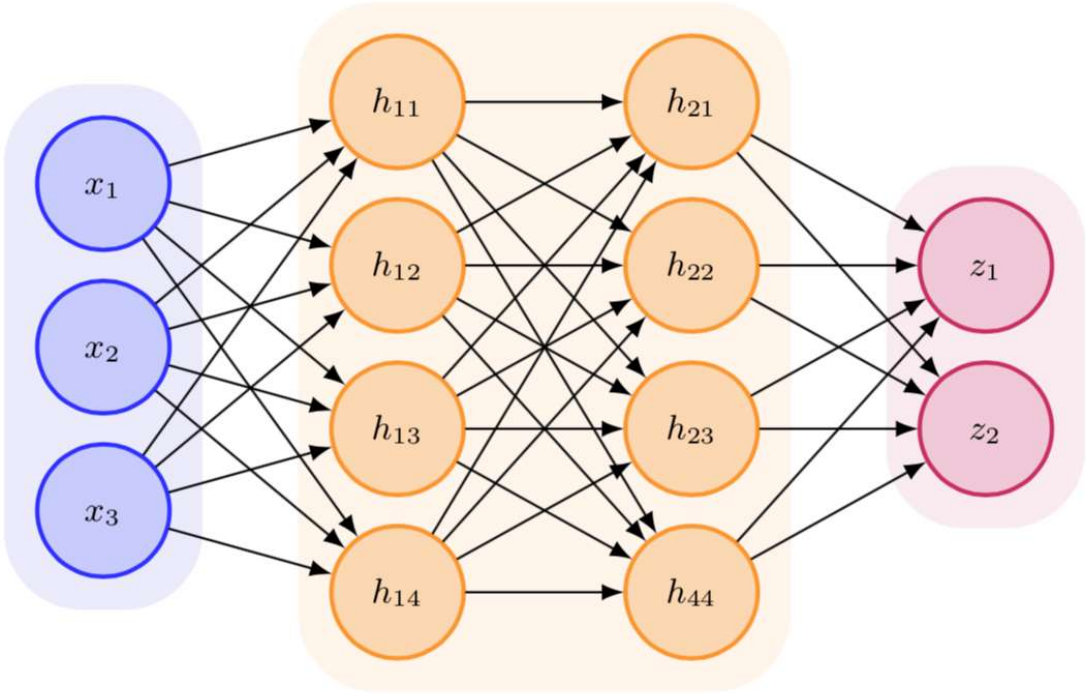
Regularizations and training neural networks

Reading:

Deep Learning, 7 (intro), 7.1, 7.2, 7.4, 7.5, 7.7, 7.9, 7.11, 7.12 (skim), 8.7.1



Initializations



Initializations matter.

Small random weight initialization

How about small random weight initializations?

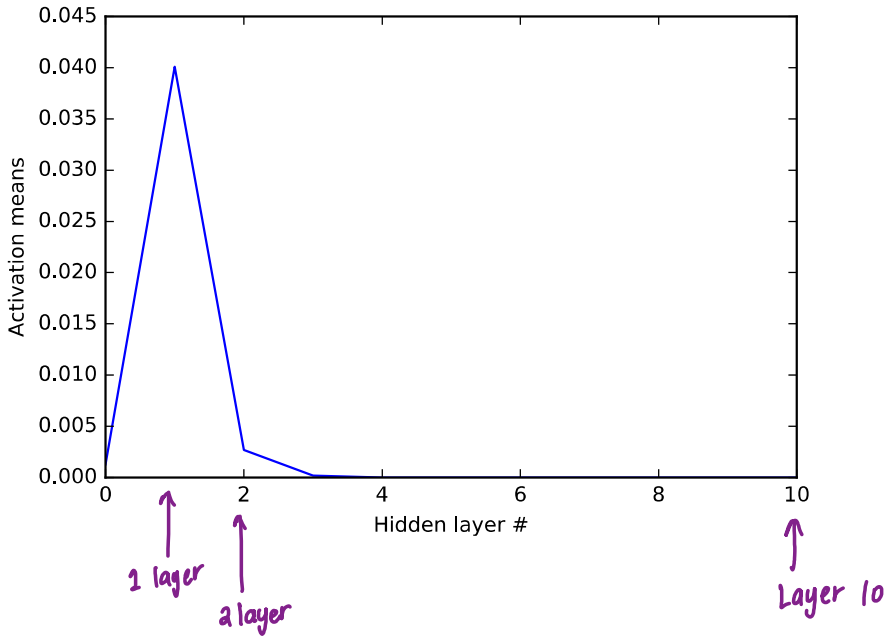
The thought is that we don't set them large enough to bias training.

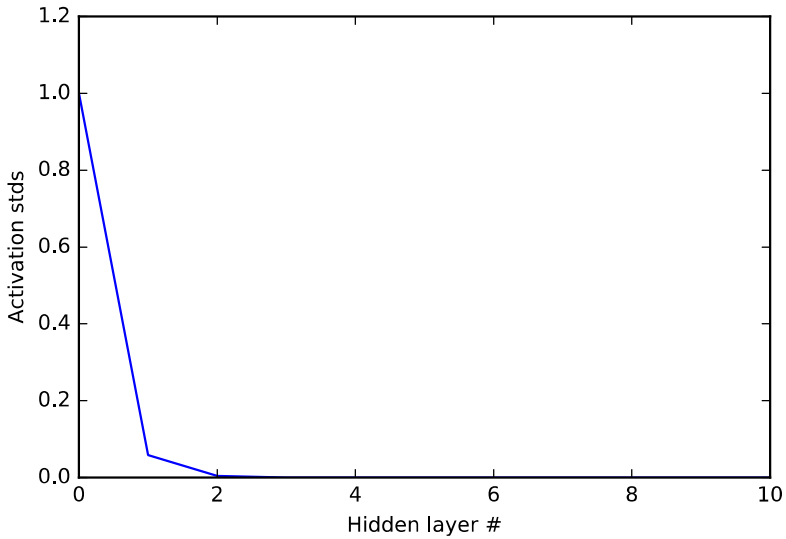
```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize small weights
    W = np.random.randn(layer_sizes[i], H.shape[0]) * 0.01
    Z = np.dot(W, H)
    H = Z * (Z > 0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

Small random weight initialization

ReLu in this example.

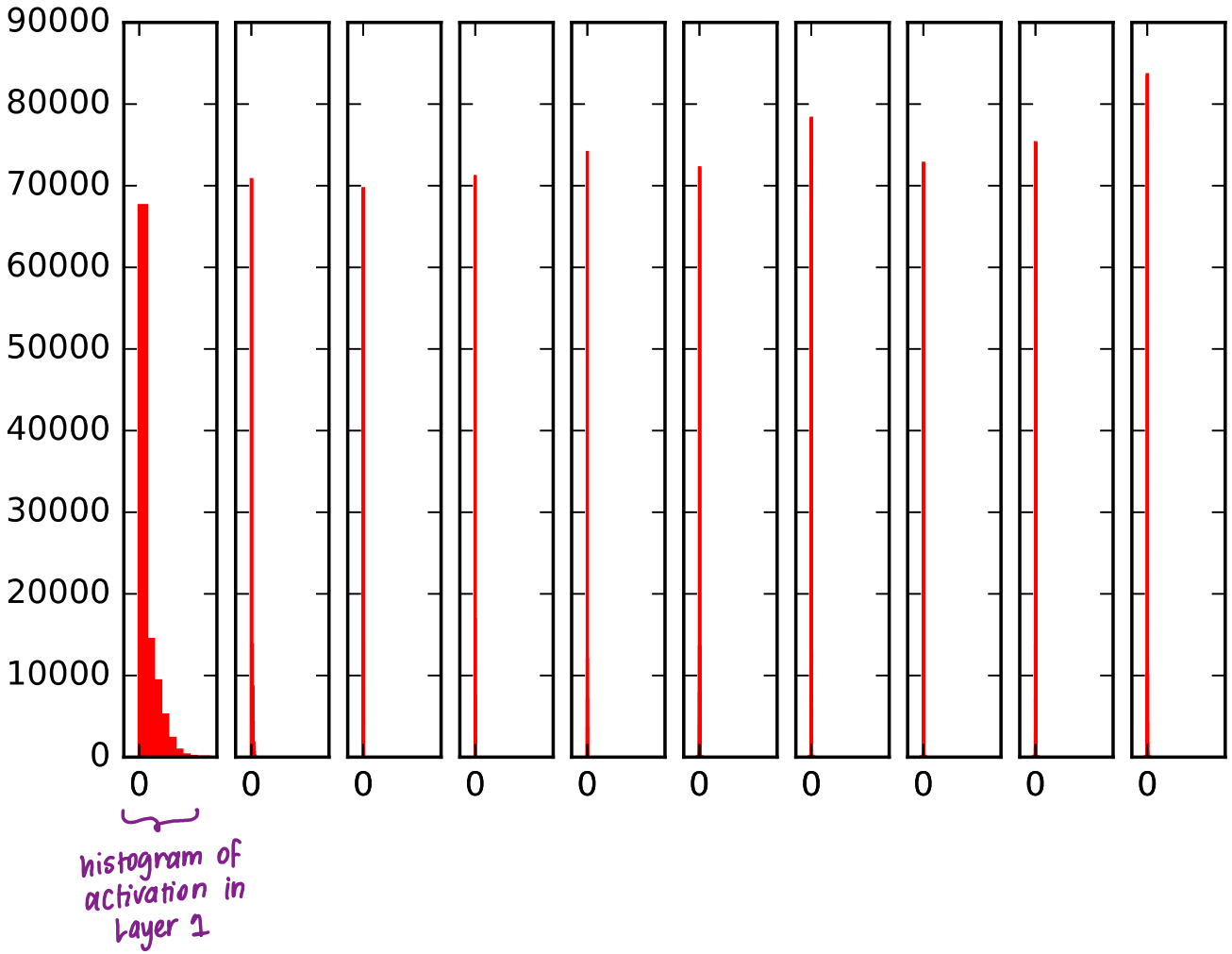
Empirically, these initializations cause all activations to decay to zero.





Small random weight initialization

Distribution of each layer's activations:



Small random weight initialization

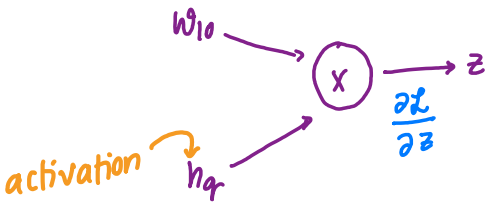
It's bad for learning if the output activations are close to zero. Why?

Think about the value of: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$

$z = w_{10} h_q$

$w_{10} \leftarrow w_{10} - \varepsilon \frac{\partial \mathcal{L}}{\partial w_{10}}$

If weights are small and activations go to zero, there's not going to be any learning



$\frac{\partial \mathcal{L}}{\partial w_{10}} = \frac{\partial \mathcal{L}}{\partial z} h_q^T$

Small random weight initialization

It's bad for learning if the output activations are close to zero. Why?

Think about the value of: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$

```
# Now backprop
dLdH = 100*np.random.randn(100, 1000) # 1000 losses for examples
loss_grads = [dLdH]
grads = []

for i in np.flip(np.arange(1,11), axis=0):
    loss_grad = loss_grads[-1]
    dLdZ = loss_grad * (Z[i] > 0)
    grad_W = np.dot(dLdZ, Hs[i-1].T)
    grad_h = np.dot(Ws[i-1].T, dLdZ)
    loss_grads.append(grad_h)
    grads.append(grad_W)
grads = list(reversed(grads))
loss_grads = list(reversed(loss_grads))
```

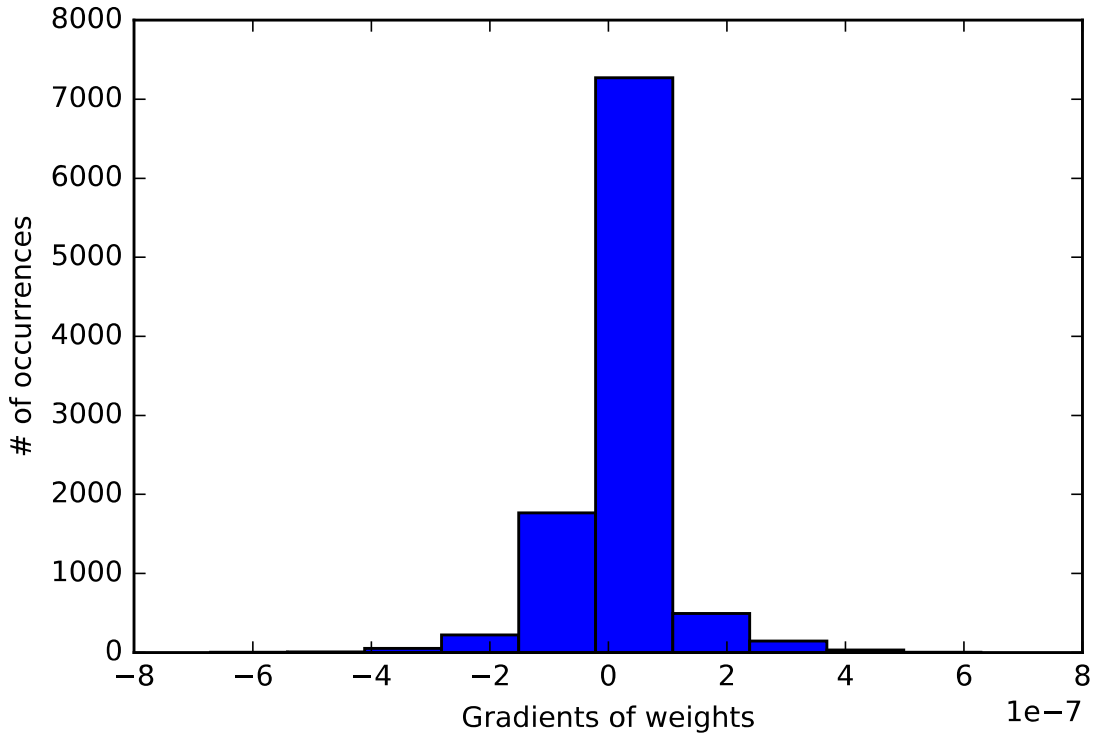
Small random weight initialization

It's bad for learning if the output activations are close to zero. Why?

Think about the value of: $\frac{\partial \ell}{\partial \mathbf{W}}$

Small, random weight initializations will not lead to learning.

Gradients for the last layer with respect to **W**:



Small random weight initialization

In practice, small weight initializations may be appropriate for smaller neural networks.

Large random weight initialization

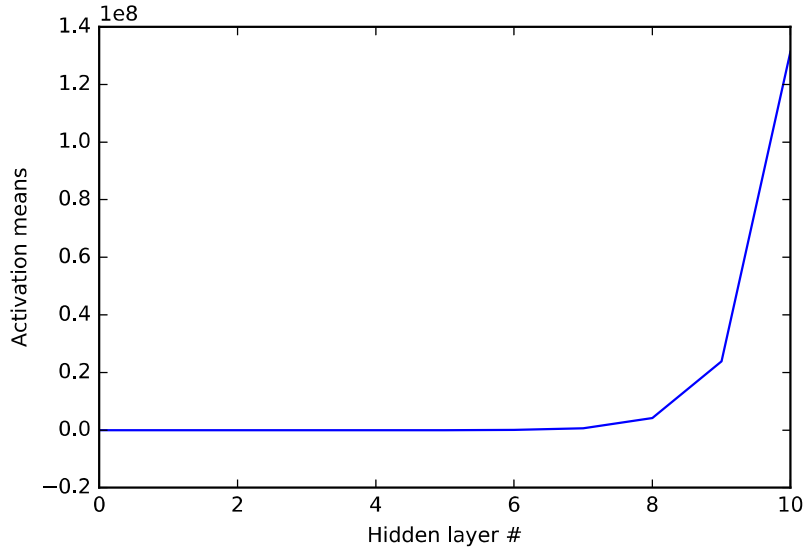
How about larger random weight initializations?

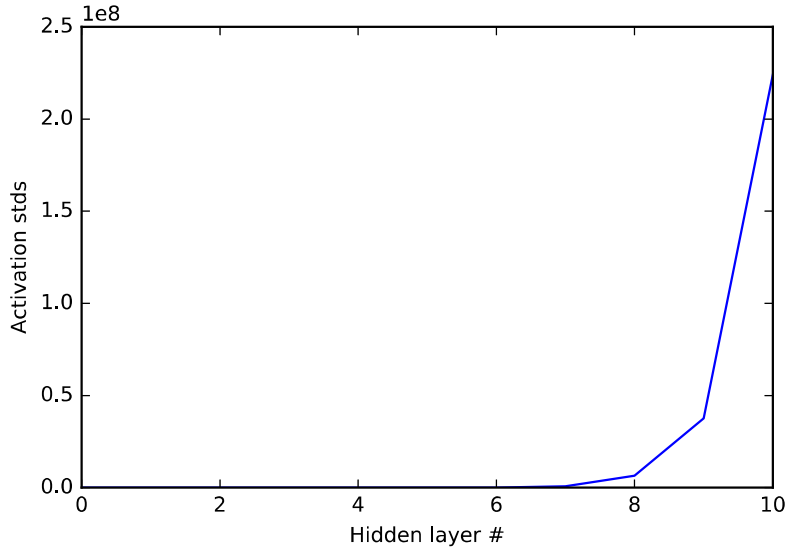
```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize large weights
    W = np.random.randn(layer_sizes[i], H.shape[0]) * 1
    Z = np.dot(W, H)
    H = Z * (Z > 0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

Large random weight initialization

Empirically, these cause the units to explode.

no learning



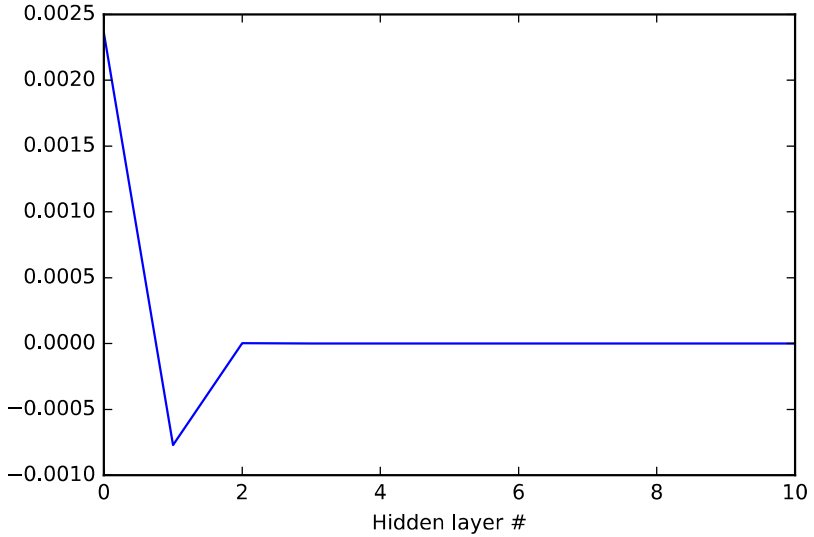


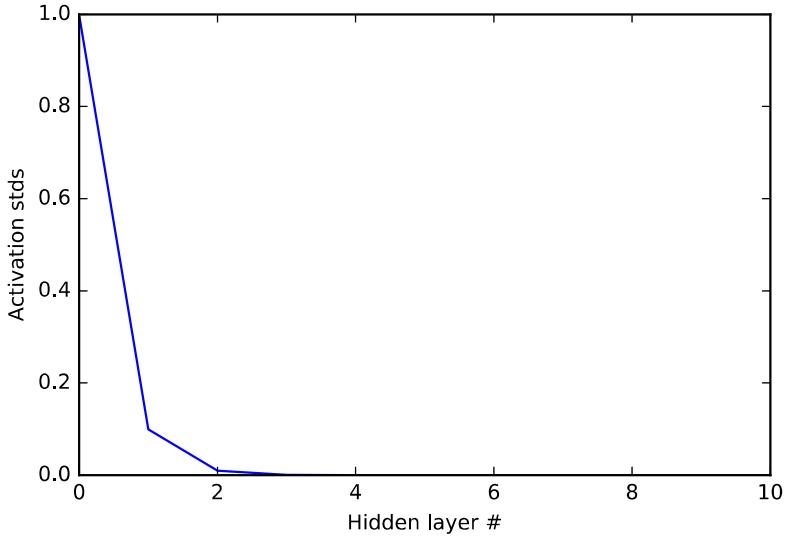
What happens to the gradients?

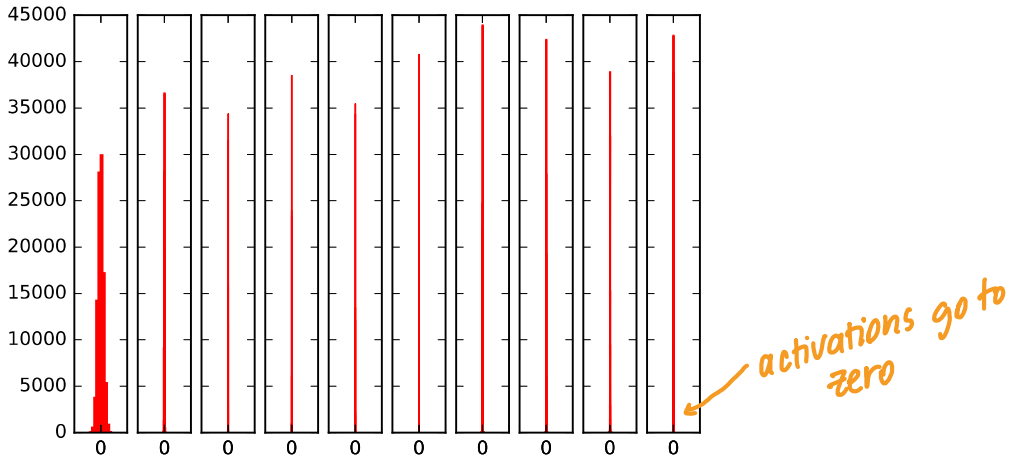
Again, this is a problem.

Will this problem occur for other activations?

What happens when we use the tanh() activation with small initialization?

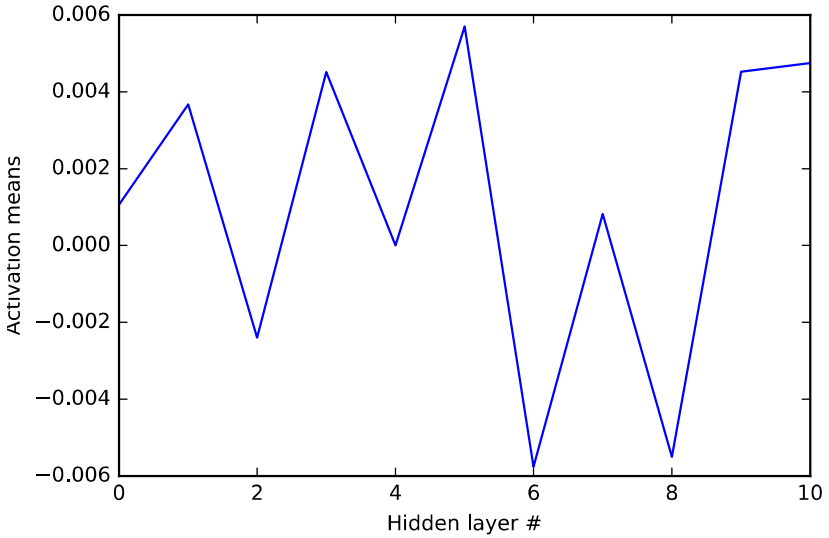


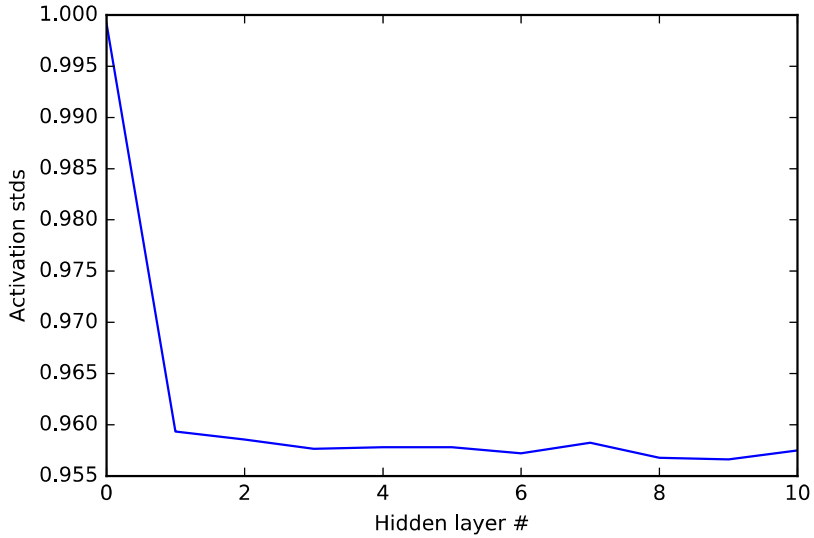


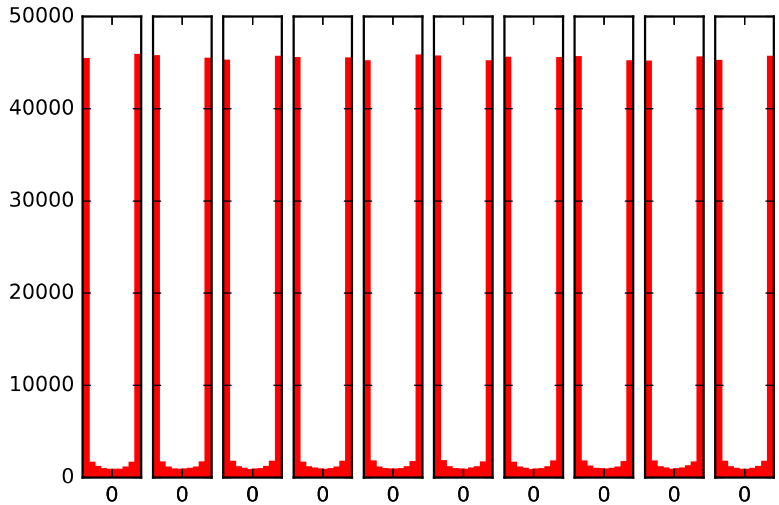


Will this problem occur for other activations?

What happens when we use the tanh() activation with large initialization?







saturates activations.
most activations will be ±1.

If not small and not large initialization, then what?

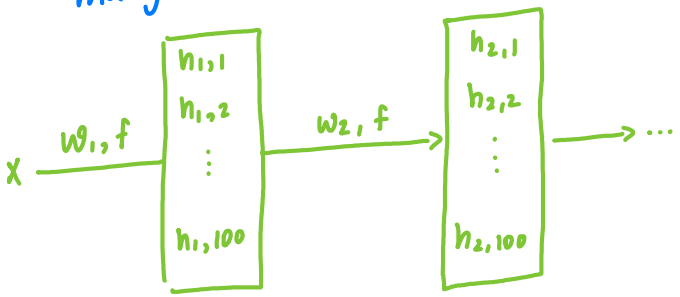
The next idea is to try an intermediate initialization.

But what intermediate value?

Xavier initialization

One initialization is from Glorot and Bengio, 2011, referred to commonly as the Xavier initialization. It intuitively argues that the variance of the units across all layers ought be the same, and that the same holds true for the backpropagated gradients.

↳ default initialization type for many implementations.



w_1, w_2 - matrix
f - activation function

simplifying assumptions

(1) $\mathbb{E}[h_{1,1}] = \mathbb{E}[h_{1,2}] = \mathbb{E}[h_{1,3}] = \cdots = \mathbb{E}[h_1]$

all units in layer 1 have same mean.

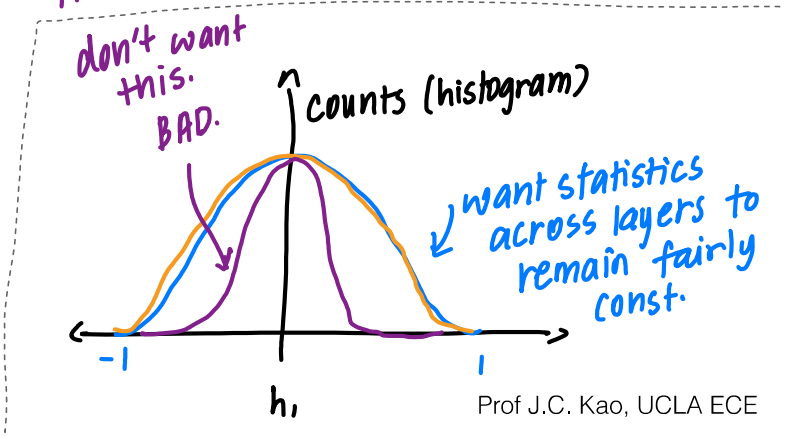
$\text{Var}(h_{1,1}) = \text{Var}(h_{1,2}) = \text{Var}(h_{1,3}) = \cdots = \text{Var}(h_1)$

all units in layer 1 have same variance.

Constraint: $\text{Var}(h_1) \approx \text{Var}(h_2) \approx \text{Var}(h_3) \approx \cdots \approx \text{Var}(h_L)$

$\text{Var}(\nabla_{h_1} \mathcal{L}) \approx \text{Var}(\nabla_{h_2} \mathcal{L}) \approx \text{Var}(\nabla_{h_3} \mathcal{L}) \approx \cdots \approx \text{Var}(\nabla_{h_L} \mathcal{L})$

variance of backpropogated gradients across all layers should remain approx the same.



Xavier initialization

For simplicity, we assume the input has equal variance across all dimensions. Then, each unit in each layer ought to have the same statistics. For simplicity we'll denote h_i to denote a unit in the ith layer, but the variances ought be the same for all units in this layer.

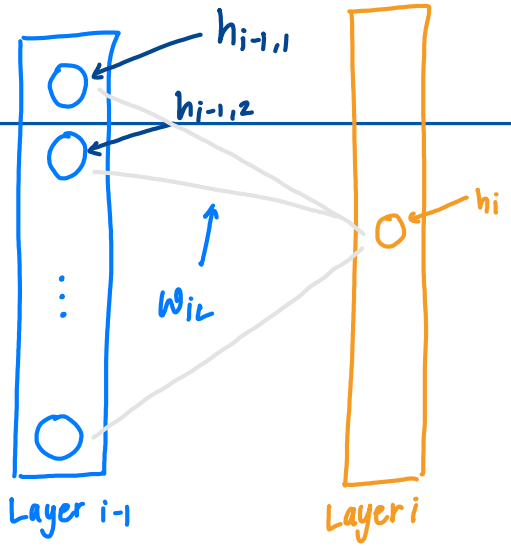
Concretely, the heuristics mean that

$$\text{var}(h_i) = \text{var}(h_j)$$

and

$$\text{var}(\nabla_{h_i} J) = \text{var}(\nabla_{h_j} J)$$

Xavier initialization



Xavier initialization (cont.)

simplifying assumption.

If the units are linear, (2) then

$$h_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} h_{i-1,j}$$

assumption (3)

Further, if the w_{ij} and h_{i-1} are independent, and all the units in the $(i-1)$th layer have the same statistics, then using the fact that

$$\begin{aligned} \text{var}(wh) &= \mathbb{E}^2(w)\text{var}(h) + \mathbb{E}^2(h)\text{var}(w) + \text{var}(w)\text{var}(h) \\ &= \text{var}(w)\text{var}(h) \qquad \text{if } \mathbb{E}(w) = \mathbb{E}(h) = 0 \end{aligned}$$

($\mathbb{E}(w)$ → 0, $\mathbb{E}(h)$ → 0) (4)

↳ assumption.

then,

$$\text{var}(h_i) = \text{var}(h_{i-1}) \cdot \sum_{j=1}^{n_{\text{in}}} \text{var}(w_{ij})$$

Note: Assumptions are questionable

$\text{var}(h_{i-1,j}) = \text{var}(h_i)$ [due to simplifying assumption from prev. slide]

variance of a unit in ith layer

Xavier: $\text{var}(h_i) = \text{var}(h_{i-1})$ [assumption from before]

$\Rightarrow \sum_{j=1}^{n_{\text{in}}} \text{var}(w_{ij}) = 1$ → assume weights w_{ij} have same stats

$n_{\text{in}}\,\text{var}(w) = 1 \Rightarrow \boxed{\text{var}(w) = \frac{1}{n_{\text{in}}}}$

Xavier initialization

Now if the weights are identically distributed, then we get that for the unit activation variances to be equal,

$$\mathrm{var}(w_{ij}) = \frac{1}{n_{\mathrm{in}}}$$

for each connection j in layer i. The same argument can be made for the backpropagated gradients to argue that:

$$\mathrm{var}(w_{ij}) = \frac{1}{n_{\mathrm{out}}}$$

Xavier initialization

Xavier initialization (cont).

To incorporate both of these constraints, we can average the number of units together, so that

$$\text{var}(w_{ij}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Hence, we can initialize each weight in layer i to be drawn from:

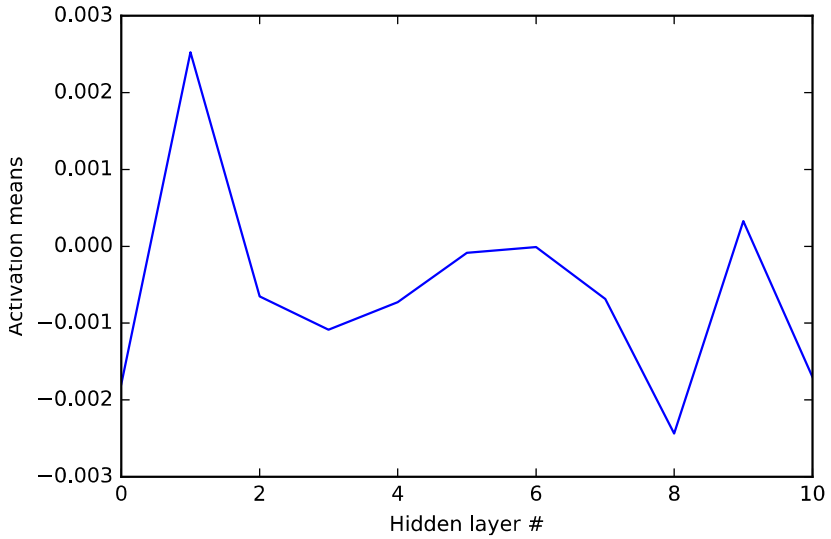
$$\mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

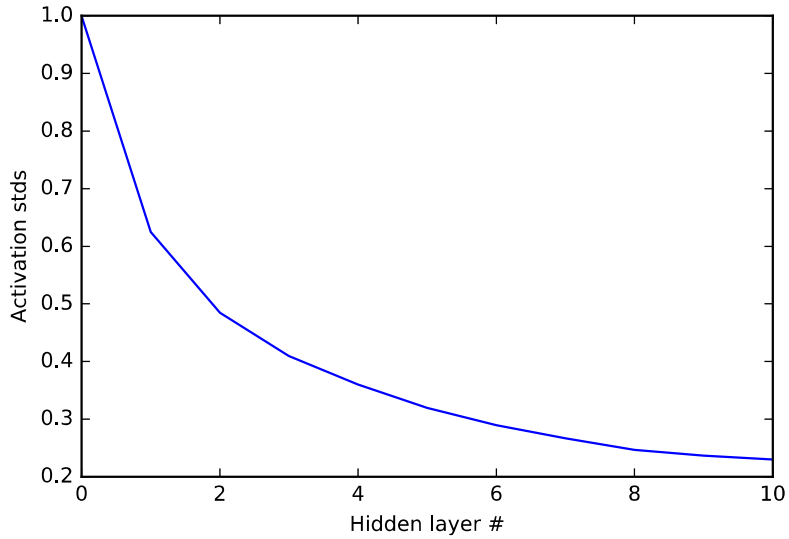
$n_{avg} = \frac{n_{in} + n_{out}}{2}$ } Some people do this to incorporate the backprop condition.

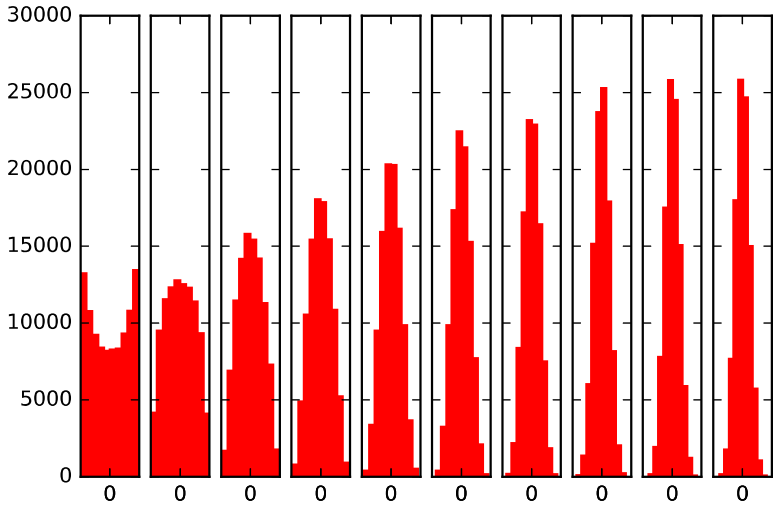
Xavier initialization with tanh

```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize Xavier
    W = np.random.randn(layer_sizes[i], H.shape[0]) * np.sqrt(2) / (np.sqrt(100 + 100))
    Z = np.dot(W, H)
    H = np.tanh(Z)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

Xavier initialization with tanh







Xavier initialization

A few notes:

- The Xavier initialization (either one) typically leads to dying ReLU units, though it is fine with $\tanh$ units.
- He et al., 2015 suggest the normalizer $2/n_{\text{in}}$ when considering ReLU units. If linear activations prior to the ReLU are equally likely to be positive or negative, ReLU kills half of the units, and so the variance decreases by half. This motivates the additional factor of 2.
- Glorot and Bengio, 2015, ultimately suggest the weights be drawn from:

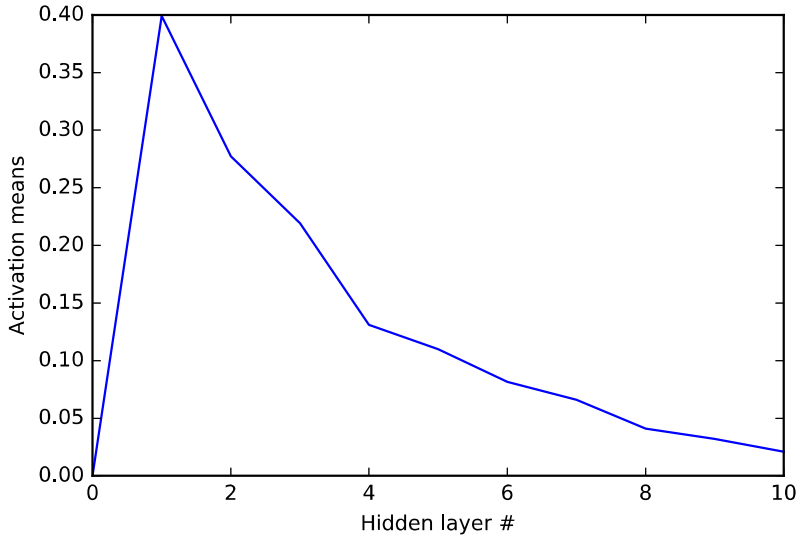
better in practice

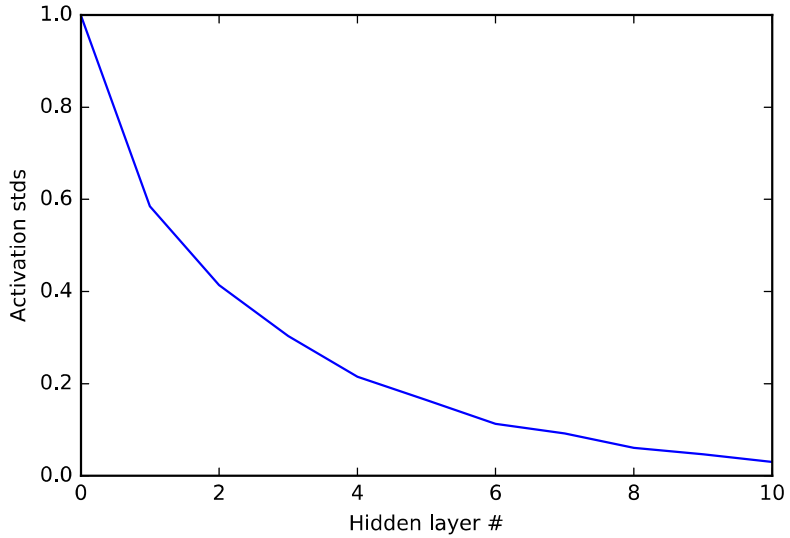
$$U\left(-\frac{\sqrt{6}}{n_{\text{in}}+n_{\text{out}}}, \frac{\sqrt{6}}{n_{\text{in}}+n_{\text{out}}}\right)$$

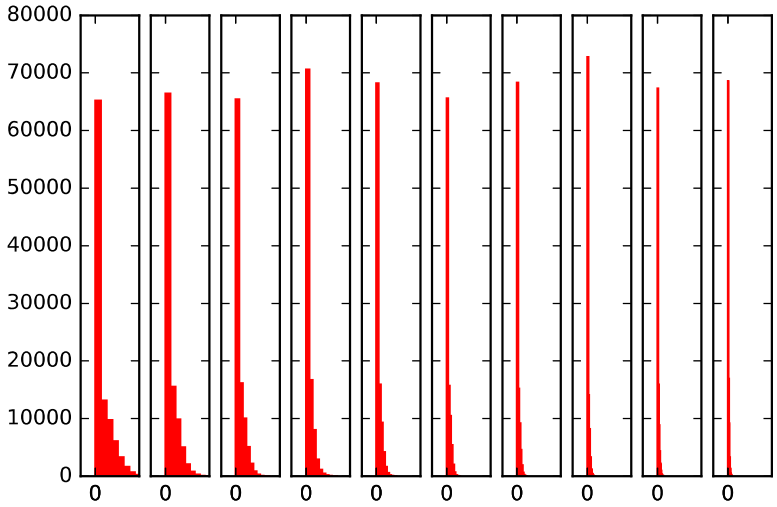
simpler to implement

$\hookrightarrow \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$

Xavier initialization with ReLU



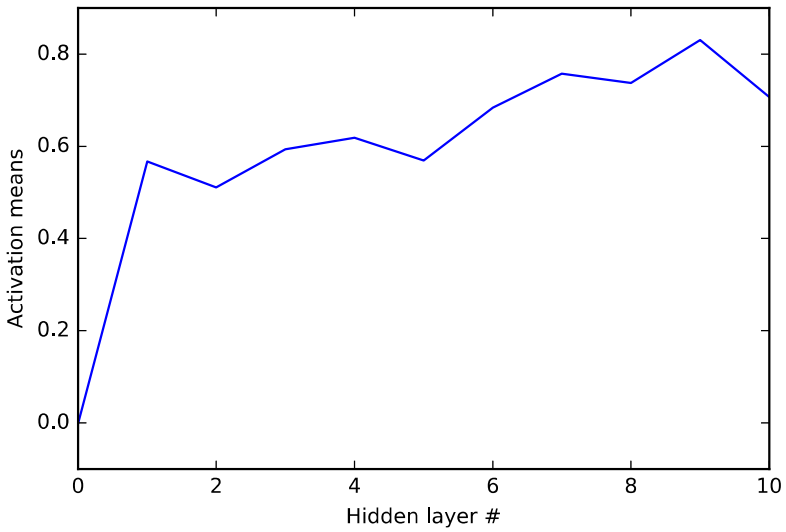


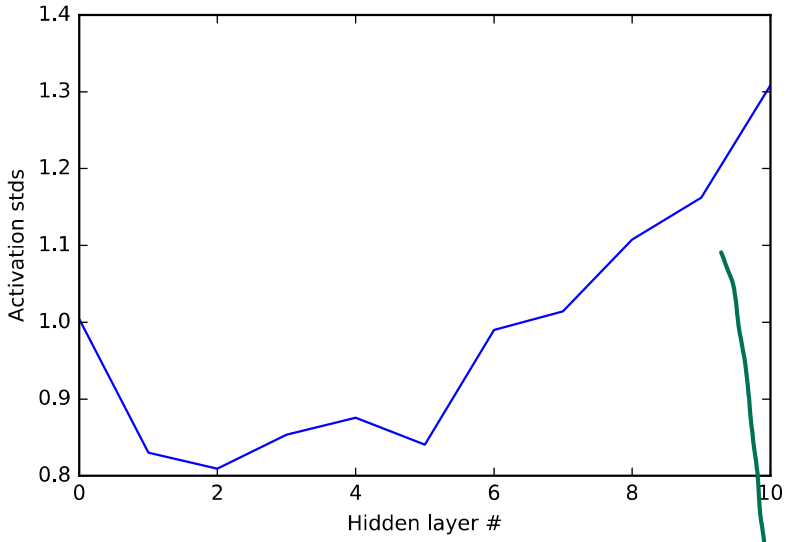


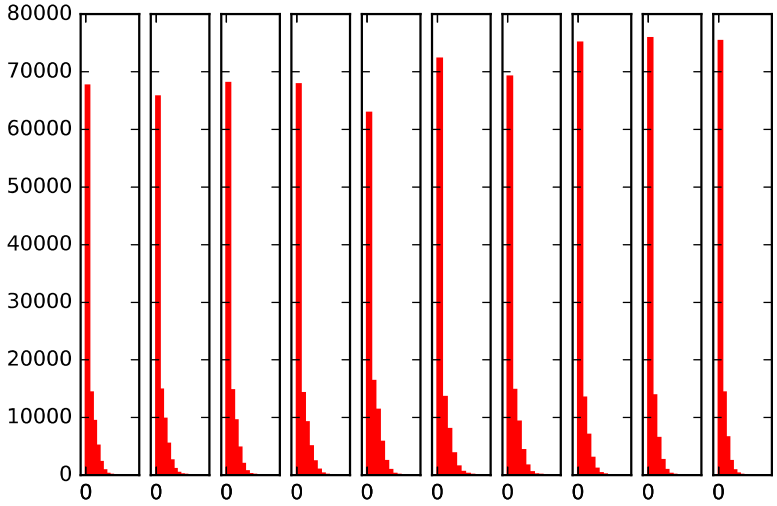
He (MSRA) initialization with ReLU

```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize Xavier
    W = np.random.randn(layer_sizes[i], H.shape[0]) * np.sqrt(2) / np.sqrt(100)
    Z = np.dot(W, H)
    H = Z * (Z>0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

He (MSRA) initialization with ReLU







if you change to $\frac{2}{N_{in}}$

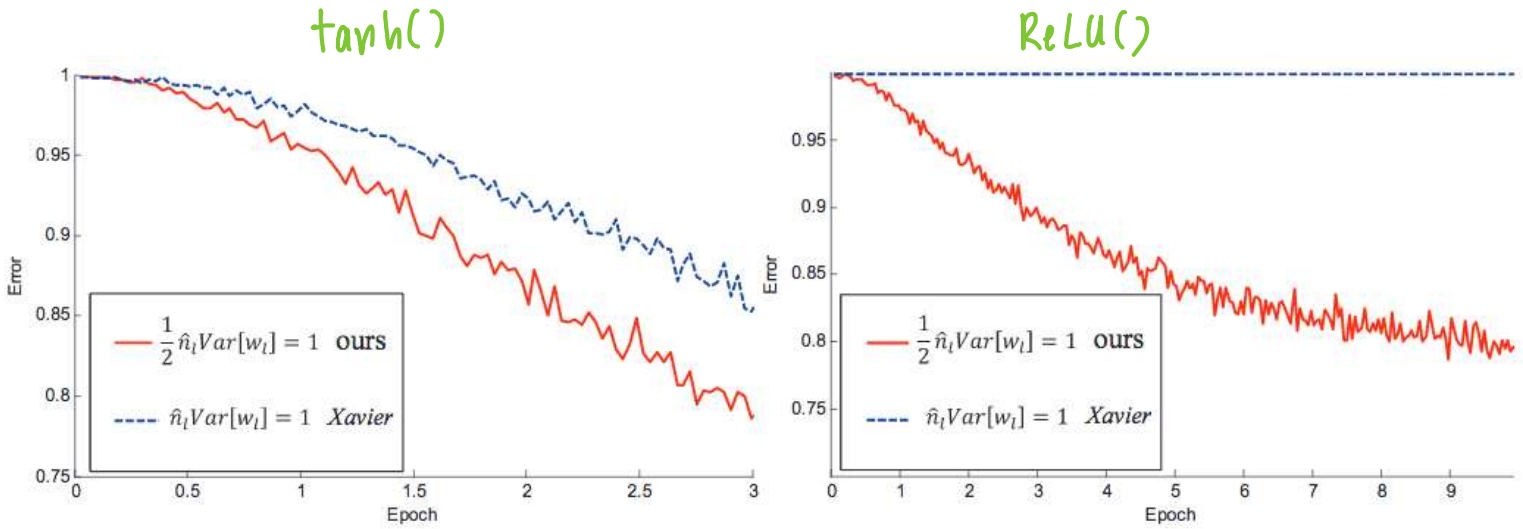
your ReLU activation survives at layer 10

Initialization take home points

Key Takeaway: Networks can be sensitive to initialization.

Initialization is a **very** important aspect of training neural networks and remains an active area of research.

A factor of two in the initialization can be the difference between the network learning well or not learning at all.



He et al., 2015

Sussillo and Abbott, "Random walk initialization for training very deep feedforward networks," 2014.
He et al., "Delving deep into rectifiers; surpassing human-level performance on ImageNet classification," 2015.
Mishkin and Matas, "All you need is a good init," 2015.

Initialization take home points

E.g., Mishkin and Matas, “All you need is a good init,” 2015.

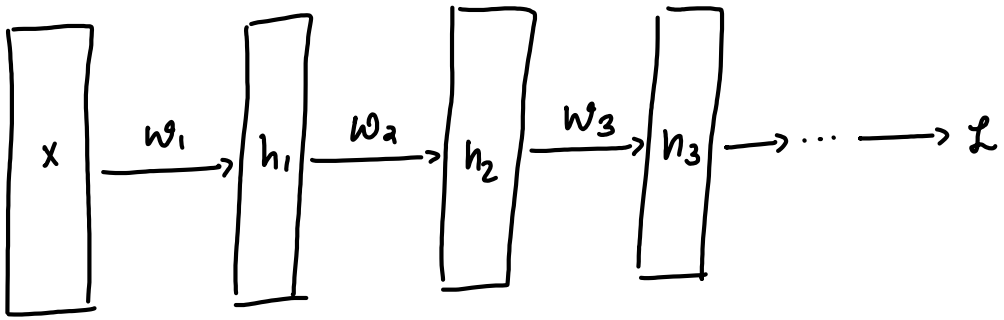
Init method	maxout	ReLU	VLReLU	tanh	Sigmoid
LSUV	**93.94**	**92.11**	92.97	89.28	n/c
OrthoNorm	93.78	91.74	92.40	89.48	n/c
OrthoNorm-MSRA scaled	–	91.93	**93.09**	–	n/c
Xavier	91.75	90.63	92.27	**89.82**	n/c
MSRA	n/c†	90.91	92.43	89.54	n/c

BATCH NORMALIZATION → average over a batch.

Avoiding saturation, high variance activations, etc.

We will motivate batchnorm through internal covariate shift, though new research suggests that batchnorm works primarily due to making the loss surface smoother.

An obstacle to standard training is that the distribution of the inputs to each layer changes as learning occurs in previous layers. As a result, the unit activations can be very variable. Another consideration is that when we do gradient descent, we're calculating how to update each paramer *assuming the other layers don't change*. But these layers may change drastically.



$\frac{\partial \mathcal{L}}{\partial w_1}$ $\frac{\partial \mathcal{L}}{\partial w_2}$ $\frac{\partial \mathcal{L}}{\partial w_3}$ $\frac{\partial \mathcal{L}}{\partial w_{softmax}}$

Don't want to run into an issue where update to w_3 is "undone" by updates to w_1, w_2.

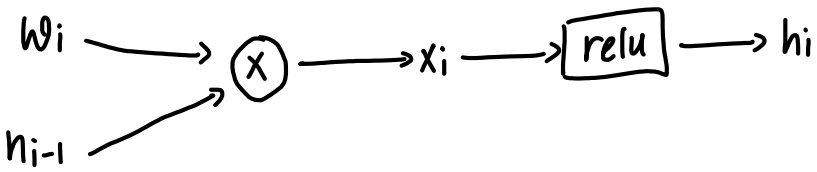
main issue with this approach: we assume that other layers don't change, which is not true.

how do I change gradient to make loss better, assuming statistics of h_2 don't change.

[affine]: $x_i = W_i h_{i-1} + b_i$

Batch normalization

The idea of batch-normalization is to make the output of each layer have unit statistics. Learning then becomes simpler because parameters in the lower layers do not change the statistics of the input to a given layer. This makes learning more straightforward.

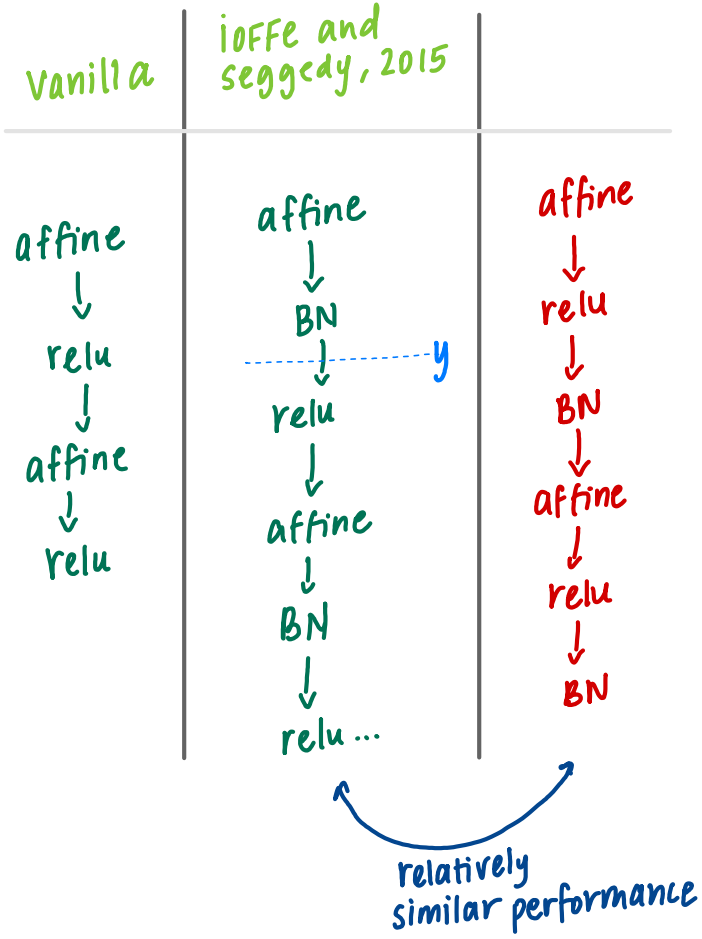


$h_i = \text{relu}(x_i)$

$\mathbb{E}[x_i] = 0$

$\text{Var}(x_i) = 1$

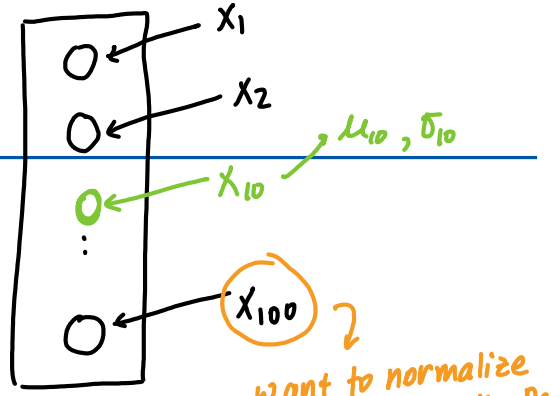
} unit statistics



Other norms:
"Layer norm"
"Group Norm"

Biases network to use unit statistics but has flexibility to allow it to deviate from that.

Batch normalization



Batch normalization (cont.)

(Ioffe and Szegedy, 2015), introduced batch normalization.

- Normalize the unit activations:

$\mathbb{E}(\hat{x}_i) = 0$, $\mathrm{Var}(\hat{x}_i) = 1$

mean 0 s.d. close to 1

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

mean and variance across a single batch.

prevents division by zero

with

batch size (e.g. 256)

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

j^{th} example

and ε small.

Note: if you have batch size of 1, then no diff. b.c. we compute statistics across a batch.

- Scale and shift the normalized activations:

output of batch norm layer

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

$\mathbb{E}\, y_i = \beta_i$, $\mathrm{var}(y_i) = \gamma_i^2$ — output of batch norm layer generally has nonzero mean and non-unit var.

Importantly, the normalization and scale / shift operations are included in the computational graph of the neural network, so that they participate not only in forward propagation, but also in backpropagation.

Running 256 forward passes gives 256 values of a neuron (x_{10}), then we compute "average" value of x_{10} across these 256 batches.

learnable params. ⟹ prevent from overly restricting the network.

Batch normalization (cont.)

A few notes on batch normalization's implementation:

- The reason the scaling is on a per unit basis is primarily computational efficiency. It is possible to normalize the entire layer via $\Sigma^{-1/2}(\mathbf{x} - \mu)$ where μ, Σ are the mean and covariance of $\mathbf{x}$. However, this requires computation of a covariance matrix, its inverse, and the appropriate terms for backpropagation (including computation of the Jacobian of this transform).
- The scale and shift layer is inserted in case it is better that the activations not be zero mean and unit variance. As γ_i and β_i are parameters, it is possible for the network to rescale the activations. In fact, it could learn $\gamma_i = \sigma_i$ and $\beta_i = \mu_i$ to undo the normalization.

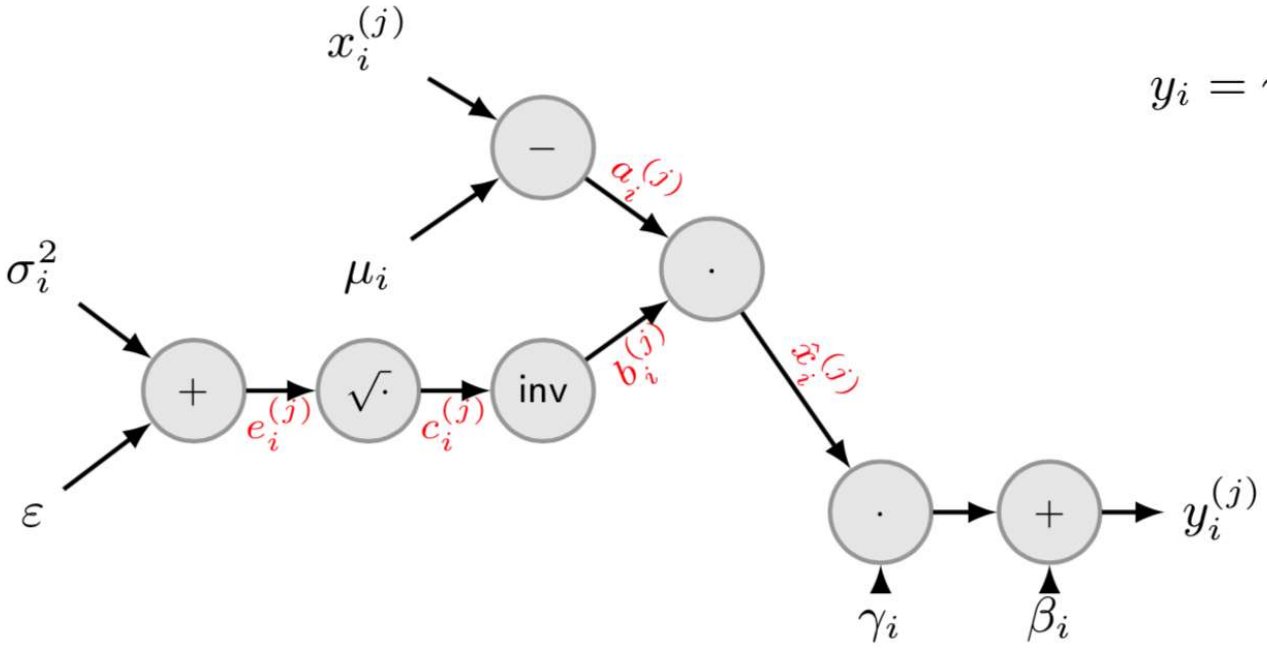
Batch normalization

one unit (i)
one example (j)

Batch normalization computational graph

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

$$y_i = \gamma_i \hat{x}_i + \beta_i$$



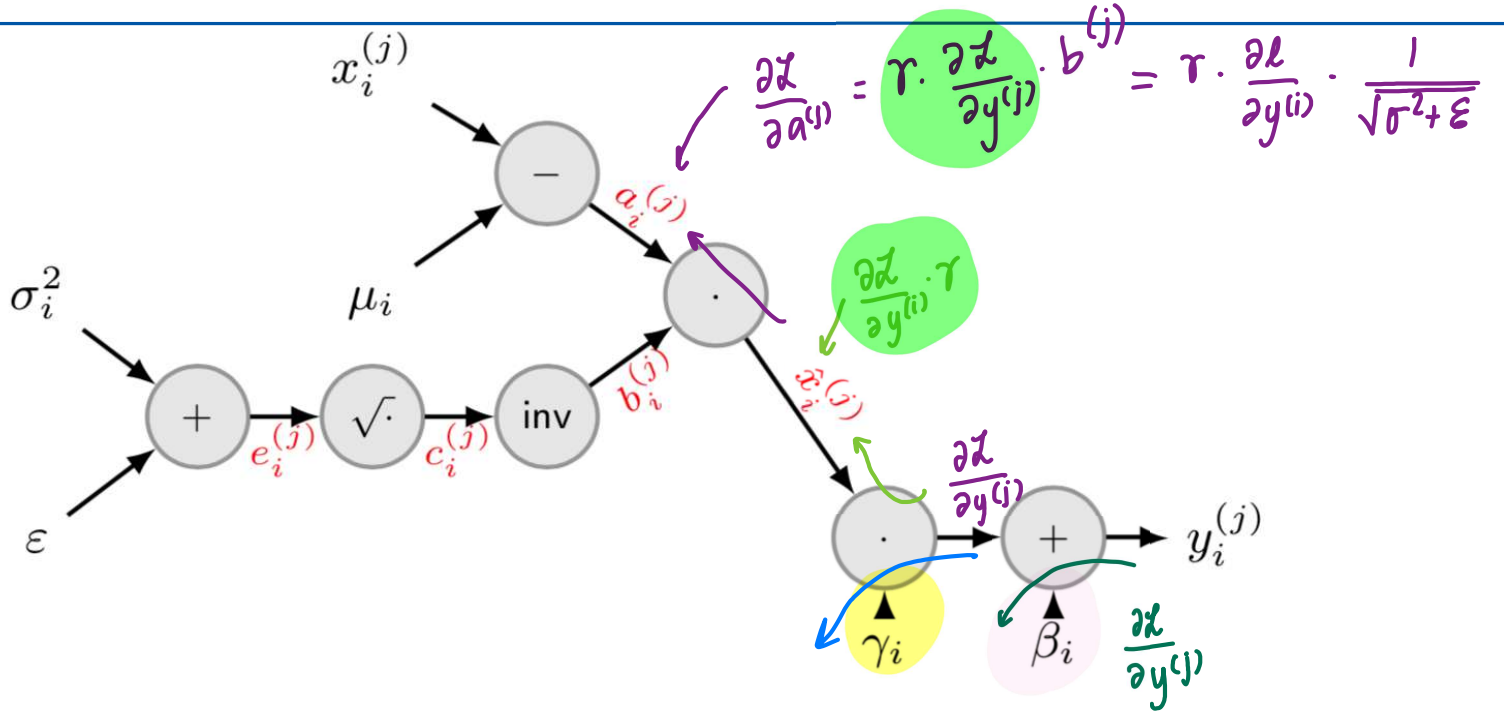
Gradients from backprop on next page.

We will drop the subscript i for convenience.

Note: In this case the order of backprop gradient does not matter b.c. all scalars.

Batch normalization

$$b^{(i)} = \frac{1}{\sqrt{\sigma^2+\varepsilon}}$$

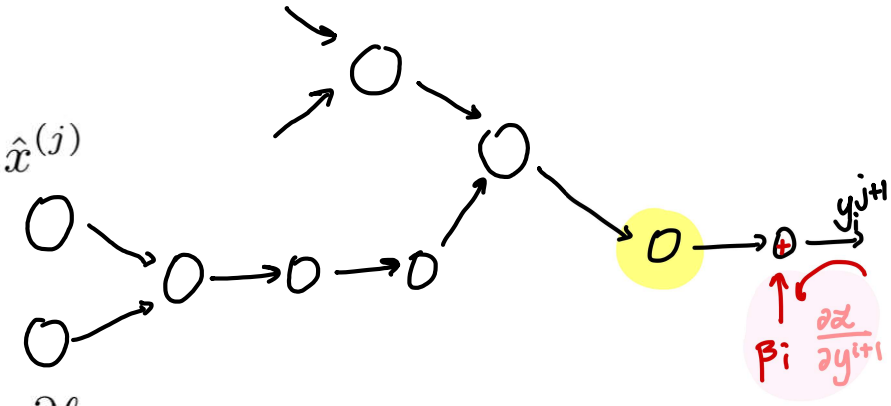


$$\frac{\partial \ell}{\partial \beta} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}} \hat{x}^{(j)}$$

$$\frac{\partial \ell}{\partial \hat{x}^{(j)}} = \frac{\partial \ell}{\partial y^{(j)}} \gamma$$

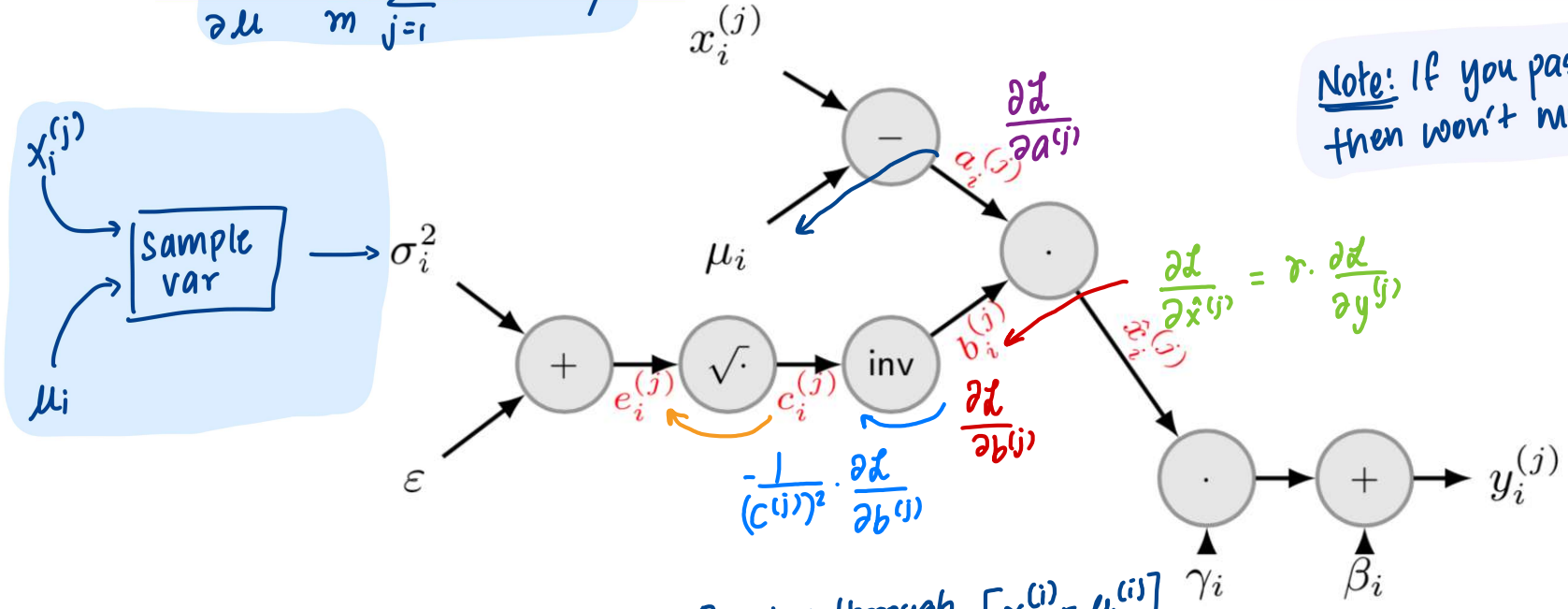
$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$



Batch normalization

$$\sigma^2 = \frac{1}{m}\sum_{j=1}^{m}(x^{(j)} - \mu)^2$$

$$\frac{\partial \sigma^2}{\partial \mu} = \frac{-2}{m}\sum_{j=1}^{m}(x^{(j)} - \mu)$$



Note: If you passed through to $x^{(i)}$, then won't multiply by (-)

Passing through $[x^{(i)} - \mu^{(i)}]$ minus sign.

$$\frac{\partial \ell}{\partial \mu} = \ominus \frac{1}{\sqrt{\sigma^2 + \epsilon}} \sum_{j=1}^{m} \frac{\partial \ell}{\partial \hat{x}^{(j)}} - \left(\frac{2}{m}\sum_{j=1}^{m}(x^{(j)} - \mu)\right)\frac{\partial \ell}{\partial \sigma^2}$$

$$\frac{\partial \ell}{\partial b^{(j)}} = (x^{(j)} - \mu)\frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial c^{(j)}} = -\frac{1}{\sigma^2 + \varepsilon}(x^{(j)} - \mu)\frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2}\frac{1}{(\sigma^2 + \varepsilon)^{3/2}}(x^{(j)} - \mu)\frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

Backpropagating through inv:

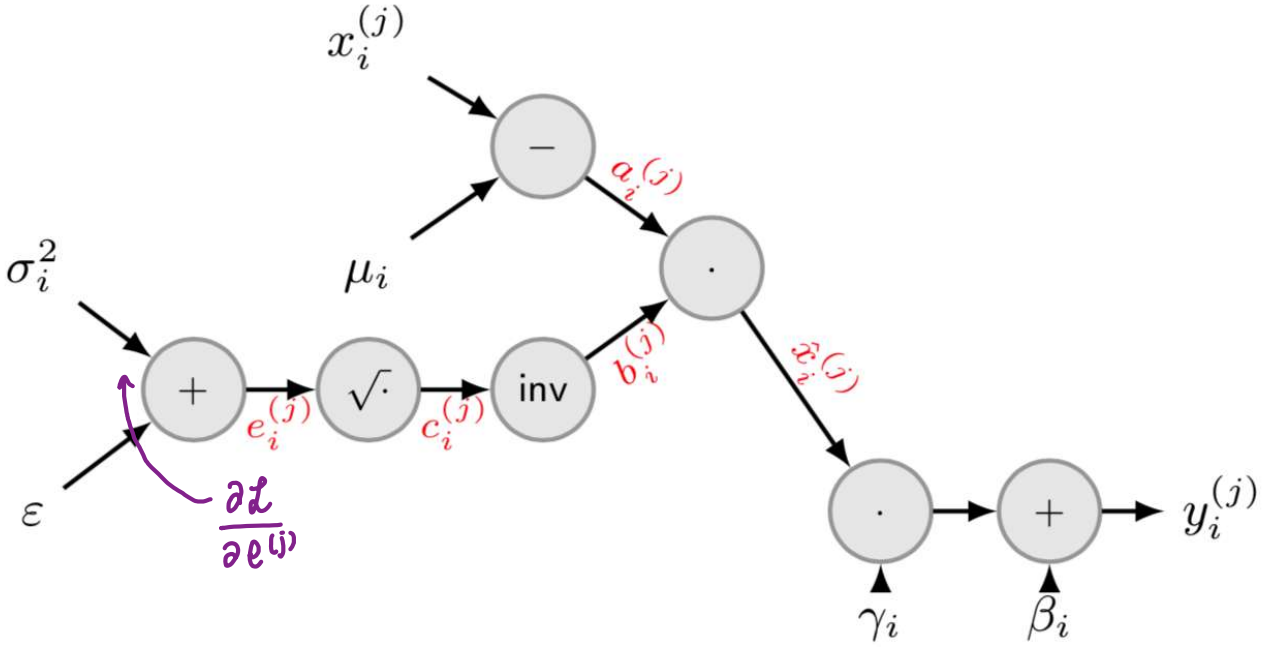
$b^{(j)} = \frac{1}{c^{(j)}}$

$\frac{\partial b^{(j)}}{\partial c^{(j)}} = -\frac{1}{(c^{(j)})^2}$

$c^{(j)} = \sqrt{e^{(j)}}$

$\frac{\partial c^{(j)}}{\partial e^{(j)}} = \frac{1}{2\sqrt{e^{(j)}}}$ $(e^{(j)} = \sigma^2 + \varepsilon)$

Batch normalization



$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} \qquad \frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

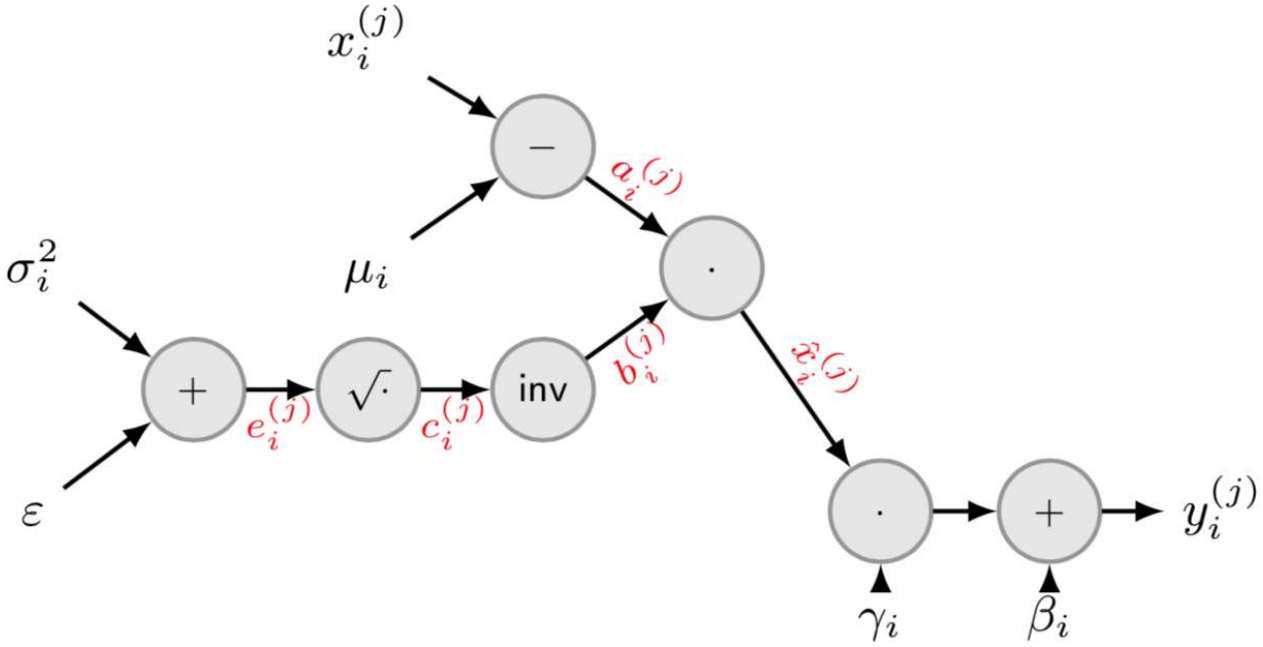
Where does summation come from.

From prev slide.

$$\frac{\partial \ell}{\partial \sigma^2} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial e^{(j)}}$$

$$= \sum_{j=1}^{m} -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

Batch normalization



$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} \qquad \frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial x^{(j)}} = \frac{\partial \ell}{\partial a^{(j)}} + \frac{\partial \sigma^2}{\partial x^{(j)}} \frac{\partial \ell}{\partial \sigma^2} + \frac{\partial \mu}{\partial x^{(j)}} \frac{\partial \ell}{\partial \mu}$$

$$= \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} + \frac{2(x^{(j)} - \mu)}{m} \frac{\partial \ell}{\partial \sigma^2} + \frac{1}{m} \frac{\partial \ell}{\partial \mu}$$

Batch normalization layer

The batch normalization layer is typically placed right before the nonlinear activation. Hence, a layer of a neural network may look like:

$$\mathbf{h}_i = f\left(\text{batch-norm}\left(\mathbf{W}_i\mathbf{h}_{i-1} + \mathbf{b}_i\right)\right)$$

Batch normalization

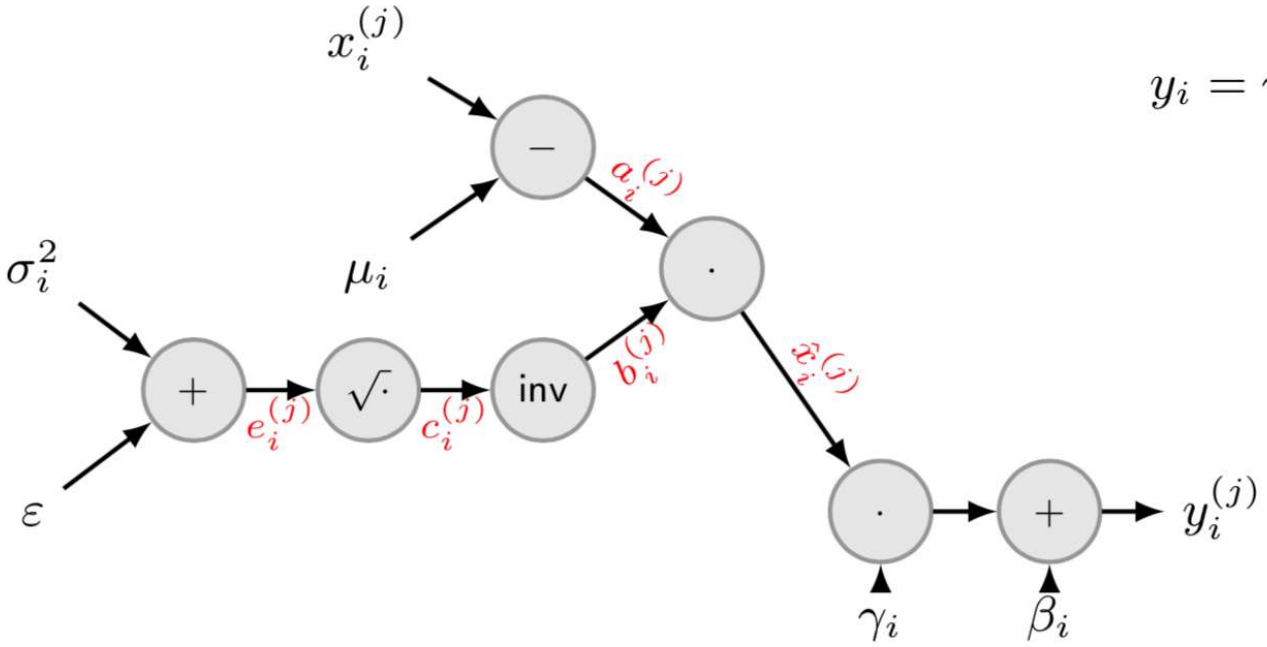
Empirically, batch normalization allows higher learning rates to be used.

Batch normalization

Batch normalization computational graph

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

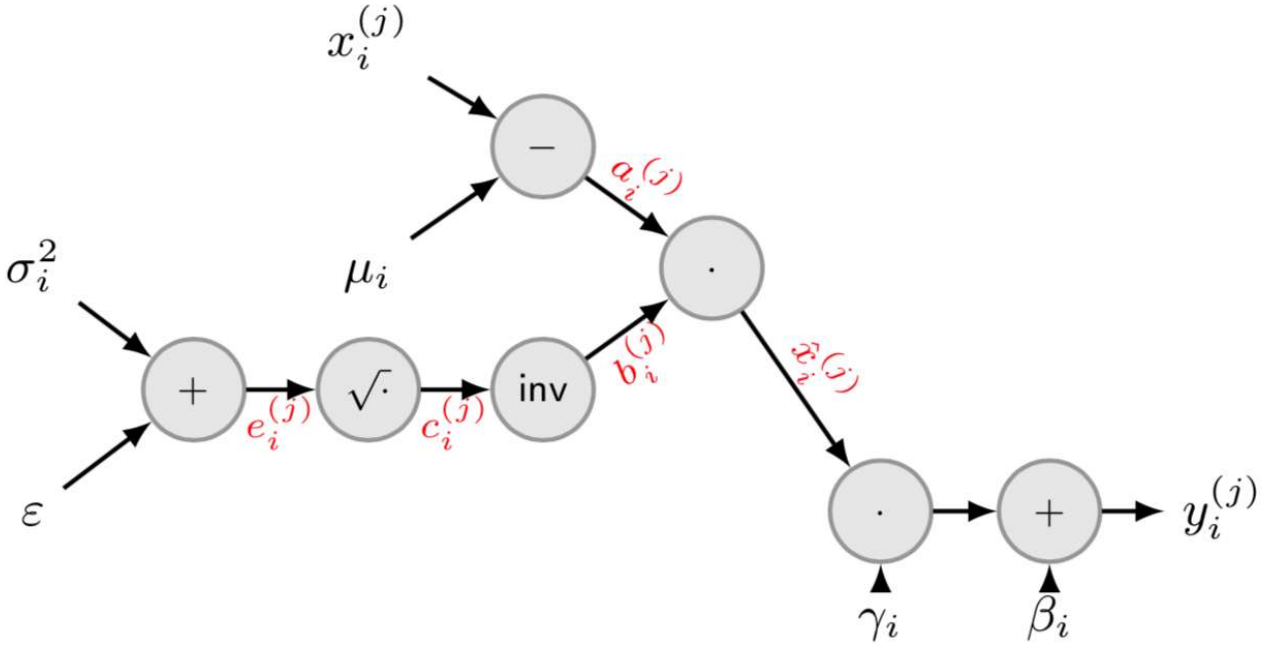
$$y_i = \gamma_i \hat{x}_i + \beta_i$$



Gradients from backprop on next page.

We will drop the subscript i for convenience.

Batch normalization



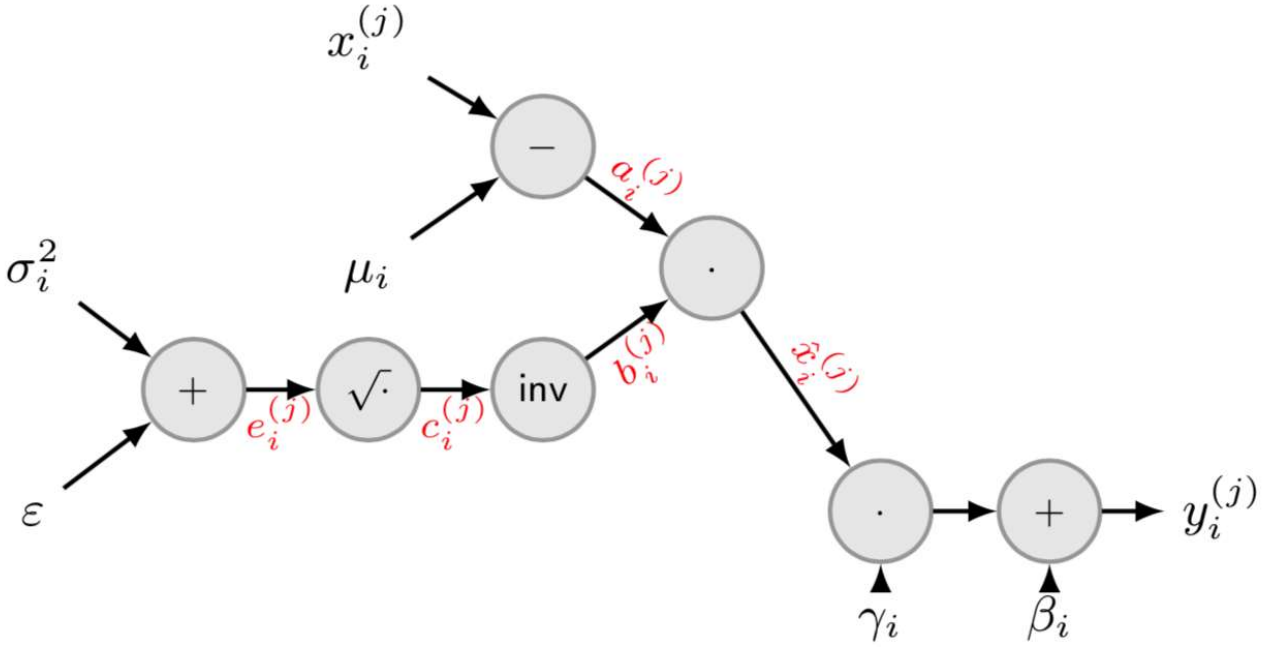
$$\frac{\partial \ell}{\partial \beta} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}} \hat{x}^{(j)}$$

$$\frac{\partial \ell}{\partial \hat{x}^{(j)}} = \frac{\partial \ell}{\partial y^{(j)}} \gamma$$

$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

Batch normalization



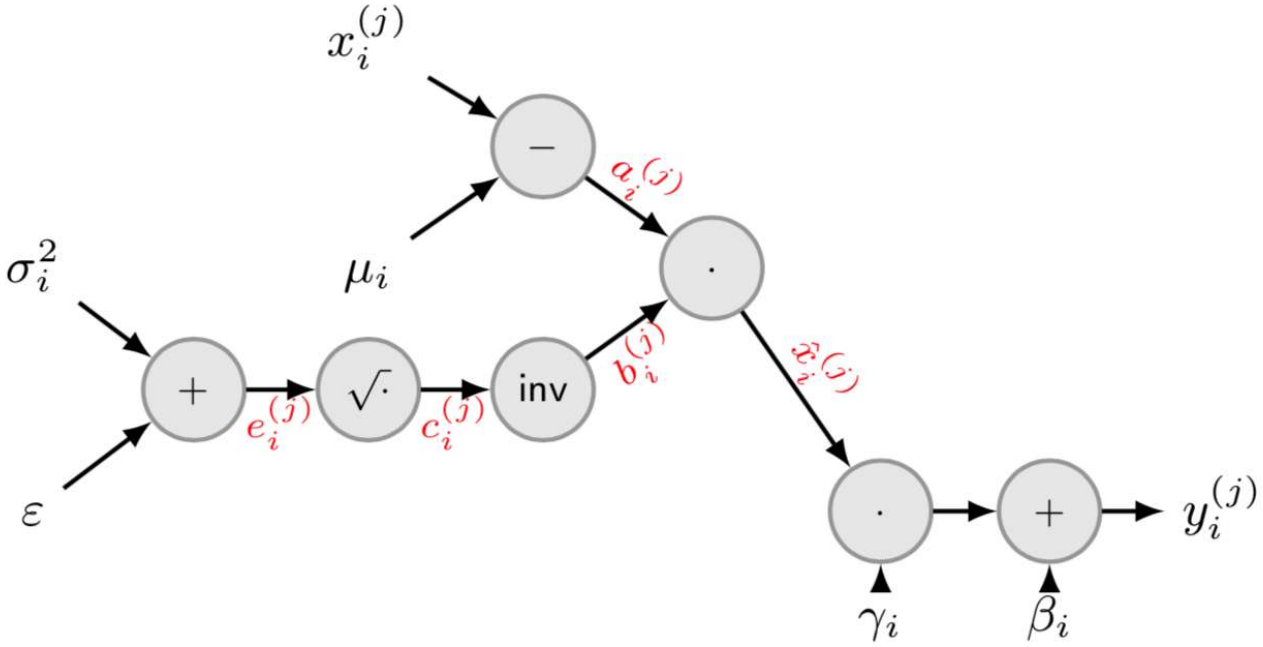
$$\frac{\partial \ell}{\partial \mu} = -\frac{1}{\sqrt{\sigma^2+\epsilon}} \sum_{j=1}^{m} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial b^{(j)}} = (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial c^{(j)}} = -\frac{1}{\sigma^2+\varepsilon} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

Batch normalization



$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial \sigma^2} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial e^{(j)}}$$

$$= \sum_{j=1}^{m} -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$