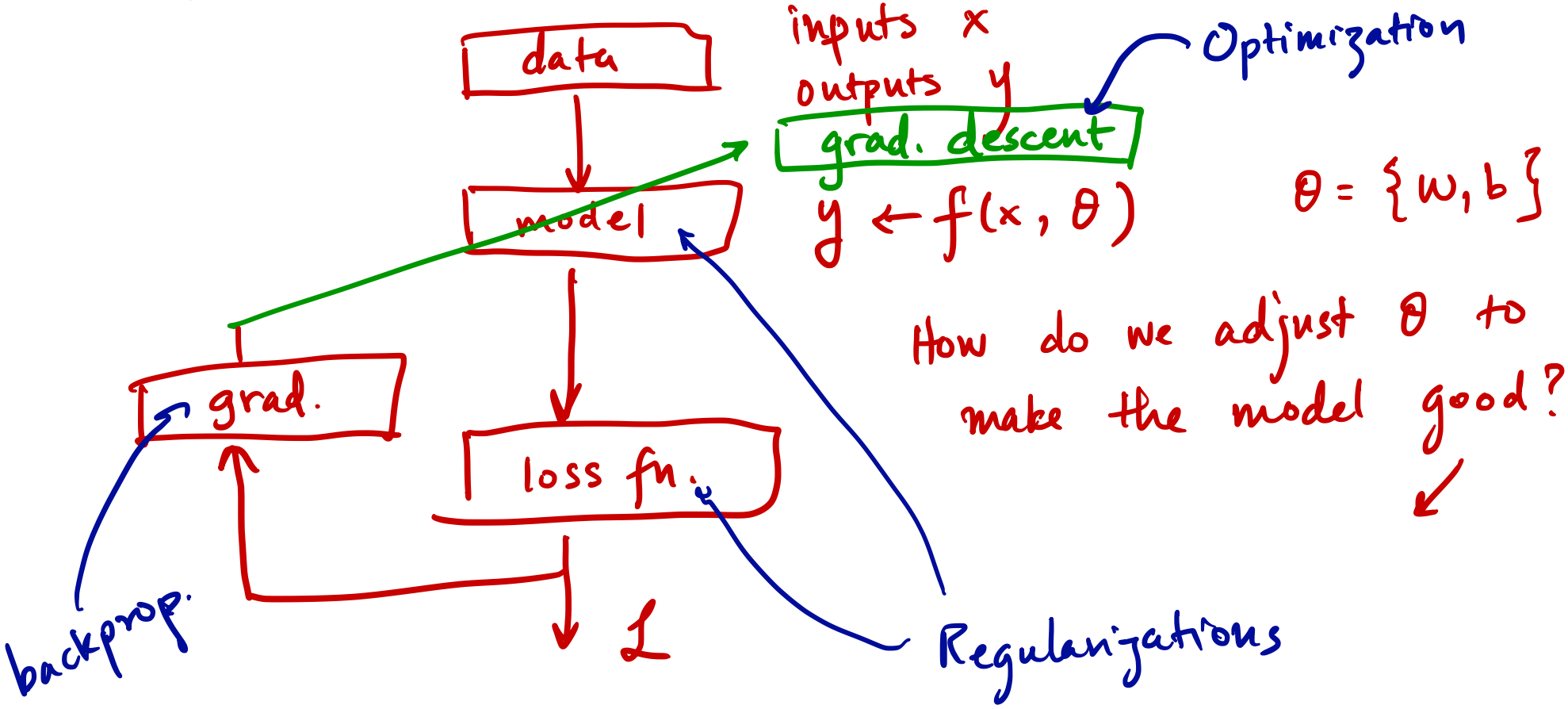
Announcements, 2018-01-~~29~~ 31

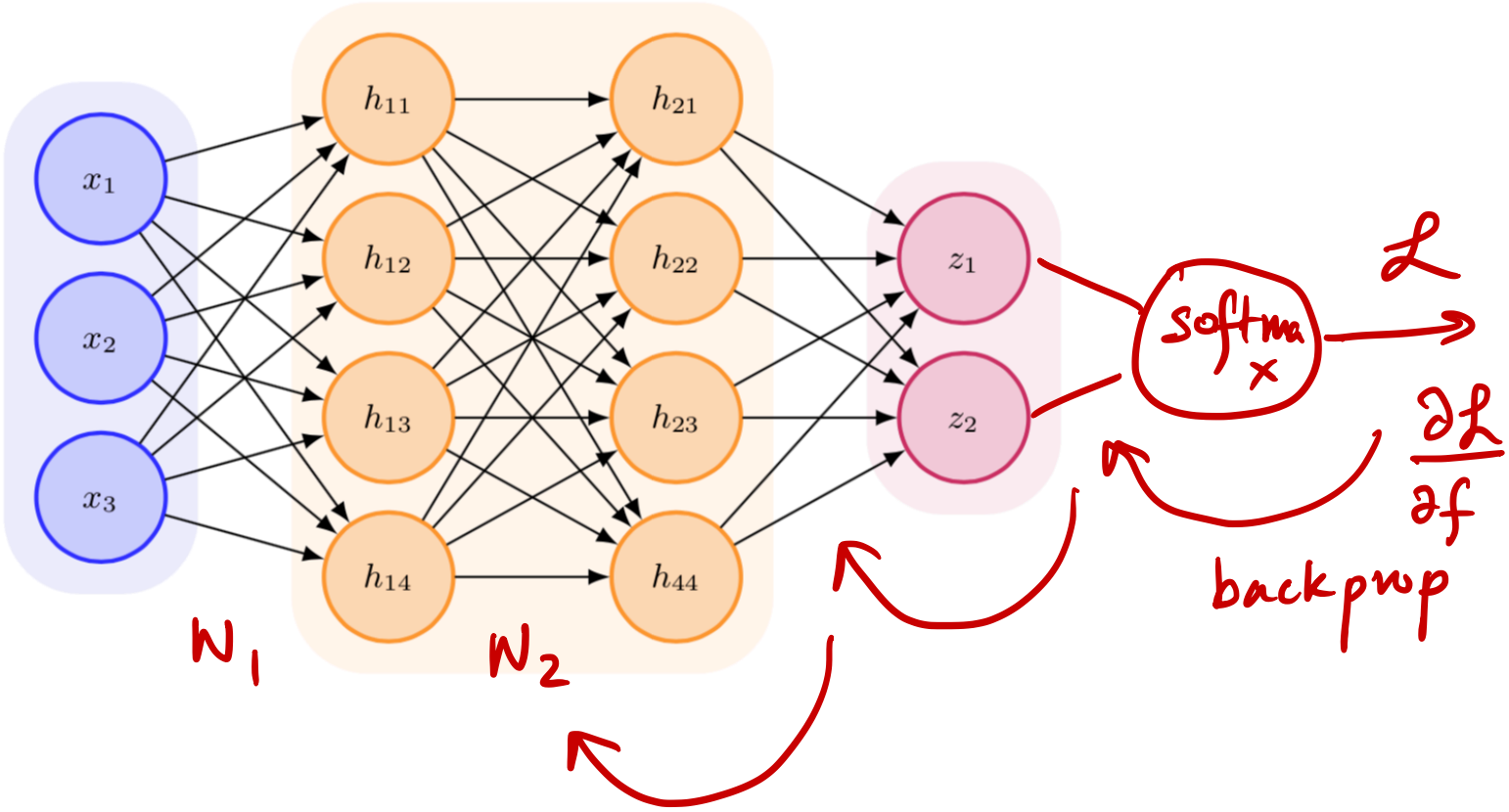
- HW #2 due Wednesday, 31 Jan 2018, at 11:59pm.
  - Be sure to get it in!
  - Be sure you submit the .py files that implement the knn, svm, and softmax classes! (This is in addition to the Jupyter notebooks.)
- HW #3 will be distributed on Wednesday, 31 Jan 2018, and will be due a week later on Wednesday, 7 Feb 2018.
- HW #1 regrades need to be in by this Fri; we will close the regrade requests then.
- Front light?

Recap from 2018-01-29

A recap of the overall problem set up for machine learning, and where each component fits in so far.



Recap from 2018-01-29



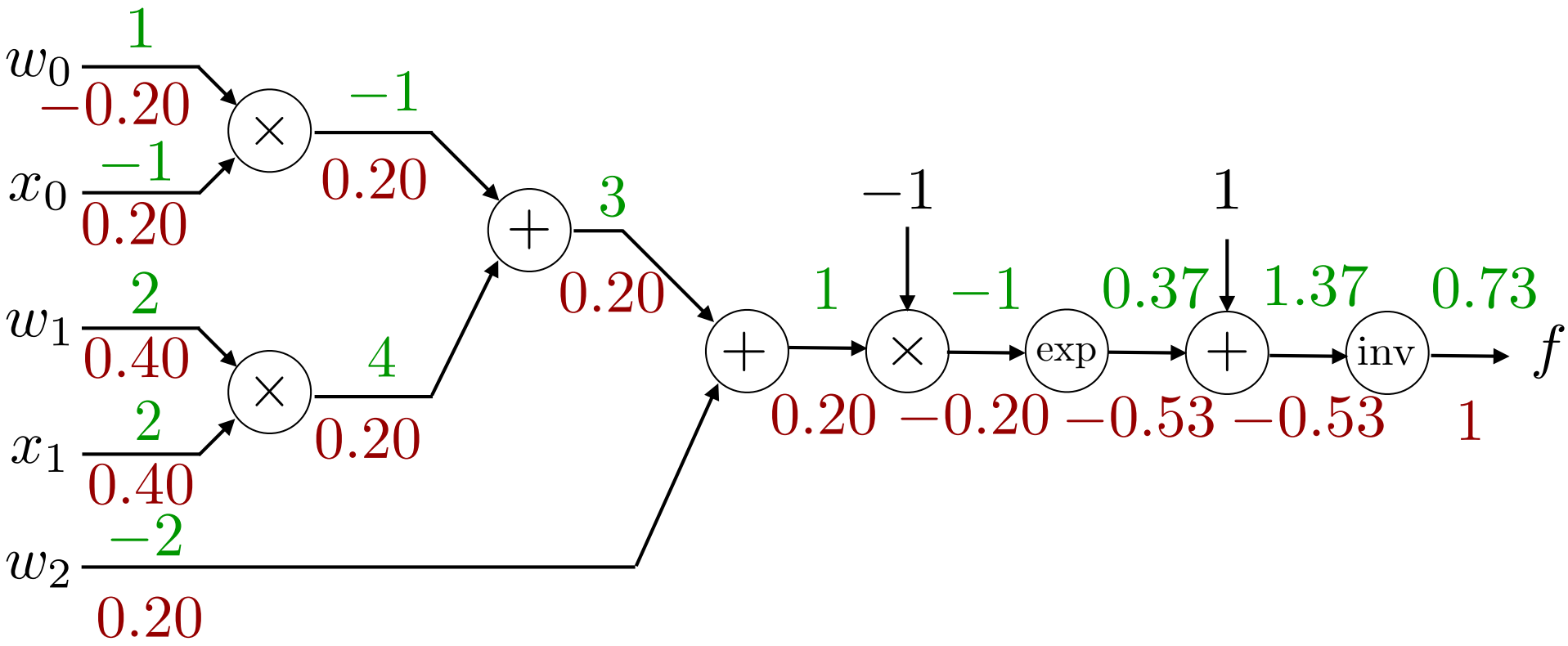
First layer: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

Second layer: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$

Third (output layer): $\mathbf{z} = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3$

Recap from 2018-01-29

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



You can take any gradient this way

With backpropagation, as long as you can **break the computation into components where you know the local gradients, you can take the gradient of anything.**

Multivariate backpropagation

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

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

$h_i = f(W_i h_{i-1} + b_i)$

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?

$y \in \mathbb{R}^n$, $x \in \mathbb{R}^m$

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

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

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

$y = f(x)$

$\Delta y \leftarrow \Delta x$

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

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

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.)

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

A word on notation:

- In the denominator layout definition, the denominator vector changes along columns.

$$\nabla_{\mathbf{x}} y_i = \begin{bmatrix} \frac{\partial y_i}{\partial x_1} \\ \vdots \\ \frac{\partial y_i}{\partial x_m} \end{bmatrix} \quad \nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \triangleq \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

$x \in \mathbb{R}^m \quad y \in \mathbb{R}^n$

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

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

$\frac{\partial y}{\partial x} \in \mathbb{R}^{m \times n}$

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

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

Example: derivative of a vector with respect to a vector

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

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 $\mathbf{x}$.

$\in \mathbb{R}^{n \times h}$

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

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

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

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & & & \\ \vdots & & & \\ w_{h1} & & \cdots & w_{hn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

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,

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

Equivalently:

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

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

$y \in \mathbb{R}^n$ $\quad z = g(y)$

$z \in \mathbb{R}^p$

$z = g(f(x))$

- 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

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

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

$z \in \mathbb{R}^p$

$$\Delta x \rightarrow \Delta z$$

$$\Delta x \rightarrow \Delta y \rightarrow \Delta z$$

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

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

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

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

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

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

Chain rule for vector valued functions (cont.)

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

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

- The chain rule is intuitive, since:

$$\begin{aligned} \Delta\mathbf{y} &\approx (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x} \\ \Delta\mathbf{z} &\approx (\nabla_{\mathbf{y}}\mathbf{z})^T \Delta\mathbf{y} \end{aligned}$$

Composing these, we have that:

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

- 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:

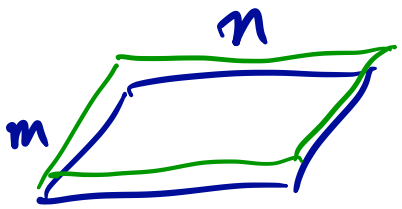
$$\nabla_{\mathbf{x}}\mathbf{z} = \nabla_{\mathbf{x}}\mathbf{y}\, \nabla_{\mathbf{y}}\mathbf{z}$$

Tensor derivatives

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

$$\frac{\partial y}{\partial w} \in \mathbb{R}^{m \times n \times m}$$

$$\frac{\partial y_i}{\partial w} \in \mathbb{R}^{m \times n}$$



$$\frac{\partial y_1}{\partial w} \qquad \frac{\partial y_2}{\partial w}$$

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.

$\mathcal{E}^{(i)} = \frac{1}{2} z^{(i)T} z^{(i)}$

Multivariate chain rule and tensor derivative example

Consider the squared loss function:

$z^{(i)} =$

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

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 $\mathbf{x}^{(i)}$.

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

$$z^{(i)} = y^{(i)} - W x^{(i)}$$

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

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

$(m \times n)$

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

$(m \times n \times 1)$

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.

$$\frac{\partial \varepsilon}{\partial W} = \frac{\partial z}{\partial W}\frac{\partial \varepsilon}{\partial z} \qquad \varepsilon = \frac{1}{2} z^T z$$

z

Multivariate chain rule example

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

$$z = y - Wx$$

$$z_k = y_k - \sum_j W_{kj} x_j$$

$$\frac{\partial z_k}{\partial W} \in \mathbb{R}^{m \times n}$$ } I have m of these.

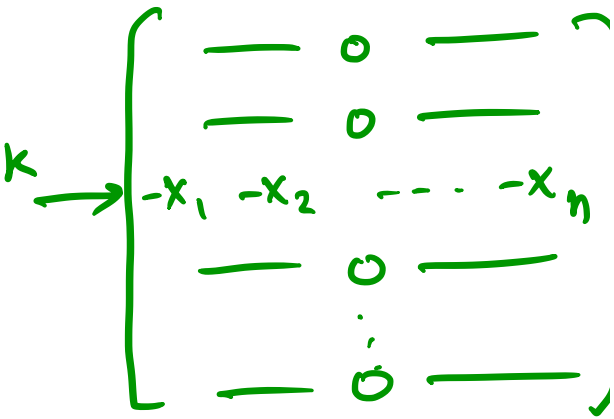
$$\frac{\partial z_k}{\partial W} = \frac{\partial}{\partial W}\left[y_k - \sum_j W_{kj} x_j \right]$$

$\frac{\partial z_k}{\partial W}$

$$\frac{\partial z_k}{\partial W_{ip}} = -\frac{\partial}{\partial W_{ip}}\left(\sum_j W_{kj} x_j \right)$$

$$(i = k) \Rightarrow \frac{\partial z}{\partial W_{kp}} = -x_p$$

$$(i \neq k) \Rightarrow \frac{\partial z}{\partial W_{ip}} = 0$$



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 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.

Multivariate chain rule example

$(Ax)_i = \sum_{j=1}^{n} a_{ij} x_j$

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

$\frac{\partial \varepsilon}{\partial W}$

$$\frac{\partial z}{\partial W} \frac{\partial \varepsilon}{\partial z} = \sum_{i=1}^{m} \frac{\partial z_i}{\partial W} \frac{\partial \varepsilon}{\partial z_i}$$

$\frac{\partial \varepsilon}{\partial z_i}$ ← scalar

$$= \frac{\partial \varepsilon}{\partial z_1} \begin{bmatrix} -x^T \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \frac{\partial \varepsilon}{\partial z_2} \begin{bmatrix} 0 \\ -x^T \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

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

scalar

$m \times 1$

z

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

$(m \times n)$

Multivariate chain rule and tensor derivative example (cont.)

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

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

$$= \sum_{i=1}^{m} \frac{\partial z_i}{\partial \mathbf{W}} \left(\frac{\partial \varepsilon}{\partial \mathbf{z}} \right)_i$$

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

$$+ \frac{\partial \varepsilon}{\partial z_m} \begin{bmatrix} \mathbf{0}^T \\ \mathbf{0}^T \\ \vdots \\ -\mathbf{x}^T \end{bmatrix}$$

$z = y - Wx$

$z = Wx$

$\nabla_x x^T y = y$

$\nabla_x Wx = W^T$

Multivariate chain rule and tensor derivative example (cont.)

Continuing the work from the previous page...

$m \times n$

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

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

$\frac{\partial \ell}{\partial z} \in \mathbb{R}^{m \times 1}$

$\frac{\partial (y - Wx)}{\partial W}$

$-x^T \quad \in \mathbb{R}^{1 \times n}$

$-\frac{\partial \ell}{\partial z} x^T$

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.

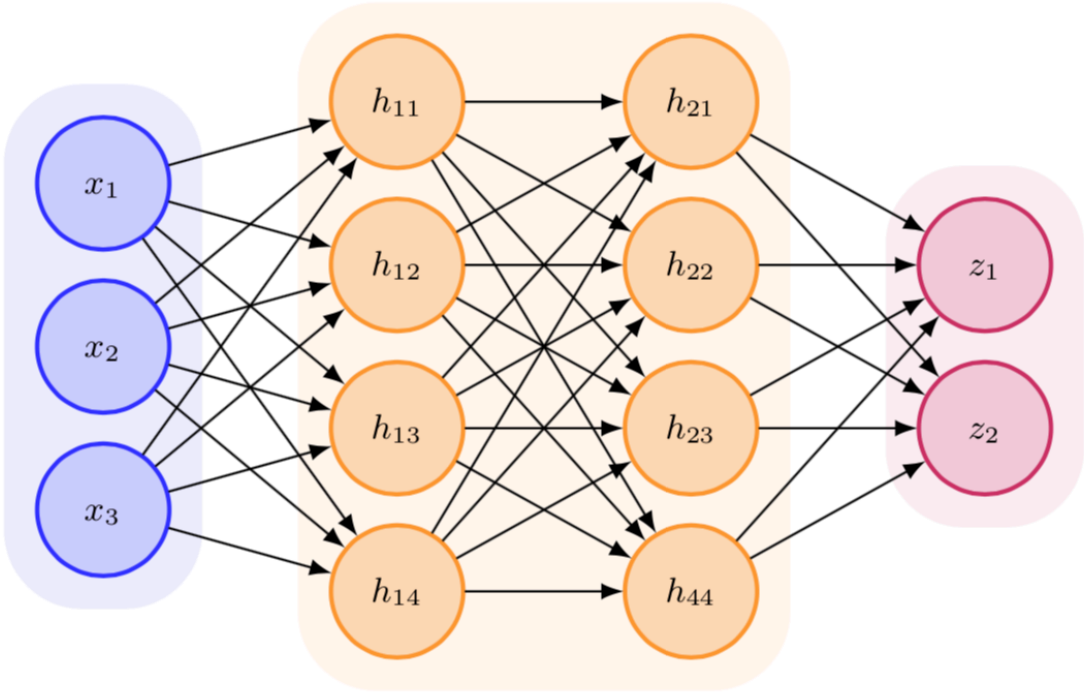
Multivariate chain rule example

$$\Delta \ell \approx \left(\frac{\partial \ell}{\partial z}\right)^T \Delta z$$

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).

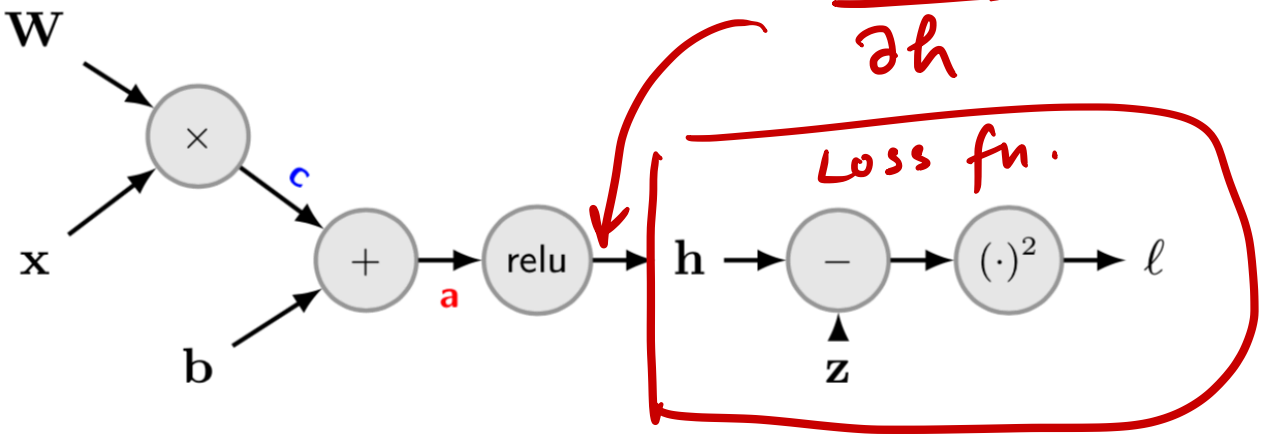
Back to backpropagation



Backpropagation for a neural network layer

Backpropagation: neural network layer

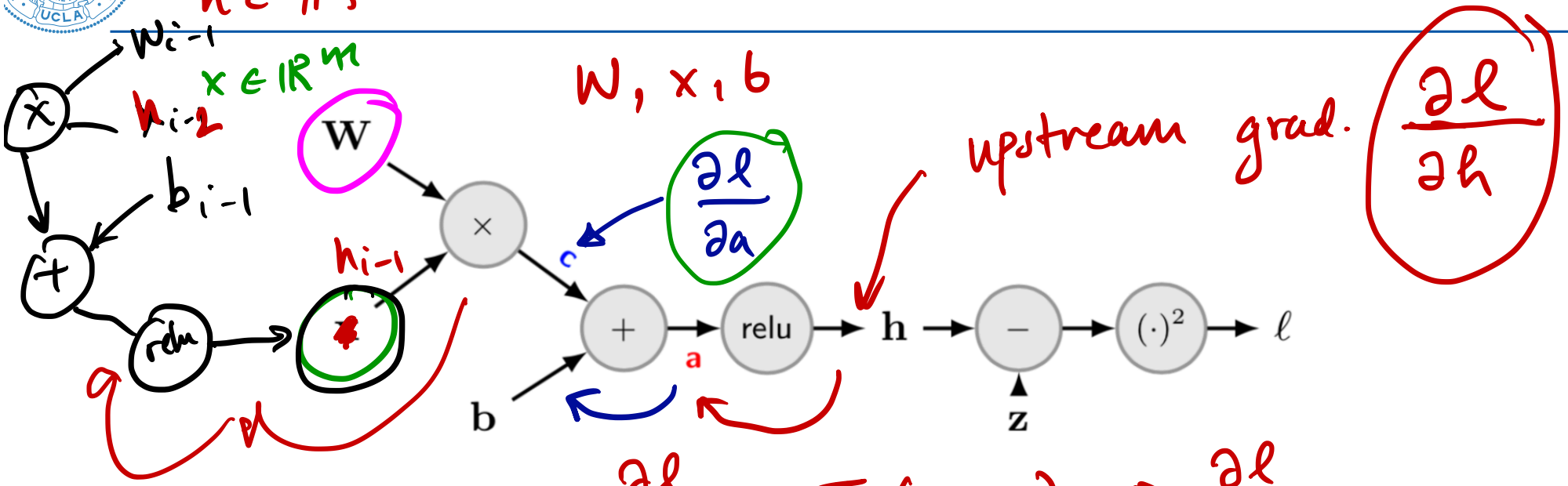
Here, $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output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



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

$h \in \mathbb{R}^{h}$

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

W, x, b

upstream grad. $\frac{\partial \ell}{\partial h}$

$$\frac{\partial \ell}{\partial a} = \mathbb{I}(a > 0) \odot \frac{\partial \ell}{\partial h}$$

$$\frac{\partial \ell}{\partial b} = \mathbb{I}(a > 0) \odot \frac{\partial \ell}{\partial h}$$

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

$(m \times h)\ (h \times 1)$

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

$(h \times m)$ $(h \times 1)\ (1 \times m)$

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.

Lecture 6: 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

NN

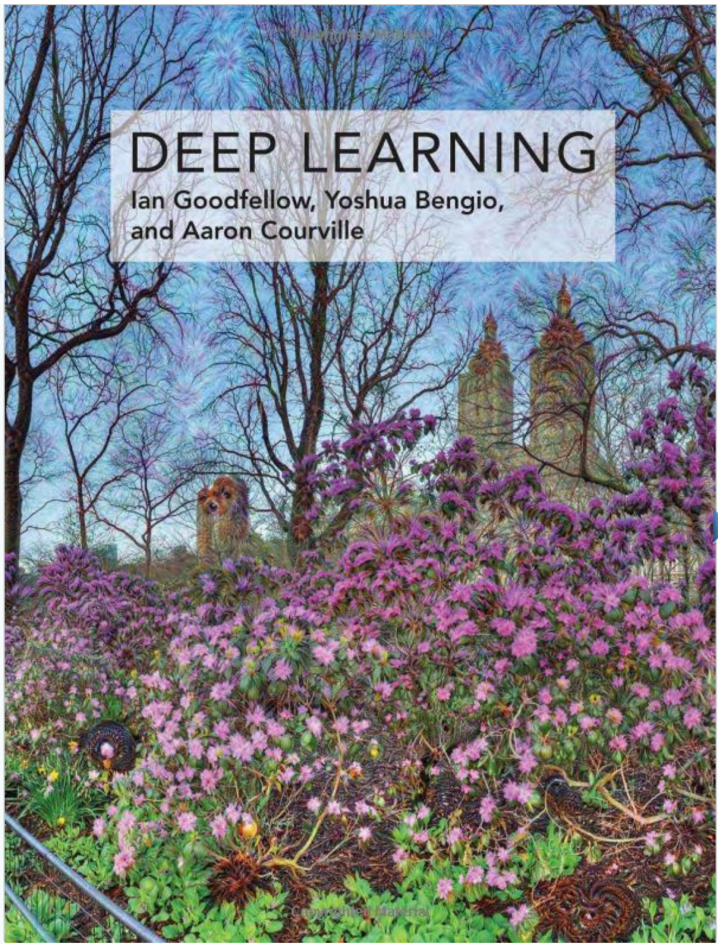
general regularizations

Announcements, 2018-01-31

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

Batch Norm



Initializations

$f(Wx+b)$

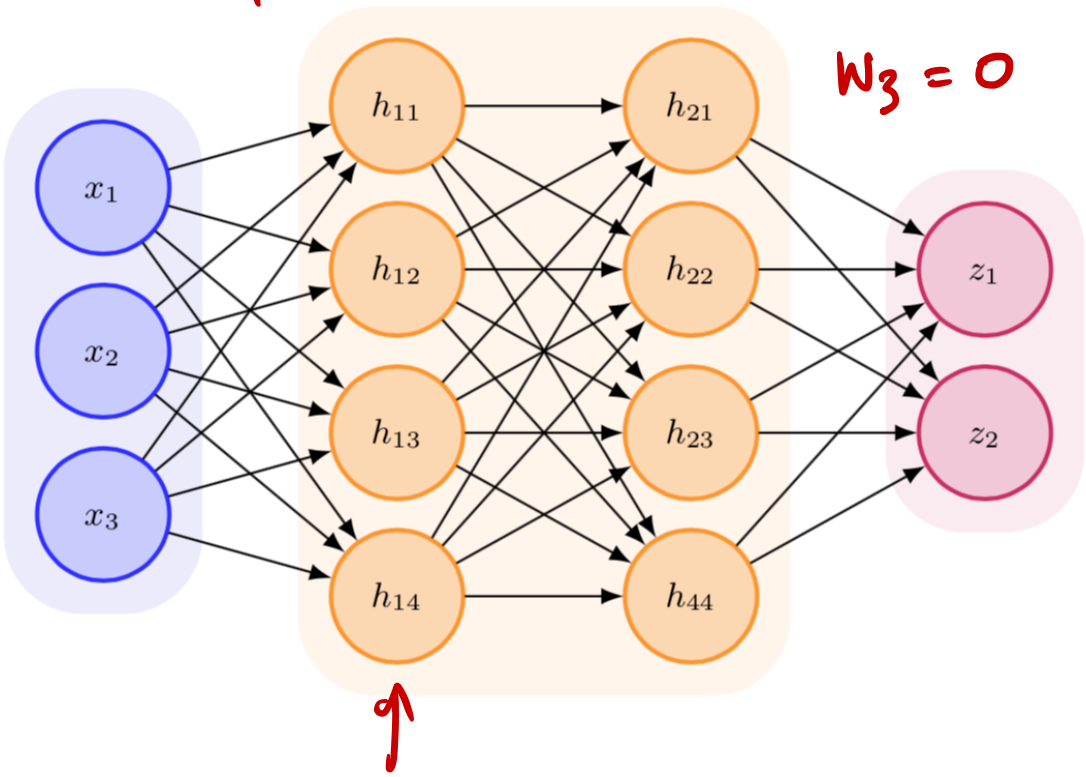
$W_1 = 0$ $W_2 = 0$ $W_3 = 0$

$f(b)$

$f(0)$

$\sigma(0) = 0.5$

$\tanh(0) = 0$



$f(b)$

Initializations matter.

What if **W** is initialized to zero?

Small random weight initialization

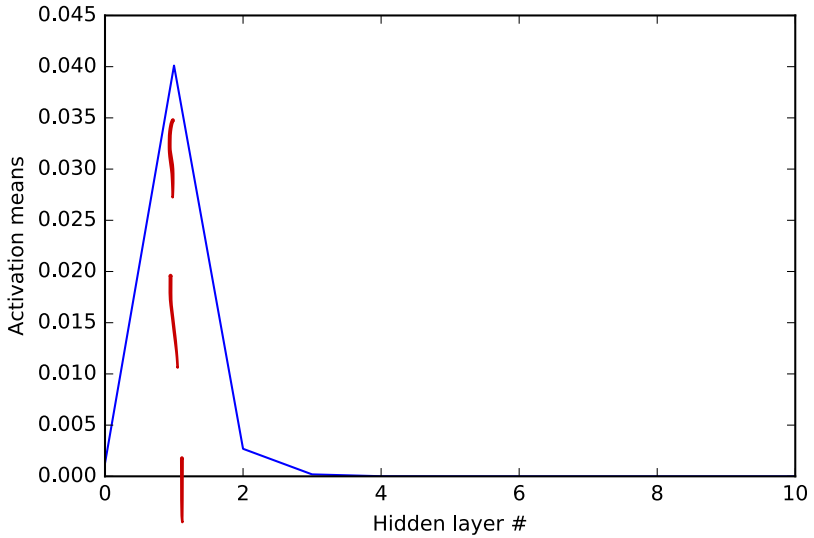
How about small random weight initializations?

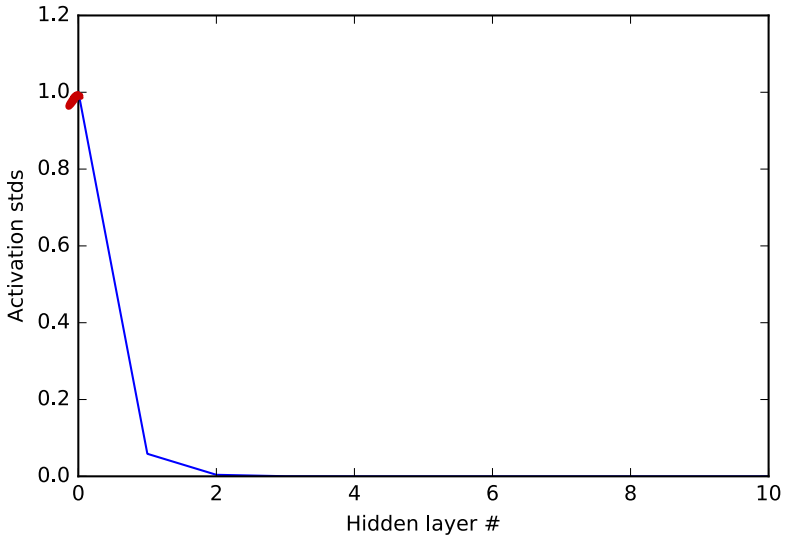
The thought is that we don't set them large enough to bias training.

```
# 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

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





h_1

$x \sim N(0, 1)$

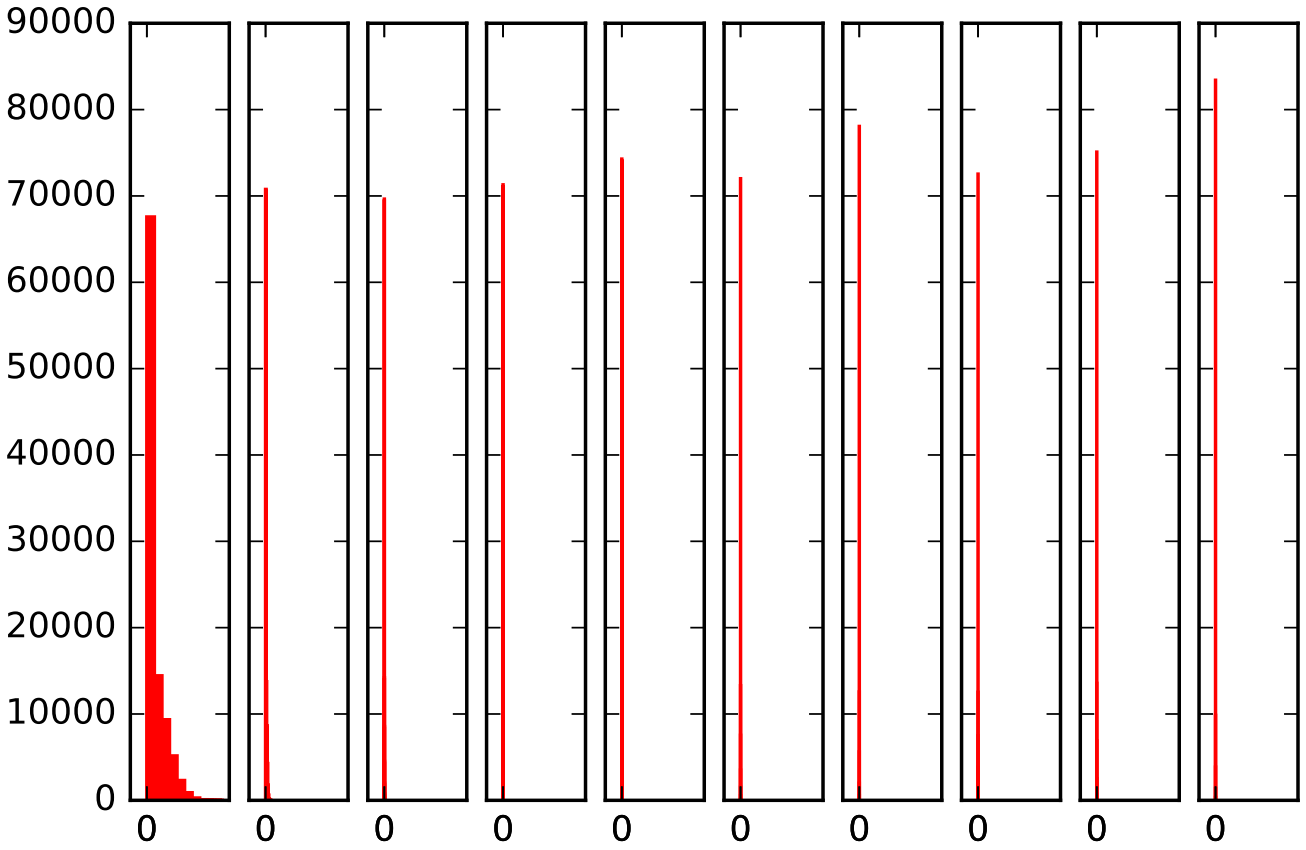
$h_1 = \text{relu}(Wx + \cancel{b}^{0})$

$h_2 = \text{relu}(Wh_1)$

Small random weight initialization

$$\frac{\partial \ell}{\partial W_{10}} = \frac{\partial \ell}{\partial c} h_{10}^T$$

Distribution of each layer's activations:



$$\frac{\partial L}{\partial h_{10}}$$

Small random weight initialization

What about backpropagation with this small weight initialization?

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

Small random weight initialization

What about backpropagation with this small weight initialization?

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

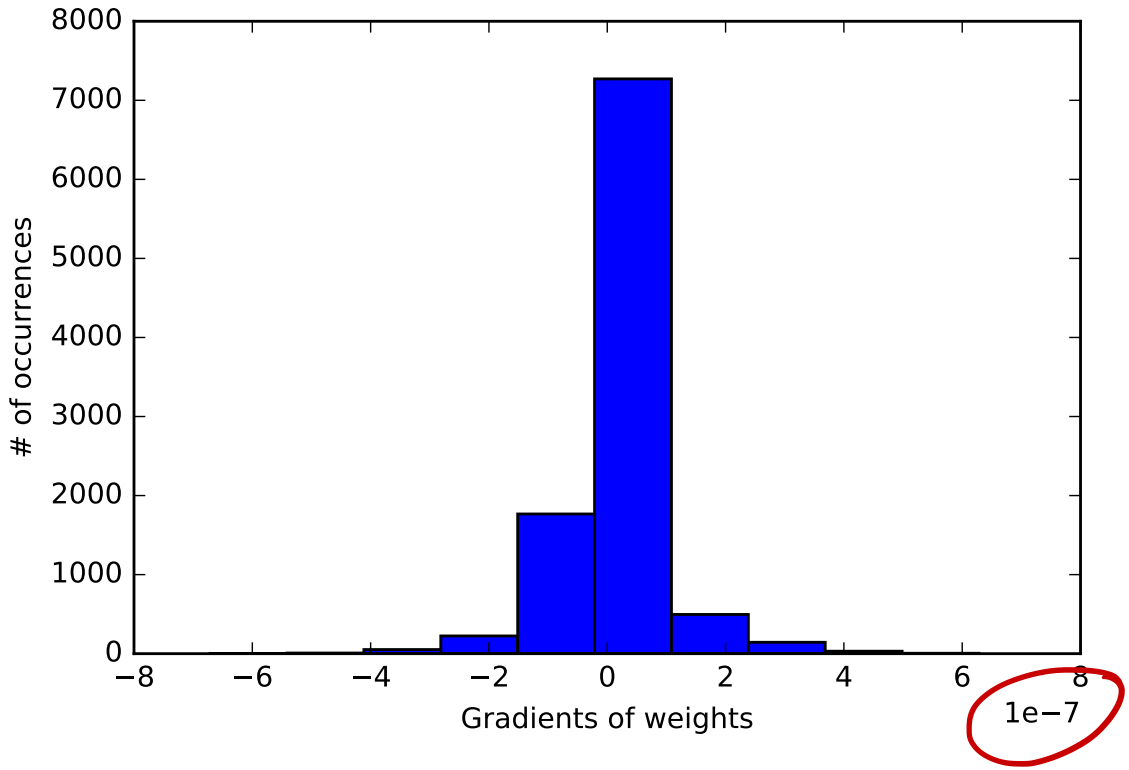
```
# Now backprop
dLdH = 100*np.random.randn(100, 1000) # 1000 losses for examples
loss_grads = [dLdH]
grads = []

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

What about backpropagation with this small weight initialization?

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

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.

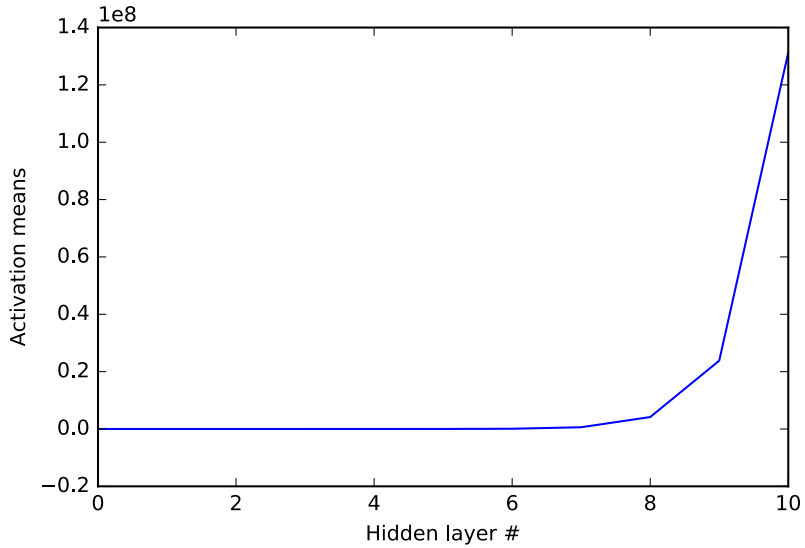
Large random weight initialization

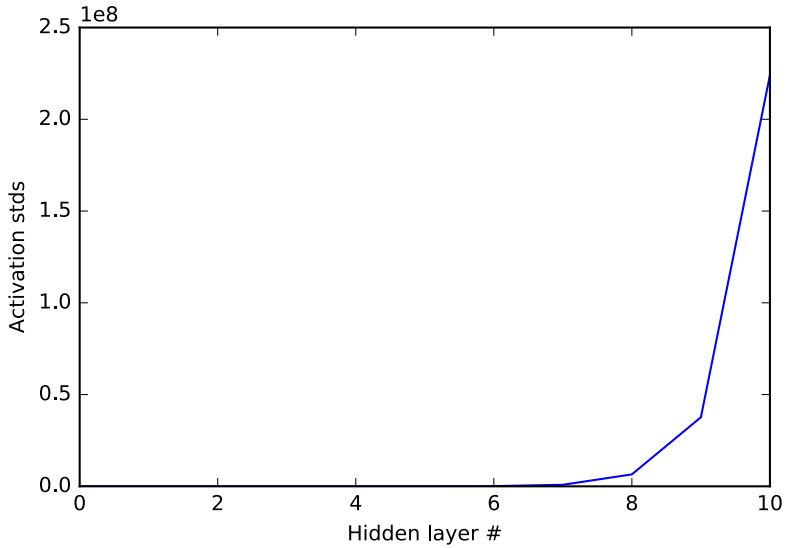
How about larger random weight initializations?

```
# 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.





What happens to the gradients?

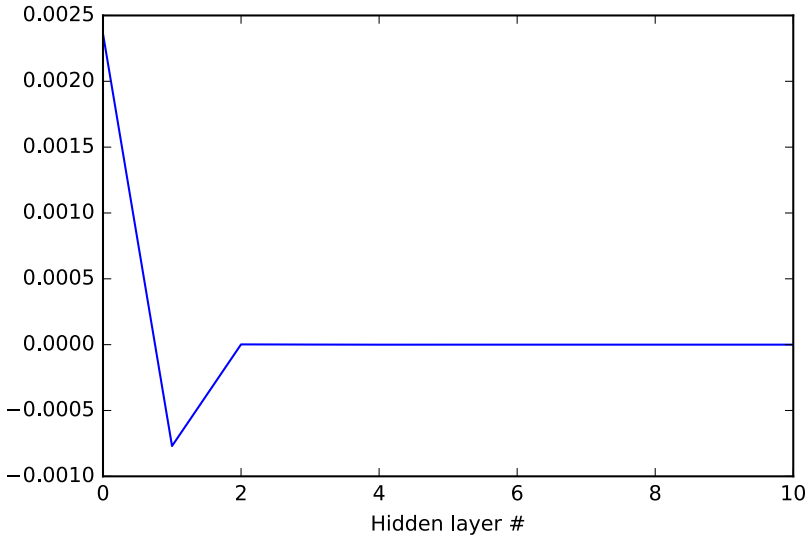
Again, this is a problem.

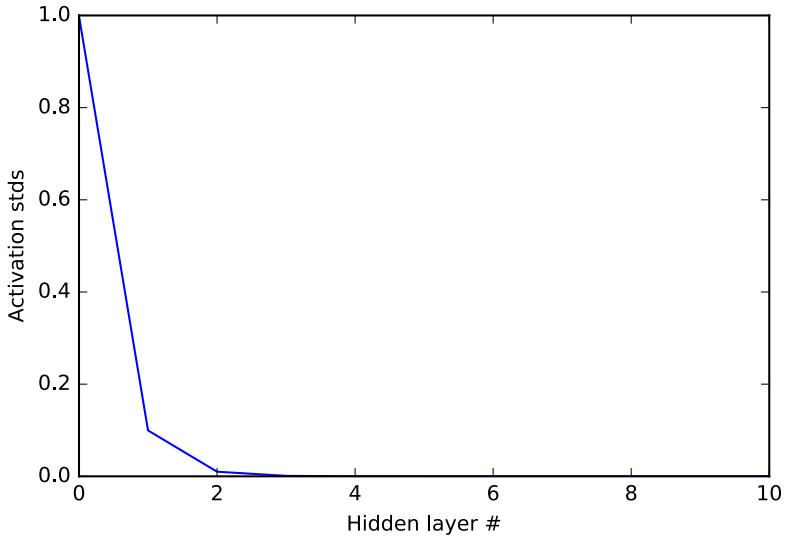
Will this problem occur for other activations?

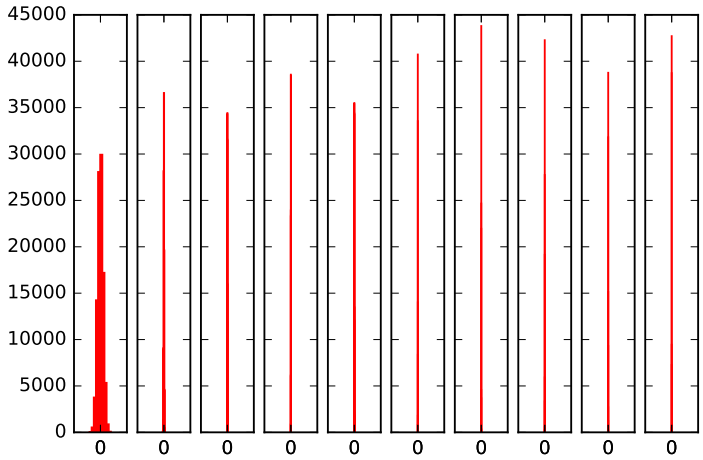
What happens when we use the tanh() activation with small initialization?

Will this problem occur for other activations?

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

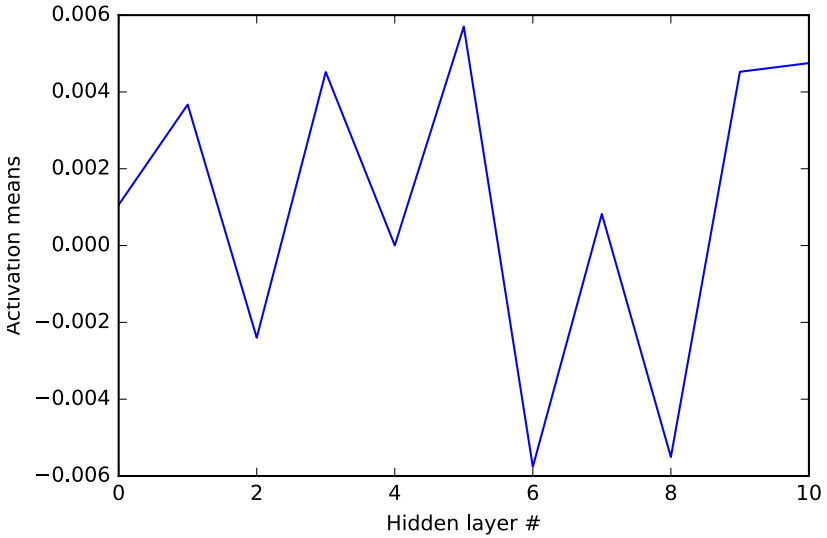


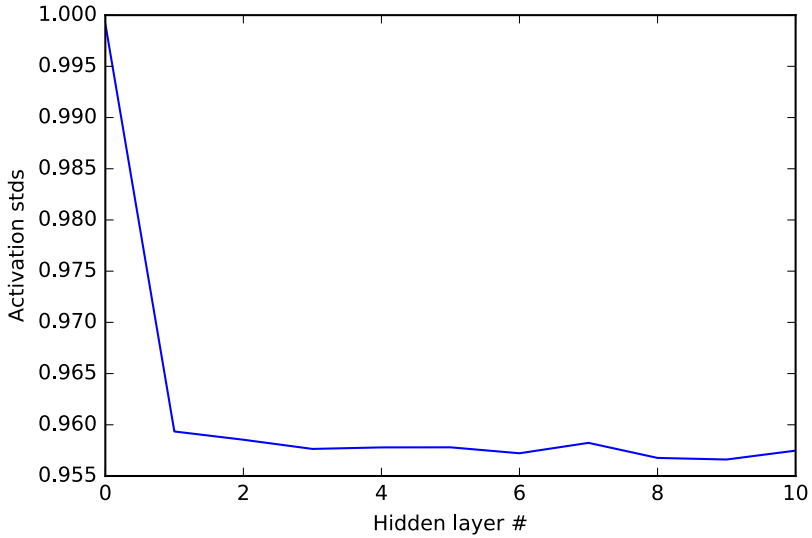


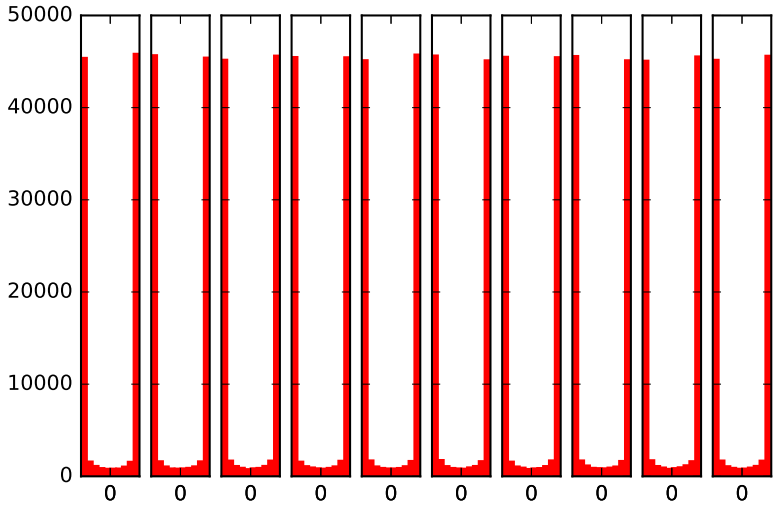


Will this problem occur for other activations?

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







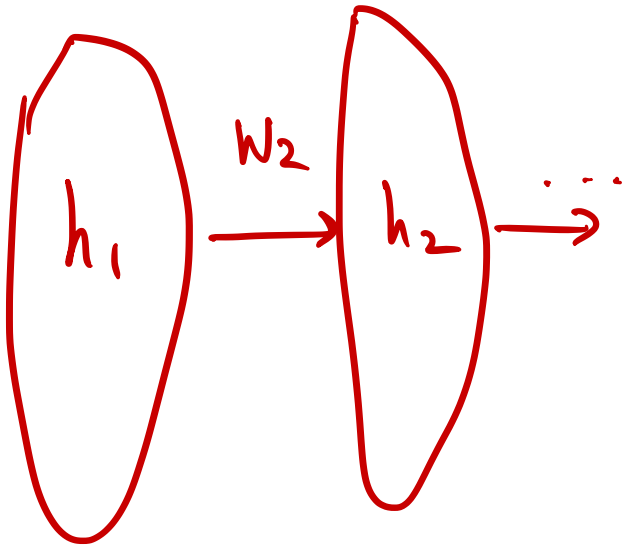
If not small and not large initialization, then what?

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.



$$\text{var}(h_1) \approx \text{var}(h_2) \approx \text{var}(h_i)$$

$$\text{var}(\nabla_{W_1} \mathcal{L}) \approx \text{var}(\nabla_{W_2} \mathcal{L})$$

Xavier initialization

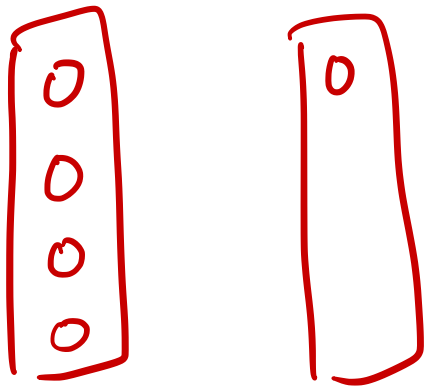
For simplicity, we assume the input has equal variance across all dimensions. Then, each unit in each layer ought to have the same statistics. For simplicity we'll denote h_i to denote a unit in the ith layer, but the variances ought be the same for all units in this layer.

Concretely, the heuristics mean that

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

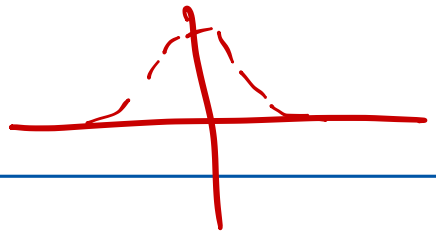
and

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

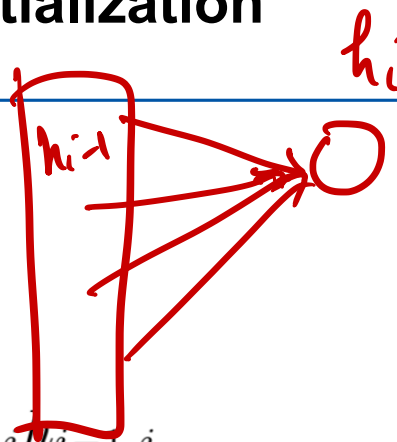


J, L, l

All units in layer i have same var.



Xavier initialization



$w_i^T h_{i-1}$

Xavier initialization (cont.)

If the units are linear, then

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

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

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

then,

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

$\text{var}(h_i) = \text{var}(h_{i-1})$

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

$= n_{in}\,\text{var}(w_{ij})$

Xavier initialization

$h = f(w^T x + b)$

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

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

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

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

$\frac{2}{n_{out}+n_{in}}$ $\frac{n_{out}+n_{in}}{2}$

Xavier initialization (cont).

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

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

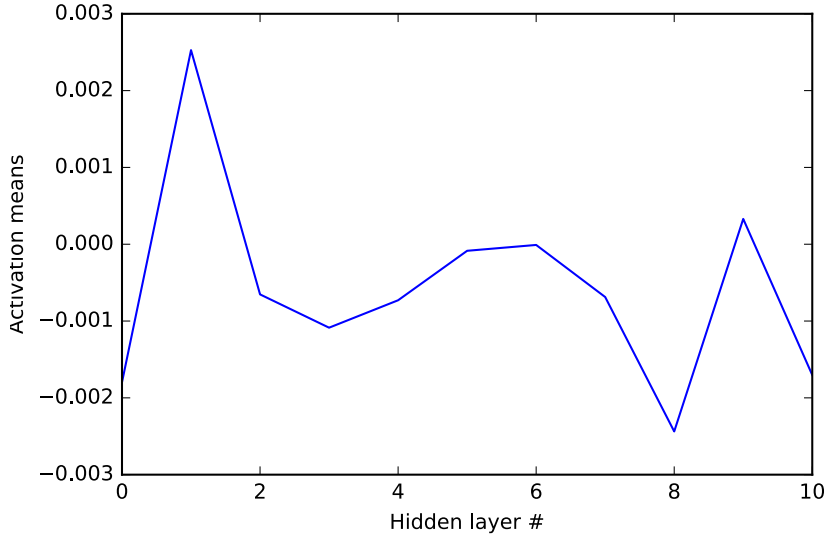
Hence, we can initialize each weight in layer i to be drawn from:

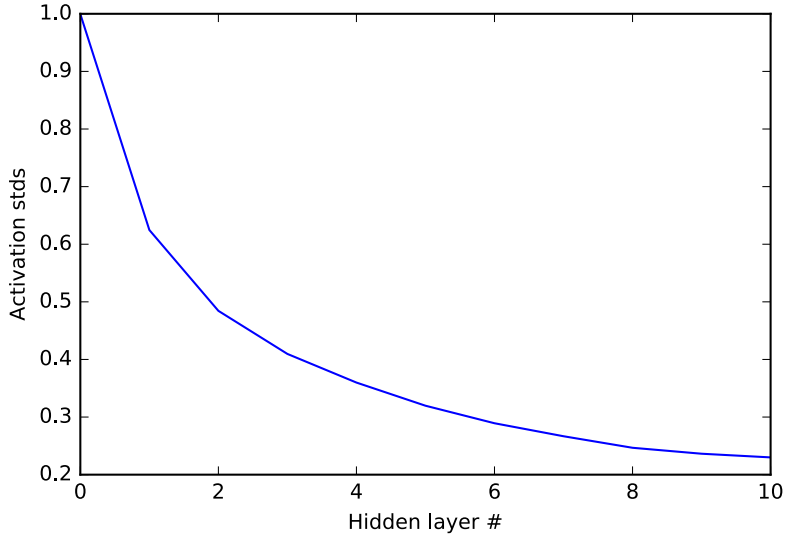
$$\mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

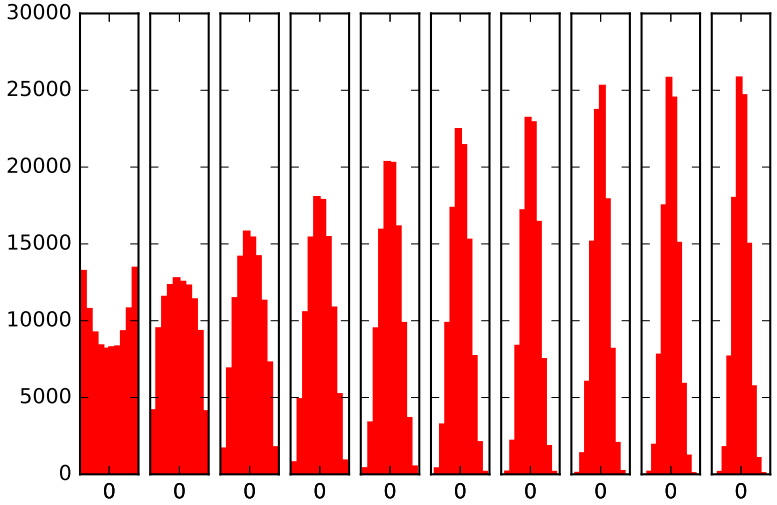
Xavier initialization with tanh

```
# 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







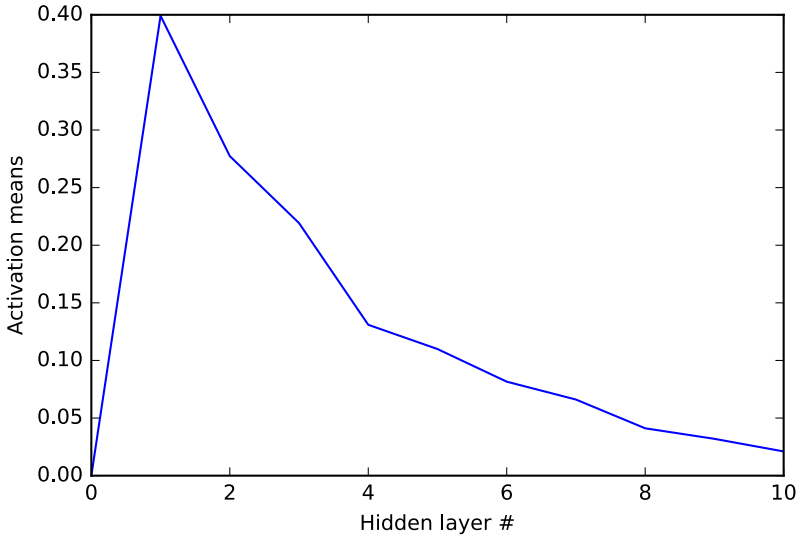
Xavier initialization

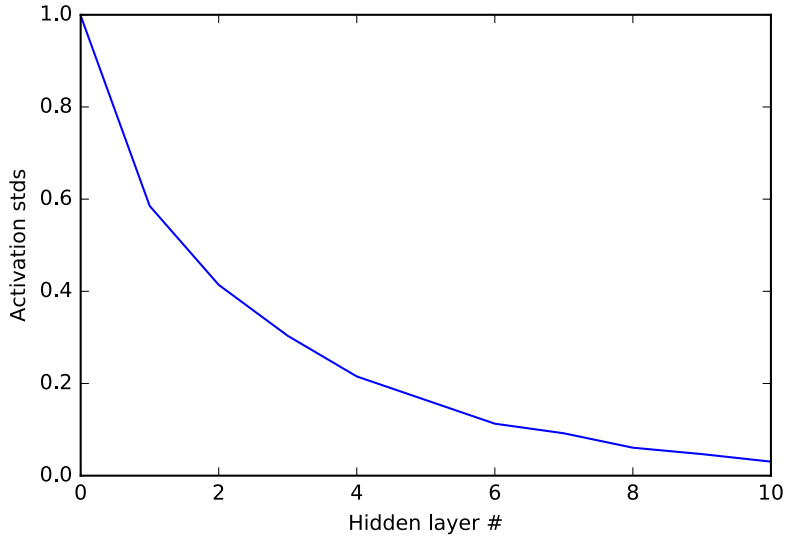
A few notes:

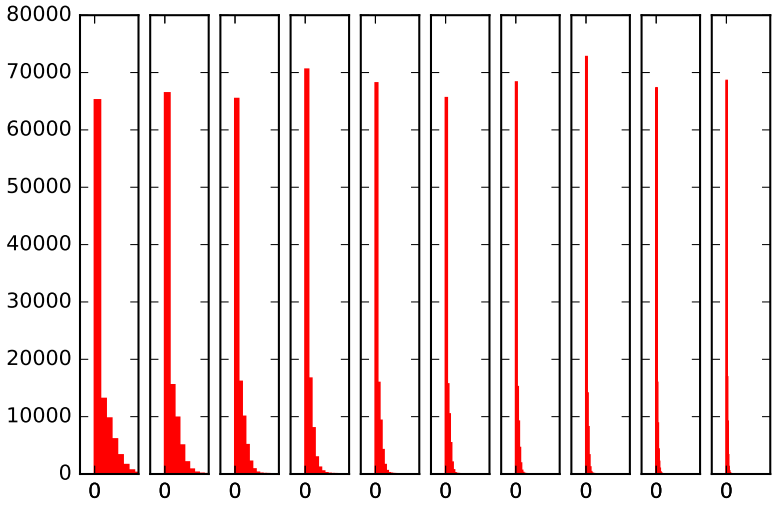
- If you use Caffe (and / or talk to other people), the Xavier initialization sets the variance to $1/n_{\text{in}}$. This is equivalent to the above form if $n_{\text{in}} = n_{\text{out}}$.
- However, the Xavier initialization (either one) typically leads to dying ReLU units, though it is fine with $\tanh$ 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:

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

Xavier initialization with ReLU



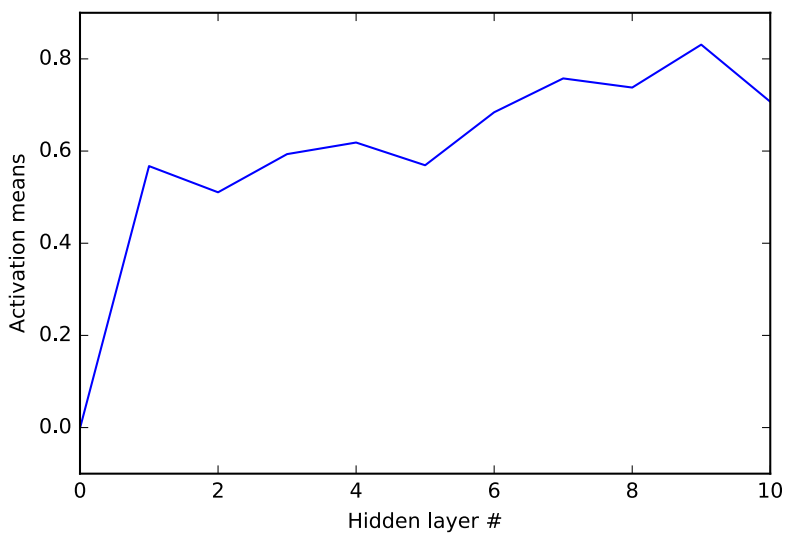


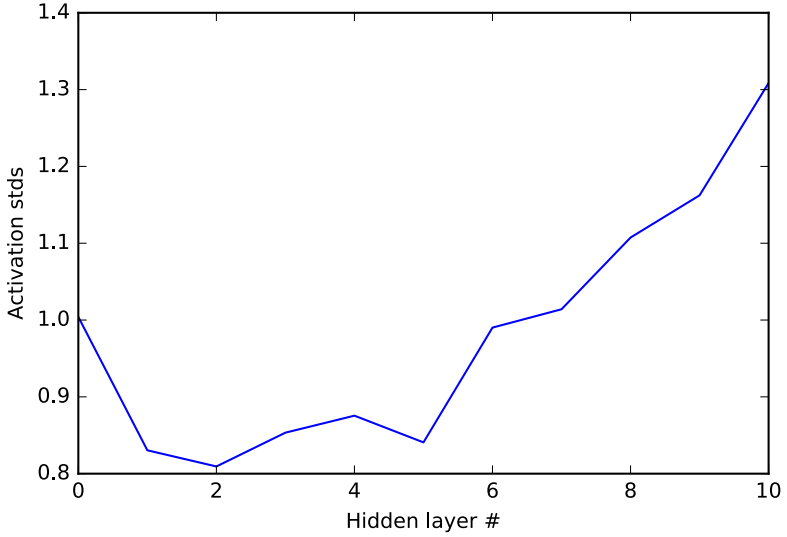


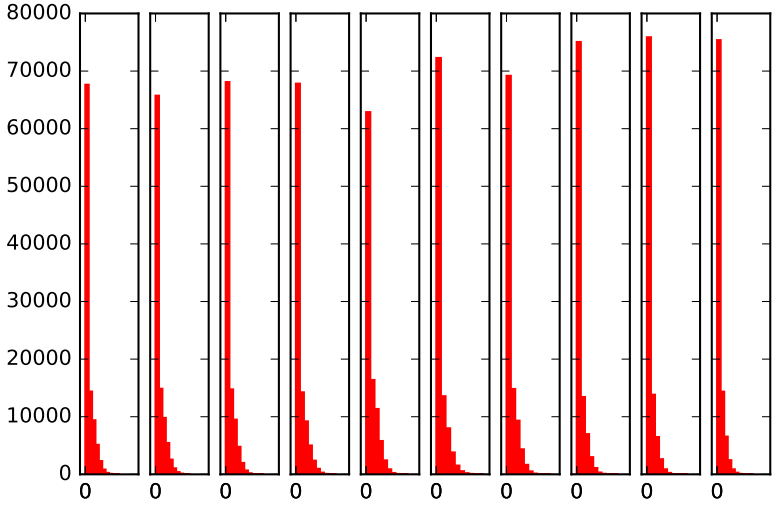
He (MSRA) initialization with ReLU

```
# 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



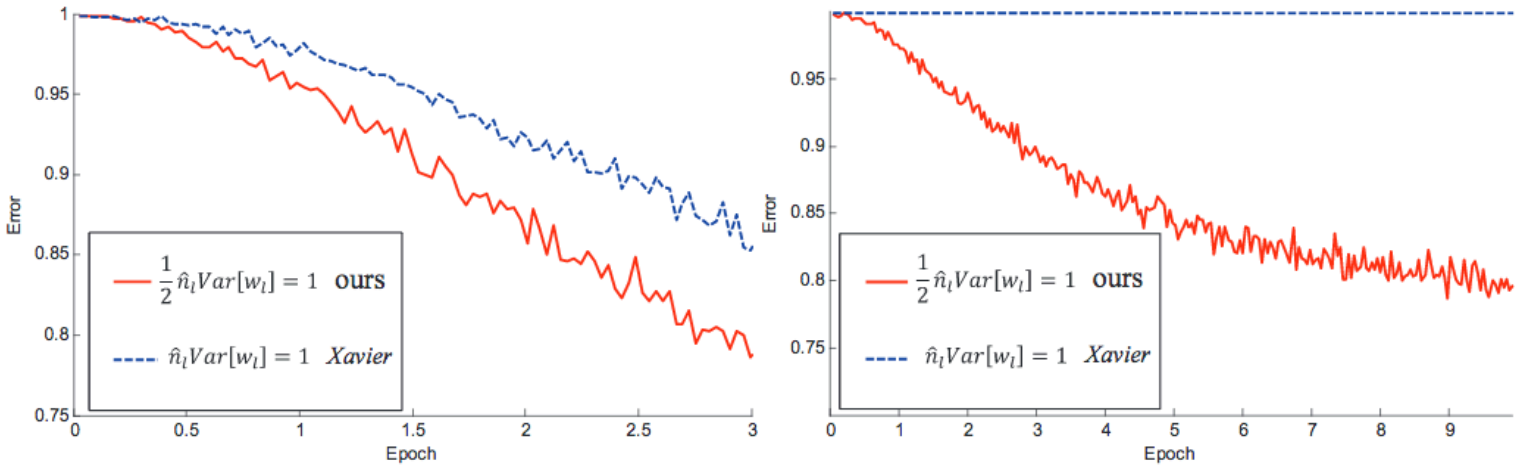




Initialization take home points

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